Module Lattice - Digital Signature Algorithm (ML-DSA) for Financial Transactions

An Industry Oriented Mini Project Report (CS755PC)

Submitted

in partial fulfilment of the

requirements for the award of the

degree of

BACHELOR OF TECHNOLOGY

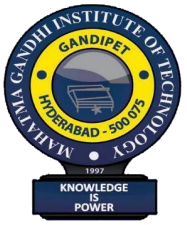
in

COMPUTER SCIENCE AND ENGINEERING

by

SHUCHITA PRAKASH

[Reg. No. 21261A0557]



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

MAHATMA GANDHI INSTITUTE OF TECHNOLOGY

HYDERABAD - 500075, TELANGANA

DECEMBER - 2024

ABSTRACT

In the FinTech industry, digital signatures play a crucial role in ensuring data integrity and authenticating the identity of participants in financial transactions. By applying digital signatures to transaction data, unauthorized modifications can be detected, and the identities of the signatories can be confirmed. This is especially important for sensitive financial information, where non-repudiation is essential: signatories cannot later deny their involvement in the transaction, providing accountability and trust. This standard specifies the use of CRYSTALS Dilithium, later renamed as ML-DSA, a set of algorithms designed to generate and verify digital signatures that meet the stringent requirements of FinTech applications. ML-DSA is believed to be resistant to attacks from even large-scale quantum computers, offering a level of security fit for the evolving needs of the FinTech industry. By adopting quantum-safe digital signatures, financial institutions can future-proof their systems, maintaining secure and verifiable transactions in an increasingly digitized and regulated environment.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

CHAPTER-1

INTRODUCTION

This project explores the implementation of ML-DSA (Module Lattice - Digital Signature Algorithm) or CRYSTALS Dilithium Algorithm within a payment gateway framework to enhance the security of digital transactions. Given the growing demand for robust cryptographic mechanisms that can withstand quantum computing threats, ML-DSA offers a novel approach by leveraging post-quantum cryptographic signatures.

The system allows users to initiate fund transfers by signing transaction data, which includes payer and payee account information along with the transfer amount. Once a transaction request is signed on the client side, the generated signature is sent to the backend for validation, ensuring both data integrity and authenticity. This end-to-end approach strengthens the transaction process by verifying each transfer's authenticity without requiring public key provision by the client.

The digital signature scheme approved in this standard is the Module-Lattice-Based Digital Signature Algorithm (ML-DSA), which is based on the Module Learning with Errors problem. ML-DSA is considered secure, even against adversaries equipped with large-scale fault-tolerant quantum computers. In particular, ML-DSA is believed to be strongly unforgeable, which implies that the scheme can detect unauthorized modifications to data and authenticate the identity of the signatory (tied to the possession of the private key). Additionally, a signature generated by this scheme can serve as evidence, demonstrating to a third party that the signature was indeed generated by the claimed signatory. This property, known as non-repudiation, ensures the signatory cannot easily repudiate the signature later.

This project demonstrates ML-DSA’s applicability in a payment gateway

environment, assessing both its feasibility and performance in securing financial transactions.

1.1 PROBLEM DEFINITION

With the advancement of quantum computing, traditional cryptographic algorithms face increasing vulnerability to quantum-based attacks, threatening the security foundations of modern digital transactions. Quantum computers, by leveraging quantum phenomena, have the potential to solve complex mathematical problems—such as integer factorization and discrete logarithms—that form the basis of widely used cryptographic schemes like RSA and ECC (Elliptic Curve Cryptography). As quantum computing power grows, these algorithms are projected to become obsolete, compromising sensitive data and enabling potential unauthorized access to financial systems.

This presents a significant risk to secure payment platforms and digital transactions, which rely on public-key cryptographic protocols for authentication, integrity, and confidentiality. Without adequate protection, transactions could be forged or tampered with, leading to data breaches and financial fraud on a large scale. Therefore, developing and implementing post-quantum cryptographic solutions is critical to ensuring the security of financial systems in a quantum-capable world.

1.2 EXISTING SOLUTION

Quantum Key Distribution (QKD) is a security technology that uses the principles of quantum mechanics to enable secure key exchange between two parties. Unlike traditional cryptographic methods that rely on the difficulty of mathematical problems, QKD achieves security through the fundamental properties of quantum particles, typically photons. By encoding key information in the quantum states of these particles, QKD creates a key that both sender (Alice) and receiver (Bob) can use to encrypt their messages securely. A unique aspect of QKD is its inherent ability to

detect any eavesdropping attempts; if an intruder tries to intercept the key exchange, the quantum states of the particles will be altered, alerting the parties and allowing them to discard the compromised key.

The process of QKD generally involves two phases: the quantum transmission of particles to establish the key and a classical communication phase to verify the key's integrity. During the quantum transmission, Alice sends individual photons encoded with information to Bob. Depending on the protocol used (such as BB84 or E91), the photons may be polarized or entangled, creating a secure channel for key distribution. The classical phase allows Alice and Bob to confirm that they share the same key without revealing it directly, using principles of randomness and probability to ensure any interception is detectable. If the key is intercepted or altered, QKD protocols ensure it is discarded, and the exchange can start over.

Disadvantages

1. **High Costs:** The infrastructure required for QKD is expensive, with significant costs for quantum communication hardware and maintenance.
2. **Distance Limitations:** QKD systems are limited by the distance over which they can reliably transmit keys, often only feasible for short to moderate ranges.
3. **Complex Infrastructure Requirements:** QKD requires specialized quantum communication channels, like optical fibers or satellite links, which are challenging to implement on a large scale.
4. **Scalability Issues:** The need for dedicated quantum channels makes it difficult to scale QKD for widespread use across different networks.
5. **Interoperability Challenges:** QKD systems do not easily integrate with existing classical networks, leading to compatibility issues.
6. **Performance Trade-offs:** Due to the sensitive nature of quantum

channels, QKD can be slow and prone to disruptions from environmental factors.

1.3 PROPOSED SOLUTION

This project proposes implementing ML-DSA (Module Lattice Digital Signature Algorithm) as a quantum-resistant solution for securing digital transactions within a payment gateway framework. ML-DSA is a post-quantum cryptographic signature scheme designed to protect data integrity and authenticate transactions in a world where quantum computers could potentially break traditional cryptographic algorithms. By incorporating ML-DSA, we aim to create a system that remains secure against both classical and quantum computing threats, ensuring that each transaction is signed and validated with post-quantum resilience.

The proposed solution involves generating and verifying ML-DSA signatures at the server end. During a transaction, a signature is generated based on key data (such as payer and payee accounts and the transaction amount). This signature is accessible to the backend, where the ML-DSA verify algorithm verifies it, ensuring that the data has not been tampered with.

This approach enhances the payment gateway's robustness against quantum attacks. By implementing this proof of concept, we aim to evaluate ML-DSA's viability as a standardized quantum-resistant solution that can be seamlessly integrated into financial systems and other critical applications vulnerable to future quantum threats.

Advantages:

1. **Quantum Resistance**: ML-DSA is designed to withstand quantum attacks, ensuring that digital transactions remain secure even as quantum computing technology advances, making it a future-proof solution for post-quantum cryptography.

2. **Enhanced Security for Digital Transactions**: The system guarantees data integrity and authenticates transactions, protecting against tampering and unauthorized modifications, which is crucial for ensuring the reliability of financial transactions.
3. **Compatibility with Existing Frameworks**: ML-DSA can be integrated into existing payment gateway systems with minimal disruption, adding an extra layer of security without requiring major changes to the current infrastructure.
4. **Scalability**: The backend-based implementation of ML-DSA allows the system to handle large volumes of transactions, ensuring that it remains efficient and effective even as transaction numbers grow.
5. **Tamper-Proof Transactions**: The signature generation and verification process ensures that any modifications to transaction data are detectable, increasing the trustworthiness of the payment system.
6. **Future-Proofing**: As quantum computing continues to develop, ML-DSA will keep transaction data secure, helping businesses and financial systems stay protected from emerging quantum threats.

1.4 INTRODUCTION TO LATTICE THEORY

Introduction to Lattices: A lattice is a structured grid of points extending infinitely in multiple dimensions, defined mathematically by a set of basis vectors. Each lattice point can be reached through integer combinations of these basis vectors. Despite their simple geometric appearance, lattices form the core of some of the hardest problems in computational mathematics, making them ideal for cryptographic application.

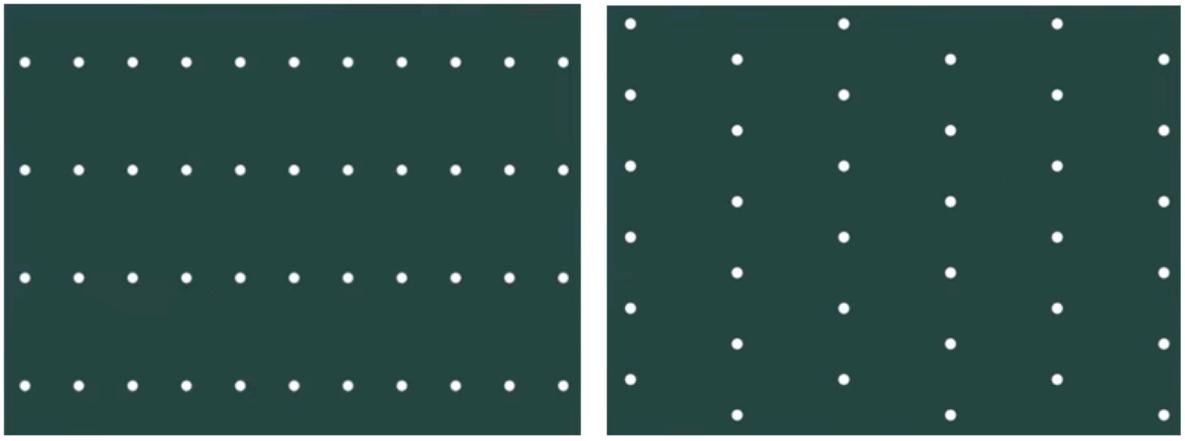


Fig. 1.1 Two different 2D lattice structures

In simple words, a lattice is a bunch of points or dots in a repeating pattern. And while these pictures look pretty simple, they're the setup for some seriously hard math problems.

Basis Vectors and Lattice Construction: A lattice is formally defined by a set of basis vectors. In two dimensions, for example, a lattice is generated by two vectors, say v1 and v2. Starting from the origin, each point in the lattice is formed by an integer combination of these vectors:

$$p=av1+bv2$$

where a and b are integers.

Changing the basis vectors alters how the lattice is described but does not change the lattice itself. A key observation is that different sets of basis vectors can generate the same lattice. For example, a "good basis" consists of vectors that are nearly perpendicular, while a "bad basis" has vectors that are nearly parallel. The latter makes certain computational problems far more difficult to solve.

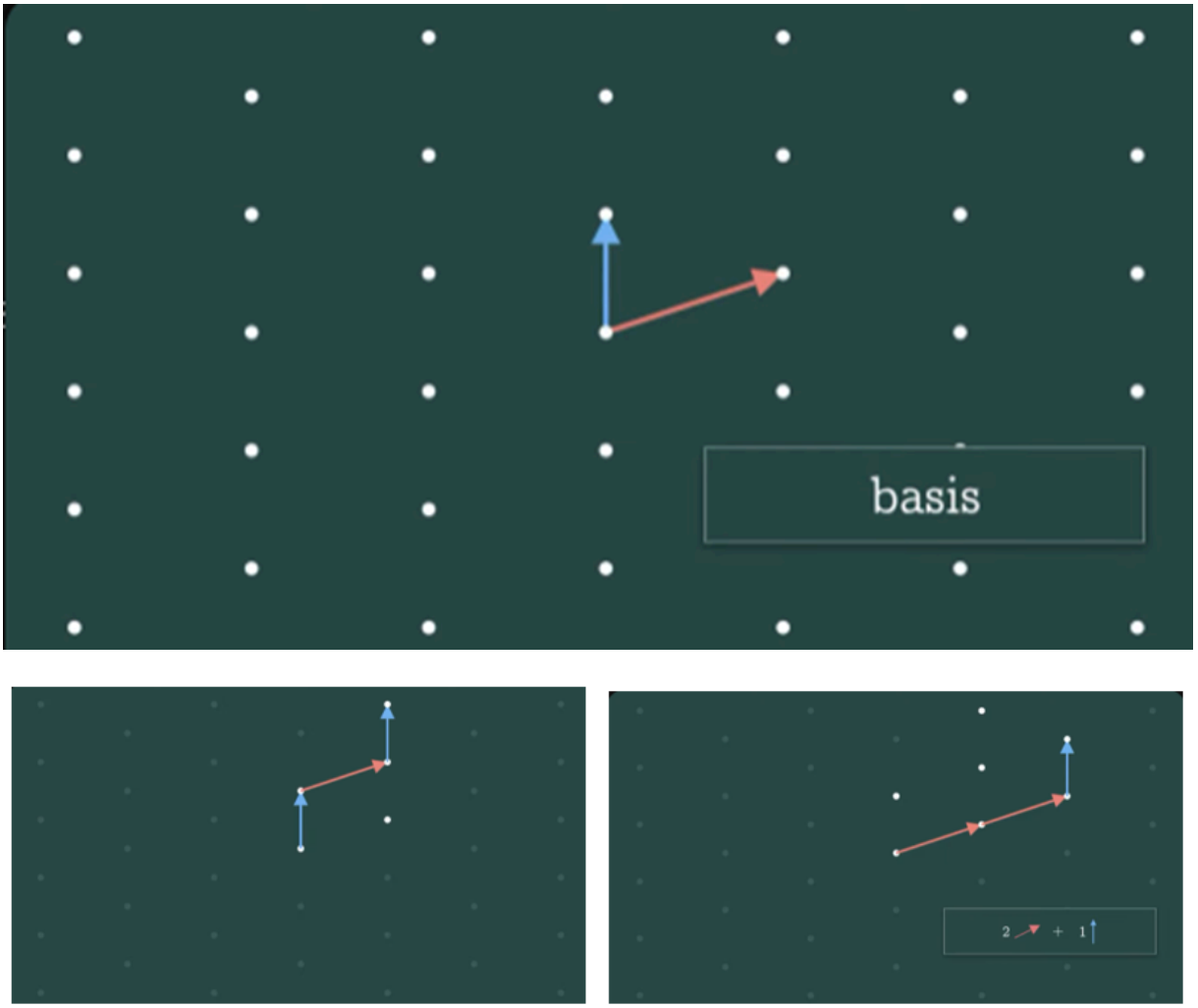


Fig. 1.2 Construction of a lattice from the basis vector

Multiple Bases for the Same Lattice: Consider the standard square lattice generated by basis vectors (1,0) and (0,1). Another pair of basis vectors, such as (7,3) and (2,1), can also generate this lattice. Though these bases differ visually, they define the same set of lattice points. This property underscores the flexibility in lattice representation and is crucial in cryptographic schemes, where one basis can be kept private while another is made public.

The **Learning With Errors (LWE)** problem is a cornerstone of modern lattice-based cryptography and serves as the foundation for many post-quantum cryptographic schemes. The LWE problem involves solving a system of noisy linear equations, where each equation includes a small

random error. This noise makes it computationally difficult to find the original solution, even with access to all other parameters.

Formally, the LWE problem can be described as follows:

> Given a random matrix **A** and a noisy vector **b**, such that **b = A·s + e**, where:
>
> a. **A** is a public random matrix.
> b. **s** is a secret vector (private).
> c. **e** is a small error vector (noise).

The goal is to recover the secret vector **s**. However, due to the addition of **e**, this problem is known to be computationally hard for both classical and quantum computers.

Lattice Problems: Lattice-based cryptography relies on the difficulty of solving certain computational problems involving lattices. Two key problems are Shortest Vector Problem and Closest Vector Problem. The Lattice problem used in ML-DSA is Learning With Errors:

Complexity in Higher Dimensions: Both SVP and CVP become significantly harder as the lattice's dimensionality increases. In higher dimensions, the number of basis vectors grows, and the relationships between lattice points become less intuitive. For instance, a 17-dimensional lattice involves combinations of 17 basis vectors, exponentially increasing the problem's complexity. This inherent difficulty, even for quantum computers, is why these problems are central to post-quantum cryptographic schemes.

Lattices and their basis vectors offer a rich mathematical framework with immense potential for cryptography. The computational hardness of SVP, CVP, and related lattice problems underpins the security of many post-quantum cryptographic algorithms. As quantum computing advances, lattice-based cryptography stands as a robust candidate for

securing future communications.

1.5 ALGORITHMS USED

ML-DSA Key Generation

The key generation algorithm ML-DSA.KeyGen takes no input and outputs a public key and a private key, which are both encoded as byte strings.

The algorithm uses an approved RBG to generate a 256-bit (32-byte) random seed ξ that is given as input to ML-DSA.KeyGen_internal (Algorithm 6), which produces the public and private keys.

Algorithm 1 ML-DSA.KeyGen()

Generates a public-private key pair.

Output: Public key $pk \in \mathbb{B}^{32+32k(\text{bitlen }(q-1)-d)}$
and private key $sk \in \mathbb{B}^{32+32+64+32\cdot((\ell+k)\cdot\text{bitlen }(2\eta)+dk)}$.

1: $\xi \leftarrow \mathbb{B}^{32}$ ▷ choose random seed
2: **if** $\xi =$ NULL **then**
3: **return** $\perp$ ▷ return an error indication if random bit generation failed
4: **end if**
5: **return** ML-DSA.KeyGen_internal (ξ)

Alg. 1.1 ML-DSA.KeyGen() algorithm

ML-DSA Signing

The signing algorithm ML-DSA.Sign takes a private key, a message, and a context string as input. It outputs a signature that is encoded as a byte string.

For the default “hedged” version of ML-DSA signing, the algorithm (at line 5) uses an approved RBG to generate a 256-bit (32-byte) random seed *rnd*. If the deterministic variant is desired, then *rnd* is set to the fixed zero string {0}. The value *rnd*, the private key, and the encoded message are input to ML-DSA.Sign_internal, which produces the signature.

Algorithm 2 ML-DSA.Sign(sk, M, ctx)

Generates an ML-DSA signature.

Input: Private key $sk \in \mathbb{B}^{32+32+64+32\cdot((\ell+k)\cdot \text{bitlen }(2\eta)+dk)}$, message $M \in \{0,1\}^*$, context string ctx (a byte string of 255 or fewer bytes).
Output: Signature $\sigma \in \mathbb{B}^{\lambda/4+\ell\cdot 32\cdot(1+\text{bitlen }(\gamma_1-1))+\omega+k}$.

1: **if** $|ctx| > 255$ **then**
2: **return** $\perp$ ▷ return an error indication if the context string is too long
3: **end if**
4:
5: $rnd \leftarrow \mathbb{B}^{32}$ ▷ for the optional deterministic variant, substitute $rnd \leftarrow \{0\}^{32}$
6: **if** $rnd =$ NULL **then**
7: **return** $\perp$ ▷ return an error indication if random bit generation failed
8: **end if**
9:
10: $M' \leftarrow$ BytesToBits(IntegerToBytes$(0,1) \parallel$ IntegerToBytes$(|ctx|,1) \parallel ctx) \parallel M$
11: $\sigma \leftarrow$ ML-DSA.Sign_internal(sk, M', rnd)
12: **return** σ

Alg. 1.2 ML-DSA.Sign() Algorithm

ML-DSA Verifying

The verification algorithm ML-DSA.Verify (Algorithm 3) takes a public key, a message, a signature, and a context string as input. The public key, signature, and context string are all encoded as byte strings, while the message is a bit string. ML-DSA.Verify outputs a Boolean value that is true if the signature is valid with respect to the message and the public key and false if the signature is invalid. The verification is accomplished by calling ML-DSA.Verify_internal (Algorithm 8) with the public key, the encoded message, and the signature.

Algorithm 3 ML-DSA.Verify(pk, M, σ, ctx)

Verifies a signature σ for a message M.

Input: Public key $pk \in \mathbb{B}^{32+32k(\text{bitlen }(q-1)-d)}$, message $M \in \{0,1\}^*$,
signature $\sigma \in \mathbb{B}^{\lambda/4+\ell\cdot 32\cdot(1+\text{bitlen }(\gamma_1-1))+\omega+k}$,
context string ctx (a byte string of 255 or fewer bytes).
Output: Boolean.

1: **if** $|ctx| > 255$ **then**
2: **return** $\perp$ ▷ return an error indication if the context string is too long
3: **end if**
4:
5: $M' \leftarrow$ BytesToBits(IntegerToBytes$(0,1) \parallel$ IntegerToBytes$(|ctx|,1) \parallel ctx) \parallel M$
6: **return** ML-DSA.Verify_internal(pk, M', σ)

Alg.1. 3 ML-DSA.Verify() Algorithm

1.6 REQUIREMENTS SPECIFICATION

Requirement Specifications describe the arti-craft of Software Requirements and Hardware Requirements used in this project.

Software Requirements

1. **OS to build:** macOS
2. **IDE:** Visual Studio Code
3. **Text editor:** Visual Studio Code 1.77 or later
4. **Framework:** HTML and CSS for frontend and Node.js(Express) for backend development
6. **Target OS(s):** Cross-platform (Windows, macOS, Linux)

Hardware Requirements

1. **Computer requirements to develop**
   a. x86_64 CPU Cores: 8
   b. Memory in GB: 8
   c. Display resolution in pixels: 1366 x 768
   d. Free disk space in GB: 10
2. **Target device requirements:**
   a. RAM: 4GB or more
   b. Storage: 1GB or more
   c. Screen type: Touch or non-touch
   d. Screen size: 5 inches or more

e. Connectivity: Wi-Fi (2.4Ghz/5Ghz) or cellular network (5G/4G/LTE)

CHAPTER 2

LITERATURE SURVEY

[1] "Recommendation for Obtaining Assurances for Digital Signature Applications"

Authors: Elaine Barker

The authors specify methods for ensuring the validity of digital signatures, highlighting the importance of authenticity in the process. They outline the necessary assurances for valid digital signatures, including the validity of domain parameters, public key validity, verification that the owner of the key pair possesses the corresponding private key, and confirmation of the identity of the key pair owner. These assurances are essential for establishing trust in the digital signature process.

https://doi.org/10.6028/NIST.SP.800-89

[2] "Lattice-Based Cryptography"

Authors: Daniele Micciancio, Oded Regev

The authors describe recent advancements in lattice-based cryptography, emphasizing its potential for post-quantum cryptography. They highlight the strong security proofs based on worst-case hardness, the efficiency of implementations, and the simplicity of lattice-based cryptographic constructions. Additionally, they note that lattice-based cryptography is believed to be secure against quantum computers. The authors focus on the practical aspects of lattice-based cryptography, rather than the detailed methods used to establish their security. They also reference other surveys and resources on the topic, such as works by Nguyen and Stern and the book by Micciancio and Goldwasser, which provide more insights into the computational complexity and applications of lattices in cryptanalysis.

https://cims.nyu.edu/~regev/papers/pqc.pdf

[3] "Digital Signature"

Authors: Ravneet Kaur, Amandeep Kaur

The authors explain the different types of encryption techniques used to

ensure the privacy of data transmitted over the internet. They focus on the concept of digital signatures, which are mathematical schemes designed to guarantee the privacy of communication, data integrity, authenticity of the digital message or sender, and non-repudiation of the sender. They describe how digital signatures can be embedded in hardware devices or stored as files on digital storage media. The authors also highlight that digital signatures are typically certified by third-party certifying authorities. In their paper, they elaborate on the key factors of digital signatures, detailing the various methods and procedures involved in signing data or messages using digital signatures, and introduce the algorithms that are commonly used for this purpose.

https://ieeexplore.ieee.org/document/6391693

[4] "Worst-case to average-case reductions for module lattices"

Authors: Adeline Langlois, Damien Stehle

The authors examine the relationship between lattice-based cryptographic schemes and the hardness of key problems, specifically the Short Integer Solution (SIS) and Learning With Errors (LWE) problems. They discuss how the efficiency of these schemes can be improved by shifting from SIS and LWE to the more compact Ring-SIS and Ring-LWE problems. However, they highlight a potential security tradeoff, as SIS and LWE are known to be at least as hard as standard lattice problems on Euclidean lattices, whereas Ring-SIS and Ring-LWE are only as hard as problems on ideal lattices. To address this, the authors define the Module-SIS and Module-LWE problems, which bridge the gap between SIS and Ring-SIS, and LWE and Ring-LWE, respectively. They prove that these new problems are at least as hard as standard lattice problems restricted to module lattices, which connect arbitrary and ideal lattices. The authors also note that the worst-case to average-case reductions for the module problems are sharp, with converse reductions, unlike in the case of Ring-SIS and Ring-LWE, where ideal lattice problems may not fully reflect the hardness of the original problems

https://doi.org/10.1007/s10623-014-9938-4

[5] "A Review Paper on Cryptography"

Authors: Abdalbasit Mohammed, Nurhayat Varol

The authors discuss the rapid growth of the internet and its integration into everyday life, which has heightened concerns regarding data security. They explain that data security ensures that information remains accessible only to the intended recipient, protecting it from unauthorized modification or tampering. To achieve this level of security, various cryptographic algorithms and techniques have been developed. These methods rely on specific algorithms to encrypt data, making it unreadable to unauthorized individuals unless decrypted using predefined algorithms known only to the sender.

https://www.researchgate.net/publication/334418542_A_Review_Paper_on_Cryptography

[6] "Post Quantum Cryptography(PQC) - An overview: (Invited Paper)"

Authors: Manoj Kumar, Pratap Pattnaik

The authors discuss the Post-Quantum Cryptography (PQC) algorithms under consideration by NIST for standardization, focusing on three key candidates: Crystals-Kyber, Classic McEliece, and Supersingular Isogeny Key Encapsulation (SIKE). They describe the hard problems that form the basis of the security for each algorithm, the algebraic structures (such as groups or finite fields) used in their computations, and the fundamental operations involved in each algorithm. Additionally, they examine the performance considerations for implementing these algorithms efficiently on conventional many-core processors. For Crystals-Kyber and SIKE, the authors explore potential solutions to enhance their performance on such processors.

https://ieeexplore.ieee.org/document/9286147

[7] "Public Key cryptography"

Authors: Dwi Liestyowati

The author describes Public Key Cryptography as a software model that involves encoding information, specifically documents contained within a file. They explain the process starts by selecting the master file or

document to be encoded. The encoding is achieved through a combination method, which includes shifting and applying a passphrase to the document's contents. The resulting encoded document is formed through an encryption function. To retrieve the original document, the contents are decrypted by reversing the process: applying the passphrase, shifting, and recombining the elements, ultimately restoring the master document through the decryption function .
https://iopscience.iop.org/article/10.1088/1742-6596/1477/5/052062/pdf

[8] "Guideline for Using Cryptographic Standards in the Federal Government: Cryptographic Mechanisms"

Authors: Elaine Barker

The authors provide guidance for the federal government on how to use cryptography and NIST's cryptographic standards to protect sensitive but unclassified digital information during both transmission and storage. They discuss the cryptographic methods and services that should be used to ensure the security of this information, offering a framework for safeguarding data in these contexts .
https://doi.org/10.6028/NIST.SP.800-175Br1

[9] "Digital Signature Standard (DSS)"

Authors: NIST

The authors describe a standard that specifies a suite of algorithms designed to generate digital signatures. These signatures serve two main purposes: detecting unauthorized modifications to data and authenticating the identity of the signatory. Additionally, the authors explain that a recipient of signed data can use the digital signature as evidence to prove to a third party that the signature was generated by the claimed signatory. This process is referred to as non-repudiation, as it ensures that the signatory cannot easily deny their signature at a later time.
https://doi.org/10.6028/NIST.FIPS.186-5

[10] "Federal Information Processing Standard(FIPS)"

Authors: NIST

The authors explain that a Federal Information Processing Standard (FIPS) is a standard developed within the Information Technology Laboratory and published by NIST, a part of the U.S. Department of Commerce. They describe how FIPS covers various topics in information technology, with the goal of achieving a common level of quality or ensuring interoperability across federal departments and agencies.
https://csrc.nist.gov/glossary/term/federal_information_processing_standard

[11] "Module-Lattice-Based Digital Signature Standard"

Authors: NIST

The authors explain that digital signatures serve to detect unauthorized modifications to data and authenticate the identity of the signatory. They further state that recipients of signed data can use a digital signature as evidence to demonstrate to a third party that the signature was indeed generated by the claimed signatory. This concept, known as non-repudiation, ensures that the signatory cannot easily deny having signed the data at a later time. The authors introduce ML-DSA, a set of algorithms designed for generating and verifying digital signatures. According to them, ML-DSA is considered secure even against adversaries equipped with large-scale quantum computers.
https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.204.pdf

S.No	Author(s)	Title	Year	Journal /Source	Methodology	Merits	Demerits
1	Elaine Barker	Recommendation for Obtaining Assurances for Digital Signature Applications	2006	NIST Special Publication (Link)	Specifies methods to ensure digital signature validity, focusing on verifying domain parameters, public key validity, and key pair ownership.	Provides a detailed framework for establishing authenticity and trust in digital signatures, ensuring secure digital transactions.	Primarily addresses federal standards; may not address all needs of non-federal or commercial digital signature applications.

2	Daniele Micciancio, Oded Regev	Lattice-Based Cryptography	2009	NYU (available at Link)	Overview of lattice-based cryptographic methods focusing on post-quantum security and practical cryptographic constructions.	Strong security proofs based on worst-case hardness; secure against quantum attacks; relatively efficient implementations.	Limited detail on the specific mathematical techniques for security; theoretical focus, with limited real-world implementations discussed
3	Ravneet Kaur, Amandeep Kaur	Digital Signature	2012	IEEE Xplore (Link)	Explanation of digital signature schemes; discusses methods to secure data privacy, integrity, authenticity, and non-repudiation.	Provides a clear overview of digital signatures, explaining key aspects such as certification, verification, and algorithms used.	Limited focus on practical applications or comparisons between various digital signature algorithms.
4	Adeline Langlois, Damien Stehle	Worst-case to Average-case Reductions for Module Lattices	2014	Springer (Journal of Cryptology) (Link)	Examines the relationship between SIS, LWE, Ring-SIS, Ring-LWE, and Module-SIS/Module-LWE problems; explores security tradeoffs and reductions.	Provides sharp reductions for module lattices, bridging the gap between standard and ideal lattices; improves understanding of hardness assumptions.	Focuses primarily on theoretical aspects; the practical implications of these reductions are not extensively explored.
5	Abdalbasit Mohammed, Nurhayat Varol	A Review Paper on Cryptography	2019	Research Gate (Link)	Review and analysis of cryptographic algorithms; discusses data encryption and decryption techniques for securing information.	Provides a comprehensive overview of cryptographic techniques, including their applications for enhancing data security on the internet.	Lacks a detailed empirical analysis or evaluation of each algorithm's performance; limited focus on practical implementations or real-world case studies.
6	Manoj Kumar, Pratap Pattnaik	Post Quantum Cryptography (PQC) -	2020	IEEE Xplore (Link)	Overview of PQC algorithms, focusing on	Provides insights into the structure and computational	Limited to a high-level overview; lacks a detailed

		An Overview: (Invited Paper)			Crystals-Kyber, Classic McEliece, and SIKE; discusses their security basis, algebraic structures, and performance on many-core processors.	basis of PQC candidates; includes performance considerations for practical implementation.	analysis of each algorithm's security parameters or in-depth performance benchmarks.
7	Dwi Liestyowati	Public Key Cryptography	2020	IOP Science (Link)	Describes a public key cryptography model involving encryption and decryption via shifting and passphrase application to documents.	Provides a straightforward description of the encryption and decryption process for securing document contents.	Focuses on a specific encoding technique without addressing broader cryptographic algorithms or applications.
8	Elaine Barker	Guideline for Using Cryptographic Standards in the Federal Government: Cryptographic Mechanisms	2020	NIST Special Publication (Link)	Provides guidance on implementing NIST cryptographic standards to protect sensitive but unclassified information during transmission and storage.	Offers a practical framework tailored for federal agencies, promoting consistent cryptographic practices for data security.	Focused on federal applications; may not fully address cryptographic needs of non-federal organizations or broader use cases.
9	NIST	Digital Signature Standard (DSS)	2023	NIST FIPS Publication (Link)	Specifies a set of algorithms for generating digital signatures for data integrity, authentication, and non-repudiation.	Ensures secure authentication and non-repudiation, widely used for trusted digital communication and data integrity.	Limited to predefined algorithms, which may not cover all emerging needs or applications in non-federal contexts.
10	NIST	Federal Information Processing Standard (FIPS)	N/A	NIST Glossary (Link)	Development of standards within the Information Technology Laboratory to ensure quality and	Provides uniform standards across federal agencies, ensuring data security, interoperability,	Limited to federal applications; may not address all requirements of non-federal or private sector

					interoperability across federal agencies	and quality.	entities.
11	NIST	Module-Lattice-Based Digital Signature Standard (FIPS 204)	2021	NIST FIPS Publication (Link)	Specifies a digital signature standard based on module lattices, focusing on their use in secure signatures.	Establishes cryptographic standards to enhance digital security in federal applications.	Limited to federal use; potential adaptability issues for broader non-governmental use cas

Table 2.1 Literature Survey

CHAPTER 3

METHODOLOGY

In the development of the Payment Gateway with Post-Quantum Cryptographic Signatures, a systematic approach was applied to enhance secure transaction handling while integrating post-quantum cryptographic methods. Below is an outline of the methodology used:

3.1 TECHNOLOGIES USED:

In developing the Payment Gateway with Post-Quantum Cryptographic Signatures, a carefully selected stack of frontend and backend technologies was utilized to ensure secure, efficient, and seamless transaction handling:

a) **Noble Post-Quantum Library**: This library was essential for implementing ML-DSA (Module Lattice - Digital Signature Algorithm), a post-quantum signature scheme. Noble's library provided a robust framework for generating and validating cryptographic signatures that are resistant to quantum attacks, ensuring the long-term security of transaction data.

b) **JavaScript**: JavaScript was used for both frontend and backend development, creating a cohesive environment for consistent data handling and communication between client and server. Its versatility allowed for rapid development, especially given its compatibility with various libraries needed for cryptographic functions.

c) **HTML/CSS**: HTML and CSS formed the foundation of the user interface, enabling the creation of a responsive and user-friendly layout for the payment gateway. HTML structured the frontend elements, while CSS provided visual styling, making the application accessible and intuitive for users.

d) **Express.js**: This lightweight web application framework for Node.js

was used for backend API development. Express.js allows for easy routing and handling of HTTP requests, supporting secure and efficient transaction processing between the frontend client and the server.

3.2 DEVELOPMENT PROCESS:

The development process followed a structured methodology to effectively manage the project:

a) **Requirement Analysis** : Analyzed the project requirements with a focus on secure payment processing and identified post-quantum cryptographic techniques as essential for protection against future quantum computing threats.

b) **Technology Selection** : Selected Noble Post-Quantum Library for implementing the ML-DSA signature scheme to secure transactions against quantum-based attacks. Node.js and Express.js were chosen for backend development due to their scalability and efficient handling of asynchronous operations.

c) **System Design and Architecture** : The Payment Gateway system uses a server-client model, where users enter transaction details on a secure frontend, and the server handles signature generation and validation, ensuring all cryptographic processes are securely processed and verified on the server side, with results then displayed to the user.

d) **Implementation** : Developed the frontend with HTML/CSS and JavaScript, focusing on user-friendly design for easy payment processing. The backend implemented the cryptographic signing and validation processes using ML-DSA, ensuring all transaction data remains secure and authenticated.

e) **Testing and Validation** : Conducted unit and integration tests to verify the functionality and reliability of cryptographic signatures. Simulated various attacks to validate the gateway's resilience and

identified any potential security vulnerabilities in the cryptographic implementation.

CHAPTER 3

DESIGN

4.1 DESIGN:

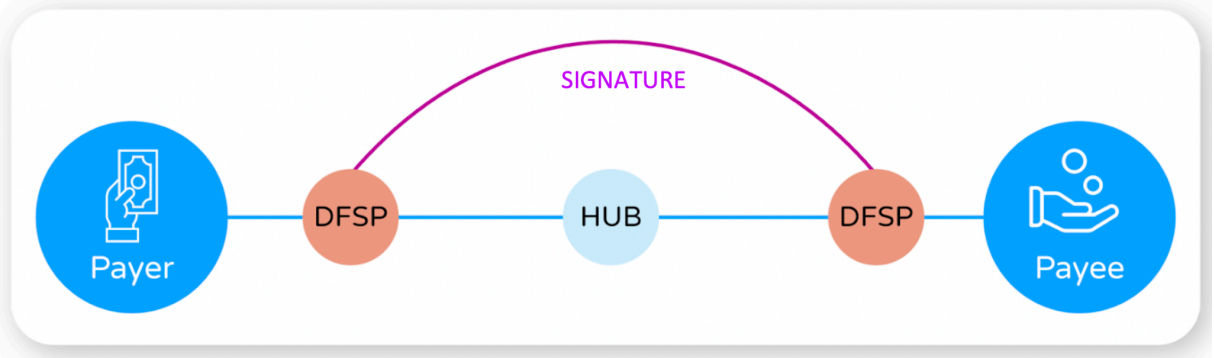


Fig 4.1 Design of Payment Gateway

Figure Overview:

The figure represents a simplified model of a digital payment system involving two Digital Financial Service Providers (DFSPs), a central Hub, and a Payer and Payee.

Key Components:

1. **Payer and Payee:** Individuals or entities initiating and receiving the payment.
2. **Payer DFSP:** The financial institution or service provider associated with the Payer.
3. **Payee DFSP:** The financial institution or service provider associated with the Payee.
4. **Hub:** A central server that facilitates communication and transaction processing between DFSPs.

Transaction Flow

1. **Transaction Initiation**:

i. The Payer initiates a transaction request through the **Payer DFSP**.
ii. The **Payer DFSP** converts the transaction request into a **JavaScript Object** format, preparing it for processing.

2. **Transaction Processing and Handoff to Hub**:
   i. Once the **Payer DFSP** has converted the transaction to a JavaScript object, it forwards this data to the **Hub**.
   ii. The **Hub** receives the transaction data and performs a **digital signature** on it. This signature serves as a cryptographic guarantee of the data's authenticity and integrity.
   iii. The **Hub** also **stores the signature** to allow future validation requests, in case there's a need to verify that the transaction was indeed processed correctly.
3. **Fund Transfer and Signature Validation**:
   i. The **Hub** then routes the transaction to the **Payee DFSP**, which, in turn, delivers the transaction details to the **Payee**.
   ii. If a validation request is made (e.g., if either DFSP wants to verify the transaction), the **Hub** uses the stored signature to confirm the data's authenticity, validating the transaction's integrity and ensuring no tampering has occurred.

4.2 System Architecture

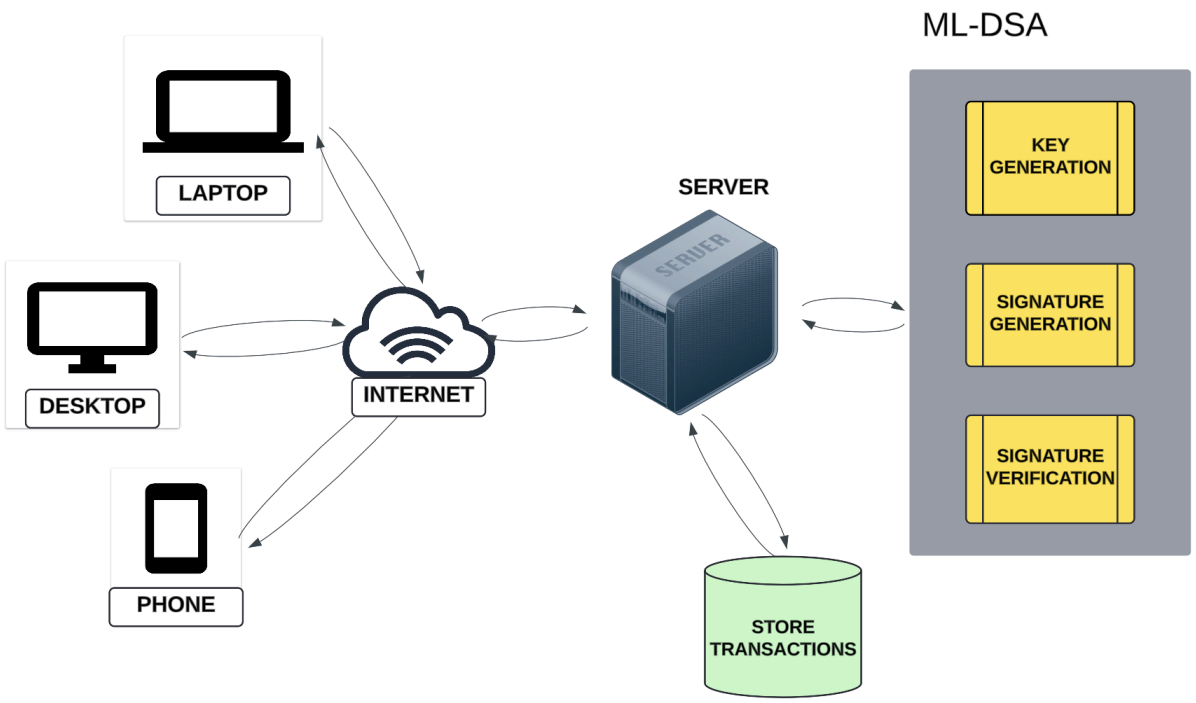


Fig 4.2 System Architecture of Payment Gateway

System Architecture Explanation for Project Report

This system architecture illustrates the integration of the **Module Lattice Digital Signature Algorithm (ML-DSA)** within a secure transaction system. Below is a detailed explanation of the components and their interactions:

1. Client Devices

The architecture supports multiple types of client devices, including laptops, desktop and handheld mobile devices or phones.

These devices represent users accessing the system through various platforms. They initiate transactions or communications over the internet.

2. Internet

The **Internet** serves as the communication channel, enabling client devices to interact with the central server. This ensures remote accessibility of the system, allowing users to submit requests or receive responses in real time.

3. Server

The **Server** is the core component that manages and processes client requests. It performs several key functions:

1. Receives transaction data from client devices.
2. Coordinates with the **ML-DSA module** for cryptographic operations.
3. Stores the processed transactions in a database for record-keeping and analysis.

4. Module Lattice Digital Signature Algorithm (ML-DSA) Module

This module ensures the security and integrity of the system by implementing post-quantum cryptographic algorithms. It is composed of three critical operations:

1. **Key Generation**:
    a. Generates public and private keys based on lattice-based cryptography.
    b. Ensures that keys are resistant to quantum computing attacks.
2. **Signature Generation**:
    a. Uses the private key to sign transaction data or messages.
    b. The signature guarantees the authenticity and integrity of the message.
3. **Signature Verification**:
    a. Verifies the received signatures using the corresponding

public key.

b. Ensures that the message has not been tampered with and originates from a trusted source.

The use of lattice-based cryptography makes the system resistant to attacks from quantum computers, providing robust security for sensitive transactions.

5. Transaction Storage

The **Store Transactions** component represents a database system.

1. It securely stores transaction records for audit, verification, and future reference.
2. The stored data may include user requests, signed transactions, and verification logs.

This ensures accountability and supports data retrieval when needed.

System Workflow

1. **Clients** (e.g., Laptop, Desktop, Phone) submit requests or transactions over the **Internet**.
2. The **Server** receives these requests and forwards them to the **ML-DSA module** for cryptographic processing:
    a. Keys are generated.
    b. Transactions are digitally signed.
    c. Signatures are verified.
3. Processed transaction data is securely stored in the **database**.
4. The server sends appropriate responses back to the clients, ensuring secure communication.

Key Features of the Architecture

1. **Post-Quantum Security**:
   Leveraging lattice-based cryptography ensures that the system remains secure even against future quantum computing threats.
2. **Scalability**:
   The system supports multiple client devices simultaneously, enabling widespread use.
3. **Data Integrity**:
   Digital signatures ensure that all transactions are authentic and tamper-proof.
4. **Reliability**:
   Transaction data is persistently stored, ensuring reliability and traceability.

This system architecture effectively integrates **Module Lattice Digital Signature Algorithm (ML-DSA)** to provide a secure, quantum-resistant framework for handling sensitive transactions. The combination of robust cryptographic mechanisms and efficient data management ensures both security and reliability for users.

4.3 SIGNATURE AND VALIDATION - PROCESS FLOW

Overview

This section outlines the authentication process implemented using the ml-dsa (Module Lattice - Digital Signature Algorithm). The process involves key generation, signature creation, and signature verification to ensure secure communication.

Key Generation

1. **Algorithm Selection:** The ml-dsa algorithm is chosen for its security and efficiency.
2. **Key Pair Generation:**
    a. The ml_dsa65.keygen() function is invoked to generate a public-private key pair.
    b. The private key is securely stored and used for signing requests.
    c. The public key is shared with authorized parties for verification purposes.

Signature Creation

1. **Request Preparation:** The requestOptions are constructed, containing the payload and relevant headers.
2. **Signature Generation:** The ml_dsa65.sign(secretKey, payload) function is called to generate a digital signature using the private key and the request payload.

Signature Verification

1. **Signature Extraction:** The SignatureObject is extracted from the requestOptions, containing the signature and headers.
2. **Signature Validation:** The ml_dsa65.verify(publicKey, payload, sign) function is used to verify the signature using the public key, the original payload, and the received signature.
3. **Header Validation:** The headers within the SignatureObject are validated against expected values to ensure integrity and authenticity.

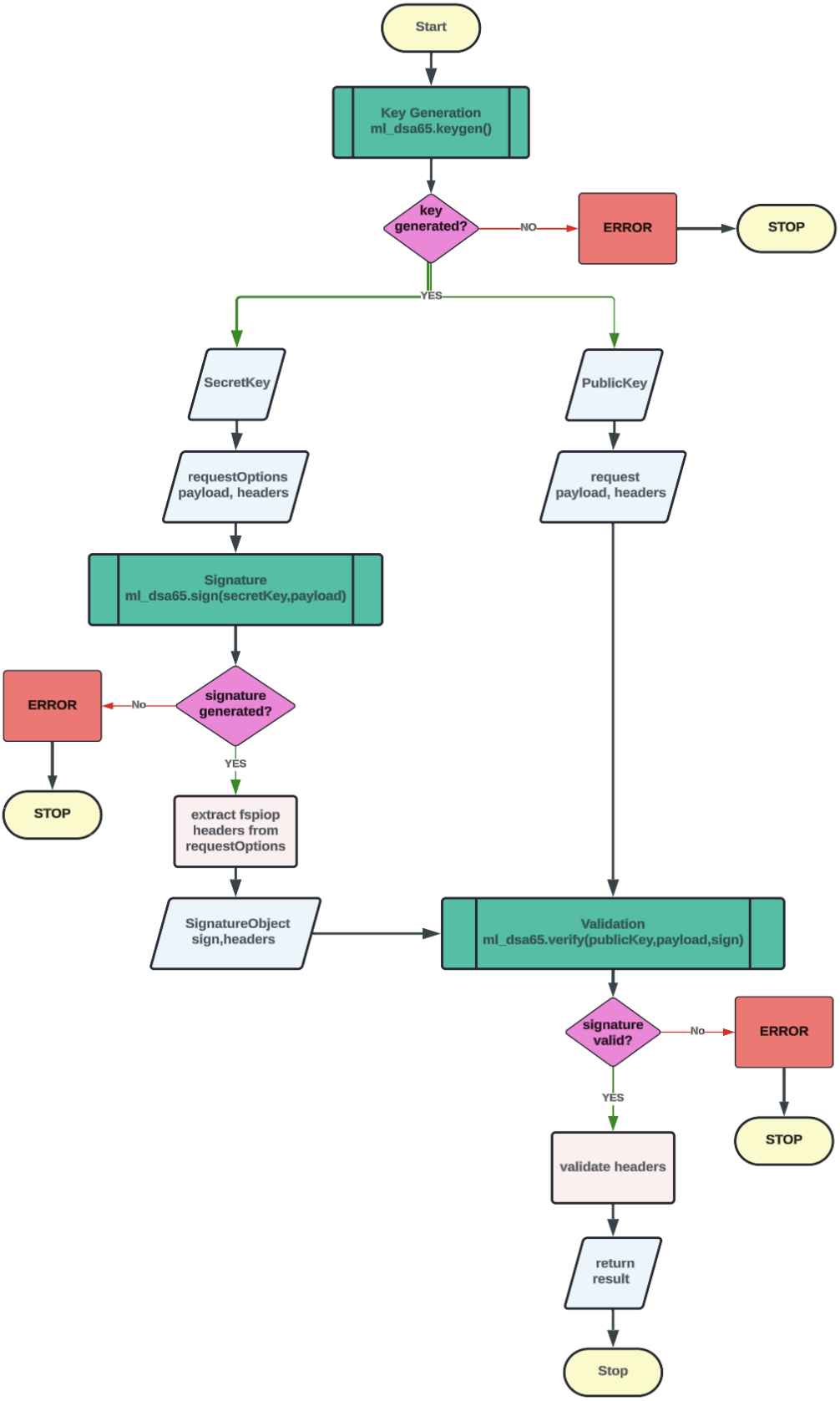


Fig 4.3 Flow Chart of Payment Gateway

4.4 Use Case Diagram

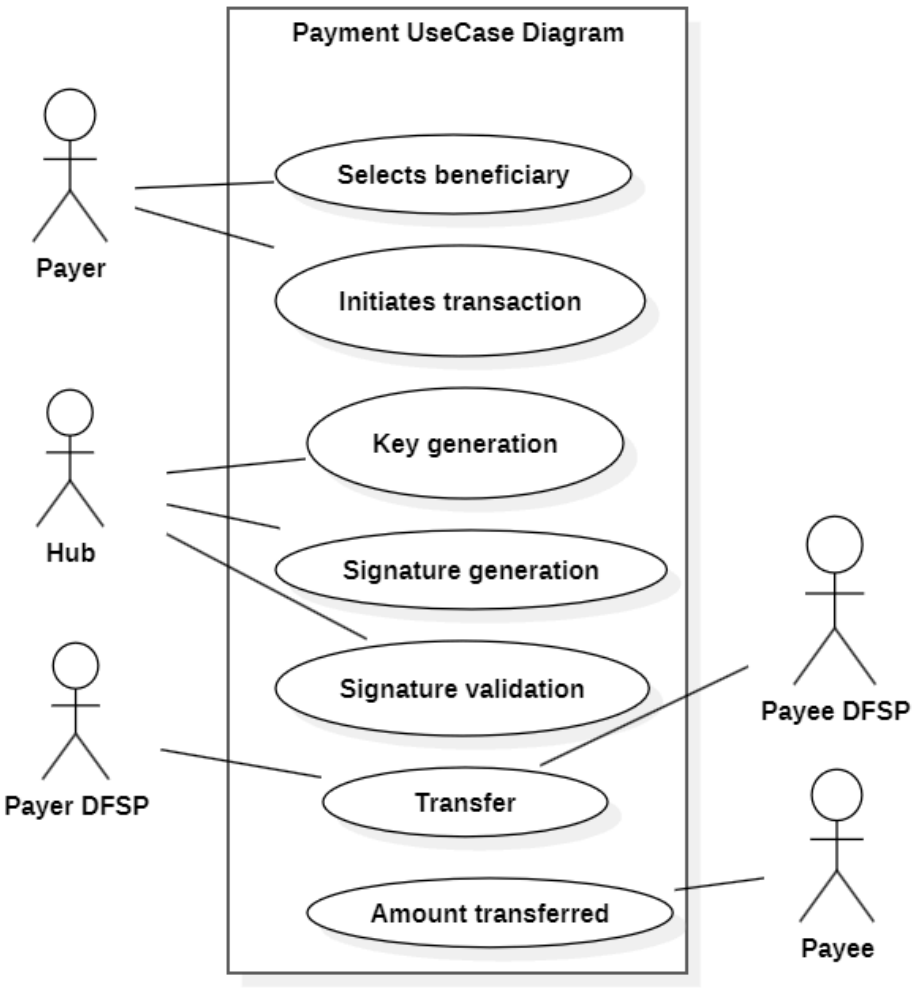


Fig. 4.4 Payment Use Case Diagram

The diagram is a **Payment Use Case Diagram**, showing the interaction between different actors in a payment system and the corresponding use cases.

Actors:

1. **Payer**:
    a. Initiates the payment process.
    b. Interacts with use cases such as selecting a beneficiary and initiating a transaction.

2. **Hub**: Involved in key generation, signature generation, and validation to ensure secure transactions.
3. **Payer DFSP** (Digital Financial Services Provider): Participates in key payment operations such as transfer initiation and validation.
4. **Payee DFSP**: Ensures that the payment details are correctly handled and validated on behalf of the payee.
5. **Payee**: The recipient of the transferred amount.

Use Cases:

1. **Selects Beneficiary**: Payer selects the beneficiary for the payment.
2. **Initiates Transaction**: Payer starts the payment process by providing necessary details.
1. **Key Generation**: Hub generates cryptographic keys for secure transaction processing.
2. **Signature Generation**: Hub or Payer DFSP digitally signs the transaction.
3. **Signature Validation**: Payee DFSP validates the transaction's digital signature to confirm its integrity.
4. **Transfer**: The amount is moved from the payer's account to the payee's account.
5. **Amount Transferred**: Final confirmation that the payment has been successfully completed, and the funds are available to the payee.

This diagram outlines the secure flow of a financial transaction, emphasizing the roles of key actors in generating, validating, and completing payments. It highlights the critical cryptographic processes for ensuring security and accuracy in digital payments.

4.5 State Chart Diagram

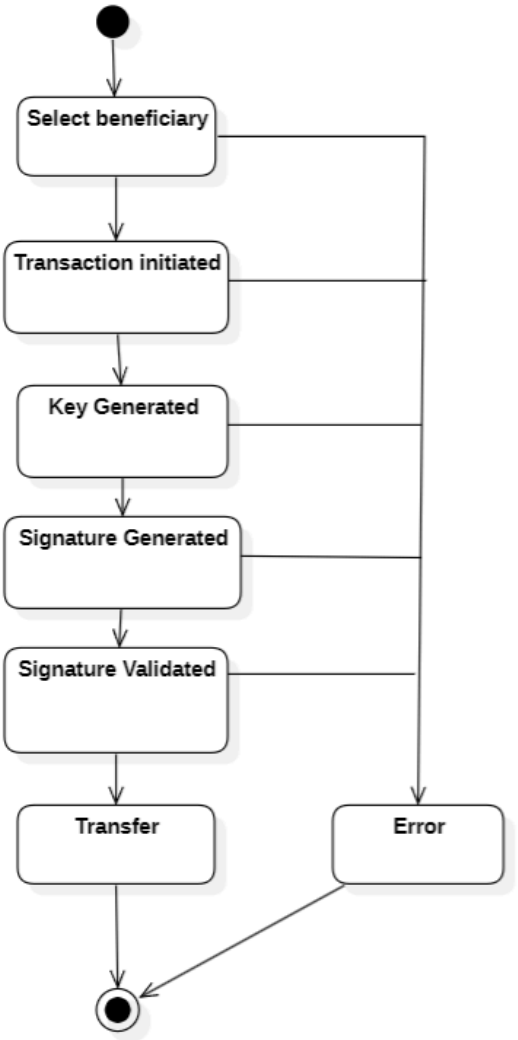


Fig. 4.5 State Chart Diagram

States:

1. **Initial State** (Black Dot): Represents the starting point of the process.
2. **Select Beneficiary**: The user (payer) selects the beneficiary for the transaction.
3. **Transaction Initiated**: The transaction is started after the beneficiary is selected, and payment details are provided.
4. **Key Generated**: Cryptographic keys are generated for securing the transaction.

5. **Signature Generated**: A digital signature is created to authenticate the transaction.
6. **Signature Validated**: The system validates the generated digital signature to ensure the integrity and authenticity of the transaction.
7. **Transfer**: The funds are transferred from the payer to the payee upon successful validation.
8. **Error**: Represents a state where the process fails at any point due to issues like invalid signature or insufficient funds.
9. **Final State** (Black Dot with Circle): Indicates the successful completion of the payment process.

CHAPTER 5

IMPLEMENTATION

The project uses the @noble/post-quantum library for ML-DSA operations. It employs a basic server-client model, where the hub serves as the server and the payer and the payee serve as clients. Below is a basic implementation showcasing key functionalities:

Code Implementation:

javascript

```
// Install the library
npm install @noble/post-quantum

// Importing ML-DSA
import { ml_dsa65 } from '@noble/post-quantum/ml-dsa';

// Key generation
const seed = new TextEncoder().encode('not a safe seed');
const aliceKeys = ml_dsa65.keygen(seed);

// Message and signing
const msg = new Uint8Array(1);
const sig = ml_dsa65.sign(aliceKeys.secretKey, msg);

// Verification
const isValid = ml_dsa65.verify(aliceKeys.publicKey, msg, sig);

console.log('Signature valid:', isValid);
```

This implementation highlights the core features of ML-DSA within a server-client architecture. The server acts as a trusted intermediary, ensuring the authenticity and integrity of financial transactions. This approach demonstrates how quantum-resistant cryptographic algorithms can secure financial systems in a post-quantum era.

CHAPTER 6

TESTING

S.no.	Description	Input	Expected Outcome	Real Outcome	Result	Reason
1.	Key Generation	-	Public and Secret Keys	Error	Fail	No input seed given
2.	Key Generation	Random seed	Public and Secret Keys	Public and Secret Keys	Pass	
3.	Signature Generation	private key, transaction data	signature	error	Fail	key format disturbed while exchanging keys between different programs
4.	Signature Generation	private key, transaction data	signature	signature	Pass	
5.	Signature Verification	public key, transaction data, signature	true	false	Fail	Transaction data format changed while fetching information from client
6.	Signature Verification	public key, transaction data, signature	true	false	Fail	headers are different
7.	Signature Verification	public key, transaction data, signature	true	true	Pass	
8.	Make Payment	beneficiary, payer acc. no., payee acc. no., amount	signature	error	Fail	beneficiary acc. no. and payee acc. no. does not match in server

9.	Make Payment	beneficiary, payer acc. no., payee acc. no., amount	signature	signature	Pass	Auto fills payee acc. no. from beneficiary

Table 6.1 Test Cases

The testing process incorporated both **black-box** and **white-box** methodologies to ensure the system's functionality and reliability. Black-box testing focused on validating the system's behavior by providing inputs such as random seeds, transaction data, and account details, and checking the expected outcomes. For instance, cases like key generation and making payments revealed issues such as missing input seeds or mismatched account numbers, which were addressed to ensure the correct operation of the system.

White-box testing was applied to examine internal processes and logic, particularly in modules like signature generation and verification. It helped identify critical issues such as disturbed key formats during program exchanges and inconsistencies in transaction data formats and headers. These insights enabled targeted debugging and improvements. The combination of both testing methods ensured a thorough evaluation of the system, ultimately achieving successful outcomes for critical operations.

CHAPTER 7

RESULTS

7.1 OUTPUTS

1. **Fund Transfer Page :** Start by opening the find transfer page. User is logged in by default by the account of Jax Smith. Payer account number is auto filled.

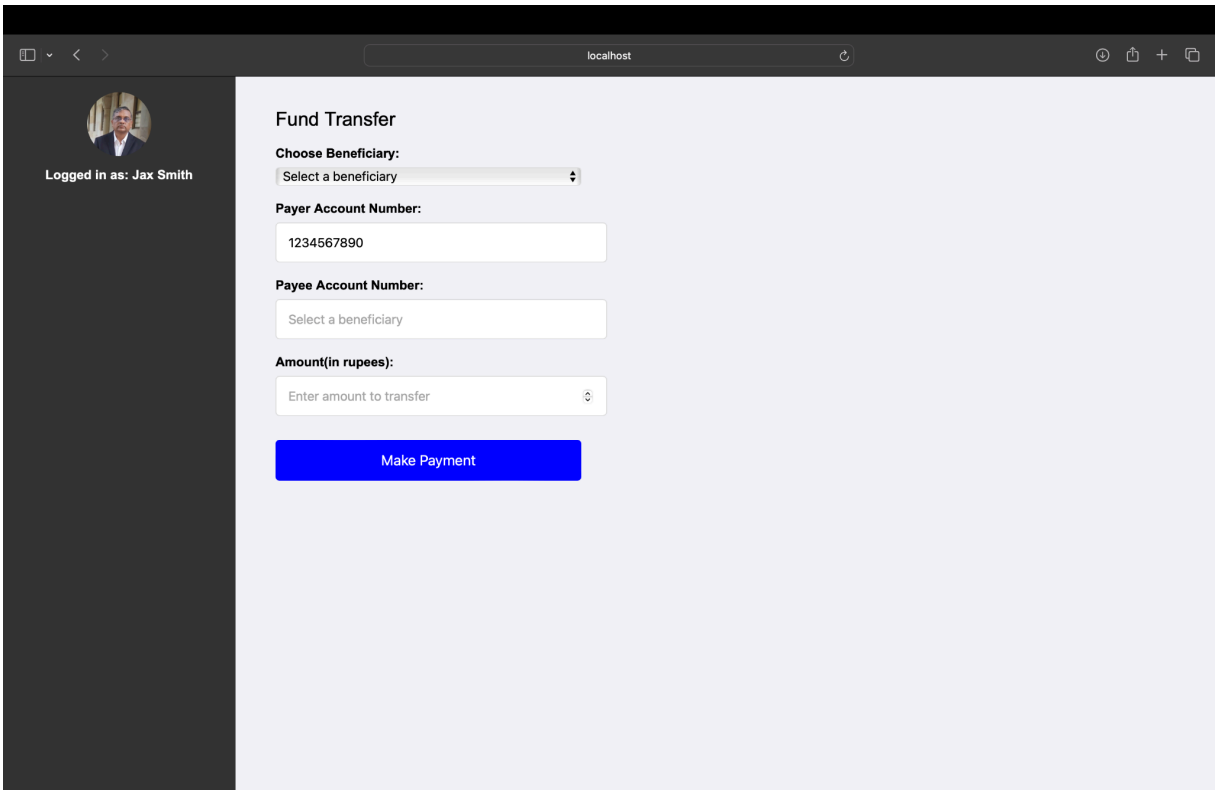


Fig. 7.1 Fund Transfer Page

2. **Choose a Beneficiary:** Choose a beneficiary to make payment from the already added beneficiaries. Once the beneficiary is chosen, the payee account number is auto-filled.

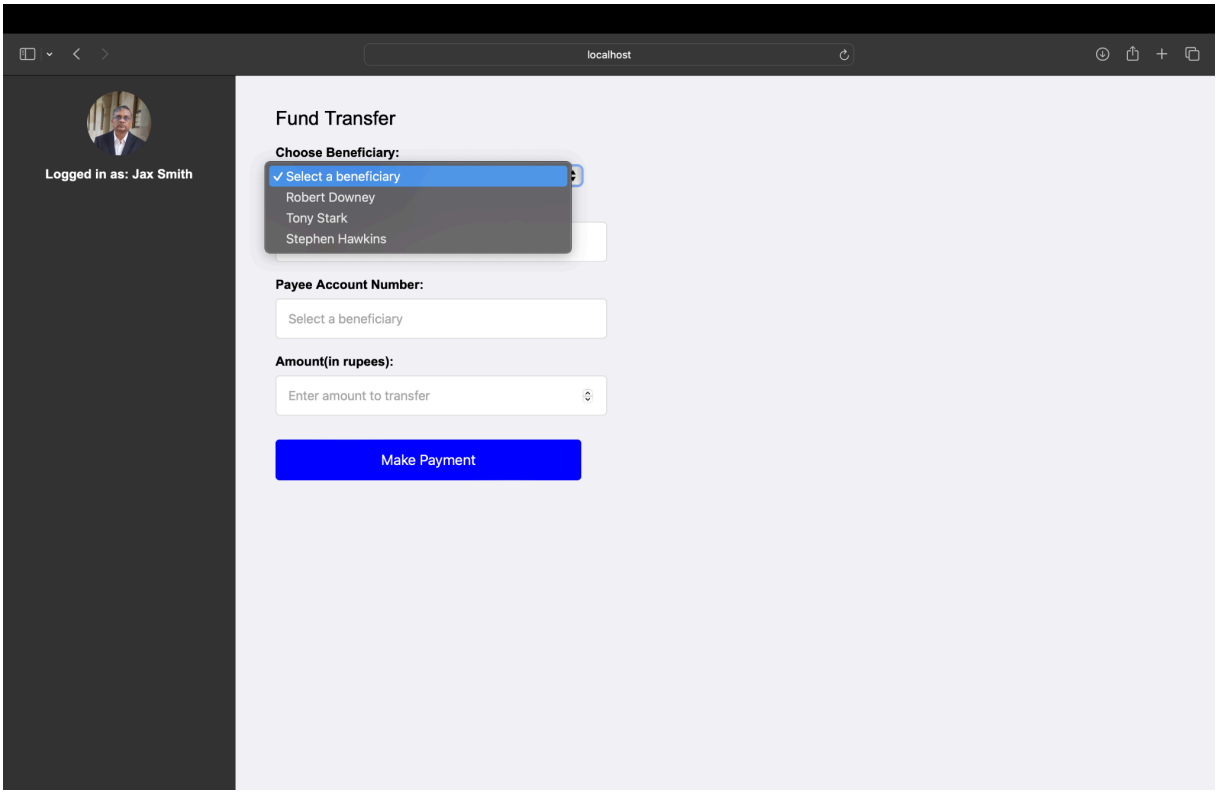


Fig. 7.2 Select Beneficiary

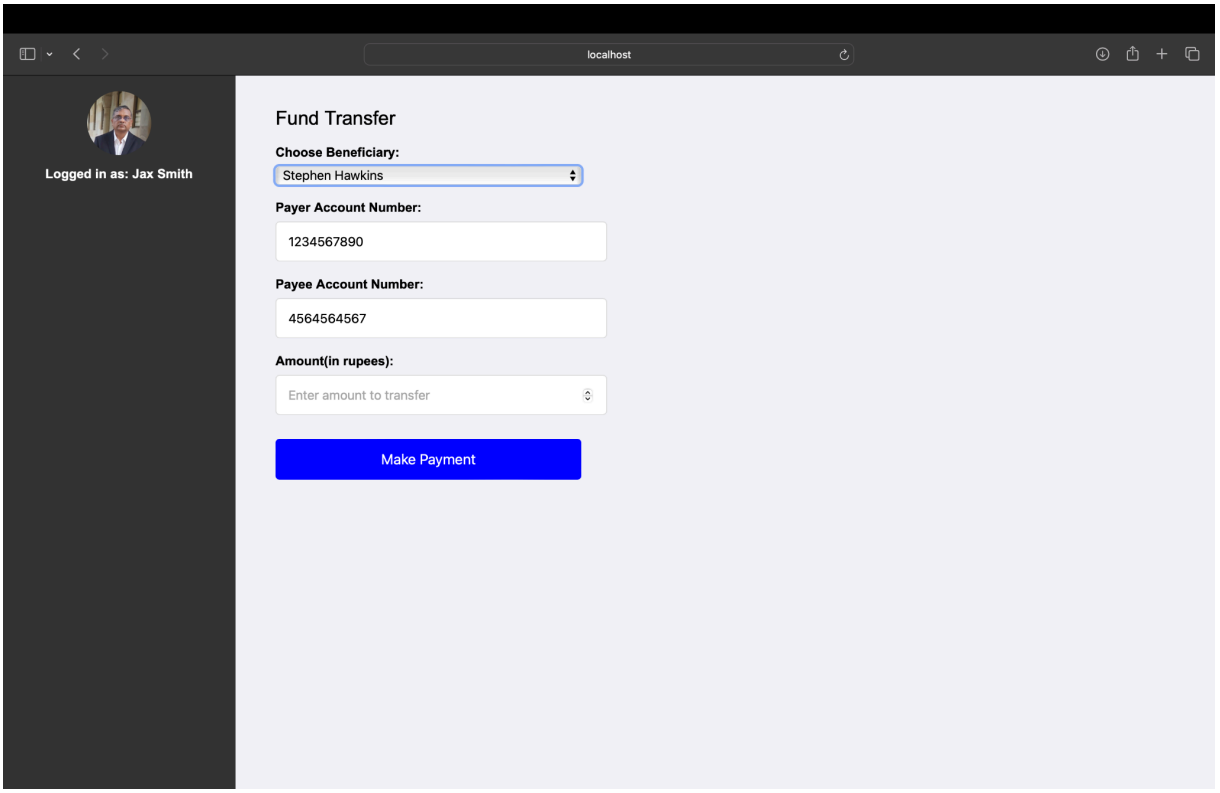


Fig. 7.3 Selected Beneficiary

3. **Enter Amount :** Only positive numbers greater or equal to one are allowed to be taken up as the amount for transfer.

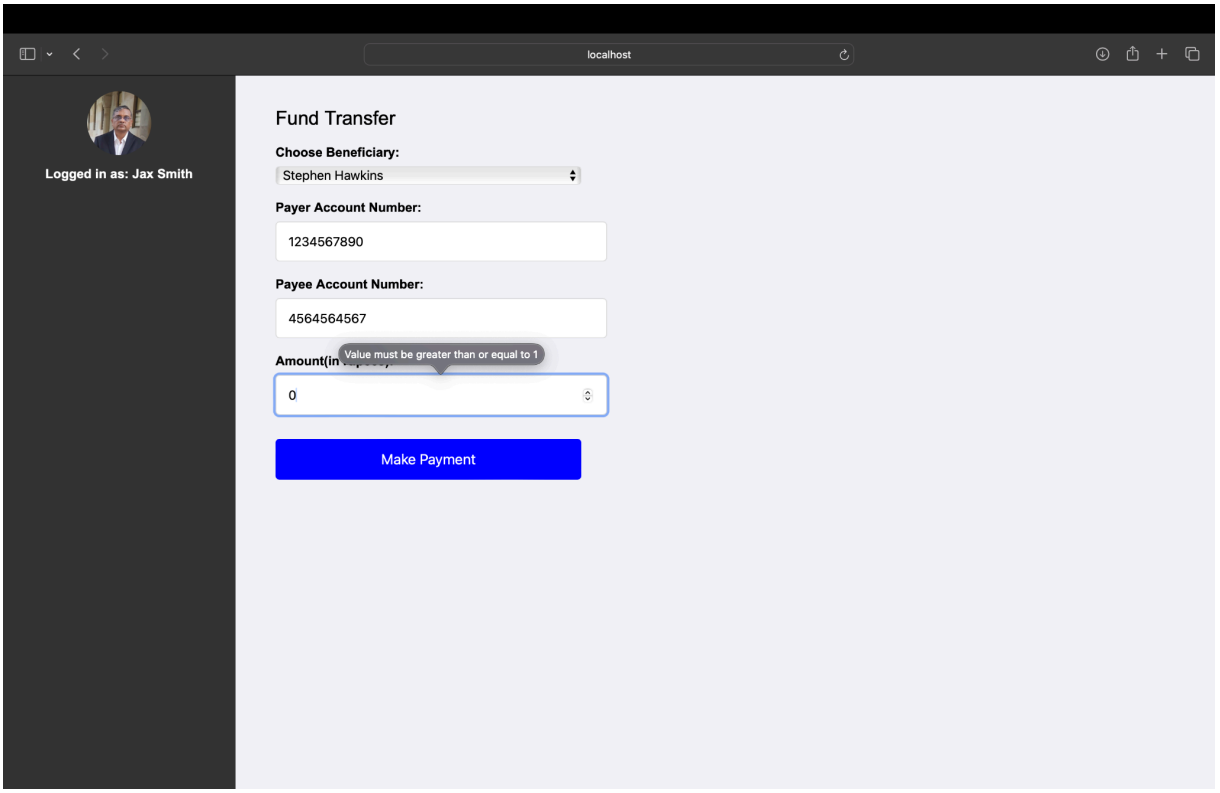


Fig. 7.4 Payee and Payer Account Details

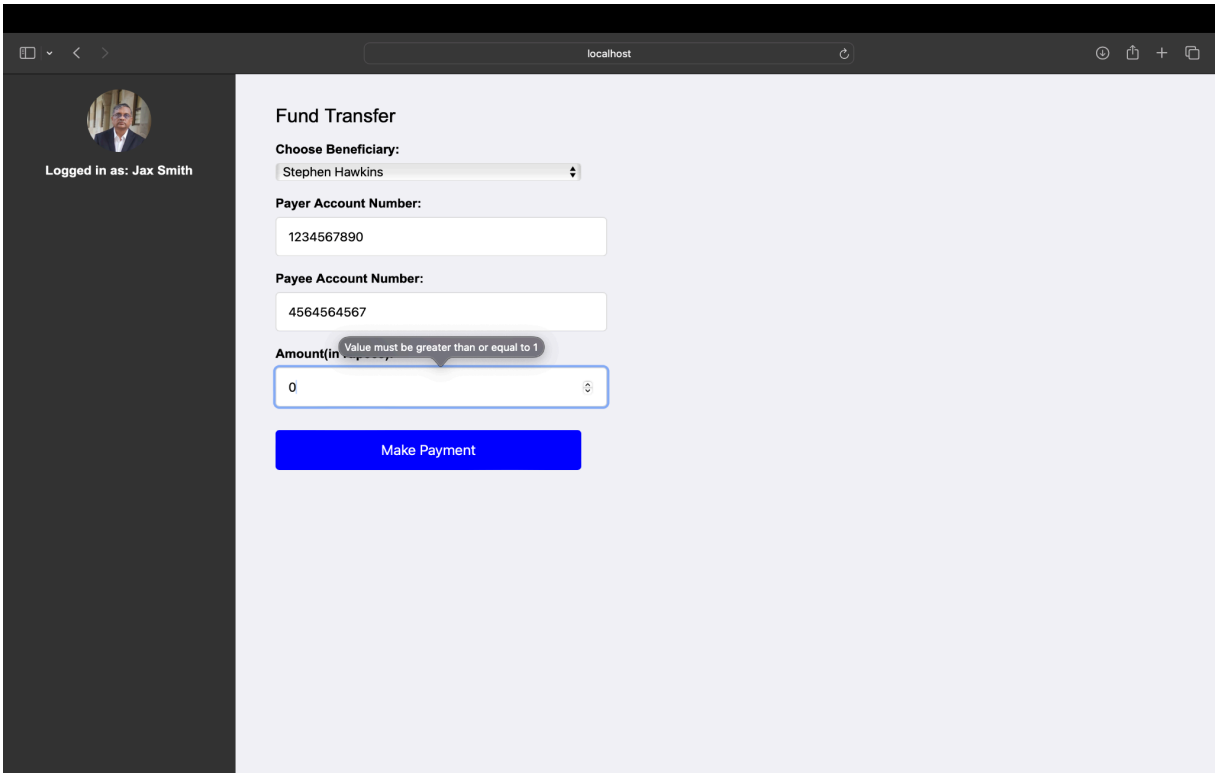


Fig. 7.5 Select the Payment Amount

4. **Make Payment and Generate Signature :** After all the fields are filled correctly, the payment is made, the user input is converted into a javascript object and signature is generated over that object. If the signature is generated successfully, a pop up message is displayed and then the signature generated is displayed on the screen. A validation button will also appear to validate the signature generated on the server side.

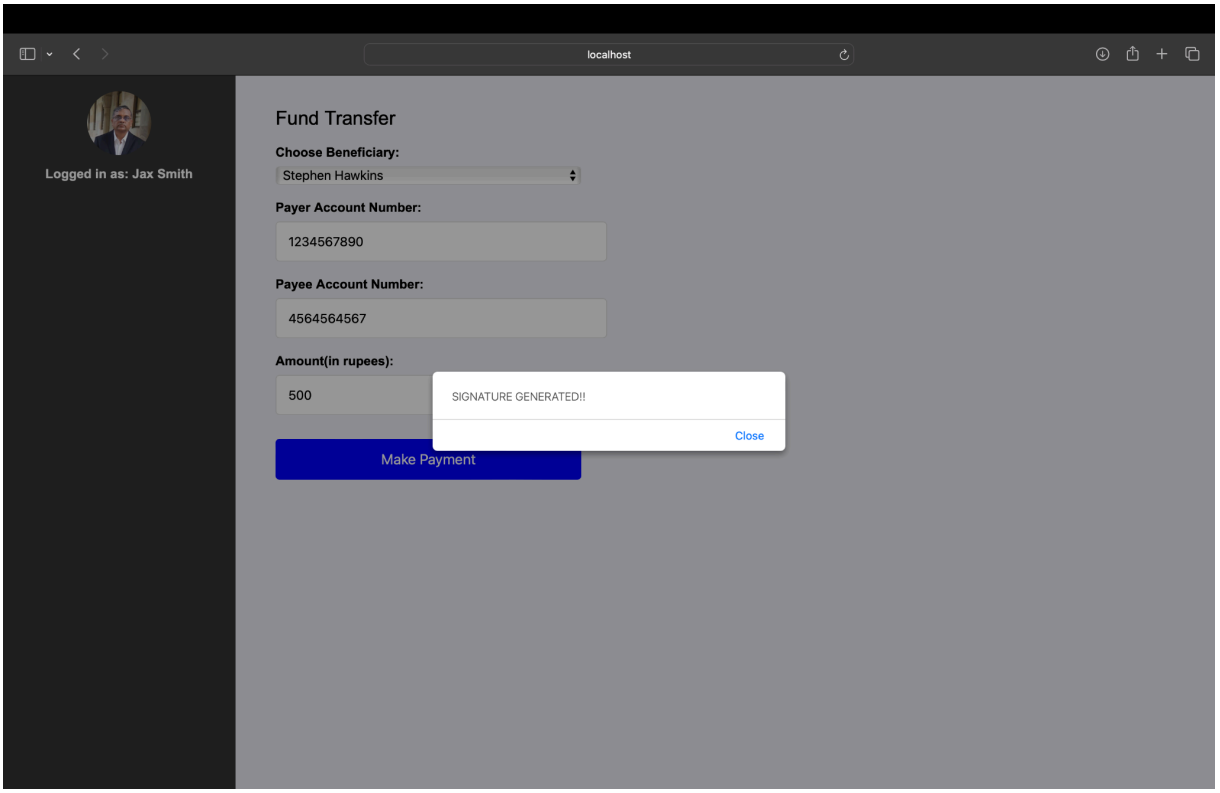


Fig. 7.6 Make the Payment

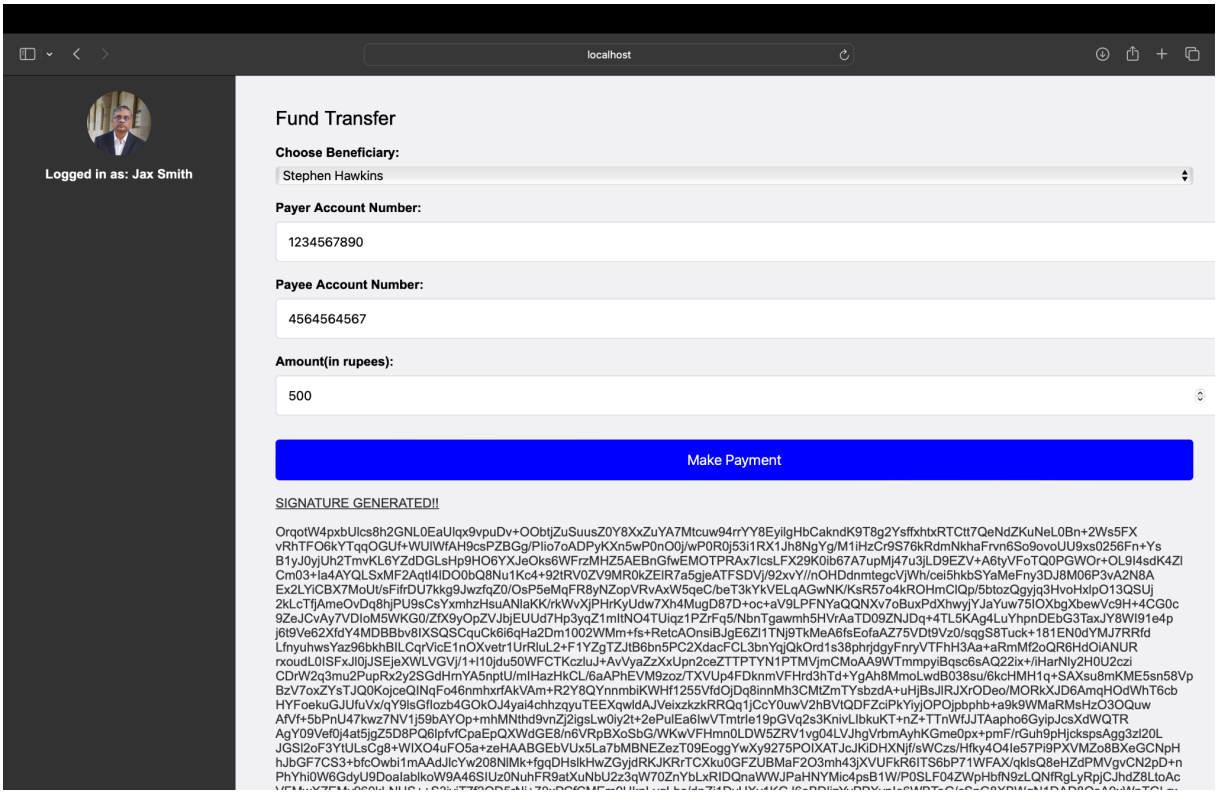


Fig. 7.7 Generate the Signature

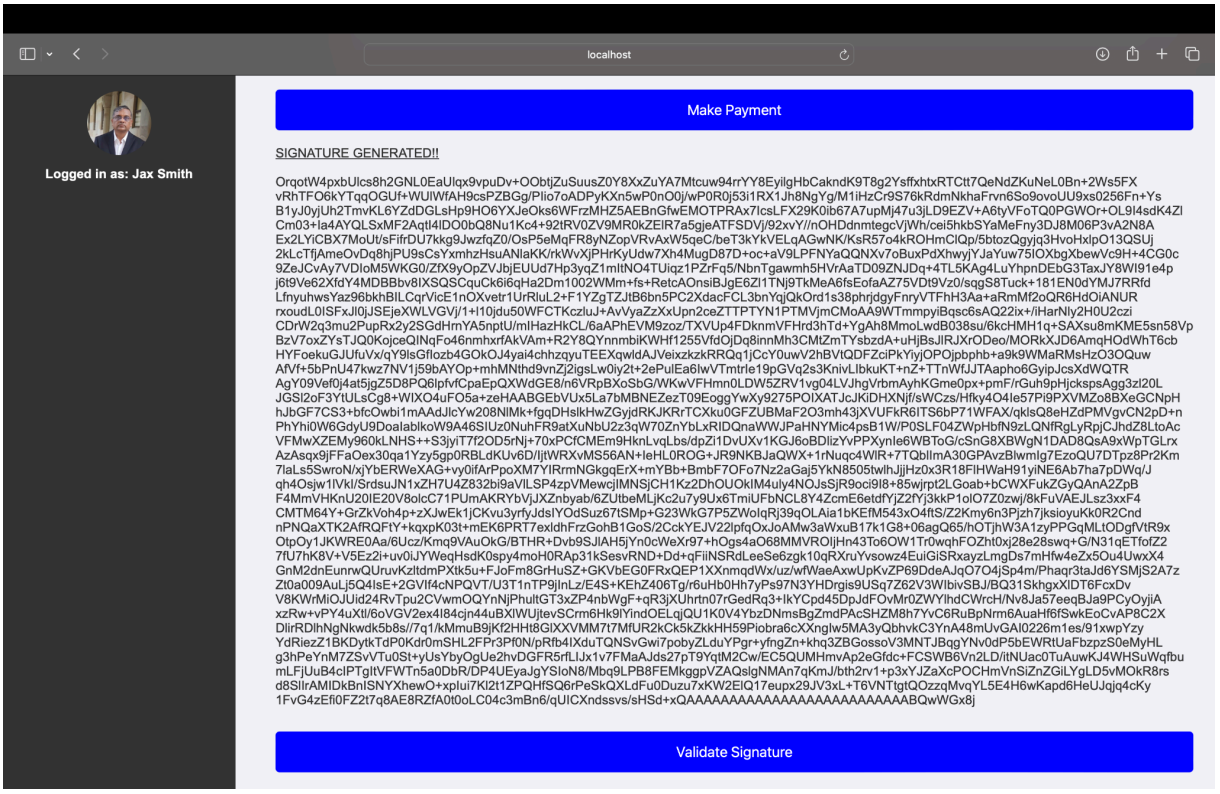


Fig. 7.8 Request Signature Validation

5. **Validation of Signature :** On clicking the Validate Signature button, the signature is validated on the server side.

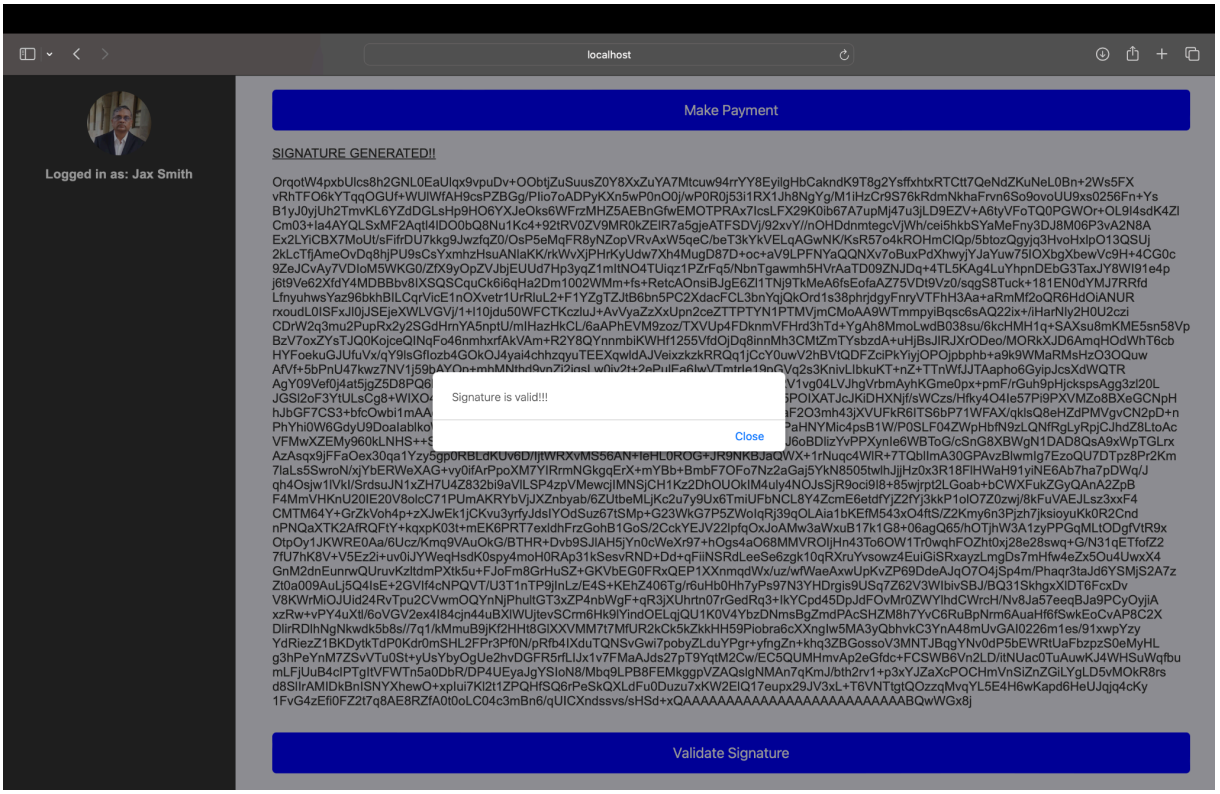


Fig. 7.9 Validation of Signature

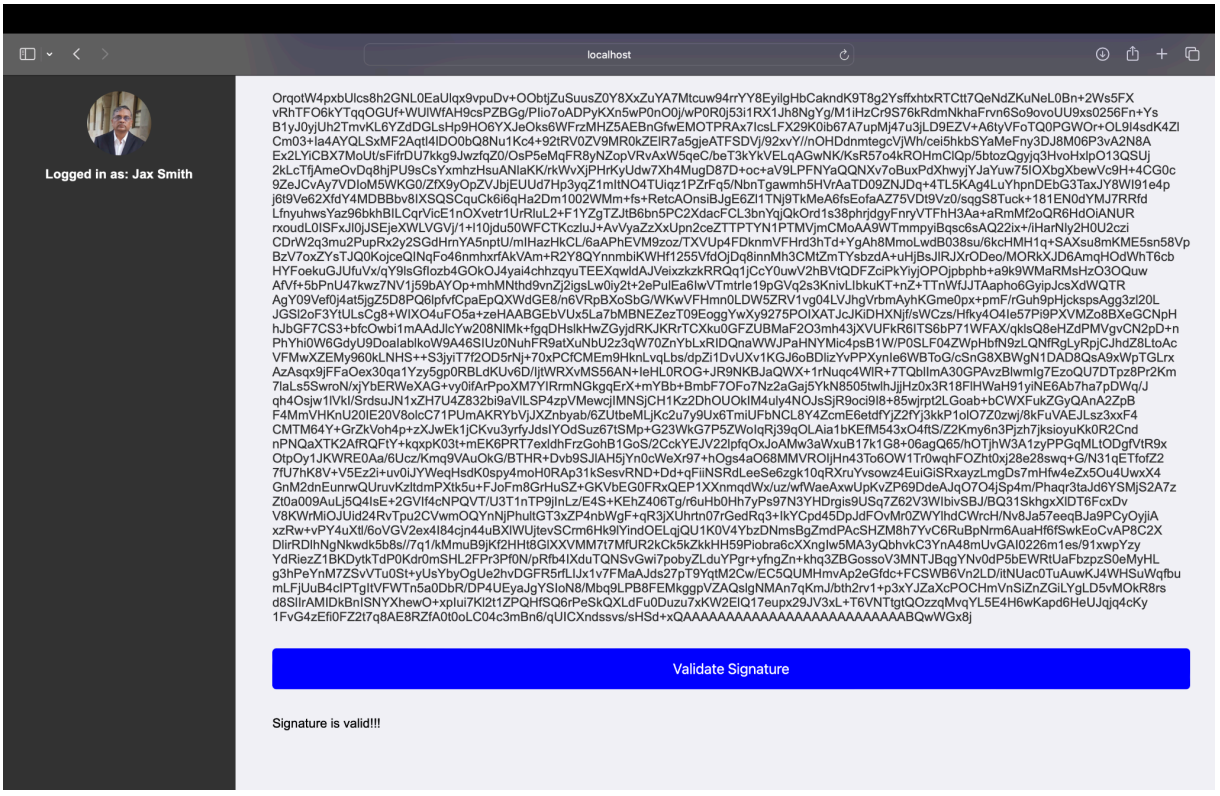


Fig. 7.10 Signature is verified at server

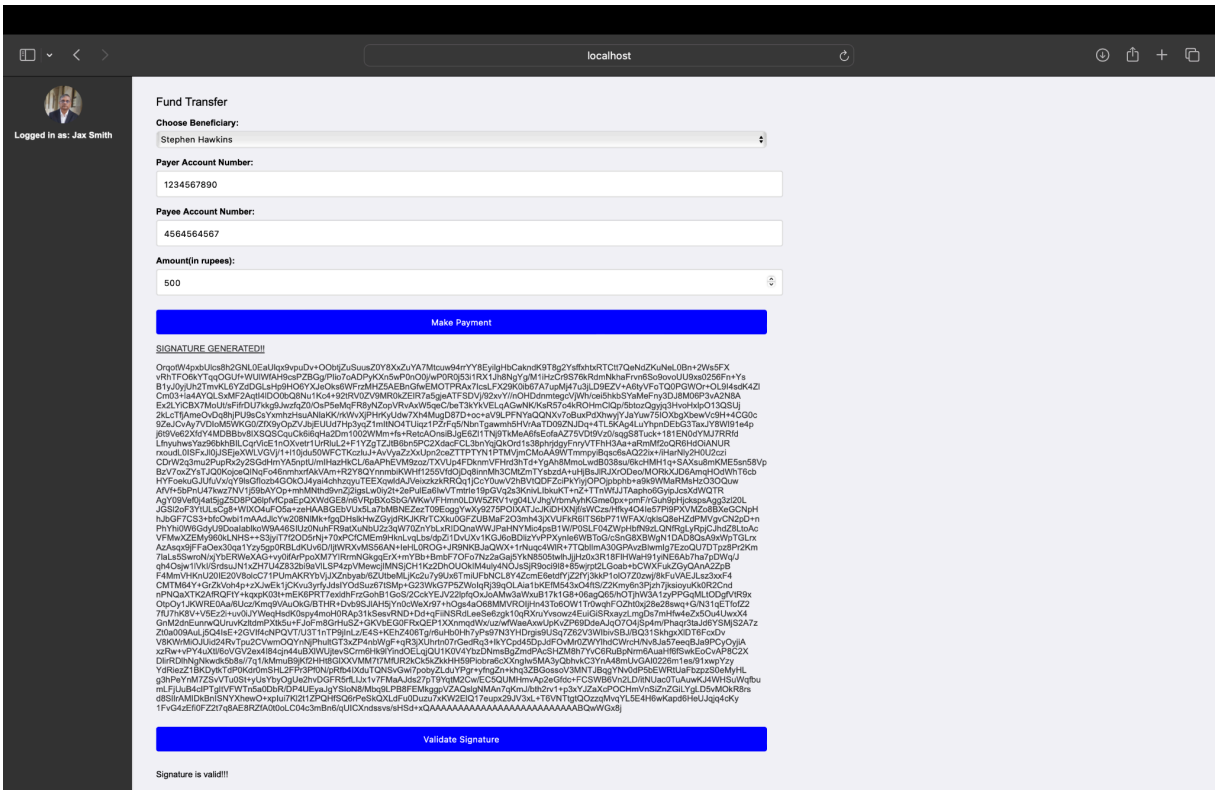


Fig. 7.11 Overall picture of the transaction

7.2 Comparing performance of different algorithms

The table outlines key characteristics of several cryptographic algorithms, including traditional methods like RSA and ECC, as well as post-quantum alternatives like ML-DSA, ML-KEM, and SLH-DSA. Each algorithm differs in its speed, key size, signature size, time of creation, and resistance to quantum attacks, which is increasingly vital as quantum computing advances.

RSA (Rivest-Shamir-Adleman) is one of the oldest and most widely used cryptographic algorithms. Developed in the 1970s and popularized in the 1990s, RSA offers a balance of security and performance for its time. However, its relatively large key and signature sizes, ranging from 256 bytes to 2 kilobytes, make it less efficient than modern alternatives. More importantly, RSA is not secure in a post-quantum world, as it is vulnerable to quantum algorithms like Shor's algorithm, which can break its underlying mathematical structure.

Elliptic Curve Cryptography (ECC), developed in the 1980s, became prominent in the 2010s due to its efficiency. ECC provides strong security with significantly smaller key sizes—ranging from 32 bytes to 256 bytes—compared to RSA. This compactness also extends to its signature sizes, which range from 48 bytes to 128 bytes, making ECC ideal for resource-constrained environments such as mobile devices. However, like RSA, ECC is not resistant to quantum attacks, posing a risk in the era of quantum computing.

In contrast, post-quantum cryptography introduces algorithms like ML-KEM (Module Lattice Key Encapsulation Mechanism) and ML-DSA (Module Lattice Digital Signature Algorithm), which are designed to resist quantum attacks. ML-KEM is particularly fast and efficient for key encapsulation, with key sizes ranging from 1.6 kilobytes to 31 kilobytes and a signature size of 1 kilobyte. This makes it an attractive option for

secure key exchange in post-quantum environments.

ML-DSA, the focus of your project, offers a balanced performance, with key sizes between 1.3 kilobytes and 2.5 kilobytes and signature sizes from 2.5 kilobytes to 4.5 kilobytes. Although not as fast as ML-KEM, ML-DSA provides robust security for digital signatures, making it ideal for applications requiring authentication and non-repudiation. Created in the 1990s and gaining traction in the 2020s, ML-DSA is built on lattice-based cryptography, providing strong resistance to quantum attacks.

SLH-DSA (Stateless Hash-Based Digital Signature Algorithm) represents a different trade-off. It leverages the security of hash functions in a stateless manner, providing post-quantum security with extremely small key sizes, ranging from 32 bytes to 128 bytes. However, this comes at the cost of very large signature sizes, from 17 kilobytes to 50 kilobytes, which can impact performance and storage. Despite its slower speed, SLH-DSA is an attractive choice in scenarios where key size minimization is critical, and quantum security is a must.

Overall, the evolution from RSA and ECC to ML-based and hash-based algorithms like SLH-DSA highlights the shift towards cryptographic solutions that can withstand the computational power of quantum computers. ML-DSA and SLH-DSA each offer unique advantages, ensuring their relevance in a post-quantum future.

	Speed	Key size	Sig size	Created in	Popularized in	Post-quantum?
RSA	Normal	256B - 2KB	256B - 2KB	1970s	1990s	No
ECC	Normal	32 - 256B	48 - 128B	1980s	2010s	No
ML-KEM	Fast	1.6 - 31KB	1KB	1990s	2020s	Yes
ML-DSA	Normal	1.3 - 2.5KB	2.5 - 4.5KB	1990s	2020s	Yes
SLH-DSA	Slow	32 - 128B	17 - 50KB	1970s	2020s	Yes

Table 7.1 Comparing performance of Different Algorithms

It is suggested to use ECC + ML-KEM for key agreement, SLH-DSA for signatures.

ML-KEM and ML-DSA are lattice-based, so they're less "proven". There's some chance of advancement, which will break this algorithm class. SLH-DSA, while being slow, is built on top of older, conservative primitives.

Symmetrical algorithms like AES and ChaCha (available in noble-ciphers)

suffer less from quantum computers. For AES, simply update from AES-128 to AES-256.

Security: The library has not been independently audited yet. There is no protection against side-channel attacks.

Speed: Noble is the fastest JS implementation of post-quantum algorithms. WASM libraries can be faster. For SLH-DSA, SHAKE slows everything down 8x, and -s versions do another 20-50x slowdown.

OPs/sec	Keygen	Signing	Verification	Shared secret
ECC ed25519	10270	5110	1050	1470
ML-KEM-768	2300			2000
ML-DSA44	670	120	620	
SLH-DSA-SHA2-128f	250	10	167	

Table 7.2 Comparing Speed of different Algorithms

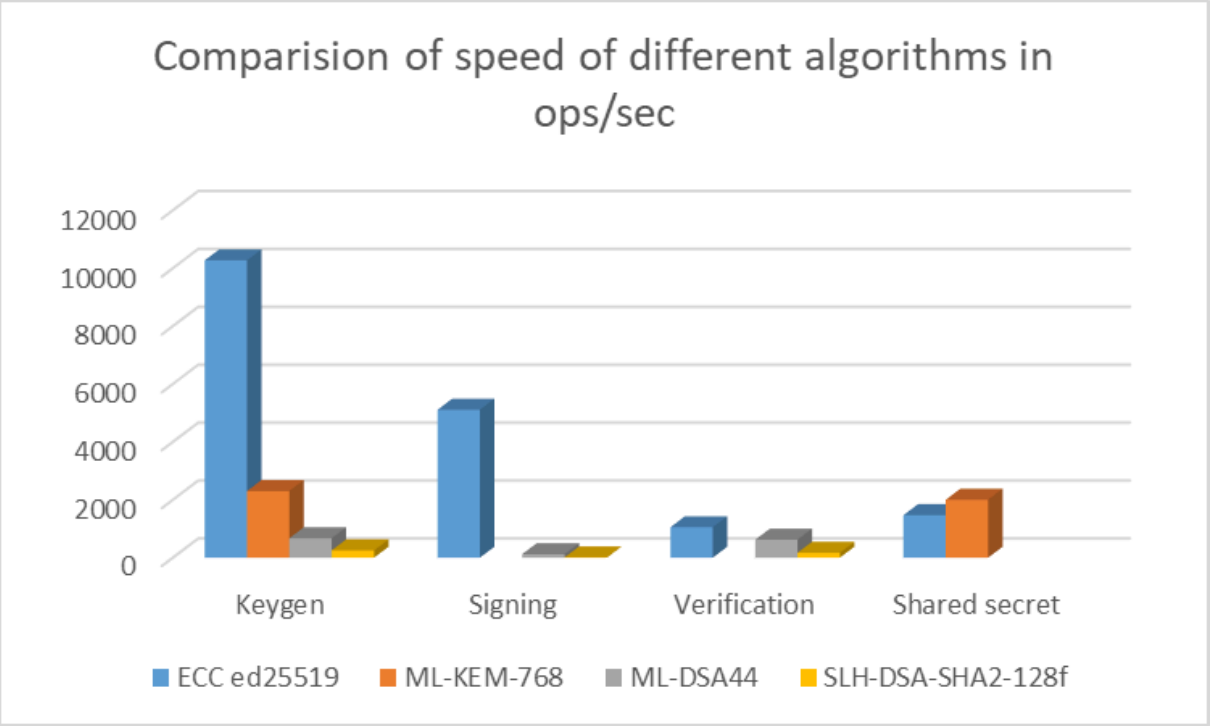


Fig. 7.12 Comparing Speeds of different Algorithms

Detailed benchmarks on Apple M2:

keygen

├─ML-DSA44 x 669 ops/sec @ 1ms/op

├─ML-DSA65 x 386 ops/sec @ 2ms/op

└─ML-DSA87 x 236 ops/sec @ 4ms/op

sign

├─ML-DSA44 x 123 ops/sec @ 8ms/op

├─ML-DSA65 x 120 ops/sec @ 8ms/op

└─ML-DSA87 x 78 ops/sec @ 12ms/op

verify

├─ML-DSA44 x 618 ops/sec @ 1ms/op

├─ML-DSA65 x 367 ops/sec @ 2ms/op

└─ML-DSA87 x 220 ops/sec @ 4ms/op

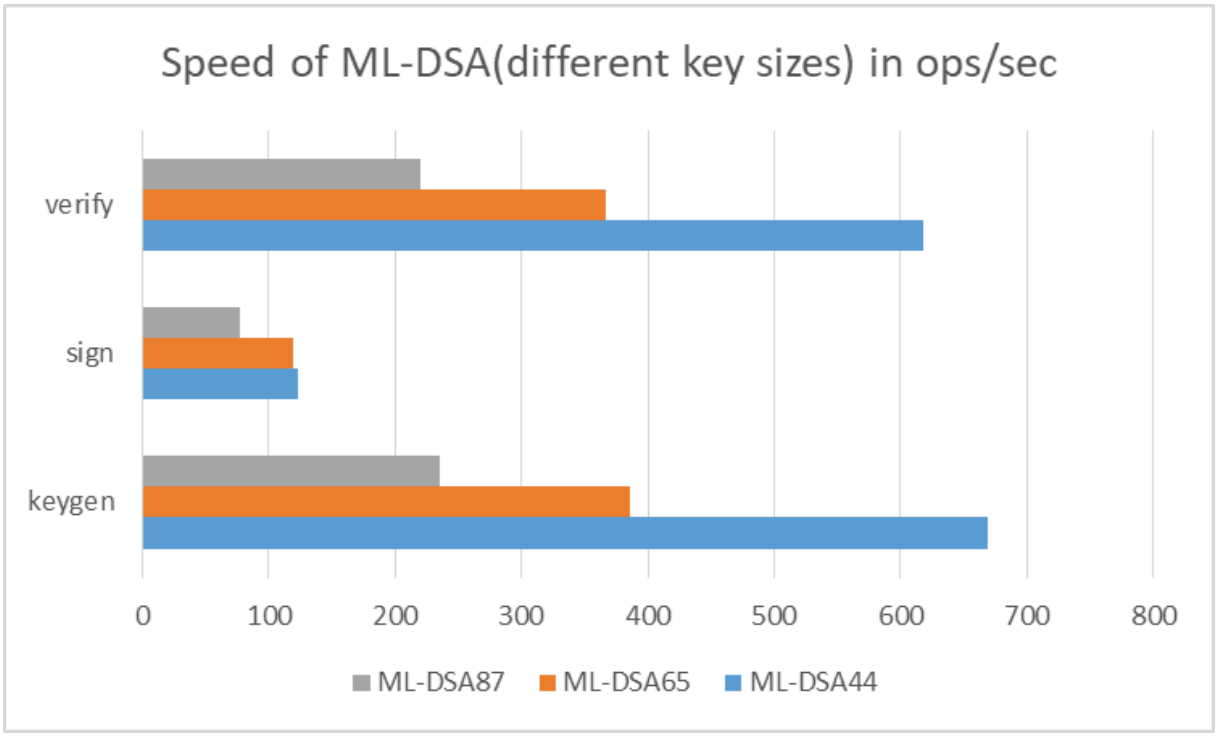


Fig. 7.13 Speed of ML-DSA(different key sizes) in ops/sec

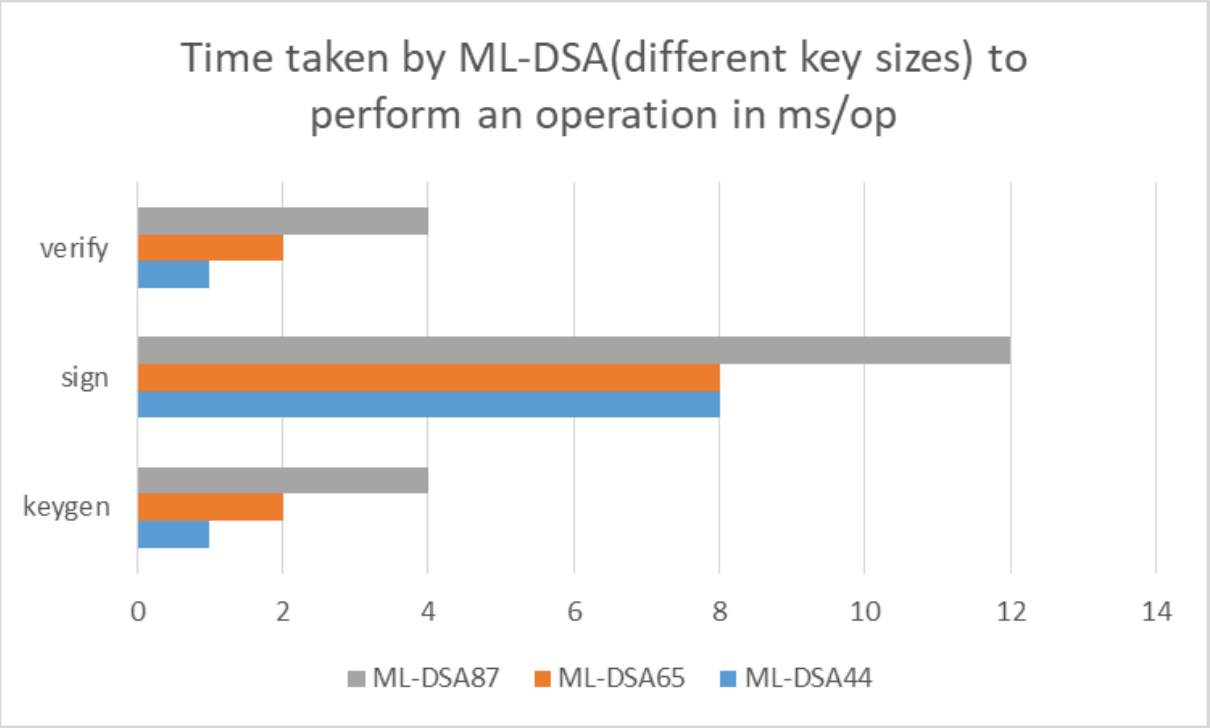


Fig. 7.14 Time taken by ML-DSA(different key sizes) to perform an operation in ms/op

CHAPTER 8

CONCLUSION AND FUTURE SCOPE

8.1 CONCLUSION

With the advancement of quantum computing, the vulnerabilities of traditional cryptographic algorithms to quantum-based attacks are becoming increasingly evident. Algorithms like RSA and ECC, which form the foundation of modern digital security, are at risk of being rendered obsolete due to the ability of quantum computers to solve complex problems such as integer factorization and discrete logarithms. This poses a critical threat to the security of digital transactions and payment platforms, potentially leading to data breaches and financial fraud.

However, the implementation of post-quantum cryptographic solutions, such as ML-DSA, addresses these vulnerabilities by providing quantum-resistant security. Through the development of a secure server-client model leveraging ML-DSA, this project demonstrates the feasibility of protecting financial transactions against quantum threats. The problem definition has been met, ensuring authentication, data integrity, and confidentiality in a quantum-capable world. This marks a significant step towards safeguarding modern financial systems for the future.

8.2 FUTURE SCOPE

As the fintech landscape continues to evolve, the incorporation of post-quantum cryptography will be vital in maintaining user trust and regulatory compliance. Future work may include:

i. **Scalability Improvements:** Further optimization of the cryptographic algorithms to ensure they can handle increased transaction volumes without compromising performance.
ii. **Integration with Existing Systems:** Exploring how these cryptographic methods can be seamlessly integrated with legacy

systems and current financial infrastructure to enhance overall security.

iii. **User Education and Awareness:** Developing educational initiatives to inform users about the importance of post-quantum security measures and how they protect their financial assets.

iv. **Research on New Algorithms:** Continuous evaluation and potential adoption of emerging post-quantum algorithms as the field evolves, ensuring that the application remains at the forefront of security advancements.

v. **Collaboration with Regulatory Bodies:** Working with regulatory authorities to establish standards and guidelines that incorporate post-quantum cryptography, ensuring compliance and promoting widespread adoption across the fintech industry.

These initiatives will not only fortify the application against future threats but also contribute to the broader adoption of post-quantum security measures in financial technology.

BIBLIOGRAPHY

[1] Barker, E. (2006). *Recommendation for Obtaining Assurances for Digital Signature Applications*. NIST Special Publication. https://doi.org/10.6028/NIST.SP.800-89

[2] Micciancio, D., & Regev, O. (2009). *Lattice-Based Cryptography*. NYU. https://cims.nyu.edu/~regev/papers/pqc.pdf

[3] Kaur, R., & Kaur, A. (2012). *Digital Signature*. IEEE Xplore. https://ieeexplore.ieee.org/document/6391693

[4] Langlois, A., & Stehlé, D. (2014). *Worst-case to Average-case Reductions for Module Lattices*. Springer, Journal of Cryptology. https://doi.org/10.1007/s10623-014-9938-4

[5] Mohammed, A., & Varol, N. (2019). *A Review Paper on Cryptography*. ResearchGate. https://www.researchgate.net/publication/334418542_A_Review_Paper_on_Cryptography

[6] Kumar, M., & Pattnaik, P. (2020). *Post Quantum Cryptography (PQC) - An Overview: (Invited Paper)*. IEEE Xplore. https://ieeexplore.ieee.org/document/9286147

[7] Liestyowati, D. (2020). *Public Key Cryptography*. IOP Science. https://iopscience.iop.org/article/10.1088/1742-6596/1477/5/052062/pdf

[8] Barker, E. (2020). *Guideline for Using Cryptographic Standards in the Federal Government: Cryptographic Mechanisms*. NIST Special Publication. https://doi.org/10.6028/NIST.SP.800-175Br1

[9] National Institute of Standards and Technology (2023). *Digital Signature Standard (DSS)*. Federal Information Processing Standards Publication (FIPS) 186-5. https://doi.org/10.6028/NIST.FIPS.186-5

[10] National Institute of Standards and Technology. *Federal Information Processing Standard (FIPS)*. NIST Glossary. https://csrc.nist.gov/glossary/term/federal_information_processing_standard

[11] National Institute of Standards and Technology (2021). *Module-Lattice-Based Digital Signature Standard (FIPS 204)*. Federal Information Processing Standards Publication. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.204.pdf

APPENDIX

LIST OF ABBREVIATIONS

ML-DSA - Module Lattice Digital Signature Algorithm

DFSP - Digital Financial Service Provider

PQC - Post-Quantum Cryptography

ECC - Elliptic Curve Cryptography

RSA - Rivest-Shamir-Adleman (an encryption algorithm)

NIST - National Institute of Standards and Technology

AES - Advanced Encryption Standard

SHA - Secure Hash Algorithm

FSPIOP - Financial Services Provider Interoperability Protocol

JWS - JSON Web Signature

URL - Uniform Resource Locator

HTML - Hypertext Markup Language

CSS - Cascading Style Sheets

JS - JavaScript

API - Application Programming Interface

CPU - Central Processing Unit

SOURCE CODE:

signer.mjs

```
import { ml_dsa44, ml_dsa65, ml_dsa87 } from
'@noble/post-quantum/ml-dsa';
import safeStringify from 'fast-safe-stringify';


class JwsSignerQ{

   constructor(config){
       this.logger = console;

       if(!config.signingKey) {
           throw new Error('Signing key must be supplied as
config argument');
       }

       this.alg = 'ml_dsa65';

       this.signingKey = config.signingKey;
       //console.log(this.signingKey);
   }

   sign(requestOptions){
       const payload = safeStringify(requestOptions.body) ||
safeStringify(requestOptions.data);
       this.logger.isDebugEnabled && this.logger.debug(`JWS
Signing request: ${safeStringify(requestOptions)}`);

       if(!payload) {
           throw new Error('Cannot sign with no body');
       }
```

```
        // get the signature and add it to the header
        requestOptions.headers['fspiop-signature'] =
this.getSignature(requestOptions);

        if (requestOptions.body && typeof requestOptions.body
!== 'string') {
            requestOptions.body =
safeStringify(requestOptions.body);
        }
        if (requestOptions.data && typeof requestOptions.data
!== 'string') {
            requestOptions.data =
safeStringify(requestOptions.data);
        }

    }

    getSignature(requestOptions){
        this.logger.isDebugEnabled && this.logger.debug(`Get
JWS Signature: ${safeStringify(requestOptions)}`);
        const payload = safeStringify(requestOptions.body) ||
safeStringify(requestOptions.data);

        if(!payload) {
            throw new Error('Cannot sign with no body');
        }

        // Note: Property names are case sensitive in the
protected header object even though they are
        // not case sensitive in the actual HTTP headers
        const headerObject = {
            alg: this.alg,
```

```
            'FSPIOP-URI':
requestOptions.headers['fspiop-uri'],
            'FSPIOP-Source':
requestOptions.headers['fspiop-source']
        };

        // set destination in the protected header object if
it is present in the request headers
        if (requestOptions.headers['fspiop-destination']) {
            headerObject['FSPIOP-Destination'] =
requestOptions.headers['fspiop-destination'];
        }

        // set date in the protected header object if it is
present in the request headers
        if (requestOptions.headers['date']) {
            headerObject['Date'] =
requestOptions.headers['date'];
        }

        const token = ml_dsa65.sign(this.signingKey, payload);
        //console.log(requestOptionsString);
        //console.log(token);
        //console.log(safeStringify(token));
        //console.log(safeStringify(requestOptions));

        const signatureObject = {
            signature : token,
            headers : headerObject
        }
        console.log(signatureObject.signature);
        global.signatureObject = signatureObject;
        return signatureObject;
```

```
    }
}

export default JwsSignerQ;
```

validatorQ.mjs

```
import { ml_dsa44, ml_dsa65, ml_dsa87 } from
'@noble/post-quantum/ml-dsa';
import { sign } from 'crypto';
import safeStringify from 'fast-safe-stringify';

const SIGNATURE_ALGORITHMS = ['ml_dsa65'];

class JwsValidatorQ{

    constructor(config) {
        this.logger = console;

        if(!config.validationKeys) {
            throw new Error('Validation keys must be supplied
as config argument');
        }

        this.validationKeys = config.validationKeys;
    }

    validate(request) {
        try{
            const headers= request.headers;
            const payload = safeStringify(request.body) ||
safeStringify(request.payload);
```

```
            this.logger.isDebugEnabled &&
this.logger.debug(`Validating JWS on request with headers:
${safeStringify(headers)} and body:
${safeStringify(payload)}`);

            if(!payload) {
                throw new Error('Cannot validate JWS without a
body');
            }

            // first check we have a public (validation) key
for the request source
            if(!headers['fspiop-source']) {
                throw new Error('FSPIOP-Source HTTP header not
in request headers. Unable to verify JWS');
            }



            const pubKey =
this.validationKeys[headers['fspiop-source']];
            //console.log('_____________public key
validate_____________________');
            //console.log(pubKey);

            if(!pubKey) {
                throw new Error(`JWS public key for
'${headers['fspiop-source']}' not available. Unable to verify
JWS. Only have keys for:
${safeStringify(Object.keys(this.validationKeys))}`);
            }

            const result=headers['fspiop-signature'];
            //console.log(result);
```

```
            const isValid = ml_dsa65.verify(pubKey, payload,
result.signature);
            console.log(isValid);

            this._validateProtectedHeader(headers,
result.headers);

            this.logger.isDebugEnabled &&
this.logger.debug(`JWS verify result:
${safeStringify(result)}`);

            this.logger.isDebugEnabled &&
this.logger.debug(`JWS valid for request
${safeStringify(request)}`);

            request.headers['result']=isValid;

            return isValid;
        }
        catch(err){
            this.logger.isDebugEnabled &&
this.logger.debug(`Error validating JWS: ${err.stack ||
safeStringify(err)}`);
            throw err;
        }

    }

    /**
     * Validates the protected header and checks it against
the actual request headers.
     * Throws an exception if a discrepancy is detected or
validation fails.
```

```
    */
    _validateProtectedHeader(headers, decodedProtectedHeader)
{
        // check FSPIOP-Source is present and matches
        if(!decodedProtectedHeader['FSPIOP-Source']) {
            throw new Error(`Decoded protected header does not
contain required FSPIOP-Source element:
${safeStringify(decodedProtectedHeader)}`);
        }
        if(!headers['fspiop-source']) {
            throw new Error(`FSPIOP-Source HTTP header not
present in request headers: ${safeStringify(headers)}`);
        }
        if(decodedProtectedHeader['FSPIOP-Source'] !==
headers['fspiop-source']) {
            throw new Error(`FSPIOP-Source HTTP request header
value: ${headers['fspiop-source']} does not match protected
header value: ${decodedProtectedHeader['FSPIOP-Source']}`);
        }


        // if we have a Date field in the protected header it
must be present in the HTTP header and the values should
match exactly
        if(decodedProtectedHeader['Date'] && !headers['date'])
{
            throw new Error(`Date header is present in
protected header but not in HTTP request:
${safeStringify(headers)}`);
        }
        if(decodedProtectedHeader['Date'] && (headers['date']
!== decodedProtectedHeader['Date'])) {
            throw new Error(`HTTP date header:
```

```
${headers['date']} does not match protected header Date
value: ${decodedProtectedHeader['Date']}`);
        }

        // if we have an HTTP fspiop-destination header it
should also be in the protected header and the values should
match exactly
        if(headers['fspiop-destination'] &&
!decodedProtectedHeader['FSPIOP-Destination']) {
            throw new Error(`HTTP fspiop-destination header is
present but is not present in protected header:
${safeStringify(decodedProtectedHeader)}`);
        }
        if(decodedProtectedHeader['FSPIOP-Destination'] &&
!headers['fspiop-destination']) {
            throw new Error(`FSPIOP-Destination header is
present in protected header but not in HTTP request:
${safeStringify(headers)}`);
        }
        if(headers['fspiop-destination'] &&
(headers['fspiop-destination'] !==
decodedProtectedHeader['FSPIOP-Destination'])) {
            throw new Error(`HTTP FSPIOP-Destination header:
${headers['fspiop-destination']} does not match protected
header FSPIOP-Destination value:
${decodedProtectedHeader['FSPIOP-Destination']}`);
        }
    }

}

export default JwsValidatorQ;
```

server.mjs

```
import express from 'express';
import bodyParser from 'body-parser';
import path from 'path';
import { ml_dsa44, ml_dsa65, ml_dsa87 } from
'@noble/post-quantum/ml-dsa';
import JwsSignerQ from './signer.mjs';
import JwsValidatorQ from './validator.mjs';
import crypto from 'crypto';

const app = express();
const port = 3000;

const payeeAccounts = {
    'Robert Downey': '9876543210',
    'Tony Stark': '1231231234',
    'Stephen Hawkins': '4564564567'
};

app.use(express.static(path.join(path.resolve(), 'public')));
app.use(bodyParser.urlencoded({ extended: true }));
app.use(express.json());

app.get('/', (req, res) => {
    res.sendFile(path.join(path.resolve(), 'index.html'));
});

const seed = new Uint8Array(32);
crypto.getRandomValues(seed);
const keys = ml_dsa65.keygen(seed);

let requestOptions;
```

```
app.post('/generate-signature', (req, res) => {
    console.log("in generate signature");
    const config_sign = {
        signingKey: keys.secretKey
    };

    requestOptions = {
        body: { amount: req.body.amount },
        date : Date.now(),
        headers: {
            'fspiop-source': req.body.source,
            'fspiop-destination': req.body.destination,
        }
    };

    let jwsSignerQ = new JwsSignerQ(config_sign);
    jwsSignerQ.sign(requestOptions);
    console.log(requestOptions);

    if('fspiop-signature' in requestOptions.headers){
        const base64String =
Buffer.from(requestOptions.headers['fspiop-signature'].signat
ure).toString('base64');
        console.log(base64String);
        res.json({message:"SIGNATURE GENERATED!!", signature:
base64String});
    }
    else{
        res.status(400).json({ message: "Error in generating
signature" });
    }

});
```

```
app.post('/validate-signature', (req, res) => {
    console.log("in validate signature")

    const config_validate = {
        validationKeys: {
            '1234567890': keys.publicKey
        }
    };

    console.log(req.body.validateAmount);
    const request = {
        body: { amount: req.body.validateAmount },
        date: requestOptions.date,
        headers: {
            'fspiop-source': "1234567890",
            'fspiop-destination':
payeeAccounts[req.body.validateBeneficiary],
            'fspiop-signature':
requestOptions.headers['fspiop-signature']
        }
    };

    let jwsValidatorQ = new JwsValidatorQ(config_validate);
    jwsValidatorQ.validate(request);
    const isValid = request.headers['result'];
    console.log(request);
    console.log(requestOptions);

    res.json({ validationResult: isValid ? "Signature is
valid!!!" : "Signature is invalid:((" });

});
```

```
app.listen(port, () => {
   console.log(`Server running at http://localhost:${port}`);
});
```

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
 <meta charset="UTF-8">
 <meta name="viewport" content="width=device-width,
initial-scale=1.0">
 <title>Payment Gateway</title>
 <style>
   body, html {
     font-family: Arial, sans-serif;
     margin: 0;
     padding: 0;
     display: flex;
     height: 100%;
     background-color: #f0f0f5;
   }

   .sidebar {
     position: fixed;
     width: 250px;
     background-color: #333;
     color: #fff;
     padding: 20px;
     display: flex;
     flex-direction: column;
```

```
    align-items: center;
    height: 100%;
}

.sidebar img {
    width: 80px;
    height: 80px;
    border-radius: 50%;
    margin-bottom: 15px;
}

.sidebar .user-name {
    font-weight: bold;
    margin-bottom: 10px;
}

.content {
    margin-left: 300px;
    flex: 1;
    padding: 40px;
    overflow-y: auto;
    height: 100vh;
}

.title {
    font-size: 24px;
    margin-bottom: 20px;
}

label {
    display: block;
    font-weight: bold;
    margin-top: 20px;
```

```
  }

  input, select, button {
    width: 100%;
    padding: 15px;
    margin-top: 8px;
    border-radius: 5px;
    border: 1px solid #ddd;
    font-size: 16px;
  }

  button {
    background-color: #0000FF;
    color: white;
    border: none;
    cursor: pointer;
    font-size: 18px;
    margin-top: 30px;
  }

  button:hover {
    background-color: #0000FF;
  }

  #signatureResult {
  white-space: pre-wrap;
  word-wrap: break-word;
  overflow-wrap: break-word;
  margin-top: 20px;
  font-size: 16px;
  color: #333;
  max-width: 100%;
}
```

```
  </style>
</head>
<body>
 <!-- Sidebar -->
 <div class="sidebar">
   <img src="/pic.jpg" alt="Profile Picture">
   <div class="user-name">Logged in as: Jax Smith</div>
 </div>

 <div class="content">
   <div class="title">Fund Transfer</div>

   <label for="beneficiary">Choose Beneficiary:</label>
   <select id="beneficiary" name="beneficiary"
onchange="updatePayeeAccount()">
   <option value="">Select a beneficiary</option>
   <option value="Robert Downey">Robert Downey</option>
   <option value="Tony Stark">Tony Stark</option>bene
   <option value="Stephen Hawkins">Stephen Hawkins</option>
   </select>

   <form id="paymentForm" onsubmit="generateSignature(event)"
action="/generate-signature" method="POST">

       <label for="payerAccount">Payer Account
Number:</label>
       <input type="text" id="source" name="source"
value="1234567890" readonly>

       <label for="payeeAccount">Payee Account
Number:</label>
       <input type="text" id="destination" name="destination"
placeholder="Select a beneficiary" readonly>
```

```
        <label for="amount">Amount(in rupees):</label>
        <input type="number" id="amount" name="amount"
placeholder="Enter amount to transfer" min="1" required>

        <button type="submit" >Make Payment</button>
    </form>

    <p id="signatureResult"></p>


    <form id="validationForm"
onsubmit="validateSignature(event)"
action="/validate-signature" method="POST">
      <input type="text" style="display: none;"
id="validateBeneficiary" name="validateBeneficiary">
      <input type="text" style="display: none;"
id="validateAmount" name="validateAmount">
      <button id ="validateButton" style="display: none;"
type="submit" >Validate Signature</button>
    </form>

    <p id="validationResult"></p>

 </div>

 <script>

    let selectedBeneficiary = '';
    let selectedAmount='';

    const payeeAccounts = {
      'Robert Downey': '9876543210',
```

```
    'Tony Stark': '1231231234',
    'Stephen Hawkins': '4564564567'
  };

  function updatePayeeAccount() {
    const beneficiary =
document.getElementById('beneficiary').value;
    const payeeAccountInput =
document.getElementById('destination');

    // Set the payee account number based on the selected
beneficiary
    if (beneficiary) {
      payeeAccountInput.value = payeeAccounts[beneficiary];
      selectedBeneficiary = beneficiary;
    } else {
      payeeAccountInput.value = ''; // Clear if no
beneficiary is selected
    }
  }

  async function generateSignature(event) {
    event.preventDefault(); // Prevent the form from
submitting normally

    const form = event.target;
    const formData = new FormData(form);
    const data = Object.fromEntries(formData.entries());
    selectedAmount=data.amount;

    try {
        const response = await fetch('/generate-signature',
{
```

```
            method: 'POST',
            headers: {
                'Content-Type': 'application/json',
            },
            body: JSON.stringify(data)
        });

        const result = await response.json();

        if (response.ok) {
            const formattedSignature =
result.signature.replace(/(.{120})/g, "$1\n");

document.getElementById("signatureResult").innerHTML =
`<u>${result.message}</u><br>` + "\n" + formattedSignature;
            alert(result.message);

document.getElementById("validateButton").style.display =
'block';
        } else {

document.getElementById("signatureResult").innerText =
result.message;

document.getElementById("validateButton").style.display =
'none';
        }
    } catch (error) {
        console.error("Error:", error);
        document.getElementById("signatureResult").innerText
= "An error occurred.";

document.getElementById("validateButton").style.display =
```

```
'none';
        }
    }

    async function validateSignature(event) {
        document.getElementById("validateBeneficiary").value =
selectedBeneficiary;
        document.getElementById("validateAmount").value =
selectedAmount;
        event.preventDefault();

        console.log("HELLOOOOOO");
        console.log(selectedBeneficiary);
        console.log(selectedAmount);

        const form = event.target;
        const formData = new FormData(form);
        const data = Object.fromEntries(formData.entries());

        try {
            const response = await fetch('/validate-signature',
{
                method: 'POST',
                headers: {
                    'Content-Type': 'application/json',
                },
                body: JSON.stringify(data)
            });

            const result = await response.json();

            if (response.ok) {
```

```
document.getElementById("validationResult").innerHTML =
`<br>`+result.validationResult+`<br><br>`;
            alert(result.validationResult);
        } else {

document.getElementById("validationResult").innerHTML =
result.validationResult;
        }
      }
      catch(error){

document.getElementById("validationResult").innerText =
"error";
      }



  }

 </script>
</body>
</html>
```

package.json

```
{
  "dependencies": {
    "@noble/post-quantum": "^0.2.0",
    "express": "^4.21.1",
    "fast-safe-stringify": "^2.1.1",
    "nodemon": "^3.1.7"
  }
}
```