Master's Thesis

Evaluation of Post Quantum Cryptography on IOT Hardware

submitted by

Niyati Tumkur Venugopal

from Stuttgart

Hochschule für Technik Stuttgart

Degree program	M. Sc. Software Technology
Examined by	Prof. Dr. Jan Seedorf
Supervised by	Dr. Darshana Rawal
Submitted on	September 2024

Declaration Of Originality

Master's Thesis by Niyati Tumkur Venugopal (M. Sc. Software Technology)
Student number: 1006303
Title: Evaluation of Post-Quantum Cryptography on IOT Hardware

I hereby confirm:

- That this thesis represents my own independent work.
- That only the sources explicitly listed have been used, and all material borrowed from other works, whether directly quoted or paraphrased, has been appropriately acknowledged.
- That this thesis has not been previously submitted, either in whole or in part, for any other academic examination or degree.

Signature: ______________________
Date: _________________

Abstract

The rapid development of quantum computing poses a significant threat to traditional cryptographic systems, particularly those used in securing communications over the internet. Current public-key cryptosystems like RSA and ECC are vulnerable to quantum attacks, prompting the need for post-quantum cryptographic solutions. This study focuses on the evaluation of various post-quantum algorithms, their performance, and security implications, particularly Kyber and SPHINCS+. The experiments conducted compare these post-quantum algorithms with traditional cryptographic systems, such as RSA and Diffie-Hellman, to assess the performance differences in practical scenarios.

By examining the current state of post-quantum cryptography and evaluating its practical implementation in client-server platforms, this research aims to provide insights into the performance and feasibility of these algorithms. Key metrics, including computational efficiency, key generation time, and latency, are analysed to offer a comparative understanding of post-quantum and traditional cryptographic systems.

The primary objective is to ensure that organizations can be prepared for quantum computing by adopting quantum-resistant algorithms in critical systems. This research demonstrates that post-quantum cryptography methods provide superior security compared to traditional systems, particularly in the context of future quantum computing threats. It contributes to the ongoing discourse in post-quantum cryptography by offering practical experimental results and a comparative analysis, with potential future work involving the integration of these algorithms into broader security protocols like TLS.

Contents

List Of Figures

1.Introduction

1.1 Introduction

As quantum computing technology advances, it presents a significant risk to the security of current cryptographic protocols. Traditional cryptographic algorithms, such as RSA and Elliptic Curve Cryptography (ECC), are vulnerable to attacks by quantum computers using algorithms like Shor's algorithm [1]. This presents a pressing need to develop and implement post-quantum cryptographic algorithms capable of resisting such quantum attacks. This section will explore the growing importance of post-quantum cryptography, the characteristics of various post-quantum algorithms protocol, and the associated performance and security implications. It will also describe the problem, motivations behind this research, and the objectives aimed at achieving a robust post-quantum cryptographic system.

Quantum computing threatens to undermine the security of current cryptographic systems that rely on the difficulty of mathematical problems such as integer factorization and discrete logarithms. Shor's algorithm, for instance, can efficiently solve these problems, which are the foundation of RSA and ECC encryption methods [1]. The rapid development of quantum computers thus necessitates a transition to cryptographic algorithms that are resistant to quantum attacks to ensure data security in a post-quantum era.

The urgency of this transition is underscored by the need for cryptographic systems that can withstand not only classical threats but also future quantum-based threats. This shift involves exploring and implementing new cryptographic approaches that are inherently secure against the capabilities of quantum computing [4].

1.2 Post-Quantum Cryptographic Algorithms

1.2.1 Background of Post-Quantum Algorithms

Post-quantum cryptography refers to a variety of algorithms designed to remain secure in the presence of quantum computing. These algorithms include:

- **Lattice-Based Cryptography:** Techniques such as NTRU Encrypt and NTRUSign leverage the hardness of lattice problems. These problems are considered resistant to quantum attacks due to their computational complexity [6][13].
- **Hash-Based Cryptography:** This approach, including schemes like Merkle trees, relies on the security of hash functions. Hash-based cryptographic systems provide a robust framework for secure digital signatures [6].
- **Code-Based Cryptography:** Methods such as McEliece encryption use the difficulty of decoding random linear codes. Although these methods offer strong security guarantees, they often have large key sizes [10].
- **Multivariate Polynomial Cryptography:** These schemes are based on solving systems of multivariate quadratic equations, which are difficult for quantum computers to solve efficiently [10].

These algorithms offer varying levels of security, performance, and implementation complexity, making them suitable for different applications depending on the specific requirements of the cryptographic system [10].

1.2.2 Theoretical Analysis

Evaluating the suitability of post-quantum algorithms involves assessing several factors:

- **Security:** The algorithms must provide strong resistance against quantum attacks to ensure data confidentiality and integrity [6][13].
- **Computational Efficiency:** The performance impact of implementing these algorithms must be evaluated, including computational overhead and latency].

  This evaluation process is critical to ensure that the selected post-quantum algorithms can effectively secure communications while maintaining the performance efficiency [13].

1.2.3 Experimental Analysis of Post -Quantum Algorithms: Performance and Security Considerations

Post-quantum cryptographic algorithms are essential for safeguarding digital communications against both classical and quantum attacks. This research focuses on evaluating the performance and security of these algorithms to ensure they meet practical requirements.

Key aspects of performance analysis include:

- **Computational Requirements:** The computational complexity of post-quantum algorithms affects processing time and resource usage [6].
- **Latency:** The impact on latency during cryptographic operations must be evaluated to ensure minimal delay in secure communications [6].

This research includes building a client-server platform where various post-quantum cryptographic algorithms were implemented were and compared with traditional cryptographic systems to analyse their performance under realistic conditions.

In terms of security, post-quantum cryptographic algorithms must offer robust protection against future quantum attacks. Key considerations include: Key considerations include:

- **Potential Vulnerabilities:** Analysing potential weaknesses in post-quantum algorithms helps ensure robust protection against emerging threats [10][11].
- **Robust Protection:** Ensuring that the algorithms provide strong security guarantees is essential for maintaining data security in a post-quantum era [12].

A comprehensive security analysis is necessary to verify that post-quantum algorithms effectively address the challenges posed by quantum computing [11].

1.3 Problem Description and Motivation

As quantum computing technology continues to advance, it poses a serious threat to the security of existing cryptographic systems, particularly those based on algorithms like RSA and ECC. These traditional cryptographic methods rely on the difficulty of certain mathematical problems, such as factoring large integers or solving discrete logarithms, which are currently computationally infeasible with classical computers. However, quantum computers can solve these problems efficiently using algorithms like Shor's algorithm, potentially rendering these cryptographic methods obsolete [1][2].

The primary issue at hand is that current cryptographic systems will become vulnerable to attacks once sufficiently powerful quantum computers are available. This vulnerability threatens the confidentiality, integrity, and authenticity of data transmitted across networks. As a result, there is an urgent need to transition to cryptographic algorithms that are resistant to quantum attacks. This shift involves developing and implementing post-quantum cryptographic algorithms that can secure communications even in a future where quantum computing is a reality.

The motivation behind this research is to evaluate post-quantum cryptographic algorithms to understand their performance implications and security benefits when compared to traditional algorithms in a real-world setting. The experimental focus is on building a secure client-server platform and measuring key metrics like computational time and latency. The research aims to provide solutions that will help protect sensitive information from being compromised by advances in quantum computing.

1.4 Goals and Scope

1.4.1 Objectives of the Research

The primary goal of this research is to investigate the feasibility and implications of incorporating post-quantum cryptographic algorithms in real-world systems. This involves several key objectives:

1. **Evaluate Post-Quantum Algorithms:** Analyse various post-quantum cryptographic algorithms to determine their suitability for practical use

2. **Compare with Traditional Algorithms**: Measure the performance differences between post-quantum algorithms and traditional cryptographic systems like RSA and Diffie-Hellman.
3. **Analyse Performance and Security:** Examine computational overhead, latency, and overall efficiency to ensure that the new cryptographic methods enhance security without degrading performance.

By achieving these objectives, the research aims to contribute to the development of secure cryptographic systems capable of defending against both current and future quantum threats.

1.4.2 Scope and Limitations

The scope of this research is focused specifically on evaluating and integrating post-quantum cryptographic algorithms in practical cryptographic systems. The research does not cover all aspects of post-quantum cryptography but is concentrated on practical applications relevant to securing communications and data in the face of quantum computing threats.

Several limitations are acknowledged in this research:

1. **Computational Resources:** The evaluation and implementation of post-quantum algorithms are constrained by current computational capabilities. This may affect the feasibility of certain algorithms and their performance in real-world scenarios.
2. **Evolving Technology:** Both quantum computing and cryptographic research are rapidly evolving fields. New developments in these areas may influence the relevance and effectiveness of the proposed solutions, necessitating ongoing adjustments and updates to the research findings.

Despite these limitations, the research aims to provide valuable insights into how post-quantum cryptographic algorithms can be effectively integrated into cryptographic systems to ensure continued data security in a post-quantum world.

2. Post-Quantum Cryptographic Algorithms

As the potential of quantum computers to break widely used cryptographic systems becomes more apparent, the need for post-quantum cryptographic algorithms has gained urgency. These algorithms are designed to be secure against attacks from both classical and quantum computers. Unlike traditional algorithms such as RSA and ECC, which rely on the difficulty of problems like integer factorization and discrete logarithms, post-quantum cryptography is built on mathematical problems believed to be resistant to quantum attacks. This chapter delves into the primary categories of post-quantum cryptographic algorithms, including lattice-based, code-based, hash-based cryptography, and systems based on multivariate quadratic equations. Each section explores the mathematical foundations, key algorithms, and the practical significance of these approaches in the context of ensuring long-term security.

2.1 Lattice-Based Cryptography

Lattice-based cryptography is emerging as a leading candidate for post-quantum cryptographic systems, primarily due to its strong security guarantees and efficient computation. The cryptographic strength of lattice-based systems is derived from the hardness of certain problems in high-dimensional lattice structures, which remain intractable even for quantum computers. This section provides a detailed overview of lattice-based cryptography, focusing on its mathematical foundation and the most significant algorithms developed within this framework.

Mathematical Foundation and Key Algorithms of Lattice Based Cryptography:

Lattice-based cryptography is fundamentally grounded in the complexity of lattice problems, which are geometric structures consisting of points in a multi-dimensional grid. The most notable hard problems in this domain include:

- **Learning With Errors (LWE)**: The LWE problem is a cornerstone of lattice-based cryptography. It involves solving a system of linear equations with added noise, which makes it difficult to distinguish the resulting values from random ones. This problem is assumed to be quantum-resistant, as no efficient quantum algorithm has been found to solve it [1][13].

- **Shortest Vector Problem (SVP)**: The SVP asks for the shortest non-zero vector in a lattice, a problem that is NP-hard and believed to be resistant to quantum algorithms. The difficulty of this problem underpins the security of several lattice-based cryptographic schemes [13][14].

Key algorithms in lattice-based cryptography include:

- **Kyber**: Kyber is a post-quantum key encapsulation mechanism (KEM) that is part of the NIST post-quantum cryptography standardization project. It is based on the LWE problem and is designed to be efficient in terms of both computation and bandwidth, making it well-suited for secure communication protocols [1].
- **Dilithium**: Dilithium is a digital signature scheme that leverages the hardness of lattice problems like LWE. It offers a balance of security, efficiency, and moderate key sizes, making it a strong candidate for post-quantum digital signatures [1].
- **FrodoKEM**: Another KEM based on the LWE problem, FrodoKEM does not use any additional algebraic structure, making it more conservative and potentially more secure against unforeseen attacks [13].
- **NTRUEncrypt**: NTRUEncrypt is a public-key cryptosystem based on the hardness of finding short vectors in lattices. It is one of the earliest lattice-based cryptographic systems and has been extensively analyzed, making it a robust choice for post-quantum encryption [13].
- **BLISS (Bimodal Lattice Signature Scheme)**: BLISS is a lattice-based digital signature scheme known for its efficiency and compact signature size. It is based on the Ring-LWE problem and offers strong security guarantees [14].
- **Ring-LWE**: An extension of the LWE problem, Ring-LWE allows for more efficient implementations by leveraging ring structures. It serves as the basis for various encryption and signature schemes, including NewHope, which is used in experimental quantum-safe TLS implementations [14].

Lattice-based cryptography's appeal lies not only in its security but also in its adaptability to various cryptographic tasks, including key exchange, encryption, and digital signatures. The versatility and robustness of lattice-based systems make them central to the future of cryptographic security in a quantum era.

2.2 Code-Based Cryptography

Code-based cryptography is a classical approach that has gained renewed interest in the context of post-quantum security. It is based on the hardness of problems in coding theory, particularly the problem of decoding random linear codes. This section explores the mathematical foundations of code-based cryptography and discusses key algorithms that have been proposed to resist quantum attacks.

Mathematical Foundation and Key Algorithms of Code Based Cryptography:

The mathematical foundation of code-based cryptography is rooted in error-correcting codes, which are used to detect and correct errors in data transmission. The security of code-based cryptography relies on the difficulty of the following problem:

- **Decoding a Random Linear Code**: This problem involves recovering the original message from a codeword that has been corrupted by random errors. The challenge lies in the exponential number of possible codewords, which makes decoding computationally infeasible without a special structure or knowledge. This problem is NP-hard and has remained resistant to quantum algorithms [2][18].

Key algorithms in code-based cryptography include:

- **Classic McEliece**: The McEliece cryptosystem is one of the oldest public-key cryptosystems still considered secure in the post-quantum world. It uses large Goppa codes for encryption, providing a high level of security. However, its large key sizes pose challenges for practical implementation, particularly in resource-constrained environments [1][2].
- **Niederreiter Cryptosystem**: The Niederreiter cryptosystem is a variant of McEliece that uses parity-check matrices instead of gen-

erator matrices. It offers similar security properties but with potentially different performance characteristics, depending on the specific implementation [2].

- **BIKE (Bit Flipping Key Encapsulation)**: BIKE is a key encapsulation mechanism that uses code-based techniques to achieve quantum resistance. It is designed to be efficient and secure, with smaller key sizes compared to traditional code-based systems [18].
- **LEDACrypt**: LEDACrypt is a recent proposal that combines low-density parity-check (LDPC) codes with a variant of the Niederreiter cryptosystem. It offers a good trade-off between security and efficiency, making it suitable for post-quantum applications [18].

Code-based cryptography is particularly well-suited for long-term data security, where the high computational cost of large key sizes can be offset by the need for robust encryption. Despite its challenges, such as large key sizes, the resilience of code-based systems against quantum attacks ensures their place in the future cryptographic landscape.

2.3 Hash-Based Cryptography

Hash-based cryptography is a straightforward approach that leverages the properties of cryptographic hash functions to build secure systems, particularly digital signatures. This approach is appealing due to its simplicity and strong security guarantees, even in the presence of quantum computers. In this section, we discuss the mathematical foundations of hash-based cryptography and highlight key algorithms that have been developed in this domain.

Mathematical Foundation and Key Algorithms of Hash Based Cryptography:

Hash-based cryptography is based on the use of cryptographic hash functions, which are mathematical functions that take an input and produce a fixed-size output (the hash) in a way that is computationally infeasible to reverse. The security of hash-based cryptography relies on the following properties:

- **Collision Resistance**: It should be difficult to find two different inputs that produce the same hash output.

- **Pre-image Resistance**: Given a hash output, it should be difficult to find any input that maps to that output.
- **Second Pre-image Resistance**: Given an input and its hash, it should be difficult to find a different input that produces the same hash [5].

Key algorithms in hash-based cryptography include:

- **SPHINCS+**: SPHINCS+ is a stateless hash-based signature scheme that offers strong security and post-quantum resistance. It is designed to avoid the pitfalls of stateful schemes, such as the risk of key reuse, by ensuring that each signature is independent. While the signatures are relatively large, SPHINCS+ is highly secure and versatile [5].
- **Merkle Signature Scheme**: The Merkle Signature Scheme uses a tree structure (Merkle tree) to enable the reuse of hash-based one-time signatures. This approach reduces the overhead of generating new keys for each signature, making it more efficient while maintaining security [11].
- **Lamport-Diffie Signatures**: One of the earliest hash-based signature schemes, the Lamport-Diffie scheme is simple but secure, using hash functions to create signatures that are computationally infeasible to forge. However, it is a one-time signature scheme, requiring new keys for each message [12].
- **XMSS (eXtended Merkle Signature Scheme)**: XMSS is an improvement on the Merkle Signature Scheme, designed to be secure against quantum attacks while allowing for the creation of many signatures with a single key pair. It is considered stateful, meaning it requires careful management of the signature process to avoid security risks [11].

Hash-based cryptography, particularly in the form of digital signatures, provides a simple yet powerful tool for securing communications in a post-quantum world. Its reliance on well-understood hash functions ensures that it remains a viable option for quantum-resistant security, especially in scenarios where simplicity and security are paramount.

2.4 Multivariate Quadratic Equations

Multivariate quadratic (MQ) cryptography is based on the hardness of solving systems of multivariate polynomial equations, a problem that is widely believed to be resistant to quantum attacks. This approach is particularly attractive for digital signatures, where the efficiency of MQ-based systems can provide significant advantages. This section explores the mathematical foundations of MQ cryptography and presents key algorithms that have been developed in this area.

Mathematical Foundation and Key Algorithms of Multivariate Quadratic Equations:

The mathematical foundation of multivariate quadratic cryptography lies in the Multivariate Quadratic (MQ) Problem, which involves solving systems of quadratic equations over finite fields. The problem is NP-hard, meaning it is computationally intractable to solve in general, even with quantum computers. The difficulty of this problem forms the basis of several cryptographic schemes that are designed to be secure against quantum attacks [16][14].

Key algorithms in MQ cryptography include:

- **Rainbow**: Rainbow is a digital signature scheme based on the MQ problem. It is an extension of the unbalanced oil and vinegar (UOV) scheme, which uses two sets of variables to create a system of equations that is difficult to solve. Rainbow is efficient and offers relatively small key sizes, making it a practical choice for lightweight applications [10].
- **HFE (Hidden Field Equations)**: HFE is a public key cryptosystem that uses multivariate polynomial equations over finite fields. It is designed to be secure against both classical and quantum attacks, though it requires careful parameter selection to avoid vulnerabilities. HFE has been adapted into various signature schemes, offering flexibility in its applications [9].
- **Quartz**: Quartz is an MQ-based signature scheme that builds on the HFEv- (HFE with vinegar variables) framework. It provides compact signatures and efficient verification, making it suitable for applications where signature size and speed are critical [10].
- **GeMSS (Great Multivariate Signature Scheme)**: GeMSS is a more recent development in MQ-based cryptography, offering improved security and performance compared to earlier schemes. It is designed to be secure against quantum attacks while providing efficient signature generation and verification [16].

- **MUltivariate quadratic Polynomial public key cryptosystem (MUP)**: MUP is a public key encryption scheme based on the difficulty of solving MQ problems. It is designed for secure communication and offers a balance between security and efficiency, though it is less commonly used than other MQ-based systems [16].

MQ cryptography, with its strong security foundations and efficient algorithms, holds promise for a range of applications in the post-quantum era. Its ability to provide secure digital signatures and encryption makes it a valuable tool for protecting information in the presence of quantum threats.

2.5 Summary of Differences and Applications

The post-quantum cryptographic algorithms discussed in this chapter vary significantly in their underlying mathematical foundations, performance characteristics, and practical applications. This section provides a summary of the key differences between these approaches and their respective areas of application.

Security Basis

- Lattice-Based Cryptography: Relies on the hardness of lattice problems such as LWE and SVP, offering broad applicability across encryption, key exchange, and digital signatures.
- Code-Based Cryptography: Based on decoding random linear codes, primarily used for encryption with proven resilience against quantum attacks.
- Hash-Based Cryptography: Utilizes the properties of cryptographic hash functions, particularly for digital signatures, with a focus on simplicity and security.
- Multivariate Quadratic Equations: Centers on solving systems of multivariate quadratic equations, providing efficient solutions for digital signatures with potential for broader cryptographic applications.

Key Sizes

- Lattice-Based Cryptography: Typically involves moderate key sizes, offering a good balance between security and efficiency.

- Code-Based Cryptography: Requires very large key sizes, which can be a drawback in terms of storage and transmission but provides high security.
- Hash-Based Cryptography: Features moderate key sizes but often results in larger signature sizes.
- Multivariate Quadratic Equations: Generally, involves smaller key sizes compared to lattice-based and code-based systems, making it suitable for resource-constrained environments.

Performance

- Lattice-Based Cryptography: Known for its efficiency and balanced performance, making it suitable for a wide range of applications, including real-time communications.
- Code-Based Cryptography: Despite large key sizes, offers efficient encryption and decryption processes, with a focus on long-term data security.
- Hash-Based Cryptography: Characterized by larger signatures and slower generation times but provides high security levels and straightforward implementation.
- Multivariate Quadratic Equations: Offers potentially fast verification with compact signatures, beneficial for digital signature applications and secure communication.

Applications

- Lattice-Based Cryptography: Used in key exchange protocols, digital signatures, and encryption, providing comprehensive security solutions for various scenarios.
- Code-Based Cryptography: Primarily used for encryption, offering robust security for long-term data protection, particularly in environments where key size is less of a concern.
- Hash-Based Cryptography: Best suited for digital signatures, especially where simplicity and security are prioritized, such as in secure software distribution and digital authentication.
- Multivariate Quadratic Equations: Primarily used in digital signatures, with potential for other cryptographic applications due to their efficient verification properties and compact signature size.

3.National Institute of Standards and Technology (NIST) Post-Quantum Cryptography Standardization

3.1 Introduction to NIST and Its Role in Cryptography

The National Institute of Standards and Technology (NIST) is a U.S. government agency under the Department of Commerce, founded in 1901 to promote innovation and industrial competitiveness. NIST's primary mission is to advance measurement science, standards, and technology in ways that enhance economic security and improve the quality of life. Within the field of cryptography, NIST plays a critical role in developing and maintaining standards that secure digital communications and data across both public and private sectors. **Figure 1** illustrates the timeline of NIST’s contributions to cryptography, including its role in the post-quantum cryptography standardization process.

NIST’s influence in cryptography began in the 1970s with the development of the Data Encryption Standard (DES), one of the earliest forms of encryption standards. Over the years, NIST has continued to spearhead the creation and standardization of cryptographic algorithms, most notably with the Advanced Encryption Standard (AES) and the SHA (Secure Hash Algorithm) family of cryptographic hash functions. In response to the emerging threat posed by quantum computing, NIST launched the Post-Quantum Cryptography (PQC) Standardization project. This initiative is crucial for identifying cryptographic algorithms that are resistant to the unprecedented computational capabilities of quantum computers.

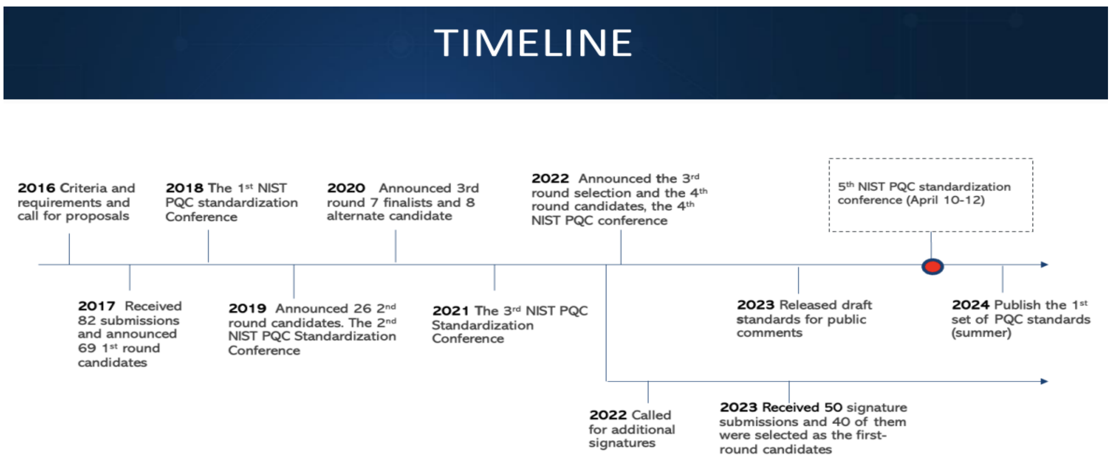
TIMELINE
2016 Criteria and requirements and call for proposals
2017 Received 82 submissions and announced 69 1st round candidates
2018 The 1st NIST PQC standardization Conference
2019 Announced 26 2nd round candidates. The 2nd NIST PQC Standardization Conference
2020 Announced 3rd round 7 finalists and 8 alternate candidate
2021 The 3rd NIST PQC Standardization Conference
2022 Announced the 3rd round selection and the 4th round candidates, the 4th NIST PQC conference
2022 Called for additional signatures
2023 Received 50 signature submissions and 40 of them were selected as the first-round candidates
2023 Released draft standards for public comments
5th NIST PQC standardization conference (April 10-12)
2024 Publish the 1st set of PQC standards (summer)

Figure 1: Overview of NIST Timeline [15]

3.2 Overview of NIST's Evaluation Process

The NIST Post-Quantum Cryptography Standardization process involves multiple rounds of evaluation. As shown in **Figure 2**, this process began in 2016 and continues to evolve with community involvement and cryptographic assessments. The figure outlines the key steps in the evaluation process, leading to the selection of quantum-resistant algorithms.

In Round 1 (2016–2019), NIST received 82 algorithm submissions, of which 69 were deemed valid. From this pool, 26 candidates were selected for further evaluation based on their cryptographic strengths and weaknesses. In Round 2 (2019–2020), NIST narrowed down the candidates to 7 selected algorithms and 8 alternates, focusing on assessing their software and hardware implementations.

By Round 3 (2020–2022), the process concentrated on 7 candidates, with 4 encryption/key encapsulation (PKE/KEM) and 3 digital signature schemes. From these, 1 PKE/KEM and 3 signature schemes were selected for standardization. Additionally, 4 candidates advanced to Round 4 (2022–present), where the focus is on PKE/KEM candidates like BIKE, Classic McEliece, and HQC. Notably, one candidate (SIKE) was attacked and subsequently withdrawn. The ongoing evaluation continues to refine the selection for final standardization.

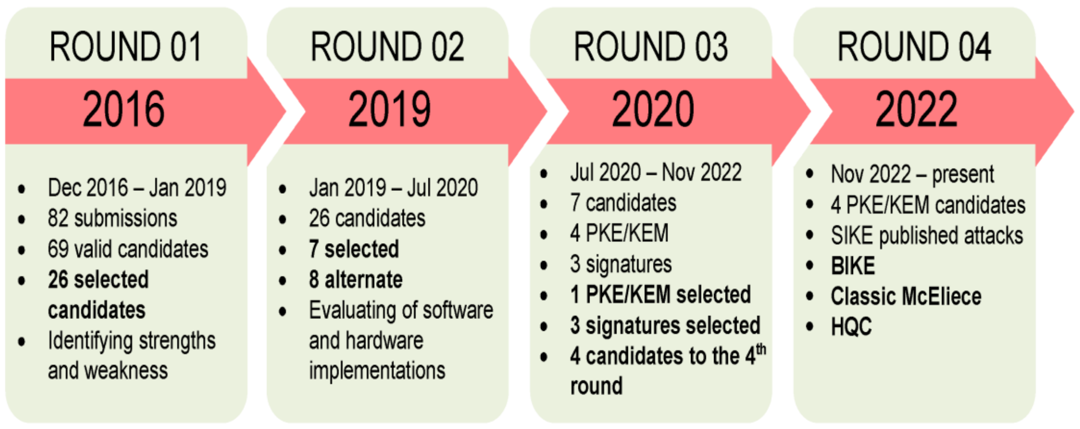

Figure 2: NIST Evaluation Process

3.2.1 Initial Submission and First Round Evaluation

The process began with an open call for submissions, inviting cryptographers from around the world to propose algorithms that could be resistant to quantum attacks. NIST received 69 submissions, covering a wide range of cryptographic techniques, including lattice-based cryptography, code-based cryptography, hash-based cryptography, and multivariate quadratic equations [15]. These submissions were evaluated based on their security against both classical and quantum attacks, as well as their performance metrics such as speed, key sizes, and memory usage.

In the first round, NIST focused on filtering out algorithms that exhibited clear weaknesses or impracticalities. This phase reduced the pool of candidates by more than half, leaving 26 algorithms that showed promise for further evaluation [16].

3.2.2 Second Round Evaluation

The second round of NIST's evaluation process was more rigorous, with a focus on deeper analysis of the remaining candidates. NIST scrutinized the algorithms for potential vulnerabilities, looking for weaknesses that could be exploited by quantum or classical attacks. The cryptographic community played a significant role in this phase, providing feedback, conducting independent reviews, and publishing research on the various candidates. The goal was to ensure that the algorithms not only offered strong theoretical security but also demonstrated practical viability for widespread adoption.

At the end of the second round, NIST selected 15 algorithms to advance to the final round. These algorithms were considered the most promising in terms of security, performance, and implementation efficiency [16].

3.2.3 Final Round and Community Involvement

The final round of NIST's PQC standardization process is currently underway, involving an even more intensive examination of the remaining algorithms. In this phase, NIST is working closely with cryptographers and industry experts to optimize the implementations of these algorithms and to identify any subtle vulnerabilities that may not have been evident in earlier rounds. Public workshops, conferences, and collaborations are integral to this process, ensuring transparency and broad community involvement.

The final selection will result in a set of standardized algorithms that can be used to secure communications and data in a quantum world. These standards will likely include algorithms for public-key encryption, digital signatures, and key exchange protocols [15][16].

3.3 Key Post-Quantum Algorithms to Consider

Among the algorithms that have advanced to the final round of NIST's PQC standardization process, several stand out due to their unique approaches and strong security foundations. These algorithms cover different categories of post-quantum cryptography, each relying on different hard mathematical problems to ensure security against quantum attacks.

- **Lattice-Based Algorithms**: This category includes algorithms like Kyber, Dilithium, and NTRUEncrypt, which are based on the hardness of lattice problems such as Learning With Errors (LWE) and Shortest Vector Problem (SVP). Lattice-based algorithms are favored for their strong security guarantees and efficiency in implementation, making them prime candidates for standardization [1][14].
- **Code-Based Algorithms**: Classic McEliece and BIKE are among the notable code-based algorithms that have progressed through NIST's process. These algorithms rely on the difficulty of decoding random linear codes, a problem that remains resistant to both classical and quantum attacks. Code-based cryptography is particularly known for its long-term security, despite the large key sizes required [2][18].
- **Hash-Based Algorithms**: SPHINCS+ is the primary hash-based algorithm in the NIST competition. It uses cryptographic hash functions to create secure digital signatures. Hash-based cryptography is valued for its simplicity and the well-understood security properties of hash functions, which remain quantum-resistant [5][11].
- **Multivariate Quadratic Equations**: Algorithms like Rainbow represent the multivariate quadratic approach to post-quantum cryptography. These algorithms are based on the difficulty of solving

systems of multivariate quadratic equations, a problem that is NP-hard and believed to be secure against quantum attacks [10][14].

3.4 Detailed Overview of Key Algorithms

The final selection of post-quantum algorithms by NIST will likely reflect the diverse needs of the cryptographic community. Below is a more detailed examination of some of the key algorithms, focusing on their usage, advantages, and potential challenges.

3.4.1 Kyber

Kyber's efficiency and security make it a top contender for NIST's PQC standards. It is particularly well-suited for secure communications protocols such as TLS (Transport Layer Security), where speed and low bandwidth are essential. Kyber's reliance on the LWE problem provides strong security guarantees, and its implementation is designed to be both straightforward and efficient, making it an ideal choice for widespread adoption in various cryptographic applications [1][13].

Equation: $A \cdot s + e$, where A is a public matrix, s is a secret vector, and e is an error vector. The security is based on the Learning with Errors (LWE) problem.

3.4.2 Dilithium

Dilithium's balanced approach to digital signatures has made it a leading candidate in NIST's standardization process. It is designed to offer moderate key and signature sizes while maintaining strong security against quantum attacks. Dilithium is expected to be widely used in applications requiring frequent and reliable digital signatures, such as in securing software updates, authenticating digital certificates, and protecting electronic documents [13][14].

Equation: Involves polynomials and matrices, typically of the form $A \cdot s = t \bmod q$, with signature generation and verification relying on the hardness of the Ring-LWE problem.

3.4.3 Classic McEliece

Classic McEliece, with its proven security track record, is likely to be included in NIST's final recommendations for post-quantum encryption. Its large key sizes make it less practical for some applications, but its unparalleled security makes it ideal for scenarios where data needs to be protected over long periods, such as in archival storage and secure email systems [2][18].

Equation: $c=m \cdot G+e$, where G is a generator matrix of a Goppa code, m is the plaintext, and e is an error vector. Decryption involves finding the original message m given ccc and the private key.

3.4.4 SPHINCS+

SPHINCS+ offers a unique combination of security and simplicity, making it a valuable option for digital signatures in the quantum era. Despite its larger signature sizes, SPHINCS+ is expected to see widespread use in scenarios where digital signature integrity is critical, such as in the distribution of secure software and the signing of legal documents. Its stateless nature simplifies implementation and reduces the potential for errors, making it an attractive choice for secure communications [5][11].

Equation: Uses hash functions H in a tree-like structure to generate signatures. The specific structure is called a Merkle tree, and the equation for a leaf node might be $H(x)$, where x is a random value.

3.4.5 Rainbow

Rainbow's efficient signature generation and verification processes make it a strong candidate for lightweight cryptographic applications. Its small signature size and fast verification times are particularly advantageous in resource-constrained environments, such as IoT devices and mobile applications. [10][16].

Equation: Solves systems of multivariate quadratic equations, generally of the form $p(x1,x2,\ldots,xn)=y$, where p is a quadratic polynomial and xi are variables. The complexity of solving these equations forms the security basis.

4.Design And Implementation Of a Client-Server Platform for Post-Quantum Cryptography Evaluation

One major goal of this thesis is the evaluation of key Post Quantum Cryptography algorithms. In order to pursue this goal, a client-server platform to test and compare the performance of post-quantum cryptographic algorithms (such as Kyber and SPHINCS+) with traditional cryptographic methods (such as RSA and Diffie-Hellman) has been designed and implemented as a research prototype. This platform allows for the comprehensive assessment of key performance metrics, such as key generation time, encryption, decryption, and signature verification, in real-world communication scenarios.

The following sections describe the key components of the experimental setup. The platform simulates a typical communication system between a client and server, where data is securely transmitted using different cryptographic algorithms. By implementing both post-quantum and traditional cryptographic methods, the platform facilitates a comparative analysis of their computational efficiency and security implications. It has been designed to evaluate the feasibility of adopting post-quantum cryptographic algorithms in real-world applications, particularly in environments that require lightweight hardware, such as IoT devices.

Through this experimental setup, key processes like key generation, encryption, decryption, and signature verification can be closely monitored and analysed. The results provide insights into the trade-offs between the enhanced security offered by post-quantum cryptography and the increased computational overhead associated with these newer algorithms. This setup is a critical tool for understanding the practical challenges and opportunities presented by the transition to quantum-resistant cryptography.

4.1 Overview of the Experimental Setup

The primary goal of this research is to evaluate the performance and security implications of post-quantum cryptographic algorithms, particularly Kyber and SPHINCS+, and compare them with traditional cryptographic systems such as RSA and Diffie-Hellman. To achieve this, a client-server communication platform was implemented using Raspberry Pi devices as IoT nodes to simulate real-world cryptographic interactions between two parties. The Raspberry Pi, acting as both the client and the server, provides an environment that closely resembles practical IoT applications, where lightweight hardware is used for secure data transmission. The experiments measure performance metrics such as key generation time, encryption time, decryption time, and signature verification time over 1000 iterations to ensure statistically significant results.

The experiments were designed to assess the computational efficiency and practical applicability of post-quantum cryptographic algorithms in comparison to well-established cryptographic systems. A TCP-based client-server architecture, as illustrated in **Figure 3**, was selected to facilitate secure data transmission. The client is responsible for fetching data from an external source, encrypting it, and transmitting it to the server. The server then decrypts the data and verifies its integrity. The experimental setup ensures that both encryption and signature verification processes are tested thoroughly.

This study also leverages the *iCity* project datasets, serving air quality data in the Stuttgart area via the SensorThings server. The data includes air quality measurements and sensor location information, which are encrypted to ensure confidentiality. Notably, access to sensor location data can be controlled to provide differential access, where one user may view the data while it is restricted for another. This functionality is implemented at the network node. In addition to security, the time required for encryption and decryption operations is evaluated, providing insights into the practical overheads associated with post-quantum cryptographic methods in a real-world context. [7]

The platform simulates a typical communication system between a client and server where data is securely transmitted using different cryptographic algorithms. By implementing both post-quantum and traditional cryptographic methods, the platform facilitates a comparative analysis of their computational efficiency and security implications.

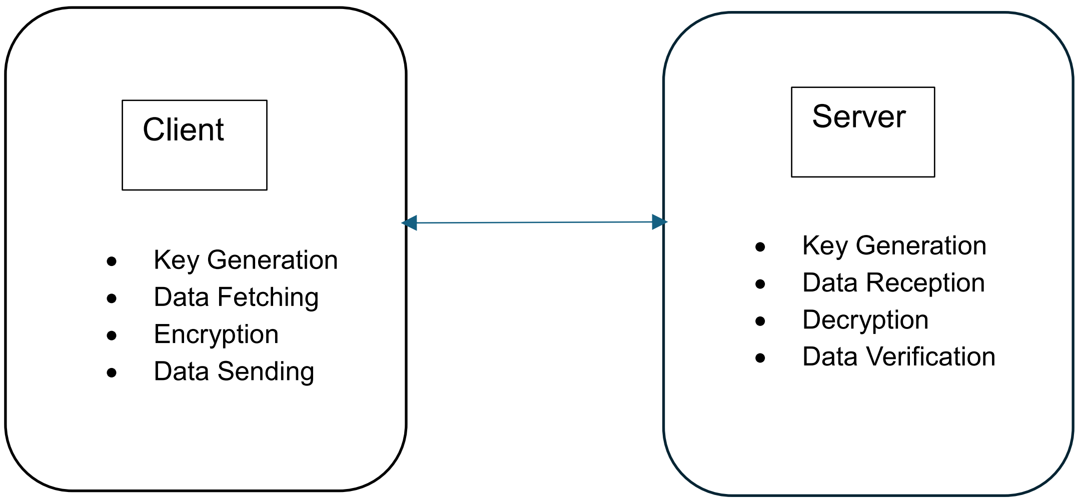

Figure 3:Client-Server Platform Architecture for Cryptographic Algorithm Evaluation

4.2 Client-Server Platform

The client-server platform has been developed that allows for testing different cryptographic algorithms in a controlled environment. The client-server architecture mimics real-world cryptographic use cases, ensuring that the performance metrics are realistic.

The client and server operate as follows:

Client-Side Operations:

1. Key Generation: Depending on the algorithm, the client generates or retrieves a key pair. For example:

    - In Kyber, a key encapsulation mechanism is used where the client generates a ciphertext and shared secret based on the server’s public key.
    - In SPHINCS+, the client generates a public-private key pair for signing the data.
    - In RSA, the client receives the server’s public key and uses it to encrypt an AES key for further data encryption.
2. Data Fetching: The client retrieves data from an external URL.
3. This data is fetched and stored for encryption in subsequent steps.
4. Encryption:
    - In Kyber, the client encrypts the data using the shared secret obtained from the key encapsulation process.
    - In SPHINCS+, the client signs the data and then encrypts it using an AES key.
    - In RSA, the client encrypts the data using an AES key, which is encrypted with the server’s public RSA key.
5. Data Transmission: The encrypted data is sent over the network to the server. In some cases, such as RSA, both the encrypted AES key and encrypted data are sent.
6. Acknowledgment: The client waits for an acknowledgment from the server indicating that the data was received and processed successfully.

Server-Side Operations:

1. Key Generation: The server generates or receives keys based on the algorithm:
    - For Kyber, the server generates its public and private key pair and sends the public key to the client.
    - For SPHINCS+, the server prepares to verify signatures using the public key from the client.

    - For RSA, the server generates an RSA key pair and sends the public key to the client.

2. Receiving Data: The server listens for incoming encrypted data from the client. Upon receiving the data, it proceeds to decrypt and verify the information.
3. Decryption:
    - For Kyber, the server decrypts the data using the private key generated in the encapsulation process.
    - For SPHINCS+, the server first decrypts the data using the AES key and then verifies the digital signature using the client’s public key.
    - For RSA, the server decrypts the AES key using its private RSA key and uses it to decrypt the encrypted data.
4. Signature Verification: In the case of SPHINCS+, the server verifies the signature to ensure the integrity and authenticity of the received data.
5. Acknowledgment: The server sends an acknowledgment to the client, confirming that the data was successfully decrypted and verified.

4.3 Flow Chart of the Experimental Process

The flow of the experimental process is detailed in **Figure 4**, which demonstrates the interaction between the client and server during the cryptographic operations. This flow includes key generation, data encryption, transmission, and decryption, followed by signature verification in the case of signature-based algorithms.

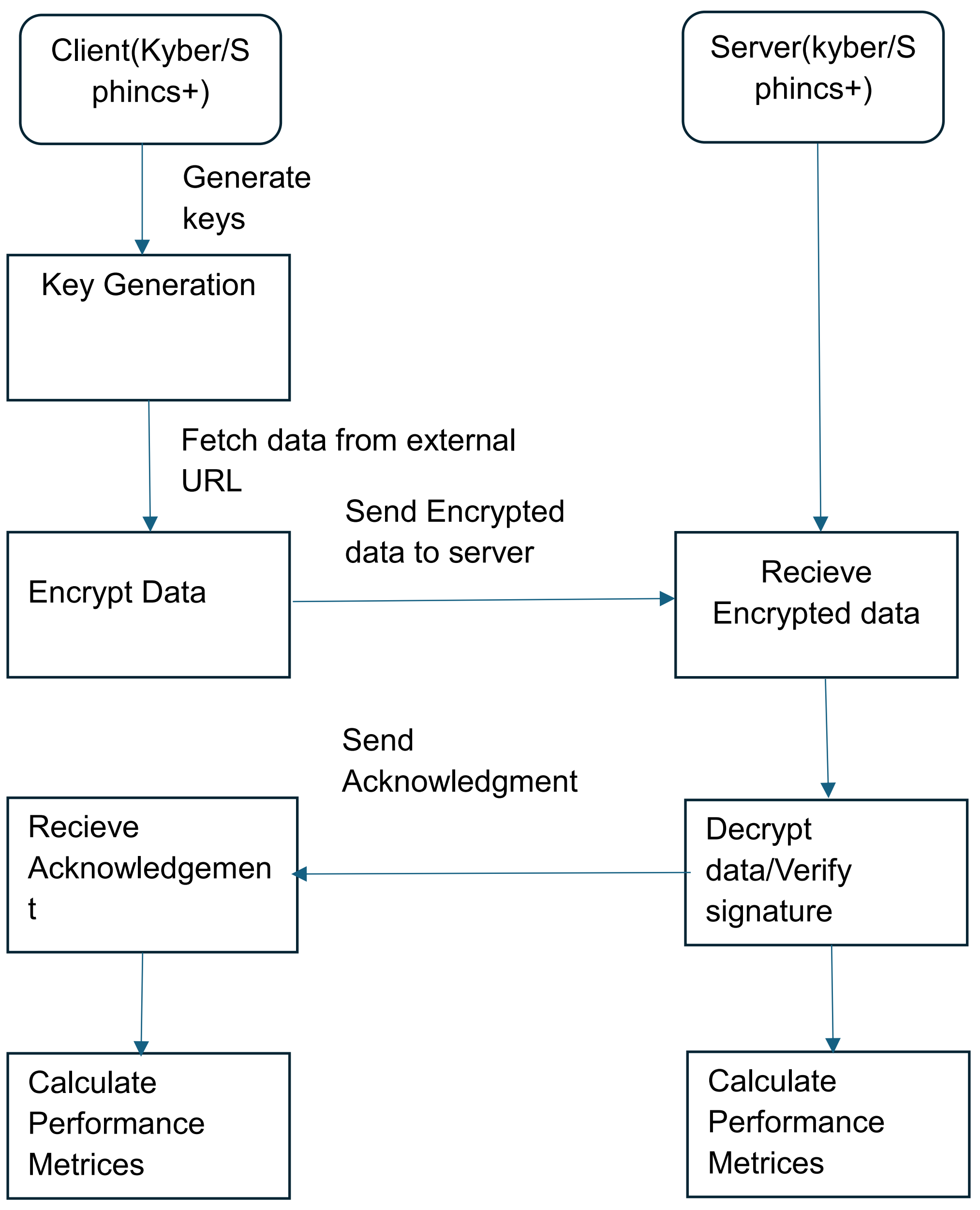

Figure 4: Flowchart of Design for evaluation process

4.4. Implementation Of Cryptographic Algorithms and Their Operations

This section focuses on the implementation of the cryptographic algorithms used in the experiments. Each algorithm follows a unique process for key generation, encryption, decryption, and, in the case of SPHINCS+, signature verification. Detailed descriptions of how Kyber, SPHINCS+, RSA, and Diffie-Hellman were implemented within the client-server platform are provided, highlighting the key operational steps and differences between post-quantum and traditional cryptographic methods. Additionally, Figure 5 highlights the difference in key encapsulation mechanisms between post-quantum algorithms and traditional Diffie-Hellman key exchange.

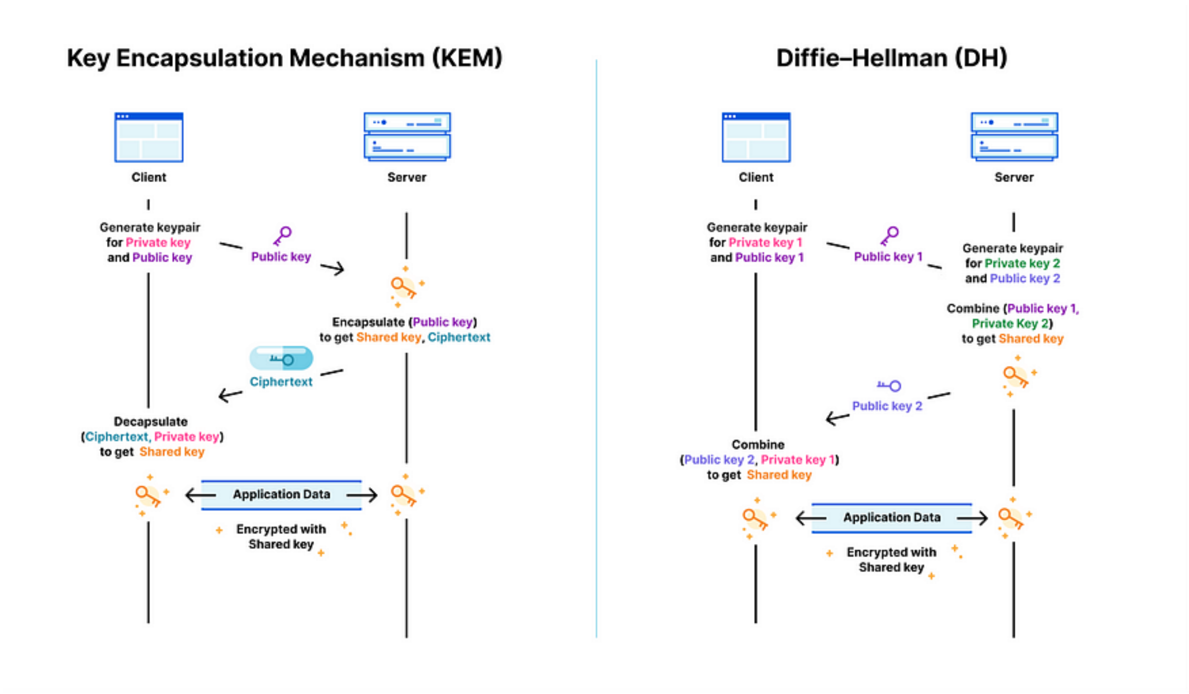
Key Encapsulation Mechanism (KEM)
Client
Server
Generate keypair for Private key and Public key
Public key
Encapsulate (Public key) to get Shared key, Ciphertext
Ciphertext
Decapsulate (Ciphertext, Private key) to get Shared key
Application Data
Encrypted with Shared key
Diffie–Hellman (DH)
Client
Server
Generate keypair for Private key 1 and Public key 1
Public key 1
Generate keypair for Private key 2 and Public key 2
Combine (Public key 1, Private Key 2) to get Shared key
Public key 2
Combine (Public key 2, Private key 1) to get Shared key
Application Data
Encrypted with Shared key

Figure 5: KEM vs Diffie-Hellman [17]

The key operations in detail for each algorithm:

4.4.1 Kyber (Post-Quantum Encryption)

- **Key Generation**: The server generates a public-private key pair based on lattice-based cryptographic principles. The public key is shared with the client, while the private key is used for decrypting the ciphertext.

Listing 4.1, the Kyber generation process is initiated:

```
if (PQCLEAN_KYBER1024_CLEAN_crypto_kem_keypair (public_key, secret_key)! = 0)
{

    fprintf(log_file, "Key pair generation failed (iteration %d).\n", i+1);

    return 1;

    }

  clock_t end_keygen = clock();

      double keygen_time = (double)(end_keygen- start_keygen) / CLOCKS_PER_SEC;

      total_keygen_time += keygen_time;

      fprintf(log_file, "Key Generation Time (iteration %d): %f seconds\n", i+1,
keygen_time)
```

This function generates the necessary public and private key pairs for establishing secure communication. The full implementation of Kyber's key generation can be found in **Appendix A.**

- **Encryption**: The client generates a shared secret and a ciphertext using the server’s public key. The shared secret acts as an AES key, which is used to encrypt the data.

  Listing 4.2 shows the encryption process:

```
 if (PQCLEAN_KYBER1024_CLEAN_crypto_kem_enc(ciphertext, shared_secret,
public_key) != 0) {

        fprintf(log_file, "Encapsulation failed (iteration %d).\n", i+1);

        return 1;

     }

     clock_t end_encap = clock();

     double encap_time = (double)(end_encap- start_encap) / CLOCKS_PER_SEC;

     total_encap_time += encap_time;

     fprintf(csv_file, "%d,%f\n", i+1, encap_time);
```

```
        fprintf(log_file, "Encapsulation Time (iteration %d): %f seconds\n", i+1,
encap_time);
```

This function encrypts the data using a shared secret derived from the server's public key. The complete Kyber client and server implementation is available in Appendix A.

- **Decryption**: The server uses the private key to decapsulate the ciphertext, recovering the shared secret. The shared secret is then used to decrypt the AES-encrypted data.

  Listing 4.3 illustrates the decryption process:

```
 if (PQCLEAN_KYBER1024_CLEAN_crypto_kem_dec(shared_secret, ciphertext,
secret_key) != 0) {
            fprintf(log_file, "Decapsulation failed (iteration %d).\n", i+1);
            return 1;
        }
        clock_t end_decap = clock();
        double decap_time = (double)(end_decap- start_decap) / CLOCKS_PER_SEC;
        total_decap_time += decap_time;
        fprintf(log_file, "Decapsulation Time (iteration %d): %f seconds\n", i+1,
decap_time);
```

This function decapsulates the ciphertext to recover the shared secret, which is then used to decrypt the encrypted data. The full Kyber decryption implementation is provided in Appendix A.

4.4.2 SPHINCS+ (Post-Quantum Signature)

- **Key Generation**: The client generates a public-private key pair for signing the data. The public key is sent to the server, which will later verify the signature.

  Listing 4.4 shows the key generation process:

```
(PQCLEAN_SPHINCSSHAKE256SSIMPLE_CLEAN_crypto_sign_keypair(public_key,
secret_key) != 0) {
        fprintf(stderr, "Key pair generation failed.\n");
        return 1;
    }
    end = clock();
    cpu_time_used = ((double)(end- start)) / CLOCKS_PER_SEC;
    printf("Key Generation Time: %f seconds\n", cpu_time_used);
    total_key_generation_time += cpu_time_used;
```

This function generates the public and private key pair for secure signing. The complete implementation of SPHINCS+ key generation can be found in **Appendix B**.

- **Signing**: The client signs the data using the private key before encrypting the data with AES.

  Listing 4.5 demonstrates the signing process:

```
If(PQCLEAN_SPHINCSSHAKE256SSIMPLE_CLEAN_crypto_sign_signature(signature,
&signature_len, ciphertext, ciphertext_len, secret_key) != 0) {
        fprintf(stderr, "Signing failed.\n");
        free(message);
        free(ciphertext);
        return 1;
    }
    end = clock();
    cpu_time_used = ((double)(end- start)) / CLOCKS_PER_SEC;
    printf("Signing Time: %f seconds\n", cpu_time_used);
```

This function securely signs the data using the client's private key. The full implementation of SPHINCS+ signing is available in **Appendix B**.

- **Encryption**: The signed data is encrypted using a randomly generated AES key.
- **Decryption and Signature Verification**: Once the encrypted data is received by the server, it is first decrypted using the AES key. After decryption, the server verifies the signature attached to the data using the public key sent by the client. This process ensures that the data is both authentic and intact.

  Listing 4.6 illustrates the signature verification process:

```
int verification_result =
PQCLEAN_SPHINCSSHAKE256SSIMPLE_CLEAN_crypto_sign_verify(signature,
signature_len, data, data_len, public_key);

end = clock();

cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;

total_verification_time += cpu_time_used;
```

  This function verifies the signature using the client's public key to ensure that the data has not been altered. The complete implementation of SPHINCS+ signature verification is provided in **Appendix B.**

4.4.3 RSA (Traditional Public-Key Encryption)

- **Key Generation**: The server generates a public-private RSA key pair and shares the public key with the client.
- **Encryption**: The client encrypts the data using a randomly generated AES key, which is itself encrypted using the server's RSA public key. The AES key and encrypted data are both sent to the server.
- **Decryption**: The server uses its RSA private key to decrypt the AES key and subsequently decrypts the AES-encrypted data.

4.4.4 Diffie-Hellman (Traditional Key Exchange)

- **Key Generation**: Both the client and server generate private-public key pairs and exchange their public keys to compute a shared secret.

  Listing 4.7 shows the Diffie-Hellman key generation process:

  ```
   start_time = time.time()
      client_private_key = parameters.generate_private_key()
      client_public_key = client_private_key.public_key()
      key_generation_time = time.time()- start_time
      key_generation_times.append(key_generation_time)
      print(f"Iteration {i+1}: Client key generation time: {key_generation_time:.6f}
  seconds")
  ```

  This function generates the private-public key pair used for Diffie-Hellman key exchange. The complete implementation of key generation can be found in **Appendix C**.

- **Encryption**: The shared secret is used as a symmetric AES key to encrypt the data.

  Listing 4.8 demonstrates AES encryption with Diffie-Hellman:

  ```
  encryptor = cipher.encryptor()
      start_time = time.time()
      ciphertext = encryptor.update(padded_data) + encryptor.finalize()
      encryption_time = time.time()- start_time
      encryption_times.append(encryption_time)
      print(f"Iteration {i+1}: Encryption time: {encryption_time:.6f} seconds")
  ```

  This function encrypts the data using the shared secret derived from the Diffie-Hellman key exchange. The full encryption process is available in **Appendix C**.

- **Decryption**: The server uses the shared secret to decrypt the AES-encrypted data.

  Listing 4.9 shows AES decryption using the shared secret:

  ```
  decryptor = cipher.decryptor()
      start_time = time.time()
      decrypted_padded_message = decryptor.update(received_data) + decryptor.finalize()
      decryption_time = time.time()- start_time
      decryption_times.append(decryption_time)
      printf("Iteration {i+1}: Decryption time: {decryption_time:.6f} seconds")
  ```

  This function decrypts the data using the shared secret from the Diffie-Hellman key exchange. The full implementation of decryption is provided in Appendix C.

4.5 Performance Metrics and Evaluation

The performance of the algorithms was evaluated based on four primary metrics:

1. **Key Generation Time**: The time required to generate the cryptographic key pair. This metric is crucial in post-quantum algorithms like Kyber and SPHINCS+, where key generation involves more complex mathematical computations compared to traditional algorithms.
2. **Encryption Time**: The time taken to encrypt the data using the selected cryptographic algorithm. This metric measures how quickly each algorithm can secure the data for transmission.
3. **Decryption Time**: The time required by the server to decrypt the data. Post-quantum algorithms often introduce additional complexity in decryption, making this metric significant in assessing their practical viability.

4. **Signature Verification Time** (only for SPHINCS+): The time taken to verify the digital signature generated by the client. This metric is critical for assessing the integrity and authenticity of data in post-quantum cryptographic systems.

Performance measurements were averaged over 1000 iterations for each cryptographic algorithm to ensure statistical reliability. The average times for each algorithm and operation were recorded and compared, providing insights into the computational overhead introduced by post-quantum algorithms relative to traditional cryptographic systems

4.6. Results

The results of this study highlight the performance characteristics of post-quantum cryptographic algorithms (Kyber and SPHINCS+) in comparison to traditional cryptosystems (RSA and Diffie-Hellman). The key performance metrics considered include key generation time, encryption time, decryption time, signing time, and signature verification time, across multiple datasets.

The datasets used for these experiments were obtained from the *iCity* project and the Frost Luftdata API. The *iCity* project provides encrypted air quality and sensor location data in the Stuttgart area, while the Frost Luftdata API offers real-time environmental data, including air quality measurements. Specifically, the datasets used are available at https://ogcapi.hft-stuttgart.de/sta/icity_data_security/v1.1, https://ogcapi.hft-stuttgart.de/sta/frost-luftdata-api/v1.1, and https://ogcapi.hft-stuttgart.de/sta/icity_data_security/v1.1/Datastreams(1). These datasets simulate real-world scenarios involving encrypted environmental data and were chosen to evaluate the performance of the cryptographic algorithms under practical conditions. [7]

The results clearly demonstrate that post-quantum algorithms like Kyber and SPHINCS+ introduce more computational overhead than traditional cryptographic systems. In particular, SPHINCS+ shows significant delays in signing operations. However, the added time is a necessary trade-off for the increased security they provide against quantum attacks. Kyber, while slightly slower in encryption and decryption than RSA, remains highly efficient in key generation and offers stronger quantum resistance.

The performance metrics for these cryptographic algorithms are summarized in Table 1, Table 2, and Table 3, using datasets from the *iCity* project [7]. These tables show consistent results across multiple datasets, confirming that Kyber and SPHINCS+ introduce higher overhead, especially in signing operations for SPHINCS+ and encryption/decryption times for Kyber. This computational overhead is counterbalanced by the superior quantum resistance offered by these algorithms, making them a viable option for secure communications in a post-quantum world.

Algorithm	**Key Generation Time (sec) (avg)**	**Encryption Time (sec) (avg)**	**Decryption Time (sec) (avg)**	**Signing Time(sec) (avg)**	**Signature Verification Time (avg)**
Kyber	0.000760	0.000927	0.001151	N/A	N/A
RSA	0.261134	0.000007	0.000250	0.0043658	0.000182
Diffie-Hellman	0.000252	0.000010 (AES encryption)+0.000253(shared key)	0.000008 (AES decryption)	N/A	N/A
Sphincs+	0.9769	N/A	N/A	11.7910	0.0163

Table1: Performance Metrics for Cryptographic Algorithms (for data https://ogcapi.hft-stuttgart.de/sta/icity_data_security/v1.1)

Algorithm	Key Generation Time (sec) (avg)	Encryption Time (sec) (avg)	Decryption Time (sec) (avg)	Signing Time(sec) (avg)	Signature Verification Time (avg)
Kyber	0.000764	0.000931	0.001157	N/A	N/A
RSA	0.252105	0.000121	0.000116	0.004618	0.000211
Diffie-Hellman	0.000352	0.000010 (AES encryption)+0.000261(shared key)	0.0000013 (AES decryption)	N/A	N/A
Sphincs+	0.978	N/A	N/A	11.741	0.01617

Table2: Performance Metrics for Cryptographic Algorithms (for data https://ogcapi.hft-stuttgart.de/sta/frost-luftdata-api/v1.1)

Algorithm	Key Generation Time (sec) (avg)	Encryption Time (sec) (avg)	Decryption Time (sec) (avg)	Signing Time(sec) (avg)	Signature Verification Time (avg)
Kyber	0.000768	0.000928	0.001152	N/A	N/A
RSA	0.249717	0.000132	0.000121	0.004619	0.000205
Diffie-Hellman	0.000349	0.00009 (AES encryption) +0.000263(shared key)	0.0000012 (AES decryption)	N/A	N/A
Sphincs +	0.9753	N/A	N/A	11.7374	0.0163

Table3: Performance Metrics for Cryptographic Algorithms (for data https://ogcapi.hft-stuttgart.de/sta/icity_data_security/v1.1/Datastreams(1))

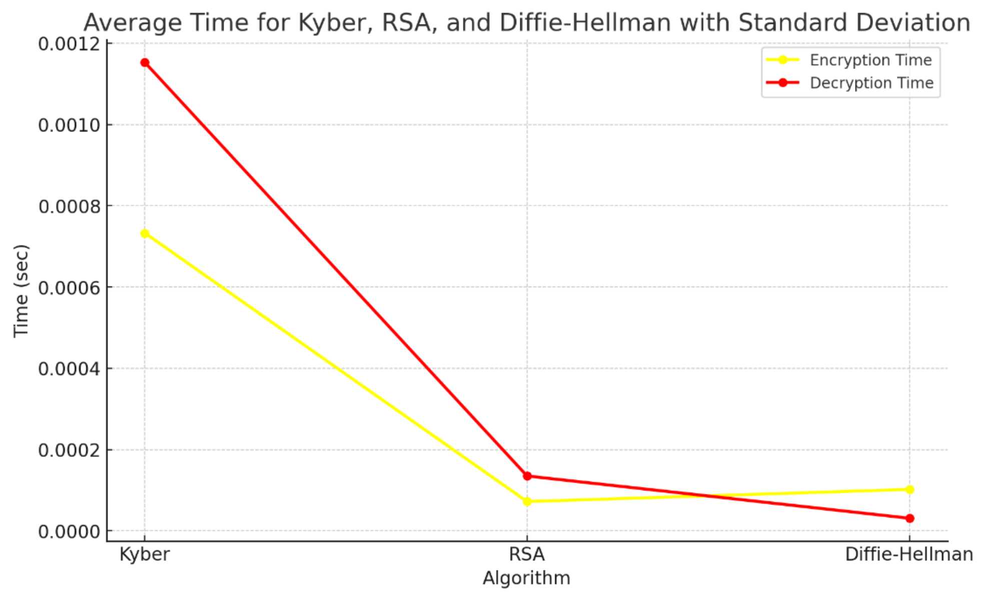

Figure 6:Comparison of Quantum Algorithms (Kyber) vs Classical Algorithms (RSA, Diffie-Hellman)

This graph Figure 6 compares the encryption, and decryption times for Kyber (a quantum-resistant algorithm) with RSA and Diffie-Hellman (classical algorithms). Kyber demonstrates its encryption and decryption times are slightly slower than RSA and Diffie-Hellman (classical algorithms). Diffie-Hellman exhibits the fastest encryption and decryption times overall, though it has a relatively slower key generation time compared to Kyber. This performance trade-off illustrates the potential computational overhead introduced by quantum-resistant algorithms like Kyber, as they prioritize security against quantum attacks.

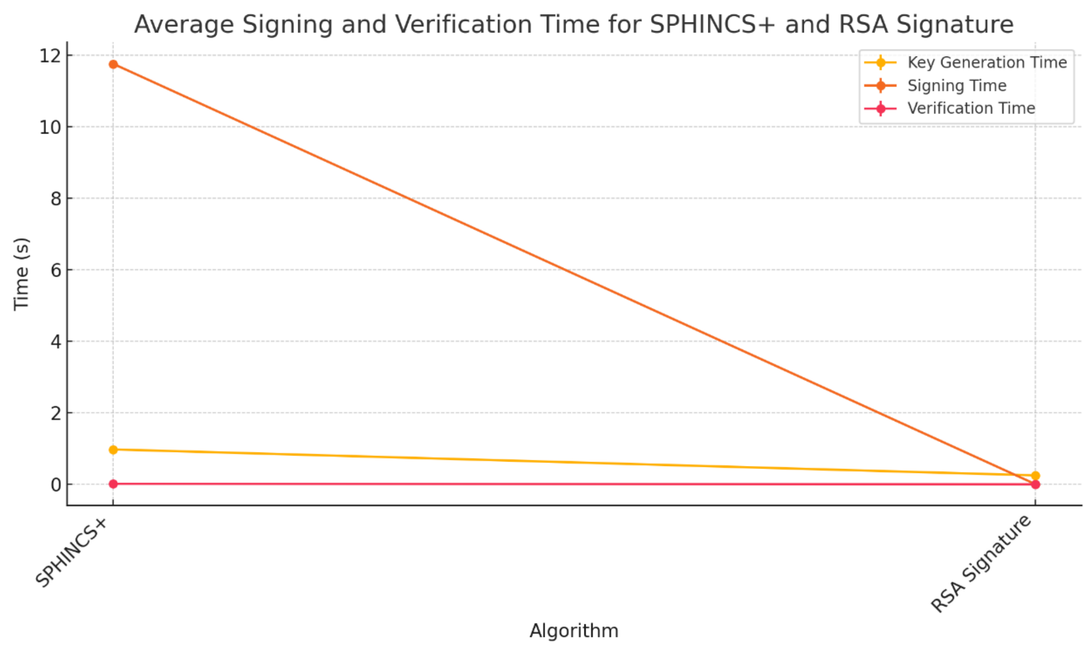

Figure 7:Comparison of SPHINCS+ and RSA Signature Times

This graph Figure 7 illustrates the signing and signature verification times for SPHINCS+ (a hash-based post-quantum algorithm) and RSA (a classical algorithm). SPHINCS+ shows significantly higher signing times compared to RSA, reflecting the computational complexity involved in its quantum-resistant design. However, the verification times for SPHINCS+ are comparable to RSA, indicating its suitability for applications requiring frequent signature verifications. The trade-off between signing time and enhanced security against quantum threats is evident in SPHINCS+.

5. Conclusion

The advent of quantum computing poses an unprecedented threat to the security of current cryptographic protocols, particularly those employed in securing communications over the Internet and in critical IoT infrastructures. Traditional cryptographic methods like RSA and Diffie-Hellman, which are based on the hardness of problems like integer factorization and discrete logarithms, are vulnerable to quantum attacks, making the transition to post-quantum cryptographic solutions a necessity. This thesis provides a thorough evaluation of post-quantum cryptography, focusing on the performance and security implications of algorithms such as Kyber and SPHINCS+, comparing them with well-established cryptosystems like RSA and Diffie-Hellman.

Through the design and implementation of a client-server platform, this study has empirically measured and compared key performance metrics including key generation time, encryption and decryption time, and signature verification time. The results indicate that post-quantum algorithms, particularly Kyber and SPHINCS+, provide robust security features while maintaining reasonable performance overhead. However, these algorithms tend to introduce higher computational costs and latency compared to traditional cryptographic systems, especially in signature verification tasks. Despite these challenges, Kyber and SPHINCS+ have demonstrated significant promise as viable alternatives for securing communications in the post-quantum era, particularly in scenarios requiring enhanced security against quantum threats.

The findings of this research underscore the importance of adopting post-quantum cryptographic systems in anticipation of future quantum-based attacks. The performance of post-quantum algorithms on IoT hardware shows that, while some optimization is needed, the integration of these algorithms into practical cryptographic systems is feasible. As the field of quantum computing continues to advance, future work should focus on refining these algorithms, improving their efficiency, and incorporating them into broader security protocols such as TLS. This will ensure the resilience of IoT systems and other critical infrastructures in the face of quantum computing advancements, providing a secure foundation for digital communications in the quantum era.

References

[1] Schwabe, P., Stebila, D. and Wiggers, T. (2020) 'Post-Quantum TLS Without Handshake Signatures'. Available at: https://eprint.iacr.org/2020/534.pdf.

[2] El-boukhari, M., Azizi, M. and Azizi, A. (2010) 'Improving TLS Security By Quantum Cryptography'. International Journal of Network Security & Its Applications (IJNSA), 2(3), pp. 1–10. DOI: 10.5121/ijnsa.2010.2306.

[3] Alkim, E., Ducas, L., Pöppelmann, T., and Schwabe, P. (2017) ‘Post-Quantum Key Exchange—A New Hope’. PQCrystals Kyber. Available at: https://github.com/pq-crystals/kyber.

[4] Open Quantum Safe 'Open Quantum Safe Project'. Available at: https://openquantumsafe.org/.

[5]wolfSSL 'Post-Quantum Cryptography'. Available at: https://www.wolfssl.com/documentation/manuals/wolfssl/appendix07.html.

[6] Niederhagen, R. and Waidner, M. (2017) 'Practical Post-Quantum Cryptography'. Fraunhofer Institute for Secure Information Technology SIT. Available at: https://sit.fraunhofer.de/en/security-solutions/quantum-safe-cryptography.html.

[7] *OGC SensorThings API*, Accessed: Sep. 25, 2024. [Online]. Available: https://ogcapi.hft-stuttgart.de/sta/frost-luftdata-api/

[8] Hülsing, A., Rijneveld, J., and Schwabe, P. (2019) 'SPHINCS+: Submission to the NIST Post-Quantum Cryptography Project'. PQClean SPHINCS+ Implementation. Available at: https://github.com/sphincs/sphincsplus.

[9] Jao, D. and De Feo, L. (2024) 'Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies'. Available at: https://arxiv.org/pdf/2401.18053.

[10] Ding, J., Chase, M.D., Lauter, K. and Zhao, Y. (2020) 'A Survey about Post-Quantum Cryptography Methods'. Available at: https://www.researchgate.net/publication/378163704_A_Survey_about_Post_Quantum_Cryptography_Methods.

[11] Bernstein, D.J., Buchmann, J. and Dahmen, E. (Eds.) (2009) 'Post-Quantum Cryptography'. Springer.

[12] Mosca, M. and Piani, M. (2013) 'A Quantum Leap in Cryptography'. Commun. ACM.

[13] Hoffstein, J., Pipher, J., Silverman, J.H. and Whyte, W. (2017) 'Practical Lattice-Based Cryptography: NTRUEncrypt and NTRUSign'. NIST. Available at: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-209-draft.pdf.

[14] ETSI Quantum-Safe Cryptography Working Group (2015) 'Quantum-Safe Cryptography and Security: An Introduction, Benefits, Enablers and Challenges'. ETSI.

[15] M. A. Moody, "NIST PQC Standards," presented at IETF 120, Vancouver, BC, Canada, Jul. 2024. Available: https://datatracker.ietf.org/meeting/120/materials/slides-120-pquip-nist-pqc-standards-00

[16] Ding, J., Chase, M.D., Lauter, K. and Zhao, Y. (2018) 'Post-Quantum Cryptography in Practice'. ACM Conference on Computer and Communications Security. Available at: https://dl.acm.org/doi/10.1145/3243734.3243816.

[17] U. Pathum, "CRYSTALS Kyber: The Key to Post-Quantum Encryption," *Medium*, Jan. 5, 2024. [Online]. Available: https://medium.com/@hwupathum/crystals-kyber-the-key-to-post-quantum-encryption-3154b305e7bd.

[18] Buchmann, J., Dahmen, E. and Hulsing, A. (2011) 'A Survey of Post-Quantum Public Key Cryptographic Schemes and Software Efficiency Aspects'. IACR Cryptology ePrint Archive. Available at: https://eprint.iacr.org/2011/623.

[19] ChatGPT ‘Providing interactive and tailored responses based on user inputs in a conversational format'. OpenAI.

[20] The Open Quantum Safe project (2020) 'PQClean: Clean Implementations of Post-Quantum Cryptography'. Available at: https://github.com/PQClean/PQClean.

Appendix

The following implementations of the Kyber, SPHINCS+, and Diffie-Hellman algorithms utilize data retrieved from the URL https://ogcapi.hft-stuttgart.de/sta/icity_data_security/v1, and are informed by the methodologies and techniques described in [3], [8], [19], and [20]. These sources provide foundational insights into post-quantum cryptography and its practical applications in secure data transmission over IoT platforms

Guidelines For Compiling and Running:

This provides detailed guidelines for setting up, compiling, and running the Kyber, Diffie-Hellman, and SPHINCS+ cryptographic algorithms implemented in C and Python. The algorithms are integrated with AES encryption for secure communication in a client-server model, designed to run 1000 iterations for performance evaluation. Below is a step-by-step guide to help in the compilation and execution of these implementations.

Prerequisites:

Before compiling and running the programs, ensure the following libraries and tools are installed on system:

- **OpenSSL**: Required for cryptographic operations, particularly AES encryption.

```
sudo apt-get install libssl-dev
```

- **libcurl**: Used to fetch data from external sources in the Kyber and SPHINCS+ implementations.

```
sudo apt-get install libcurl4-openssl-dev
```

- **Python Cryptography** (for Diffie-Hellman and RSA in Python):

```
pip install cryptography requests
```

Ensure that system has a C compiler (gcc) and a Python interpreter (python3) installed and updated

Running the Diffie-Hellman Algorithm in Python:

Running Diffie-Hellman Client and Server

1. **Install Dependencies**: Installed the required Python libraries:

```
pip install cryptography requests
```

2. **Run the Server**: Start the Diffie-Hellman server in one terminal:

```
python3 dh_server.py
```

3. **Run the Client**: Run the Diffie-Hellman client in another terminal:

```
python3 dh_client.py
```

The client fetches data from an external URL, generates keys using the Diffie-Hellman protocol, and encrypts the data using AES. The performance metrics, such as key generation and encryption times, are logged after 1000 iterations.

Running SPHINCS+ (Post-Quantum Signature) in C:

Compilation Instructions for SPHINCS+

1. **SPHINCS+ Client Compilation**: Use the following command to compile the SPHINCS+ client:

```
gcc-o sphincs_client sphincs_client.c aes_crypto.c-lcurl-lssl-lcrypto
```

2. **SPHINCS+ Server Compilation**: Compile the SPHINCS+ server using this command:

```
gcc-o sphincs_server sphincs_server.c aes_crypto.c-lcurl-lssl-lcrypto
```

A.5.2 Running SPHINCS+ Client and Server

1. **Run the Server**: Start the SPHINCS+ server:

```
./sphincs_server
```

2. **Run the Client**: Start the SPHINCS+ client:

```
./sphincs_client
```

The SPHINCS+ client generates a key pair; signs encrypted data and sends it to the server for signature verification. After 1000 iterations, the results, including key generation, signing, and verification times, are logged.

Compiling and Running the Kyber Algorithm:

Compilation Instructions

1. **Kyber Client Compilation**: Use the following command to compile the Kyber client, which uses AES encryption and the Kyber KEM:

   ```
   gcc-o client kyber_client.c aes_utils.c utils.c kem.c indcpa.c polyvec.c poly.c ntt.c reduce.c verify.c symmetric-shake.c cbd.c /path/to/PQClean/common/randombytes.c /path/to/PQClean/common/fips202.c-I .-I /path/to/PQClean/common-lcurl -lssl-lcrypto
   ```

2. **Kyber Server Compilation**: Compile the Kyber server using this command:

   ```
   gcc -o server kyber_server.c aes_utils.c utils.c kem.c indcpa.c polyvec.c poly.c ntt.c reduce.c verify.c symmetric-shake.c cbd.c /path/to/PQClean/common/randombytes.c /path/to/PQClean/common/fips202.c-I .-I /path/to/PQClean/common-lssl-lcrypto
   ```

Running the Kyber Client and Server:

1. **Run the Server**: First, start the server in one terminal:

   ```
   ./server
   ```

2. **Run the Client**: Next, run the client in another terminal:

   ```
   ./client
   ```

The client will fetch data from a specified URL, perform key encapsulation, encryption, and transmission to the server. Performance metrics for key generation, encapsulation, and encryption times will be logged in client_log.txt and server_log.txt, with results also saved in a CSV file.

Appendix A:

Kyber Client Code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <curl/curl.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <openssl/sha.h>
#include <openssl/rand.h>
#include <time.h>
#include "api.h"
#include "aes_utils.h"
#define SERVER_IP "127.0.0.1"
#define SERVER_PORT 8080
#define URL "https://ogcapi.hft-stuttgart.de/sta/icity_data_security/v1.1"
#define ITERATIONS 1000
#define CSV_FILE "client_timings.csv"
#define LOG_FILE "client_log.txt"
#define RETRY_DELAY 1000000  // 1 second

struct MemoryStruct {
    char *memory;
    size_t size;
};

static size_t WriteMemoryCallback(void *contents, size_t size, size_t nmemb, void *userp) {
    size_t totalSize = size * nmemb;
    struct MemoryStruct *mem = (struct MemoryStruct *)userp;

    char *ptr = realloc(mem->memory, mem->size + totalSize + 1);
    if (ptr == NULL) {
        printf("Not enough memory (realloc returned NULL)\n");
        return 0;
    }

    mem->memory = ptr;
    memcpy(&(mem->memory[mem->size]), contents, totalSize);
    mem->size += totalSize;
    mem->memory[mem->size] = 0;

    return totalSize;
}

int main() {
    CURL *curl_handle;
    CURLcode res;
    struct MemoryStruct chunk;
    chunk.memory = malloc(1);
    chunk.size = 0;
    curl_global_init(CURL_GLOBAL_ALL);
    curl_handle = curl_easy_init();
    curl_easy_setopt(curl_handle, CURLOPT_URL, URL);
    curl_easy_setopt(curl_handle, CURLOPT_WRITEFUNCTION, WriteMemoryCallback);
    curl_easy_setopt(curl_handle, CURLOPT_WRITEDATA, (void *)&chunk);
    curl_easy_setopt(curl_handle, CURLOPT_USERAGENT, "libcurl-agent/1.0");
    res = curl_easy_perform(curl_handle);

    if (res != CURLE_OK) {
        fprintf(stderr, "curl_easy_perform() failed: %s\n", curl_easy_strerror(res));
        return 1;
    }
    curl_easy_cleanup(curl_handle);
    curl_global_cleanup();

    uint8_t public_key[PQCLEAN_KYBER1024_CLEAN_CRYPTO_PUBLICKEYBYTES];
    uint8_t ciphertext[PQCLEAN_KYBER1024_CLEAN_CRYPTO_CIPHERTEXTBYTES];
```

```c
uint8_t shared_secret[PQCLEAN_KYBER1024_CLEAN_CRYPTO_BYTES]
double total_encap_time = 0;

FILE *csv_file = fopen(CSV_FILE, "w");
FILE *log_file = fopen(LOG_FILE, "w");
if (csv_file == NULL || log_file == NULL) {
    printf("Unable to create output files.\n");
    return 1;
}

fprintf(csv_file, "Iteration,Encapsulation Time (seconds)\n");

for (int i = 0; i < ITERATIONS; i++) {
    int sock = 0;
    struct sockaddr_in serv_addr;

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        fprintf(log_file, "Socket creation error (iteration %d)\n", i+1);
        return 1;
    }

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(SERVER_PORT);

    if (inet_pton(AF_INET, SERVER_IP, &serv_addr.sin_addr) <= 0) {
        fprintf(log_file, "Invalid address/Address not supported (iteration %d)\n", i+1);
        close(sock);
        return 1;
    }

    if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
        fprintf(log_file, "Connection Failed (iteration %d)\n", i+1);
        close(sock);
        usleep(RETRY_DELAY);
        continue;
```

```
}

// Request public key
const char *request = "Requesting Public Key";
send(sock, request, strlen(request), 0);

if (read(sock, public_key, PQCLEAN_KYBER1024_CLEAN_CRYPTO_PUBLICKEYBYTES) <= 0) {
  fprintf(log_file, "Failed to receive public key from server (iteration %d).\n", i+1);
  close(sock);
  continue;
}

clock_t start_encap = clock();
if (PQCLEAN_KYBER1024_CLEAN_crypto_kem_enc(ciphertext, shared_secret, public_key) != 0) {
  fprintf(log_file, "Encapsulation failed (iteration %d).\n", i+1);
  close(sock);
  return 1;
}
clock_t end_encap = clock();
double encap_time = (double)(end_encap - start_encap) / CLOCKS_PER_SEC;
total_encap_time += encap_time;

fprintf(csv_file, "%d,%f\n", i+1, encap_time);
fprintf(log_file, "Encapsulation Time (iteration %d): %f seconds\n", i+1, encap_time);

send(sock, ciphertext, PQCLEAN_KYBER1024_CLEAN_CRYPTO_CIPHERTEXTBYTES, 0);

unsigned char aes_key[32];
SHA256(shared_secret, sizeof(shared_secret), aes_key);

unsigned char iv[16];
if (!RAND_bytes(iv, sizeof(iv))) {
  fprintf(log_file, "Failed to generate random IV (iteration %d).\n", i+1);
  close(sock);
  return 1;
```

```
        }

        unsigned char encrypted_data[4096];
        int encrypted_data_len = aes_encrypt(chunk.memory, chunk.size, aes_key, iv, encrypted_data);
        send(sock, iv, sizeof(iv), 0);
        send(sock, encrypted_data, encrypted_data_len, 0);
        fprintf(log_file, "Encrypted data sent to server (iteration %d).\n", i+1);

        // Wait for server acknowledgment before proceeding
        char server_ack[16];
        if (recv(sock, server_ack, sizeof(server_ack), 0) <= 0) {
            fprintf(log_file, "Failed to receive server acknowledgment (iteration %d).\n", i+1);
            close(sock);
            usleep(RETRY_DELAY);
            continue;
        } else {
            fprintf(log_file, "Server acknowledged (iteration %d): %s\n", i+1, server_ack);
        }

        close(sock);  // Close the socket after each iteration
        usleep(RETRY_DELAY);  // Delay to ensure the server is ready for the next connection
    }

    fprintf(log_file, "Average Encapsulation Time: %f seconds\n", total_encap_time / ITERATIONS);
    fprintf(csv_file, "Average,%f\n", total_encap_time / ITERATIONS);

    fclose(csv_file);
    fclose(log_file);
    free(chunk.memory);

    return 0;
}
```

Kyber Server Code

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <openssl/sha.h>
#include <time.h>
#include "api.h"
#include "aes_utils.h"
#define PORT 8080
#define ITERATIONS 1000
#define CSV_FILE "server_timings.csv"
#define LOG_FILE "server_log.txt"

int main() {
  double total_keygen_time = 0;
  double total_decap_time = 0;

  FILE *csv_file = fopen(CSV_FILE, "w");
  FILE *log_file = fopen(LOG_FILE, "w");
  if (csv_file == NULL || log_file == NULL) {
    printf("Unable to create output files.\n");
    return 1;
  }

  fprintf(csv_file, "Iteration,Key Generation Time (seconds),Decapsulation Time (seconds)\n");

  for (int i = 0; i < ITERATIONS; i++) {
    int server_fd, new_socket;
    struct sockaddr_in address;
    socklen_t addrlen = sizeof(address);

    uint8_t public_key[PQCLEAN_KYBER1024_CLEAN_CRYPTO_PUBLICKEYBYTES];
    uint8_t secret_key[PQCLEAN_KYBER1024_CLEAN_CRYPTO_SECRETKEYBYTES];
```

```
}

uint8_t ciphertext[PQCLEAN_KYBER1024_CLEAN_CRYPTO_CIPHERTEXTBYTES];
uint8_t shared_secret[PQCLEAN_KYBER1024_CLEAN_CRYPTO_BYTES];

clock_t start_keygen = clock();
if (PQCLEAN_KYBER1024_CLEAN_crypto_kem_keypair(public_key, secret_key) != 0) {
  fprintf(log_file, "Key pair generation failed (iteration %d).\n", i+1);
  return 1;
}
clock_t end_keygen = clock();
double keygen_time = (double)(end_keygen - start_keygen) / CLOCKS_PER_SEC;
total_keygen_time += keygen_time;

fprintf(log_file, "Key Generation Time (iteration %d): %f seconds\n", i+1, keygen_time);

if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
  perror("Socket failed");
  exit(EXIT_FAILURE);
}

int opt = 1;
if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt))) {
  perror("setsockopt failed");
  exit(EXIT_FAILURE);
}

address.sin_family = AF_INET;
address.sin_addr.s_addr = INADDR_ANY;
address.sin_port = htons(PORT);

if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0) {
  perror("Bind failed");
  exit(EXIT_FAILURE);
}

if (listen(server_fd, 3) < 0) {
    perror("Listen failed");
    exit(EXIT_FAILURE);
}

new_socket = accept(server_fd, (struct sockaddr *)&address, &addrlen);
if (new_socket < 0) {
    perror("Accept failed");
    exit(EXIT_FAILURE);
}

// Receive request for public key from client
char client_request[64];
if (recv(new_socket, client_request, sizeof(client_request), 0) <= 0) {
    fprintf(log_file, "Failed to receive client request (iteration %d).\n", i+1);
    close(new_socket);
    close(server_fd);
    continue;
}

send(new_socket, public_key, PQCLEAN_KYBER1024_CLEAN_CRYPTO_PUBLICKEYBYTES, 0);
fprintf(log_file, "Public key sent to client (iteration %d).\n", i+1);

read(new_socket, ciphertext, PQCLEAN_KYBER1024_CLEAN_CRYPTO_CIPHERTEXTBYTES);
fprintf(log_file, "Ciphertext received from client (iteration %d).\n", i+1);

clock_t start_decap = clock();
if (PQCLEAN_KYBER1024_CLEAN_crypto_kem_dec(shared_secret, ciphertext, secret_key) != 0) {
    fprintf(log_file, "Decapsulation failed (iteration %d).\n", i+1);
    close(new_socket);
    close(server_fd);
    return 1;
}
clock_t end_decap = clock();
double decap_time = (double)(end_decap - start_decap) / CLOCKS_PER_SEC;
total_decap_time += decap_time;
```

```
        fprintf(log_file, "Decapsulation Time (iteration %d): %f seconds\n", i+1, decap_time);

        fprintf(csv_file, "%d,%f,%f\n", i+1, keygen_time, decap_time);

        unsigned char aes_key[32];
        SHA256(shared_secret, sizeof(shared_secret), aes_key);

        unsigned char iv[16];
        read(new_socket, iv, sizeof(iv));

        unsigned char encrypted_data[4096];
        int encrypted_data_len = read(new_socket, encrypted_data, sizeof(encrypted_data));

        fprintf(log_file, "Encrypted data received from client (iteration %d).\n", i+1);

        unsigned char decrypted_data[4096];
        int decrypted_data_len = aes_decrypt(encrypted_data, encrypted_data_len, aes_key, iv, decrypted_data);

        if (decrypted_data_len >= 0) {
            fprintf(log_file, "Decrypted data (iteration %d): %.100s...\n", i+1, decrypted_data);
        } else {
            fprintf(log_file, "Decryption failed (iteration %d).\n", i+1);
        }

        // Send acknowledgment to the client
        const char *ack = "Received";
        send(new_socket, ack, strlen(ack), 0);

        close(new_socket);
        close(server_fd);
    }

    fprintf(log_file, "Average Key Generation Time: %f seconds\n", total_keygen_time / ITERATIONS);
    fprintf(log_file, "Average Decapsulation Time: %f seconds\n", total_decap_time / ITERATIONS);
    fprintf(csv_file, "Average,%f,%f\n", total_keygen_time / ITERATIONS, total_decap_time / ITERATIONS);

    fclose(csv_file);
    fclose(log_file);

    return 0;
}
```

Appendix B:

Sphincs + Client Code

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <time.h>
#include <unistd.h>  // Include for sleep()
#include <curl/curl.h>
#include <openssl/aes.h>
#include <openssl/evp.h>
#include "api.h"
#include "aes_crypto.h"

#define SHA256_DIGEST_LENGTH 32  // Define SHA-256 hash length
#define NUM_ITERATIONS 1000      // Define the number of iterations
#define CSV_FILE "client_timings_sphincs.csv"
#define LOG_FILE "client_log_sphincs.txt"

// Function to download data from a URL
size_t write_data(void *ptr, size_t size, size_t nmemb, void *stream) {
    size_t written = fwrite(ptr, size, nmemb, (FILE *)stream);
    return written;
}
```

```
void download_data(const char *url, const char *file_name) {
    CURL *curl;
    FILE *fp;
    CURLcode res;
    curl = curl_easy_init();
    if (curl) {
        fp = fopen(file_name, "wb");
        curl_easy_setopt(curl, CURLOPT_URL, url);
        curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, write_data);
        curl_easy_setopt(curl, CURLOPT_WRITEDATA, fp);
        res = curl_easy_perform(curl);
        curl_easy_cleanup(curl);
        fclose(fp);
    }
}

void save_to_file(const char *filename, const uint8_t *data, size_t size) {
    FILE *file = fopen(filename, "wb");
    if (!file) {
        fprintf(stderr, "Error: Could not open file %s for writing.\n", filename);
        exit(EXIT_FAILURE);
    }
    fwrite(data, 1, size, file);
    fclose(file);
}

// Function to hash data using the EVP interface (SHA-256)
void hash_data(const unsigned char *data, size_t data_len, unsigned char *output_hash) {
    EVP_MD_CTX *mdctx;
    unsigned int hash_len;

    // Create and initialize the context
    mdctx = EVP_MD_CTX_new();
    if (!mdctx) {
        fprintf(stderr, "Failed to create hash context.\n");
        exit(EXIT_FAILURE);
    }

    // Initialize the hash function (SHA-256)
    if (1 != EVP_DigestInit_ex(mdctx, EVP_sha256(), NULL)) {
        fprintf(stderr, "Failed to initialize hash function.\n");
        EVP_MD_CTX_free(mdctx);
        exit(EXIT_FAILURE);
    }

    // Update the hash with the data
    if (1 != EVP_DigestUpdate(mdctx, data, data_len)) {
        fprintf(stderr, "Failed to update hash with data.\n");
        EVP_MD_CTX_free(mdctx);
        exit(EXIT_FAILURE);
    }

    // Finalize the hash and retrieve the result
    if (1 != EVP_DigestFinal_ex(mdctx, output_hash, &hash_len)) {
        fprintf(stderr, "Failed to finalize hash.\n");
        EVP_MD_CTX_free(mdctx);
        exit(EXIT_FAILURE);
    }

    // Free the context
    EVP_MD_CTX_free(mdctx);
}

void write_ready_flag() {
    FILE *file = fopen("ready.flag", "w");
    if (!file) {
        fprintf(stderr, "Error: Could not create ready flag.\n");
        exit(EXIT_FAILURE);
    }
    fprintf(file, "ready");
    fclose(file);
}

int main() {
    double total_key_generation_time = 0.0;
    double total_signing_time = 0.0;

    // Open CSV file for writing the results
    FILE *csv_file = fopen(CSV_FILE, "w");
    FILE *log_file = fopen(LOG_FILE, "w");

    if (!csv_file || !log_file) {
        fprintf(stderr, "Error: Could not open output files.\n");
        return 1;
    }

    fprintf(csv_file, "Iteration,Key Generation Time (seconds),Signing Time (seconds)\n");

    for (int i = 0; i < NUM_ITERATIONS; i++) {
        fprintf(log_file, "Iteration: %d\n", i + 1);

        uint8_t public_key[PQCLEAN_SPHINCSSHAKE256SSIMPLE_CLEAN_CRYPTO_PUBLICKEYBYTES];
        uint8_t secret_key[PQCLEAN_SPHINCSSHAKE256SSIMPLE_CLEAN_CRYPTO_SECRETKEYBYTES];
        uint8_t signature[PQCLEAN_SPHINCSSHAKE256SSIMPLE_CLEAN_CRYPTO_BYTES];
        size_t signature_len;

        // AES key and IV
        unsigned char key[32];  // 256-bit key
        unsigned char iv[16];   // 128-bit IV
        memset(key, 0x00, sizeof(key));  // Set key to all 0s (in real scenarios, use a secure key)
        memset(iv, 0x00, sizeof(iv));    // Set IV to all 0s

        // Download data
        const char *url = "https://ogcapi.hft-stuttgart.de/sta/icity_data_security/v1.1";
        const char *file_name = "data.json";
```

```
download_data(url, file_name);

// Read data from file
FILE *file = fopen(file_name, "rb");
if (!file) {
  fprintf(log_file, "Error: Could not open file %s for reading.\n", file_name);
  return 1;
}
fseek(file, 0, SEEK_END);
size_t message_len = ftell(file);
rewind(file);

uint8_t *message = malloc(message_len);
if (!message) {
  fprintf(log_file, "Error: Could not allocate memory for message.\n");
  fclose(file);
  return 1;
}
fread(message, 1, message_len, file);
fclose(file);

// Encrypt the data before signing
unsigned char *ciphertext = malloc(message_len + AES_BLOCK_SIZE);
int ciphertext_len = encrypt(message, message_len, key, iv, ciphertext);

// Save the ciphertext to a file
save_to_file("encrypted_data.bin", ciphertext, ciphertext_len);

// Hash the encrypted data
unsigned char data_hash[SHA256_DIGEST_LENGTH];
hash_data(ciphertext, ciphertext_len, data_hash);

// Timing variables
clock_t start, end;
double cpu_time_used;
```

```
    // Key pair generation

    start = clock();

    if (PQCLEAN_SPHINCSSHAKE256SSIMPLE_CLEAN_crypto_sign_keypair(public_key, secret_key) != 0) {

      fprintf(log_file, "Key pair generation failed.\n");

      free(message);

      free(ciphertext);

      return 1;

    }

    end = clock();

    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;

    fprintf(log_file, "Key Generation Time: %f seconds\n", cpu_time_used);

    total_key_generation_time += cpu_time_used;


    // Save public key

    save_to_file("public_key.bin", public_key, sizeof(public_key));


    // Signing the hash of the encrypted message

    start = clock();

    if (PQCLEAN_SPHINCSSHAKE256SSIMPLE_CLEAN_crypto_sign_signature(signature, &signature_len,
data_hash, SHA256_DIGEST_LENGTH, secret_key) != 0) {

      fprintf(log_file, "Signing failed.\n");

      free(message);

      free(ciphertext);

      return 1;

    }

    end = clock();

    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;

    fprintf(log_file, "Signing Time: %f seconds\n", cpu_time_used);

    total_signing_time += cpu_time_used;


    // Save signature

    save_to_file("signature.bin", signature, signature_len);


    // Write results to CSV

    fprintf(csv_file, "%d,%f,%f\n", i + 1, total_key_generation_time / (i + 1), total_signing_time / (i + 1));
```

```
        // Free allocated memory
        free(message);
        free(ciphertext);

        // Write ready flag file
        write_ready_flag();

        fprintf(log_file, "Iteration %d complete.\n", i + 1);
    }

    fprintf(csv_file, "Average,%f,%f\n", total_key_generation_time / NUM_ITERATIONS, total_signing_time /
NUM_ITERATIONS);

    fclose(csv_file);
    fclose(log_file);

    return 0;
}
```

Sphincs + Server code

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <time.h>
#include <unistd.h>  // Include for sleep()
#include <openssl/aes.h>
#include <openssl/evp.h>
#include "api.h"
#include "aes_crypto.h"

#define SHA256_DIGEST_LENGTH 32  // Define SHA-256 hash length
#define NUM_ITERATIONS 1000      // Define the number of iterations
```

```
#define CSV_FILE "server_timings_sphincs.csv"
#define LOG_FILE "server_log_sphincs.txt"

// Function to load data from a file
size_t load_from_file(const char *filename, uint8_t **data) {
    FILE *file = fopen(filename, "rb");
    if (!file) {
        fprintf(stderr, "Error: Could not open file %s for reading.\n", filename);
        exit(EXIT_FAILURE);
    }
    fseek(file, 0, SEEK_END);
    size_t size = ftell(file);
    rewind(file);

    *data = malloc(size);
    if (!*data) {
        fprintf(stderr, "Error: Could not allocate memory for reading data.\n");
        exit(EXIT_FAILURE);
    }

    fread(*data, 1, size, file);
    fclose(file);

    return size;
}

// Function to load a fixed-size array from a file
void load_array_from_file(const char *filename, uint8_t *array, size_t size) {
    FILE *file = fopen(filename, "rb");
    if (!file) {
        fprintf(stderr, "Error: Could not open file %s for reading.\n", filename);
        exit(EXIT_FAILURE);
    }

    if (fread(array, 1, size, file) != size) {
        fprintf(stderr, "Error: Could not read the complete array from file %s.\n", filename);
        fclose(file);
        exit(EXIT_FAILURE);
    }

    fclose(file);
}

// Function to hash data using the EVP interface (SHA-256)
void hash_data(const unsigned char *data, size_t data_len, unsigned char *output_hash) {
    EVP_MD_CTX *mdctx;
    unsigned int hash_len;

    // Create and initialize the context
    mdctx = EVP_MD_CTX_new();
    if (!mdctx) {
        fprintf(stderr, "Failed to create hash context.\n");
        exit(EXIT_FAILURE);
    }

    // Initialize the hash function (SHA-256)
    if (1 != EVP_DigestInit_ex(mdctx, EVP_sha256(), NULL)) {
        fprintf(stderr, "Failed to initialize hash function.\n");
        EVP_MD_CTX_free(mdctx);
        exit(EXIT_FAILURE);
    }

    // Update the hash with the data
    if (1 != EVP_DigestUpdate(mdctx, data, data_len)) {
        fprintf(stderr, "Failed to update hash with data.\n");
        EVP_MD_CTX_free(mdctx);
        exit(EXIT_FAILURE);
    }

    // Finalize the hash and retrieve the result
    if (1 != EVP_DigestFinal_ex(mdctx, output_hash, &hash_len)) {
        fprintf(stderr, "Failed to finalize hash.\n");
        EVP_MD_CTX_free(mdctx);
        exit(EXIT_FAILURE);
    }

    // Free the context
    EVP_MD_CTX_free(mdctx);
}

void wait_for_ready_flag() {
    while (access("ready.flag", F_OK) == -1) {
        usleep(10000);  // sleep for 10ms
    }
    remove("ready.flag");  // clean up the flag file
}

int main() {
    // Open CSV and log files for writing the results
    FILE *csv_file = fopen(CSV_FILE, "w");
    FILE *log_file = fopen(LOG_FILE, "w");

    if (!csv_file || !log_file) {
        fprintf(stderr, "Error: Could not open output files.\n");
        return 1;
    }

    fprintf(csv_file, "Iteration,Verification Time (seconds)\n");

    for (int i = 0; i < NUM_ITERATIONS; i++) {
        fprintf(log_file, "Iteration: %d\n", i + 1);

        // Wait for the client to signal readiness
        wait_for_ready_flag();
        uint8_t public_key[PQCLEAN_SPHINCSSHAKE256SSIMPLE_CLEAN_CRYPTO_PUBLICKEYBYTES];
        uint8_t *signature;
        uint8_t *data;
        size_t data_len;
        size_t signature_len = PQCLEAN_SPHINCSSHAKE256SSIMPLE_CLEAN_CRYPTO_BYTES;

        // Load public key from file
        load_array_from_file("public_key.bin", public_key, sizeof(public_key));

        // Load signature from file
        signature_len = load_from_file("signature.bin", &signature);

        // Load encrypted data from file
        data_len = load_from_file("encrypted_data.bin", &data);

        // Timing variables
        clock_t start, end;
        double cpu_time_used;

        // Hash the encrypted data
        unsigned char data_hash[SHA256_DIGEST_LENGTH];
        hash_data(data, data_len, data_hash);

        // Verify the signature on the hash of the encrypted data
        start = clock();
        int verification_result = PQCLEAN_SPHINCSSHAKE256SSIMPLE_CLEAN_crypto_sign_verify(signature,
signature_len, data_hash, SHA256_DIGEST_LENGTH, public_key);
        end = clock();
        cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;

        if (verification_result != 0) {
          fprintf(log_file, "Signature verification failed on iteration %d.\n", i + 1);
          free(signature);
          free(data);
          return 1;
        } else {
            fprintf(log_file, "Signature verification successful on iteration %d.\n", i + 1);
        }

        fprintf(log_file, "Verification Time: %f seconds\n", cpu_time_used);
        fprintf(csv_file, "%d,%f\n", i + 1, cpu_time_used);

        // Free allocated memory
        free(signature);
        free(data);
    }

    fclose(csv_file);
    fclose(log_file);

    return 0;
}
```

Appendix C:

Diffie-Hellman Client Code

```
import socket
import time
import requests
from cryptography.hazmat.primitives.asymmetric import dh
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.padding import PKCS7
import os

# Helper function to ensure all bytes are sent
```

```
def send_data(conn, data):
    total_sent = 0
    while total_sent < len(data):
        sent = conn.send(data[total_sent:])
        if sent == 0:
            raise RuntimeError("Socket connection broken")
        total_sent += sent

# Open log and CSV files
log_file = open('client_output_dh.txt', 'w')
csv_file = open('client_timings_dh.csv', 'w')
csv_file.write("Iteration,Key Generation Time (s),Shared Secret Time (s),AES Encryption Time (s)\n")

# Set up the client socket
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_socket.connect(('192.168.0.104', 5000))  # Replace with your server's IP

# Receive DH parameters from the server
parameters_bytes = client_socket.recv(2048)
parameters = serialization.load_pem_parameters(parameters_bytes)

for i in range(1000):
    # Measure key generation time (once per client)
    start_time = time.time()
    client_private_key = parameters.generate_private_key()
    client_public_key = client_private_key.public_key()
    key_generation_time = time.time() - start_time

    # Send Client's public key to the server
    client_public_key_bytes = client_public_key.public_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo
    )
    send_data(client_socket, client_public_key_bytes)
```

```
# Measure shared secret generation time (once per client)
start_time = time.time()
server_public_key_bytes = client_socket.recv(2048)
server_public_key = serialization.load_pem_public_key(server_public_key_bytes)
shared_secret = client_private_key.exchange(server_public_key)
shared_secret_time = time.time() - start_time

# Derive AES key using shared secret
aes_key = HKDF(
    algorithm=hashes.SHA256(),
    length=32,  # AES-256 key
    salt=None,
    info=b'handshake data',
    backend=default_backend()
).derive(shared_secret)

# Fetch data from the URL to encrypt
url = "https://ogcapi.hft-stuttgart.de/sta/icity_data_security/v1.1"
response = requests.get(url)

if response.status_code == 200:
    data_to_encrypt = response.content  # Data fetched from the URL
else:
    log_file.write(f"Failed to fetch data from the URL at iteration {i+1}\n")
    client_socket.close()
    exit()

# Prepare IV for AES encryption
iv = os.urandom(16)

# Measure AES encryption time
start_time = time.time()
cipher = Cipher(algorithms.AES(aes_key), modes.CBC(iv), backend=default_backend())
padder = PKCS7(algorithms.AES.block_size).padder()
padded_data = padder.update(data_to_encrypt) + padder.finalize()
```

```
    encryptor = cipher.encryptor()

    ciphertext = encryptor.update(padded_data) + encryptor.finalize()

    encryption_time = time.time() - start_time


    # Send IV and encrypted data to the server

    send_data(client_socket, iv)

    send_data(client_socket, len(ciphertext).to_bytes(4, byteorder='big'))

    send_data(client_socket, ciphertext)


    # Wait for acknowledgment from the server

    ack = client_socket.recv(1024)

    if ack != b'ACK':

        log_file.write(f"Failed to receive acknowledgment at iteration {i+1}\n")

        break


    # Log times for this iteration

    log_file.write(f"Iteration {i+1}: Key Generation: {key_generation_time:.6f} s, Shared Secret: 
{shared_secret_time:.6f} s, AES Encryption: {encryption_time:.6f} s\n")

    csv_file.write(f"{i+1},{key_generation_time:.6f},{shared_secret_time:.6f},{encryption_time:.6f}\n")


# Close the connection and files

client_socket.close()

log_file.close()

csv_file.close()
```

Diffie-Hellman Server Code

```
import socket

import time

from cryptography.hazmat.primitives.asymmetric import dh

from cryptography.hazmat.primitives import serialization

from cryptography.hazmat.primitives.kdf.hkdf import HKDF

from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.padding import PKCS7

# Helper function to ensure we read all bytes
def receive_data(conn, length):
    data = b''
    while len(data) < length:
        packet = conn.recv(length - len(data))
        if not packet:
            break
        data += packet
    return data

# Open log and CSV files
log_file = open('server_output_dh.txt', 'w')
csv_file = open('server_timings_dh.csv', 'w')
csv_file.write("Iteration,Key Generation Time (s),Shared Secret Time (s),AES Decryption Time (s)\n")

# Set up the server
parameters = dh.generate_parameters(generator=2, key_size=2048)

# Set up the server socket
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind(('192.168.0.104', 5000))  # Replace with your server's IP
server_socket.listen(1)
print("Server listening on port 5000...")

conn, addr = server_socket.accept()
print(f"Connection established with {addr}")

# Send DH parameters to the client
parameters_bytes = parameters.parameter_bytes(
    encoding=serialization.Encoding.PEM,
    format=serialization.ParameterFormat.PKCS3
)
conn.sendall(parameters_bytes)

for i in range(1000):
    # Measure key generation time (once per server)
    start_time = time.time()
    server_private_key = parameters.generate_private_key()
    server_public_key = server_private_key.public_key()
    key_generation_time = time.time() - start_time

    # Send Server's public key to the client
    server_public_key_bytes = server_public_key.public_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo
    )
    conn.sendall(server_public_key_bytes)

    # Measure shared secret generation time (once per server)
    start_time = time.time()
    client_public_key_bytes = receive_data(conn, 2048)
    client_public_key = serialization.load_pem_public_key(client_public_key_bytes)
    shared_secret = server_private_key.exchange(client_public_key)
    shared_secret_time = time.time() - start_time

    # Derive AES key using shared secret
    aes_key = HKDF(
        algorithm=hashes.SHA256(),
        length=32,  # AES-256 key
        salt=None,
        info=b'handshake data',
        backend=default_backend()
    ).derive(shared_secret)

    # Receive IV and encrypted data
    iv = receive_data(conn, 16)  # Receive the IV
    encrypted_data_length = int.from_bytes(receive_data(conn, 4), byteorder='big')
    received_data = receive_data(conn, encrypted_data_length)

    # Measure AES decryption time
    start_time = time.time()
    cipher = Cipher(algorithms.AES(aes_key), modes.CBC(iv), backend=default_backend())
    decryptor = cipher.decryptor()
    decrypted_padded_message = decryptor.update(received_data) + decryptor.finalize()
    decryption_time = time.time() - start_time

    # Unpad the message
    unpadder = PKCS7(algorithms.AES.block_size).unpadder()
    decrypted_message = unpadder.update(decrypted_padded_message) + unpadder.finalize()

    # Log times for this iteration
    log_file.write(f"Iteration {i+1}: Key Generation: {key_generation_time:.6f} s, Shared Secret: {shared_secret_time:.6f} s, AES Decryption: {decryption_time:.6f} s\n")
    csv_file.write(f"{i+1},{key_generation_time:.6f},{shared_secret_time:.6f},{decryption_time:.6f}\n")

    # Send acknowledgment to the client
    conn.sendall(b'ACK')

# Close the connection and files
conn.close()
server_socket.close()
log_file.close()
csv_file.close()
```