Project Title:

Post-Quantum based Key Exchange and Authentication in TLS 1.3: A Pure Post-Quantum Cryptography Approach

Domain: Post-Quantum Cryptography
Mentor: Dr. R. Gunasekaran

Team Members:

Member 1:
Name: Tejesshree S
Reg no: 2022503524
Ph no: 90250 40835
Email: gowrikavinraja@gmail.com

Member 2:
Name: Janani A
Reg no: 2022503502
Ph no: 81480 69677
Email: jananialagar.2410@gmail.com

Member 3:
Name: Kathirvel M
Reg no: 2022503060
Ph no: 72999 91398
Email: kathirvelm2022@gmail.com

NGNLab
Next Generation Networks
www.ngnlab.org

What is PQC?

- ❑ **Quantum Threat to Classical Encryption** – Algorithms like **RSA, AES** are vulnerable to quantum attacks.
- ❑ **Quantum Algorithms Break Encryption** – **Shor's** and **Grover's algorithms** can crack current cryptographic methods.
- ❑ **PQC is Quantum-Safe** – Uses encryption methods that **quantum computers cannot easily break**.
- ❑ **No Reliance on Factorization & Discrete Logs** – Unlike classical encryption, PQC is based on **harder mathematical problems**.
- ❑ **Future-Proof Security** – Protects against **"Harvest Now, Decrypt Later"** attacks.

Classical Cryptography

Classical cryptography is primarily based on problems from Number Theory, Algebra, and Finite fields.

The key mathematical problems that provide security are:

1. ***Integer Factorisation problem:***

Problem: Given a large integer N, find its prime factors.

Example: Factoring 15 is easy (3 x 5), but factoring a 2048-bit integer is extremely hard.

Mathematical Basis: Number Theory.

Hardness: The best Known classical algorithms grow exponentially with input size.

Example Cryptosystem: Rivest-Shamir-Adleman (RSA)

Why is it hard?

- Prime factorization is not known to be solved in polynomial time on a classical computer.
- The computational complexity grows superpolynomial with the size of the input.

Classical Cryptography

2. *Discrete Logarithm Problem (DLP):*

Problem: Given a prime *p,* a generator *g*, and $h = g^x$ mod *p* find *x*.

Example: if *g*=5, *h*=8, and *p*=23, find *x* such that 5^x mod 23 = 8.

Mathematical Basis: Group theory and modular Arithmetic.

Hardness: Even with index calculus algorithms, the problem grows sub-exponentially with the input size.

Example Cryptosystem: Diffie-Hellman key Exchange, DSA (Digital Signature Algorithm)

Why is it hard?

- Computing modular exponentiation is easy,but reversing it to compute a discrete logarithm is computationally infeasible for large primes.

Classical Cryptography

3. ***Elliptic Curve Discrete Logarithm Problem (ECDLP):***

Problem: Similar to the DLP but over the group of points on the elliptic curve.

Mathematical Basis: Elliptic curve theory over Finite fields.

Hardness: Harder than integer factorization and DLP for the same key size - Smaller keys provide comparable security.

Example Cryptosystem: Elliptic Curve Cryptography (ECC)

Why is it hard?

- The structure of elliptical curves make the problem more resistant to known sub-exponential attacks like index calculus.

Classical Cryptography

4. Lattice-based Problems:

Problem: Given a high-dimensional lattice, find the shortest non-zero vector (SVP) or Closest vector to a given point(CVP).

Mathematical Basis: Geometry of number and linear Algebra.

Hardness: Exponentially hard with respect to lattice dimension.

Example Cryptosystem: Learning with errors (LWE), Number Theory Research Unit (NTRU).

Why is it hard?

- No efficient algorithm is known for solving Lattice problems in high dimension.
- Lattice-based cryptography is a major area for post-quantum security.

Classical Cryptography

5. Subset sum (Knapsack) problem:

Problem: Given a set of integers and a target sum, find a subset of sums to the target.

Example: Given {3,34,4,12,5,2} and target = 9, find the subset {4,5}.

Mathematical basis: Combinators.

Hardness: NP-compete - no known polynomial-time solution.

Example Cryptosystem: Early Knapsack-based cryptosystems (later broken).

Why is it hard?

- It's NP-complete, which means no polynomial-time algorithm is known unless P=NP.

Why is Classical Cryptography considered Secure?

1. ***Exponential Time Complexity:***

Most classical cryptographic schemes require attackers to solve problems with exponential or sub-exponential complexity - infeasible for large key sizes.

2. ***No polynomial-time Algorithm:***

No polynomial-time algorithm is known for the core problems underlying classical cryptographic (on classical computers).

3. ***Use of Large key sizes:***

Increasing key size exponentially increases computational difficulty without a linear increase in processing time for legitimate encryption and decryption.

Quantum Threat to Classical Cryptography

Problem	Classical difficulty	Quantum Threat
Integer Factorization	Hard (Exponential)	Shor's Algorithm -> Polynomial Time
Discrete Logarithm	Hard (sub-exponential)	Shor's Algorithm -> Polynomial Time
ECDLP	Hard (sub-exponential)	Shor's Algorithm -> Polynomial Time
Lattice Problem	Hard (Exponential)	Believed secure against quantum.

Classical Cryptography fails against Quantum Computers

Why does classical cryptography fail against Quantum Computers?

1. *Shor's Algorithm (1994):*

Quantum Algorithm that solves both:

- Integer Factorization Problem
- Discrete Logarithm Problem

Complexity: Runs in polynomial time on a quantum Computer

$\mathcal{O}((\log N)^3)$

Where N is the input size.

Breaks classical cryptographic schemes like:

- RSA (based on factorisation)
- Diffie-Hellman (Based on discrete logs)
- ECC (based on elliptical curve discrete logs)

Classical Cryptography fails against Quantum Computers

Why does classical cryptography fail against Quantum Computers?

2. *Grover's Algorithm (1996):*

Quantum algorithm that speed up brute-force search:

- Reduces search complexity from $\mathcal{O}(2^n)$ to $\mathcal{O}(2^{n/2})$.

Affects symmetric cipher (like AES) and hash functions (like SHA) by having the effective key length:

- AES-128 → effective strength equivalent to AES-64.
- AES-256 → effective strength equivalent to AES-128.

Post-Quantum Cryptography

Post-Quantum Cryptography (PQC) is a field of cryptography that focus on developing cryptographic algorithms that are secure against attacks by quantum computers.

PQC aims to **replace current algorithms** with ones that:

- Are **resistant to quantum attacks.**
- Can still run efficiently on classical computers.
- Are ready to be adopted in real-world systems (e.g., browsers, VPNs, messaging apps).

The NIST PQC standardization project finalists like Module-Lattice Key Encapsulation Mechanism (ML-KEM) and Module-Lattice Digital Signature Algorithm (ML-DSA).

Real-world systems like Google Chrome, Cloudflare, OpenSSH, and TLS are starting to experiment with or integrate PQC algorithms.

Post-Quantum Cryptography Solution

PQC type	Mathematical Basis	Reason for hardness
Lattice-based	Geometry of numbers	Finding shortest/closest vectors in high- dimensional lattices
Code-based	Error-correcting codes	Syndrome decoding Problem
Multivariate- quadratic (MQ)	Multivariate polynomial over finite fields	Solving large nonlinear equation systems.
Hash-based	Cryptographic hash functions	Pre-image and collision resistance
Isogeny-based	Elliptical curves over finite fields	Finding isogenies between curves

Mathematical Background in PQC

1. *Lattice-based Cryptography:*

Based on the hardness of finding short or close vectors in a high-dimensional lattice.

Problem type:

- Shortest Vector Problem (SVP): Find the shortest nonzero vector in a lattice.
- Closest Vector Problem (CVP): FInd the vector in a lattice closest to a target point.
- Learning with Error (LWE): Given a noisy linear equation, recover the hidden variables.

Why is it hard?

- Lattice problems grow exponentially in complexity with the lattice dimension.
- Best known classical and quantum algorithms are inefficient for high dimensions.

Example schemes:

- NTRU (Public key encryption).
- Kyber (Key Exchange).
- Dilithium (digital signature).

Mathematical Background in PQC

2. *Code-based cryptography:*

Based on the hardness of decoding a linear code with random noise (syndrome decoding problem).

Problem:

- Given a codeword with added noise, recover the original message.

$$y = c + e$$

Where c is the codeword and e is the error vector.

Why is it hard?

- Decoding random linear codes is NP-complete.
- No efficient quantum algorithm is known.

Example Schemes:

- McElice (Public key encryption).

Mathematical Background in PQC

3. Multivariate-Quadratic (MQ) Cryptography:

Based on the difficulty systems of nonlinear polynomial equations over finite fields.

Problem:

- Solve a system of quadratic equations over a finite field:

$$P(x_{1}, x_{2,\ldots,} x_{n}) = 0$$

Why is it hard?

- Solving systems of quadratic equations over finite fields is NP-hard.
- Quantum computers don’t have an advantage here.

Example Schemes:

- Rainbow (digital signature).

Mathematical Background in PQC

4. *Hash-based Cryptography:*

Based on the security of cryptography hash functions.

Problem:

- Finding collisions or pre-image in cryptographic hash function.

Why is it hard?

- Even with Grover's algorithm, finding a pre-image still requires exponential effort: $2^{n/2}$ operations for a hash of size n.

Example schemes:

- SPHINCS (digital signature).

Mathematical Background in PQC

5. *Isogeny-based cryptography:*

Based on the hardness of finding isogenies (maps) between elliptic curves over finite fields.

Problems:

- Given two elliptical curves, finding the isogeny (morphism) between them.

Why is it hard?

- No efficient quantum algorithm is known.
- Problem grows in complexity with the size of the elliptic curve.

Example schemes:

- SIKE (Key exchange) - Recently broken by classical attack, so under review.

Already Existing Algorithms

1. ***Advanced Encryption Standard (AES):***

Type: Symmetric.

Use: Encryption.

Based on: Substitution-Permutation Network (SPN).

Mathematical Background:

- Finite field arithmetic over $\mathbf{GF(2^8)}$.
- Linear algebra (matrix transformations).

Hardness:

- Brute force complexity: $\mathbf{2^{128}}$ for AES-128.
- Resistant to known cryptanalysis methods (differential, linear attacks).
- Weakened by: Grover's algorithm (reduces complexity to $\mathbf{2^{64}}$ for AES-128).

Example:

- Block size: 128 bits.
- Key size: 128, 192, or 256 bits.
- Operations: SubBytes, ShiftRows, MixColumns, AddRoundKey.

Already Existing Algorithms

2. *Rivest-Shamir-Adleman (RSA):*

Type: Asymmetric (Public Key).

Use: Encryption, Digital Signatures.

Based on: Integer Factorization Problem.

Mathematical Background:

- Prime number theory.
- Modular arithmetic.
- Euler's theorem.

Hardness:

- Finding large prime factors is computationally hard.
- No polynomial-time algorithm exists for integer factorization on classical computers.
- Broken by: Shor's algorithm (on a quantum computer).

Example:

- Public key: (n,e) where $n = p \times q$ (product of large primes).
- Encryption: $c = m^e$ mod n.
- Decryption: $m = c^d$ mod n (where d is the modular inverse of e mode $\phi(n)$).

Already Existing Algorithms

3. *Elliptical Curve Cryptography (ECC):*

Type: Asymmetric (Public Key).

Use: Encryption, Digital Signatures.

Based on: Elliptic Curve Discrete Logarithm Problem (ECDLP).

Mathematical Background:

- Group theory over elliptic curves.
- Finite field arithmetic.

Hardness:

- Computing k from $\boldsymbol{P} = k\boldsymbol{G}$(scalar multiplication) is hard.
- No known sub-exponential classical algorithm.
- Broken by: Shor's algorithm (on a quantum computer).

Example:

- Public key: $\boldsymbol{Q} = k\boldsymbol{G}$.
- Encryption: Uses point addition and scalar multiplication.
- Security comes from the difficulty of reversing scalar multiplication.

Already Existing Algorithms

4. *Elliptic Curve Diffie-Hellman (ECDH):*

Type: Asymmetric (Public Key)

Use: Secure Key Exchange

Based on: Elliptic Curve Discrete Logarithm Problem (ECDLP)

Mathematical Background:

- Group theory over elliptic curves
- Modular arithmetic

Hardness:

- Secure because of the hardness of ECDLP.
- No efficient classical algorithm to solve ECDLP.
- Broken by: Shor's algorithm (on a quantum computer)

Example:

- Alice’s key: k_A, Bob’s key: k_B.
- Shared key: $k_{AB} = k_A \cdot k_B.\mathbf{G}$.

Already Existing Algorithms

Algorithm	Type	Mathematical Problem	Classical Hardness	Quantum Hardness
RSA	Asymmetric	Integer Factorization	Sub-exponential	Broken by Shor’s
ECC	Asymmetric	Elliptic Curve Discrete Logarithm	Sub-exponential	Broken by Shor’s
ECDH	Asymmetric	Elliptic Curve Discrete Logarithm	Sub-exponential	Broken by Shor’s
AES	Symmetric	SPN over Finite fields	Exponential (key size)	Grover reduces strength by half

Mathematical Concepts Comparison of PQC Algorithms

Problem Type	Classical Hardness	Quantum Hardness
Factorization	Sub-exponential	Polynomial (Shor’s)
Discrete Logarithm	Sub-exponential	Polynomial (Shor’s)
Lattice Problems	Exponential	Exponential
Code-based Decoding	Exponential	Exponential
MQ algorithm	NP-complete	NP-complete
Hash-based Problems	Exponential	Square-root speedup (Grover’s)
Isogeny Problem	Exponential	Exponential

Algorithms in PQC

Algorithms that are selected to the third round of NIST's Post-Quantum Cryptography standardization Process:

Category	Algorithm	Based on	Strengths	Weaknesses
KEM	ML-KEM (standardised)	Module- LWE	Fast, small, strong	-
	FrodoKEM	Standard LWE	Conservative	Large, slow
	NTRU	NTRU lattices	Mature, fast	Slightly larger
	McEliece	Code-based	Battle-tested	Huge keys
	BIKE	Code-based	Small. efficient	Under review

Algorithms in PQC

Algorithms that are selected to the third round of NIST's Post-Quantum Cryptography standardization Process:

Category	Algorithm	Based on	Strengths	Weaknesses
Signature	ML-DSA (standardised)	Module- LWE	Balanced, fast	Moderate sizes
	Falcon (Recently standardised)	NTRU lattices	compact	Hard to implement
	SPHINCS+ (standardised)	Hash-based	Very conservative	Large, slow

Possible areas to work on PQC - TLS

1. *TLS (Transport Layer Security) with PQC:*

TLS is a widely used protocol for secure communication over the internet. It secures:

- WEB Browsing (HTTPS).
- Email (SMTP, IMAP)
- Messaging.
- Secure file transfers.

Why Classical TLS is vulnerable?

- TLS relies on RSA, ECC, and DH for key exchange and authentication.
- Shor’s algorithm can efficiently break RSA and ECC once large quantum computers are available.
- Symmetric ciphers like AES and hash functions like SHA are weakened by Grover's algorithm but remain secure with longer key sizes.

Possible areas to work on PQC - TLS

i. ***NIST PQC Candidates*** (Round 4):

- Lattice-based: **ML-KEM**(key exchange), **ML-DSA** (signatures)
- Hash-based: **SPHINCS+** (signatures)

ii. ***Hybrid TLS*** – Combining classical and post-quantum algorithms:

- Example: **ECDH + ML-KEM** → Combines classical ECC with post-quantum Kyber
- Ensures security even if PQC algorithms are not yet fully trusted

iii. ***TLS 1.3 with PQC*** – Using Open Quantum Safe (OQS) and libOQS:

- OQS supports hybrid key exchange with Kyber
- **curl** and **OpenSSL** are being adapted to support PQC-based TLS

Example:

- Key Exchange → ML-KEM (Module-Lattice Key Encapsulation Mechanism).
- Signature → ML-DSA (Module-Lattice Digital Signature Algorithm.
- Symmetric encryption → AES--256-GCM (resistant to Grover’s attack with increased key size)

Possible areas to work on PQC - Vanets

2. *Vehicle-to-Vehicle (V2V) and Vehicle-to-Everything (V2X) with PQC:*

V2V and V2X are communication protocols that enable vehicles to exchange information with each other and with infrastructure (traffic lights, road signs, etc.).

Security Risks in V2V/V2X:

- Vehicles rely on **digital signatures** and **key exchange** to authenticate and secure communication.
- Classical ECDSA (Elliptic Curve Digital Signature Algorithm) is vulnerable to quantum attacks.
- Man-in-the-middle (MITM) and replay attacks are possible if key exchange is compromised.

Possible areas to work on PQC - Vanets

i. **ML-DSA** and **Falcon** → For digital signatures in V2V and V2X.

- Resistant to quantum attacks.
- Fast verification ensures low latency in real-time vehicle communication.

ii. **ML-KEM**→ For key exchange between vehicles.

- Secure and fast even in low-power environments.
- Low bandwidth overhead, suitable for real-time communication.

iii. **Hash-based schemes** → For message integrity and authentication.

- SPHINCS+ ensures long-term security.
- Resistant to Grover’s algorithm.

Example:

- V2V/V2X handshake → ML-KEM.
- Message signing → ML-DSA.
- Integrity → SPHINCS+.

Possible areas to work on PQC - IoT

IoT refers to a network of interconnected devices (sensors. Smart home devices, wearables, etc.) that communicate wirelessly over the internet.

Challenges in IoT Security:

- IoT devices have **low processing power** and **limited memory**.
- Traditional RSA and ECC are computationally expensive for IoT.
- Quantum attacks threaten existing key exchange and signature schemes.

PQc solutions for IoT:

i. ***Lightweight PQC*** – Efficient algorithms suitable for low-power devices:

- **ML-KEM** – Efficient for key exchange in IoT
- **ML-DSA** – Lightweight and fast signing

ii. ***Hash-based signatures*** – For firmware integrity and secure updates:

- **SPHINCS+** – Stateless and quantum-safe
- Resistant to side-channel attacks

Possible areas to work on PQC - IoT

PQc solutions for IoT:

iii. ***Key Exchange***

- Efficient and low-latency.
- ML-KEM’s small key size and fast operations make it suitable for constrained IoT networks..

Example:

- Secure IoT handshake → ML-KEM.
- Firmware signing → SPHINCS+.
- Integrity → SHA-512 (to resist Grover’s attack).

Hybrid Approach for Transition

Because PQC algorithm are still being standardized and tested, a hybrid approach is often used: Combine classical and post-quantum algorithms to provide fallback in case of vulnerabilities.

Component	Classical Algorithm	PQC Replacement
Key Exchange	ECDH	ML-KEM
Digital Signature	ECDSA	ML-DSA, Falcon
Symmetric Encryption	AES-256	AES-256 (increase key size for quantum resistance)
Hashing	SHA-256	SHA-512

Transport Layer Security

- ❑ TLS 1.3 Handshake establishes a secure communication session between a client and server.
- ❑ The typical TLS 1.3 Handshake follows the given steps:

1. *ClientHello* - Client initiates handshake by sending ClientHello message (TLS version, ClientRandom/*KeyShare,* CipherSuites…)

2. *ServerHello* - Server responds with ServerHello message (TLS version selected, ServerRandom/*KeyShare*, Selected CipherSuite…)

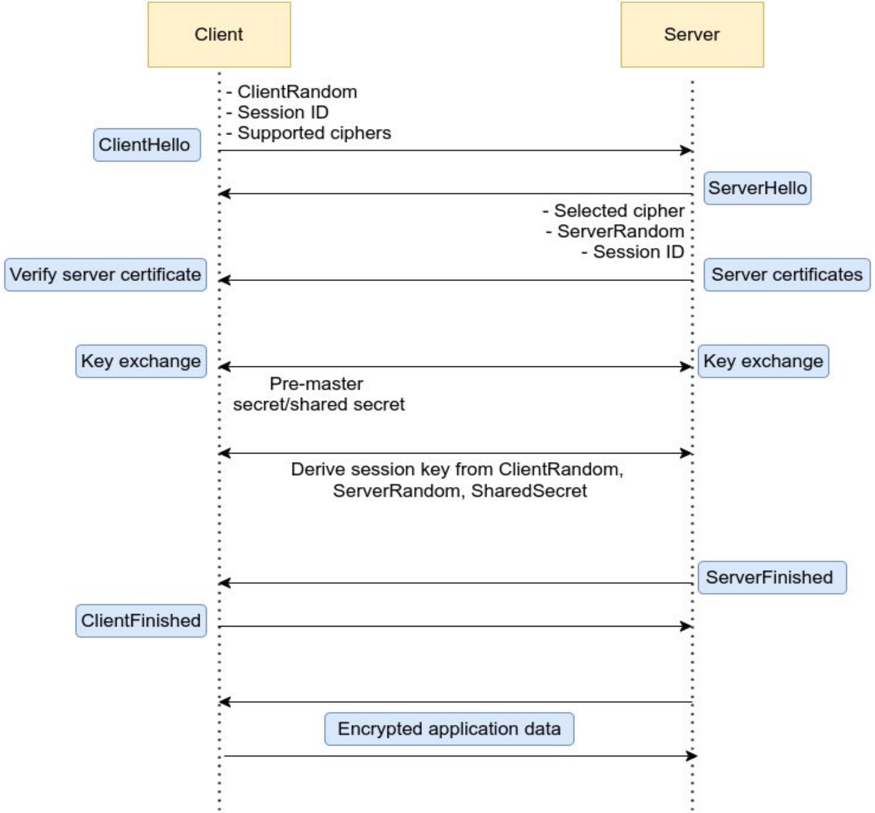
3. Server and Client derive the same *SharedSecret* from their private and other party's public key.

4. *ServerCertificate* - Server sends its X.509 certificate (ServerPublicKey, CA…) and *ServerFinished*. Client verifies ServerCertificate and sends *ClientFinished*.

5. The *SharedSecret* is fed to Key Derivation Function (KDF) along with handshake transcript which includes random nonces from both Client and Server to derive a set of symmetric keys for *SessionKeys.*

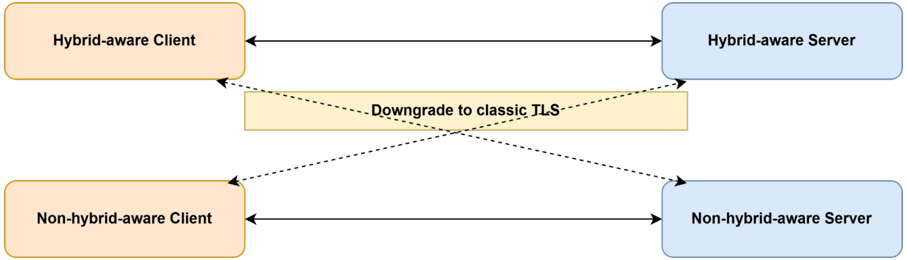
6. TLS Handshake completed and **encrypted communication** is valid using symmetric session key.

Transport Layer Security



Hybrid Key Exchange in TLS 1.3

- Combines classical key exchange with post-quantum key exchange algorithm.
- The shared secrets from each side are then combined (by concatenation and processed through a key derivation function) to produce a final master secret. This master secret is used in TLS's key schedule to derive further handshake keys.
- This approach ensures that even if one of the key exchanges is broken by future quantum attacks, the overall secret remains protected as long as atleast one component remains secure. It also allows for compatibility with Hybrid and Non -Hybrid aware systems.



Hybrid Authentication in TLS 1.3

- Ensures that TLS authentication remains secure against both classical and quantum attacks.

Hybrid digital signatures:

- Both classical and post-quantum signatures are used for authentication. Server certificate contains both signatures, allowing verification to be done in either method.

KEMTLS (Key Encapsulation Mechanism TLS):

- Instead of digital signatures, KEMs are used for authentication.
- Client sends an ephemeral KEM public key in ClientHello, and server encapsulates a shared secret using it.
- The server's static KEM public key is included in certificate and authentication is done via key encapsulation (ClientKemCipherText) rather than signatures, where Client encapsulates a shared secret using server's static KEM and verifies if decapsulation was successful based on ServerFinished message

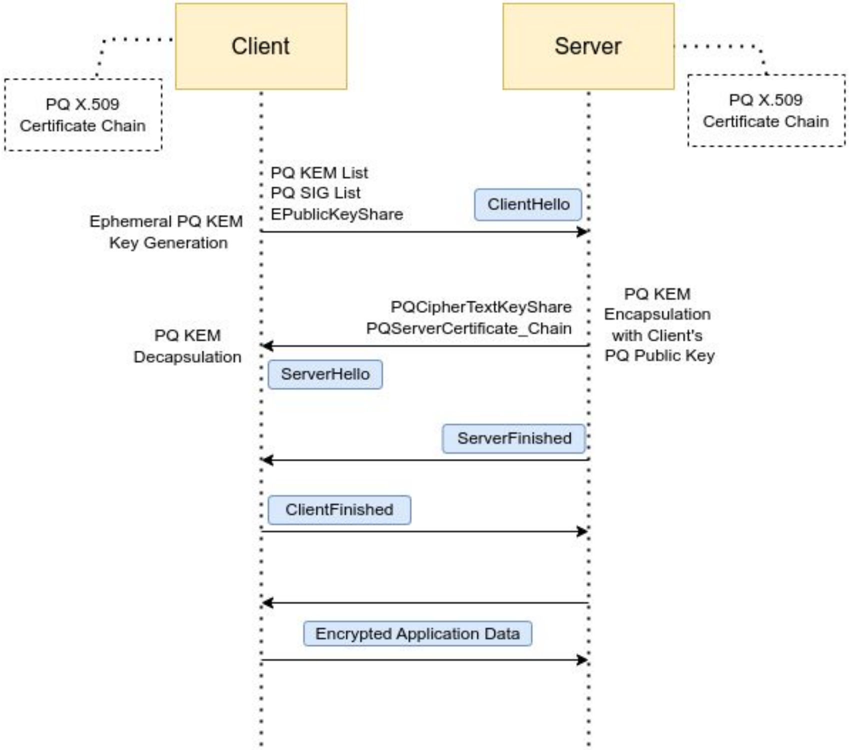
Pure Post-Quantum Cryptography in TLS 1.3

Replacing classical key exchange (RSA/ECDH) and digital signatures (RSA/ECDH) with post-quantum cryptographic alternatives (ML-KEM and ML-DSA)

The steps in pure PQC TLS 1.3 handshake includes:

1. *ClientHello* - TLS version, ClientRandom, KeyShare (ClientPQCKEMPublicKey), CipherSuites (PQC KEMs and PQC SIGs)
2. *ServerHello* - TLS version selected, ServerRandom, KeyShare (Encaps PQC KEM Ciphertext), Selected CipherSuite)
3. Client decapsulates the KEM ciphertext using its PQC private key to derive *SharedSecret*
4. *ServerCertificate* - X.509 certificate with PQC digital signature, ServerFinished. Client verifies ServerCertificate's PQC signature
5. SharedSecret is fed to KDF (Key Derivation Function) along with ClientRandom, ServerRandom to generate *SessionKeys*

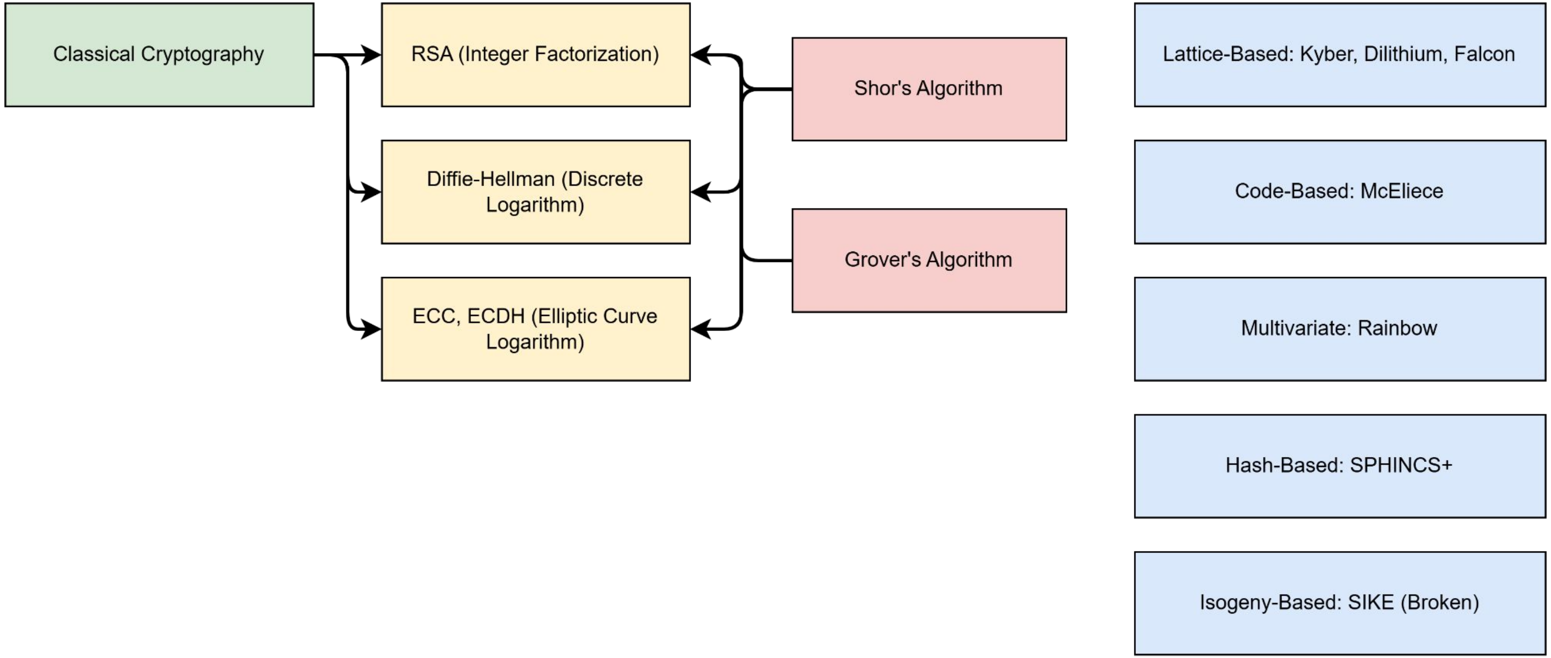
Pure Post-Quantum Cryptography in TLS 1.3


Client
Server
PQ X.509 Certificate Chain
PQ X.509 Certificate Chain
PQ KEM List
PQ SIG List
EPublicKeyShare
ClientHello
Ephemeral PQ KEM Key Generation
PQCipherTextKeyShare
PQServerCertificate_Chain
PQ KEM Encapsulation with Client's PQ Public Key
PQ KEM Decapsulation
ServerHello
ServerFinished
ClientFinished
Encrypted Application Data

Tools Used

Tool Name	Description	Why is it used?
OpenSSL	- Open-source library for **TLS/SSL** protocols and cryptographic functions.	- OpenSSL is being extended to support PQC algorithms through **libOQS** and **OQS-provider**.
libOQS (Open Quantum Safe)	- C library that provides implementations of post-quantum cryptographic algorithms. - Supports key exchange and digital signature schemes (ML-KEM, ML-DSA, SLH-DSA, etc.).	- Implements PQC algorithms selected by NIST. - Provides a consistent API for integrating PQC into OpenSSL and other cryptographic tools.
OQS-Provider (Open Quantum Safe Provider)	- Plugin for OpenSSL 3.x that enables PQC algorithms through libOQS.	- Integrates libOQS into OpenSSL by registering PQC algorithms and handling key exchange or signature operations.
NGINX	- High-performance web server and reverse proxy.	- Supports OpenSSL-based TLS, allowing integration with OQS.
cURL (Client URL)	- Command-line tool for transferring data using various protocols (HTTP, HTTPS).	- Allows testing of PQC-secured TLS connections.

Post-Quantum Algorithm Study



Classical Cryptography

RSA (Integer Factorization)

Diffie-Hellman (Discrete Logarithm)

ECC, ECDH (Elliptic Curve Logarithm)

Shor's Algorithm

Grover's Algorithm

Lattice-Based: Kyber, Dilithium, Falcon

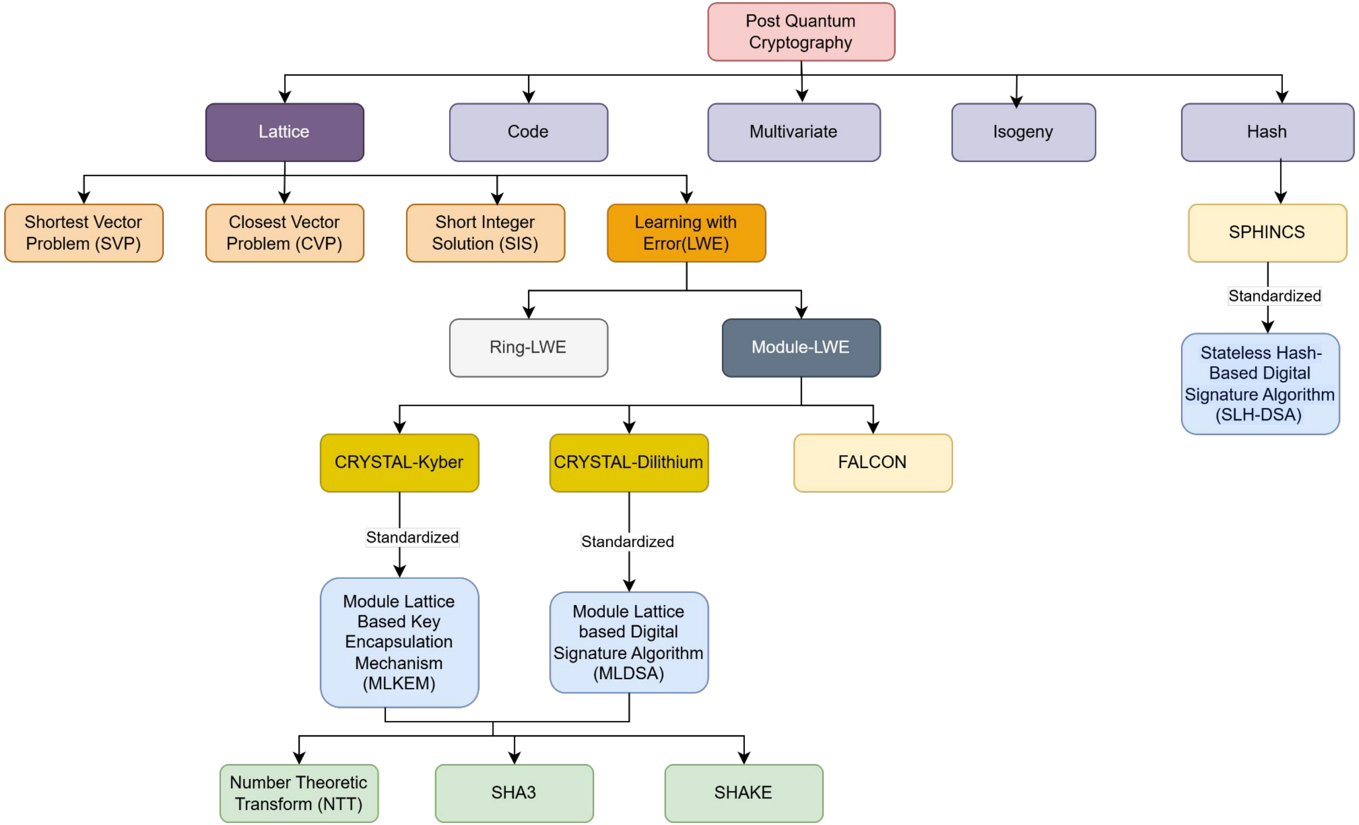
Code-Based: McEliece

Multivariate: Rainbow

Hash-Based: SPHINCS+

Isogeny-Based: SIKE (Broken)

Post-Quantum Algorithm Study


Post Quantum Cryptography
Lattice
Code
Multivariate
Isogeny
Hash
Shortest Vector Problem (SVP)
Closest Vector Problem (CVP)
Short Integer Solution (SIS)
Learning with Error(LWE)
SPHINCS
Standardized
Stateless Hash-Based Digital Signature Algorithm (SLH-DSA)
Ring-LWE
Module-LWE
CRYSTAL-Kyber
CRYSTAL-Dilithium
FALCON
Standardized
Standardized
Module Lattice Based Key Encapsulation Mechanism (MLKEM)
Module Lattice based Digital Signature Algorithm (MLDSA)
Number Theoretic Transform (NTT)
SHA3
SHAKE

Module-Lattice Key Encapsulation Mechanism (ML-KEM)

Step 1: ***Key Generation (Bob)***

- Bob picks
  - a random secret matrix S and a small error matrix E
  - a public random matrix A
- Bob computes his public key: P = A.S + E mod q
- Bob shares (A,P) as his public key

Step 2: ***Encapsulation (Alice)***

Alice wants to send Bob a shared secret key securely.

- Alice picks a message m, hashes it to get a noise vector r.
- Alice computers ciphertext (encapsulation of key)
  - U = A.r + e1 mod q
  - V = P.r + e2 + H(m) mod q
- The shared secret becomes K = G(m), where G and H are cryptographic hash functions
- Alice sends (U,V) to Bob

Module-Lattice Key Encapsulation Mechanism (ML-KEM)

Step 3: *Decapsulation (Bob)*

Bob recovers the shared secret from ciphertext sent by Alice.

- Using Bob's secret key S, he computes:
    - $V' = S^T.U \bmod q$
- Bob retrieves m by reversing error correction process.
- Bob computes the shared secret K = G(m).

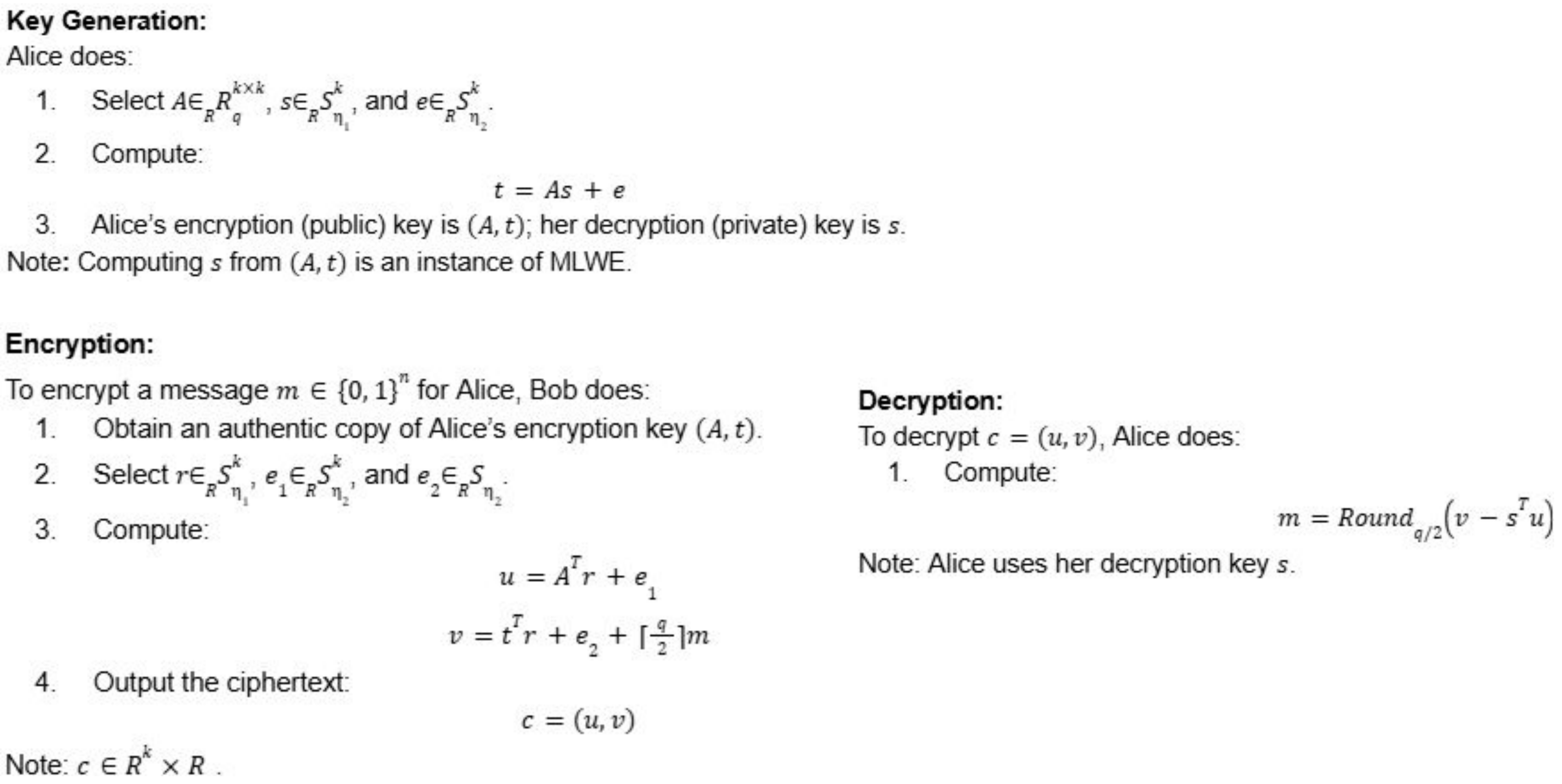
Now, both Alice and Bob share the same secret key K..

Domain Parameters:

For concreteness, we'll use the ML-KEM-768 domain parameters:

- q=3329
- n=256
- k=3
- 1=2
- 2=2

Module-Lattice Key Encapsulation Mechanism (ML-KEM)



Key Generation:

Alice does:

1. Select $A \in_R R_q^{k\times k}$, $s \in_R S_{\eta_1}^k$, and $e \in_R S_{\eta_2}^k$.
2. Compute:

$$t = As + e$$

3. Alice's encryption (public) key is (A, t); her decryption (private) key is s.

Note: Computing s from (A, t) is an instance of MLWE.

Encryption:

To encrypt a message $m \in \{0, 1\}^n$ for Alice, Bob does:

1. Obtain an authentic copy of Alice's encryption key (A, t).
2. Select $r \in_R S_{\eta_1}^k$, $e_1 \in_R S_{\eta_2}^k$, and $e_2 \in_R S_{\eta_2}$.
3. Compute:

$$u = A^T r + e_1$$

$$v = t^T r + e_2 + \lceil \tfrac{q}{2} \rfloor m$$

4. Output the ciphertext:

$$c = (u, v)$$

Note: $c \in R^k \times R$.

Decryption:

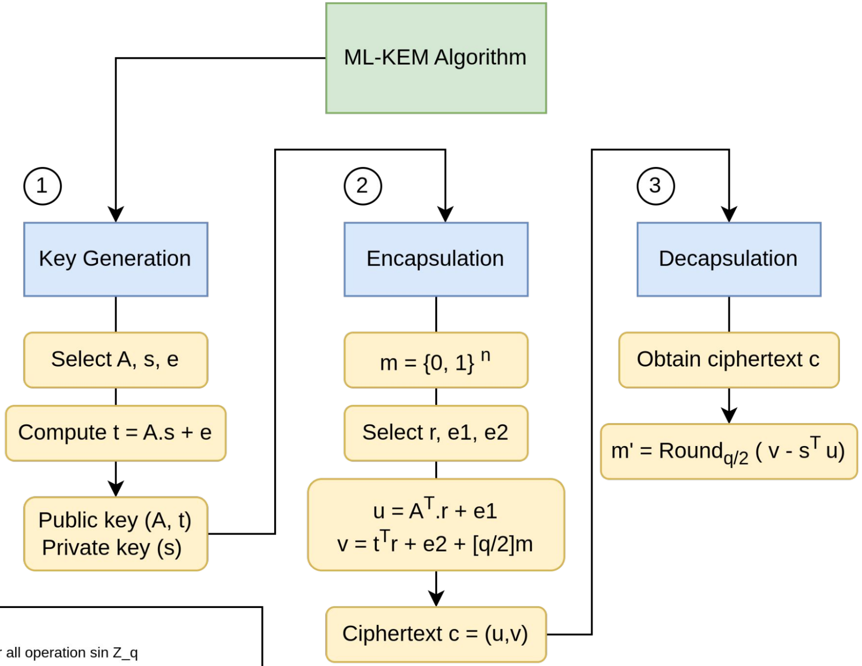
To decrypt $c = (u, v)$, Alice does:

1. Compute:

$$m = Round_{q/2}\left(v - s^T u\right)$$

Note: Alice uses her decryption key s.

Module-Lattice Key Encapsulation Mechanism (ML-KEM)



ML-KEM Algorithm

1 Key Generation

Select A, s, e

Compute t = A.s + e

Public key (A, t)
Private key (s)

2 Encapsulation

m = {0, 1} ^{n}

Select r, e1, e2

u = A^{T}.r + e1
v = t^{T}r + e2 + [q/2]m

Ciphertext c = (u,v)

3 Decapsulation

Obtain ciphertext c

m' = Round$_{q/2}$ (v - s^{T} u)

Parameters:

q - Modulus for all operation sin Z_q

n - Degree of polynomials in the ring Z_q[x]/(x^{n} + 1)

k - Number of polynomials (vectors) used in keys and ciphertext (varies based on security level)

η_1 - Parameter for noise distribution in key generation and encapsulation

η_2 - Parameter for noise distribution in encapsulation

Module-Lattice Key Encapsulation Mechanism (ML-KEM)

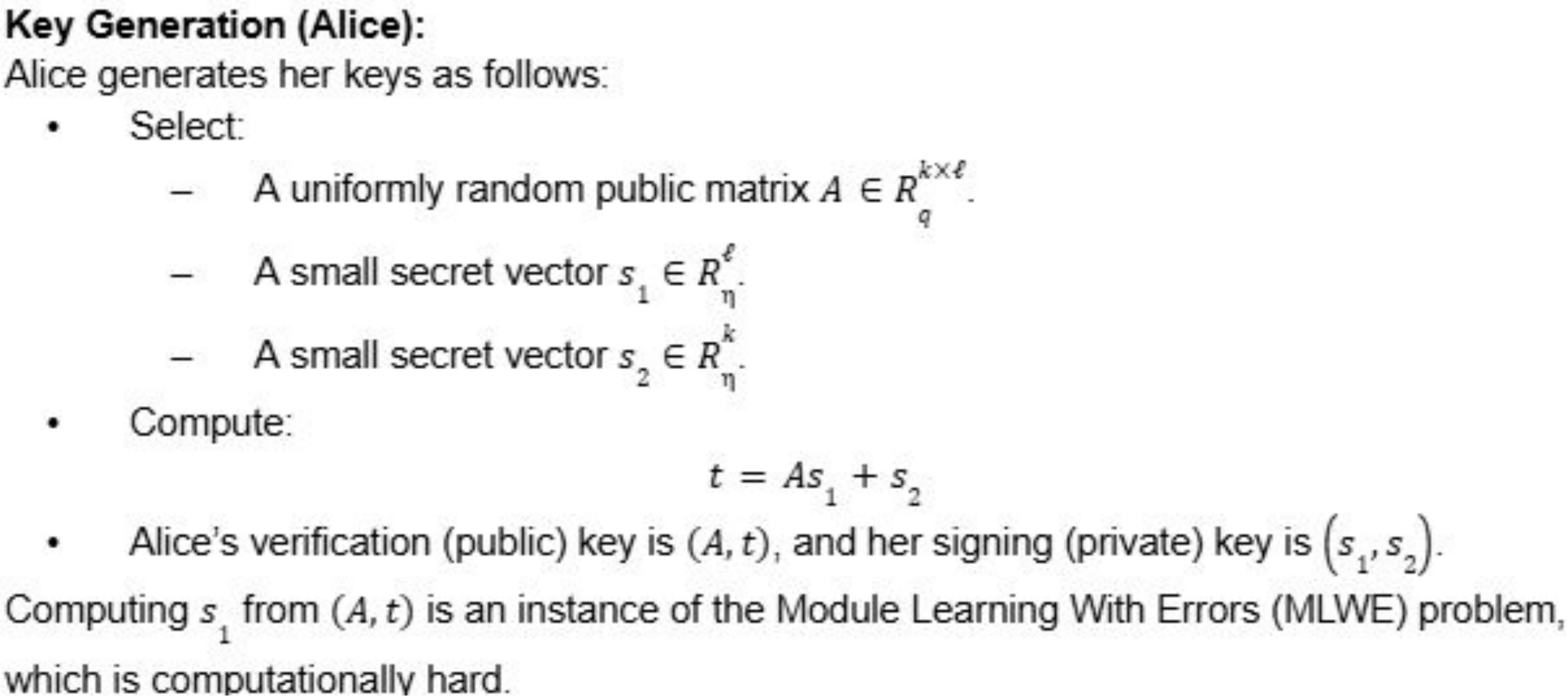
Security Level		Encapsulation Key (bytes)	Decapsulation Key (bytes)	Ciphertext (bytes)	Shared_SecretKey (bytes)
1	ML-KEM-512	800	1632	768	32
3	ML-KEM-768	1184	2400	1058	32
5	ML-KEM-1024	1568	3168	1568	32

Module-Lattice Digital Signature Algorithm (ML-DSA)

Domain Parameters:

- Field modulus: $q = 2^{23} - 2^{13} + 1$
- Dimension parameter: $n = 256$
- Matrix size: $(k, \ell) = (8, 7)$
- Secret key bounds: $\eta = 2$
- Commitment bound: $\gamma_1 = 2^{19}$
- Challenge length: $\tau = 60$
- Response bound: $\beta = \tau\eta = 120$
- Hash function: $H : \{0, 1\}^* \to B_\tau$

Module-Lattice Digital Signature Algorithm (ML-DSA)



Key Generation (Alice):

Alice generates her keys as follows:

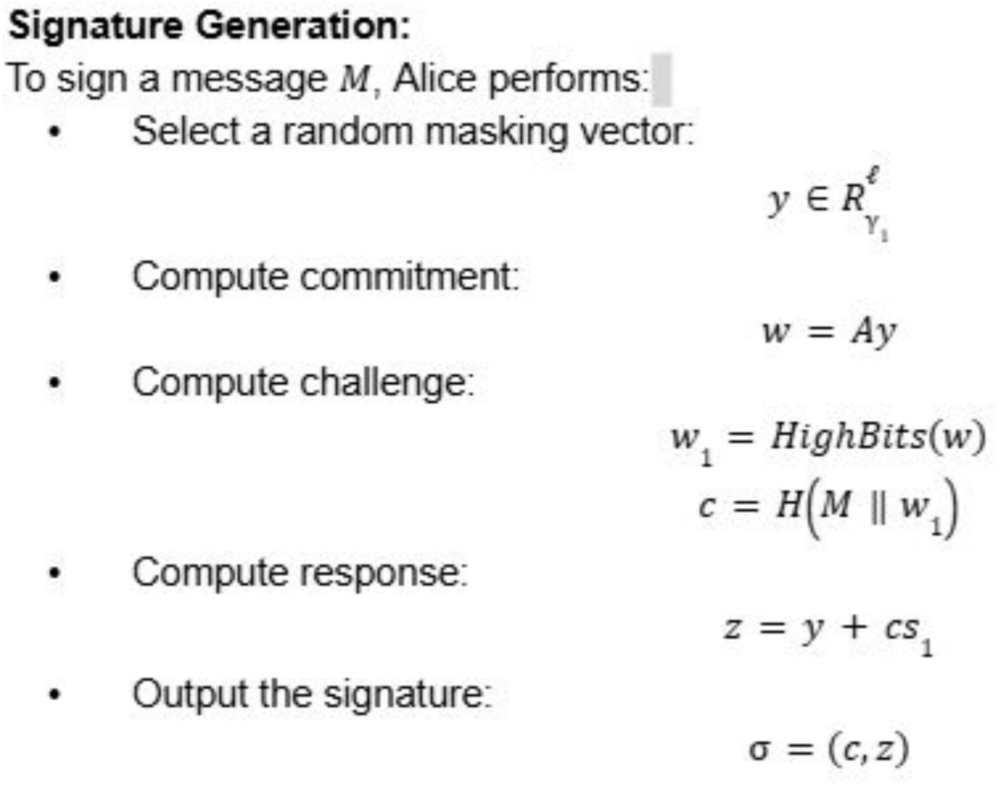
- Select:
  - A uniformly random public matrix $A \in R_q^{k\times\ell}$.
  - A small secret vector $s_1 \in R_\eta^{\ell}$.
  - A small secret vector $s_2 \in R_\eta^{k}$.
- Compute:

$$t = As_1 + s_2$$

- Alice's verification (public) key is (A, t), and her signing (private) key is (s_1, s_2).

Computing s_1 from (A, t) is an instance of the Module Learning With Errors (MLWE) problem, which is computationally hard.

Module-Lattice Digital Signature Algorithm (ML-DSA)



Signature Generation:

To sign a message M, Alice performs:

- Select a random masking vector:

$$y \in R_{\gamma_1}^{\ell}$$

- Compute commitment:

$$w = Ay$$

- Compute challenge:

$$w_1 = HighBits(w)$$
$$c = H\left(M \parallel w_1\right)$$

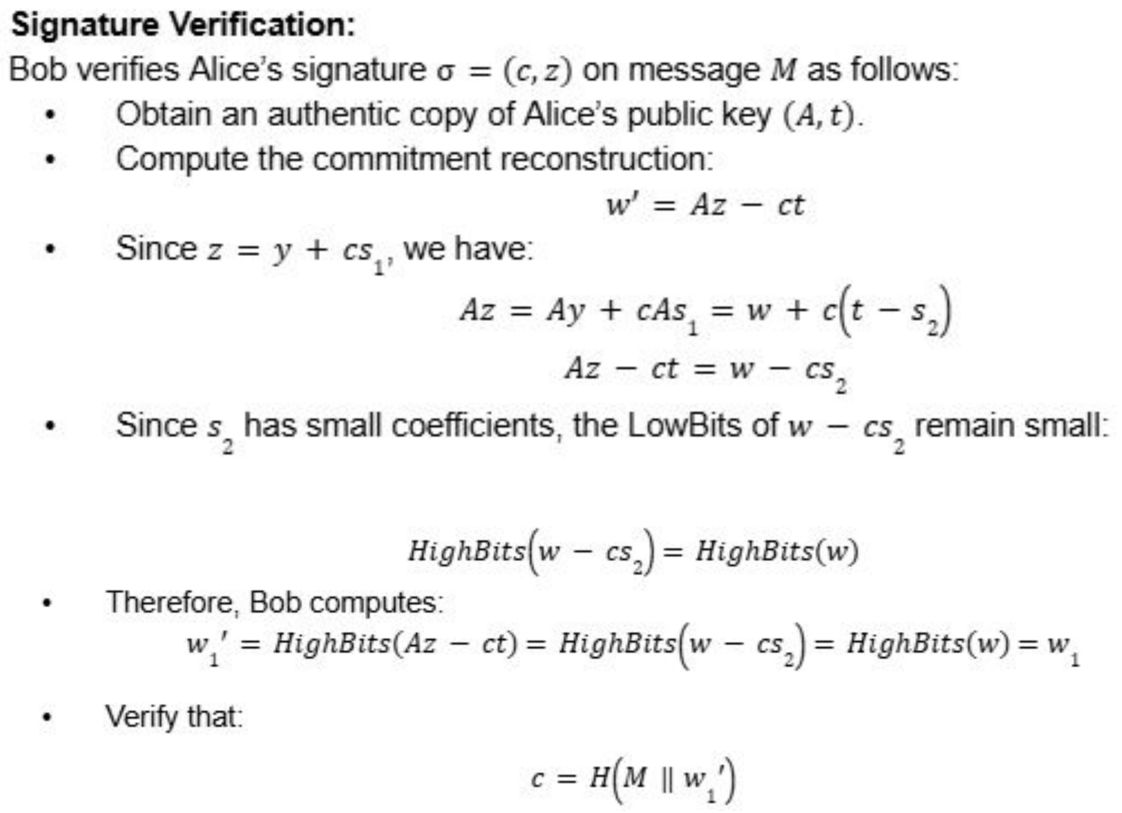
- Compute response:

$$z = y + cs_1$$

- Output the signature:

$$\sigma = (c, z)$$

Module-Lattice Digital Signature Algorithm (ML-DSA)



Signature Verification:

Bob verifies Alice's signature $\sigma = (c, z)$ on message M as follows:

- Obtain an authentic copy of Alice's public key (A, t).
- Compute the commitment reconstruction:

$$w' = Az - ct$$

- Since $z = y + cs_1$, we have:

$$Az = Ay + cAs_1 = w + c(t - s_2)$$

$$Az - ct = w - cs_2$$

- Since s_2 has small coefficients, the LowBits of $w - cs_2$ remain small:

$$HighBits(w - cs_2) = HighBits(w)$$

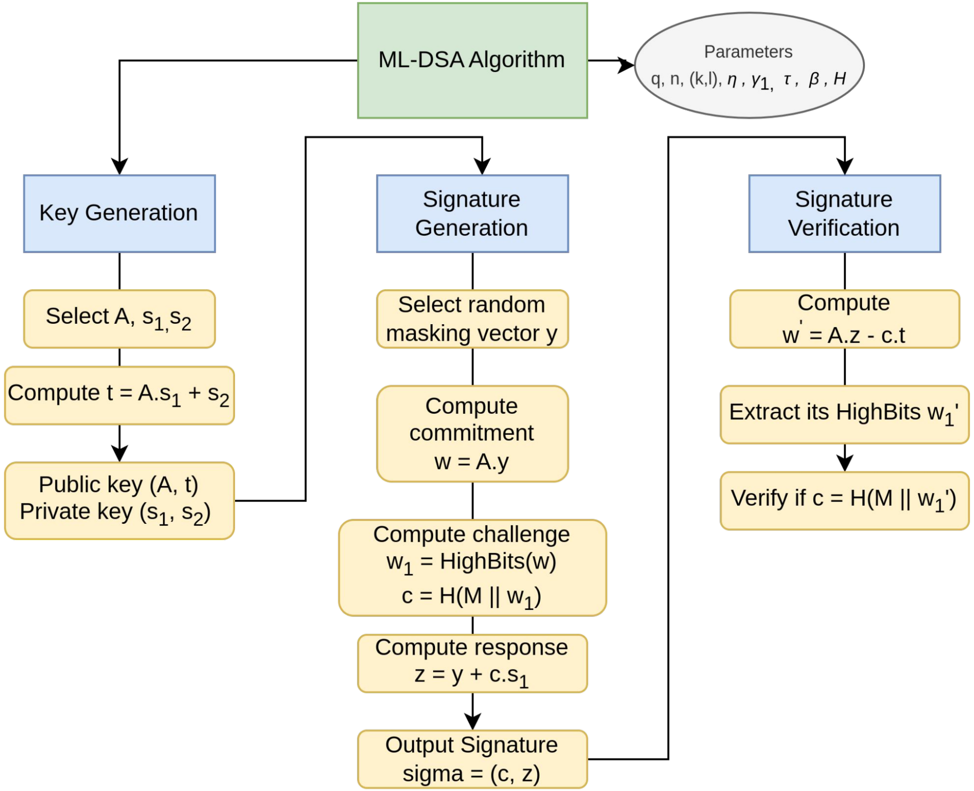
- Therefore, Bob computes:

$$w_1' = HighBits(Az - ct) = HighBits(w - cs_2) = HighBits(w) = w_1$$

- Verify that:

$$c = H(M \,\|\, w_1')$$

Module-Lattice Digital Signature Algorithm (ML-DSA)



ML-DSA Algorithm

Parameters

q, n, (k,l), η , γ_1, τ , β , H

Key Generation

Select A, $s_{1,}s_2$

Compute t = A.s_1 + s_2

Public key (A, t)
Private key (s_1, s_2)

Signature Generation

Select random masking vector y

Compute commitment
w = A.y

Compute challenge
w_1 = HighBits(w)
c = H(M || w_1)

Compute response
z = y + c.s_1

Output Signature
sigma = (c, z)

Signature Verification

Compute
w' = A.z - c.t

Extract its HighBits w_1'

Verify if c = H(M || w_1')

Module-Lattice Key Encapsulation Mechanism (ML-KEM)

Security Level		Private Key (bytes)	Public Key (bytes)	Signature Size (bytes)	Security Level
1	ML-DSA-44	2560	1312	2420	1
3	ML-DSA-65	4032	1952	3309	3
5	ML-DSA-87	4896	2592	4627	5

ML-KEM - Use cases

1. Strong Quantum Security

- Based on Module-LWE, a hard lattice problem.
- These problems are considered resistant to quantum attacks, even against large-scale quantum computers.

2. Efficiency

- Fast key generation, encapsulation, and decapsulation.
- Low memory and CPU usage compared to other PQC KEMs

3. Compact size

- ML-Kem has relatively small public keys and ciphertexts.
- Important for TLS and constrained environments.

4. Flexibility & security levels

- Comes in multiple variants: ML-KEM512, ML_KEM768, ML-KEM1024.
- We can choose the desired security level.

ML-KEM - Use cases

5. Standardized & supported

- NIST standardised this algorithm as **FIPS 203**.
- Widely supports by:
  - ➢ libOQS
  - ➢ OpenSSL
  - ➢ Browsers and servers in experimental PQC-TLS

ML-DSA - Use cases

1. String Security

- Based on Module-Lattice problems, similar to ML-KEM.
- Resistant to both Classical and quantum attacks.

2. Efficient Signing & Verification

- Fast Signature generation and verification.
- More efficient than many other PQC signature schemes like SPHINCS+.

3. Reasonable Signature SIze

- Stikes balance between size and performance.
- Signature sizes are smaller than FALCON and SPHINCS+.
- Key sizes are moderate and acceptable for most protocols like TLS or X.509 certificates.

4. Side-Channel Resistant

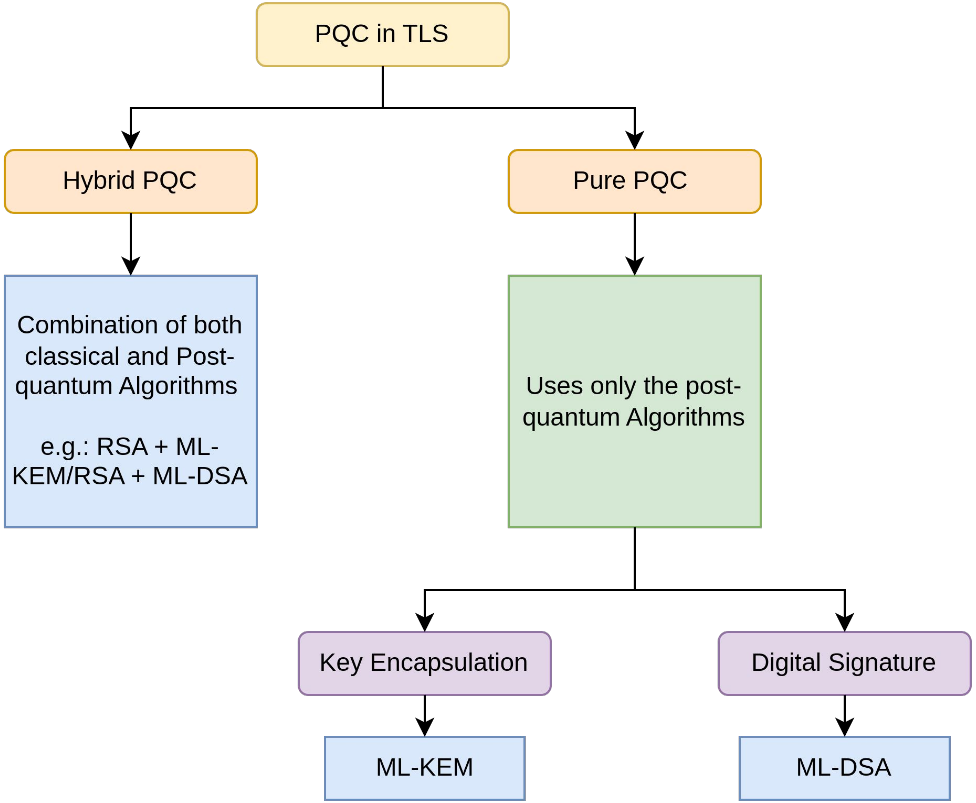
- Easier to implement in a constant-time, side-channel-resistant way than Falcon.

ML-DSA - Use cases

5. Chose as the primary Signature Standard

- NIST chose it as primary post-quantum signature scheme.
- Falcon and SPHINCS+ are also accepted but as alternatives:
  - ➢ Falcon -> very compact, but complex to implement.
  - ➢ SPHINCS+ -> stateless hash-based, but much larger and slower.

Post-Quantum Cryptography in TLS 1.3 - Approach Overview



PQC in TLS

Hybrid PQC

Pure PQC

Combination of both classical and Post-quantum Algorithms

e.g.: RSA + ML-KEM/RSA + ML-DSA

Uses only the post-quantum Algorithms

Key Encapsulation

Digital Signature

ML-KEM

ML-DSA

Hybrid Post-Quantum Cryptography in TLS

The goal is to provide security during the transition from classical cryptography (like RSA,ECC) to post-quantum cryptography (resistant to quantum computers).

- If quantum computers break classical crypto -> PQC still secures it.
- If PQC is too new or has flaws -> classical crypto still provides protection.

Using both together ensures stronger protection during this uncertain phase.

Benefits:

- Future-proofing against quantum threats.
- Stranger security during the transition period.
- Compatibility with existing systems.

Challenges:

- Larger handshake messages.
- Increased processing time.
- Needs updated software/hardware support.

Pure Post-Quantum Cryptography in TLS

Pure Post-quantum cryptography uses only the post-quantum cryptography algorithms without involving any classical cryptography algorithm.

Key encapsulation -> ML-KEM

Digital Signature -> ML-DSA

Benefits:

- Quantum resistant.
- No classical cryptography.
- Efficient, faster implementation.
- Smaller Ciphertexts/keys.

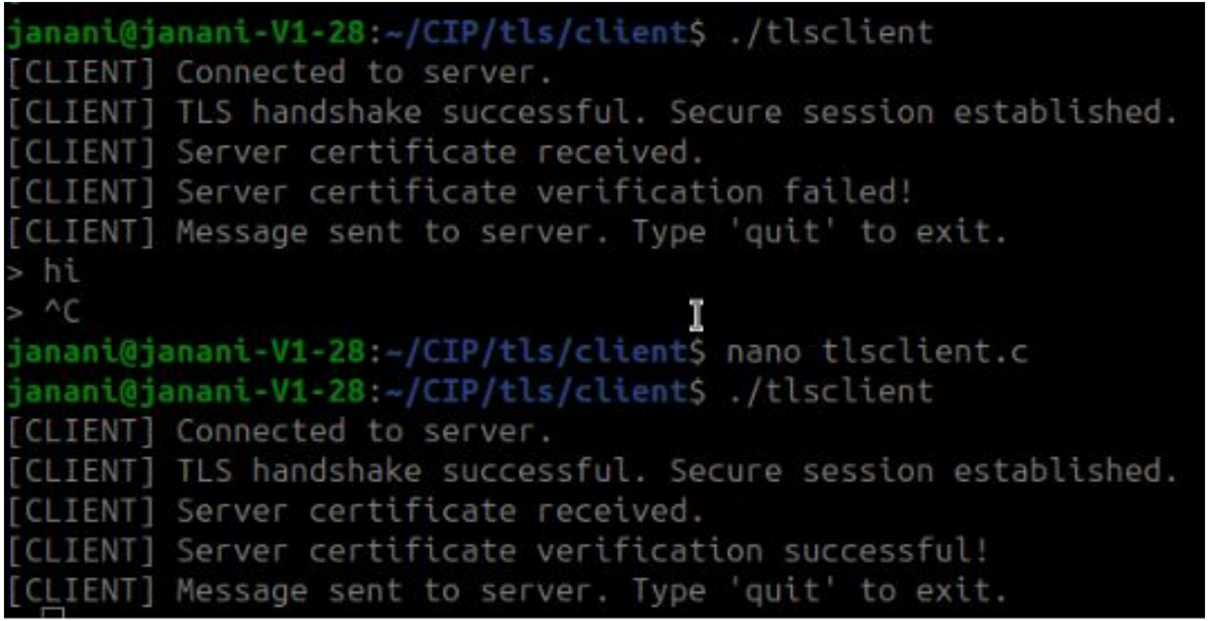
Challenges:

- Still maturing
- Adoption.
- Larger sizes than RSA/ECDSA.
- Compatibility, not all systems support PQC.

TLS Handshake - Implementation

- The TLS1.3 implementation in OpenSSL is performed in localhost by,
    - Creating separate directories for Alice, Bob, and CA.
    - Generating CA root certificate using SHA-256 and X.509 format
    - Self-signing the CA root certificate and signing the certificates of Alice and Bob
    - Loading CA certificate and initiating the handshake.
    - Performing key exchange, deriving shared secret, and thereby establishing encrypted channel.
    - Further exchanging messages over TLS.

TLS Handshake - Implementation



```
janani@janani-V1-28:~/CIP/tls/client$ ./tlsclient
[CLIENT] Connected to server.
[CLIENT] TLS handshake successful. Secure session established.
[CLIENT] Server certificate received.
[CLIENT] Server certificate verification failed!
[CLIENT] Message sent to server. Type 'quit' to exit.
> hi
> ^C
janani@janani-V1-28:~/CIP/tls/client$ nano tlsclient.c
janani@janani-V1-28:~/CIP/tls/client$ ./tlsclient
[CLIENT] Connected to server.
[CLIENT] TLS handshake successful. Secure session established.
[CLIENT] Server certificate received.
[CLIENT] Server certificate verification successful!
[CLIENT] Message sent to server. Type 'quit' to exit.
```

The overall implementation is given in this Document.

Pure PQC TLS with OpenSSL and libOQS

Rebuild OpenSSL to support PQC integration. Integrate libOQS for PQC algorithms. Implement Pure PQC based TLS handshake using OQS’s test server.

Setting up the developmental Environment

Step 1: *Install the dependencies that are needed for the project*

Install required build tools and libraries:

```
sudo apt update
sudo apt -y install git build-essential perl cmake autoconf libtool
zlib1g-dev
```

Set up workspace:

```
export WORKSPACE=~/quantumsafe
export BUILD_DIR=$WORKSPACE/build
mkdir -p $BUILD_DIR/lib64
ln -s $BUILD_DIR/lib64 $BUILD_DIR/lib
```

Pure PQC TLS with OpenSSL and libOQS

Rebuild OpenSSL to support PQC integration. Integrate libOQS for PQC algorithms. Implement Pure PQC based TLS handshake using OQS’s test server.

Step 2: ***Build OpenSSL***

Clone and build OpenSSL:

```
cd $WORKSPACE
git clone https://github.com/openssl/openssl.git
cd openssl

./Configure --prefix=$BUILD_DIR no-ssl no-tls1 no-tls1_1 no-afalgeng
no-shared threads -lm
make -j $(nproc)
make -j $(nproc) install_sw install_ssldirs
```

Pure PQC TLS with OpenSSL and libOQS

Rebuild OpenSSL to support PQC integration. Integrate libOQS for PQC algorithms. Implement Pure PQC based TLS handshake using OQS’s test server.

Step 3: ***Build libOQS***

Clone and build libOQS:

```
cd $WORKSPACE
git clone https://github.com/open-quantum-safe/liboqs.git
cd liboqs
mkdir build && cd build

cmake -DCMAKE_INSTALL_PREFIX=$BUILD_DIR -DBUILD_SHARED_LIBS=ON
-DOQS_USE_OPENSSL=OFF -DCMAKE_BUILD_TYPE=Release -DOQS_BUILD_ONLY_LIB=ON
-DOQS_DIST_BUILD=ON ..
make -j $(nproc)
make -j $(nproc) install
```

Pure PQC TLS with OpenSSL and libOQS

Rebuild OpenSSL to support PQC integration. Integrate libOQS for PQC algorithms. Implement Pure PQC based TLS handshake using OQS's test server.

Update OpenSSL configuration to support oqs-provider:

```
sed -i 's/default = default_sect/default = default_sect\noqsprovider =
oqsprovider_sect/g' $BUILD_DIR/ssl/openssl.cnf
sed -i 's/\[default_sect\]/\[oqsprovider_sect\]\nactivate = 1\n/'
$BUILD_DIR/ssl/openssl.cnf
```

Set environmental variables:

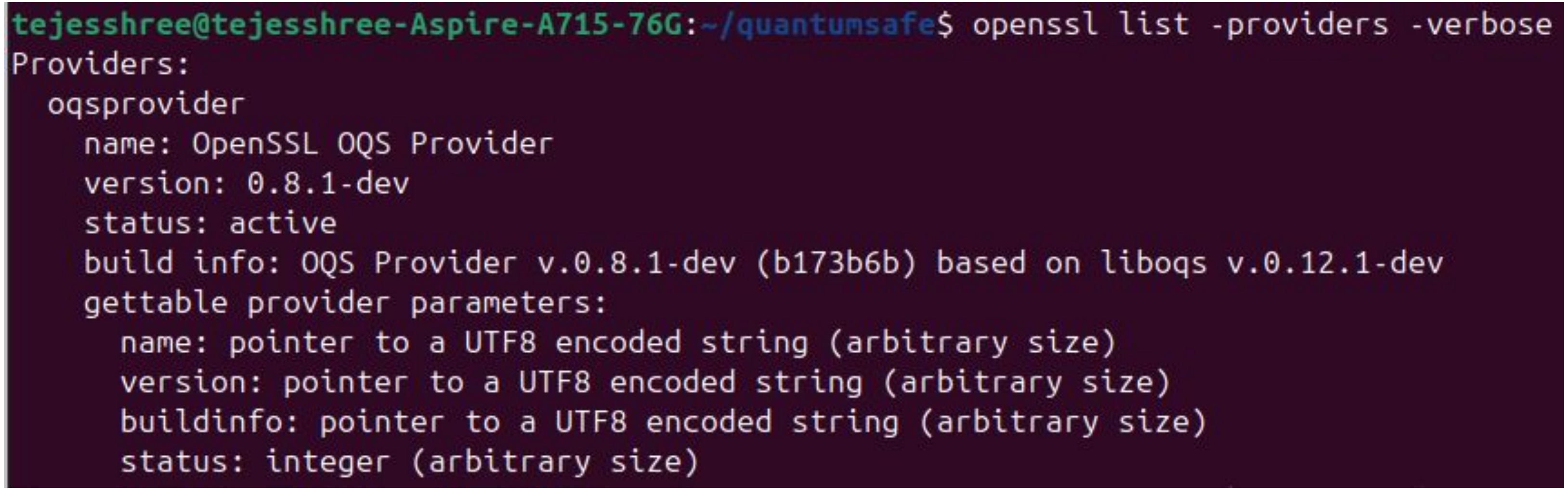
```
export OPENSSL_CONF=$BUILD_DIR/ssl/openssl.cnf
export OPENSSL_MODULES=$BUILD_DIR/lib
```

Verify installation of oqs-provider:

```
$BUILD_DIR/bin/openssl list -providers -verbose -provider oqsprovider
```

Pure PQC TLS with OpenSSL and libOQS

Rebuild OpenSSL to support PQC integration. Integrate libOQS for PQC algorithms. Implement Pure PQC based TLS handshake using OQS's test server.



```
tejesshree@tejesshree-Aspire-A715-76G:~/quantumsafe$ openssl list -providers -verbose
Providers:
  oqsprovider
    name: OpenSSL OQS Provider
    version: 0.8.1-dev
    status: active
    build info: OQS Provider v.0.8.1-dev (b173b6b) based on liboqs v.0.12.1-dev
    gettable provider parameters:
      name: pointer to a UTF8 encoded string (arbitrary size)
      version: pointer to a UTF8 encoded string (arbitrary size)
      buildinfo: pointer to a UTF8 encoded string (arbitrary size)
      status: integer (arbitrary size)
```

Pure PQC TLS with OpenSSL and libOQS

Step 5: ***Build cURL***

Clone and build cURL:

```
cd $WORKSPACE
git clone https://github.com/curl/curl.git
cd curl

autoreconf -fi
./configure \
  LIBS="-lssl -lcrypto -lz" \
  LDFLAGS="-Wl,-rpath,$BUILD_DIR/lib64 -L$BUILD_DIR/lib64
-Wl,-rpath,$BUILD_DIR/lib -L$BUILD_DIR/lib" \
  CFLAGS="-O3 -fPIC" \
  --prefix=$BUILD_DIR \
  --with-ssl=$BUILD_DIR \
  --with-zlib=/ \
  --enable-optimize --enable-libcurl-option \
  --disable-manual --without-libidn2 --without-librtmp
```

Pure PQC TLS with OpenSSL and libOQS

```
make -j $(nproc)
make -j $(nproc) install
```

Step 6: ***Test Quantum-Safe TLS***

The CA certificate for the Open Quantum Safe (OQS) test server hosted at: http://test.openquantumsafe.org

Download CA Certificate of the test server:

```
$BUILD_DIR/bin/curl -vk https://test.openquantumsafe.org/CA.crt --output $BUILD_DIR/ca.cert
```

`--output $BUILD_DIR/ca.cert` → Saves the certificate locally as ca.cert in the build directory.

Pure PQC TLS with OpenSSL and libOQS

Test a quantum-safe TLS handshake with a server of specified port number using post-quantum key exchange and signature algorithm:

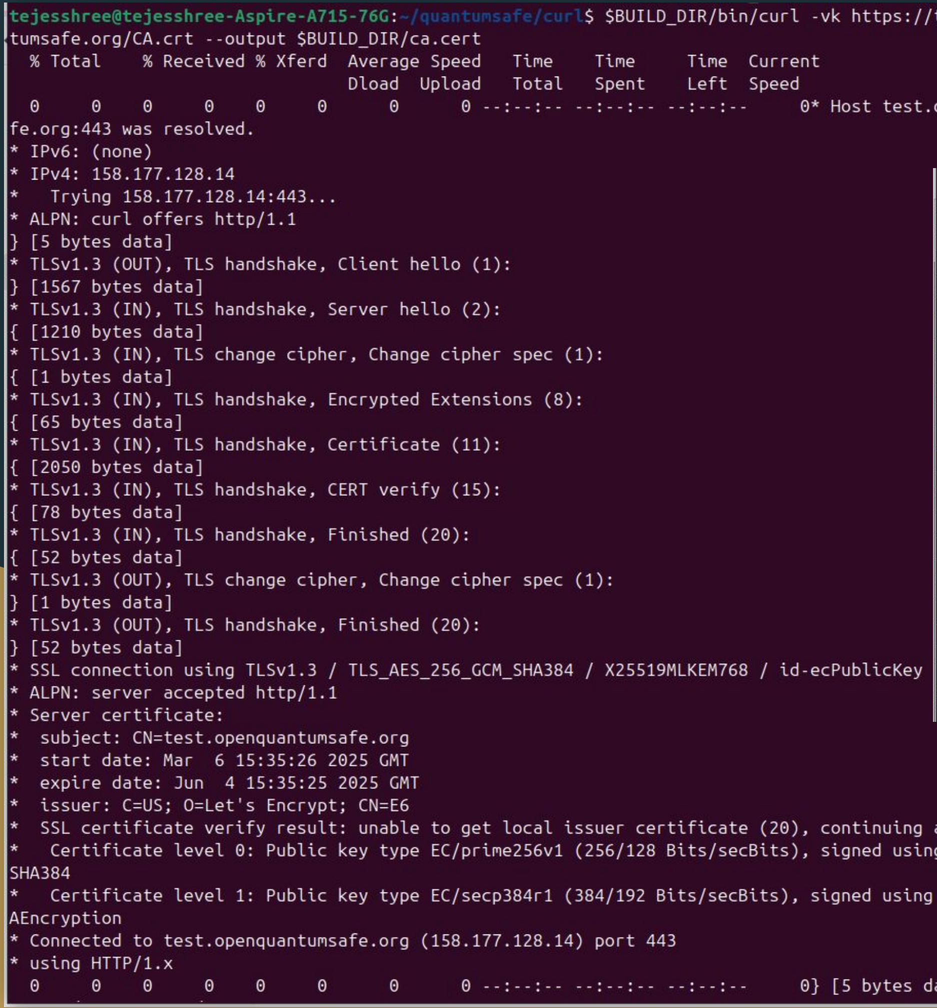
```
$BUILD_DIR/bin/curl -v --curves MLKEM512 --cacert $BUILD_DIR/ca.cert
https://test.openquantumsafe.org:6346/
```

--curves MLKEM512 → Specifies the key exchange algorithm.

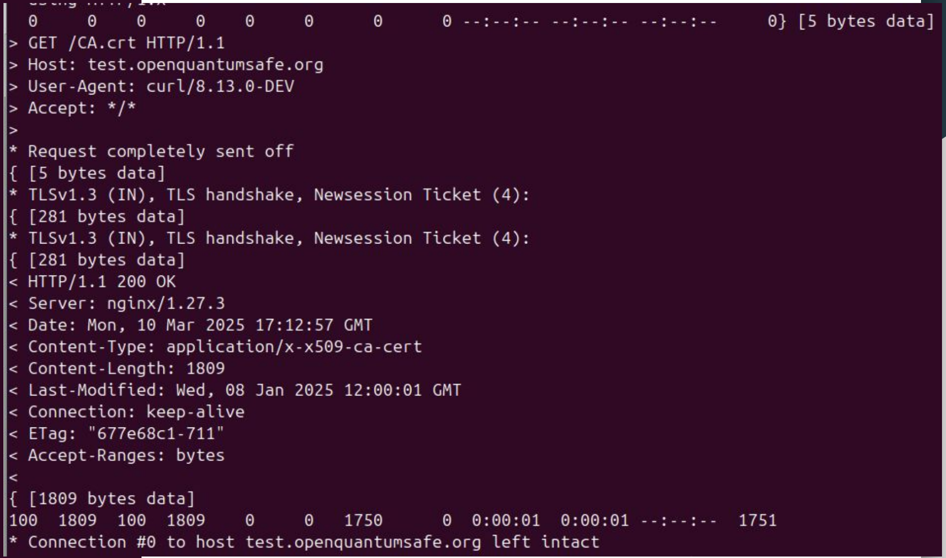
https://test.openquantumsafe.org:6346/ →Connects to the OQS test server running on port **6346**, which supports partially quantum-safe key exchange and signature. (Root CA is still signed by RSA only)

The output Document is given here.

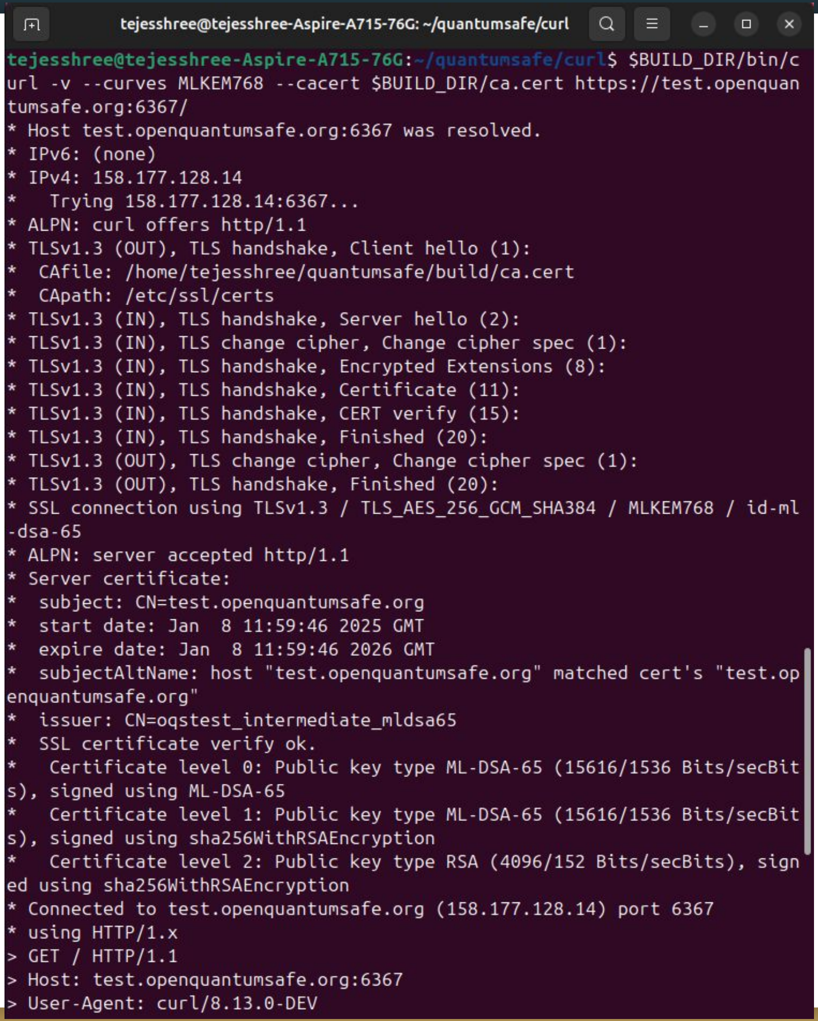
Downloaded CA certificate



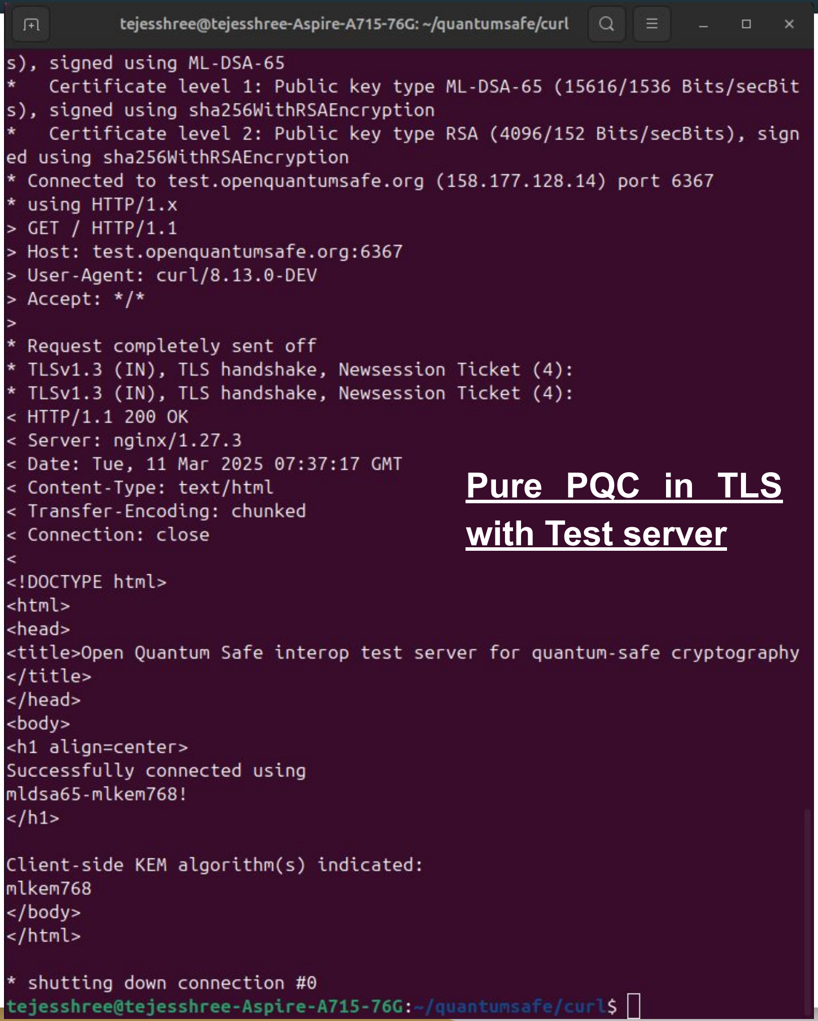
```
tejesshree@tejesshree-Aspire-A715-76G:~/quantumsafe/curl$ $BUILD_DIR/bin/curl -vk https://test.openquan
tumsafe.org/CA.crt --output $BUILD_DIR/ca.cert
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0* Host test.openquantumsa
fe.org:443 was resolved.
* IPv6: (none)
* IPv4: 158.177.128.14
*   Trying 158.177.128.14:443...
* ALPN: curl offers http/1.1
} [5 bytes data]
* TLSv1.3 (OUT), TLS handshake, Client hello (1):
} [1567 bytes data]
* TLSv1.3 (IN), TLS handshake, Server hello (2):
{ [1210 bytes data]
* TLSv1.3 (IN), TLS change cipher, Change cipher spec (1):
{ [1 bytes data]
* TLSv1.3 (IN), TLS handshake, Encrypted Extensions (8):
{ [65 bytes data]
* TLSv1.3 (IN), TLS handshake, Certificate (11):
{ [2050 bytes data]
* TLSv1.3 (IN), TLS handshake, CERT verify (15):
{ [78 bytes data]
* TLSv1.3 (IN), TLS handshake, Finished (20):
{ [52 bytes data]
* TLSv1.3 (OUT), TLS change cipher, Change cipher spec (1):
} [1 bytes data]
* TLSv1.3 (OUT), TLS handshake, Finished (20):
} [52 bytes data]
* SSL connection using TLSv1.3 / TLS_AES_256_GCM_SHA384 / X25519MLKEM768 / id-ecPublicKey
* ALPN: server accepted http/1.1
* Server certificate:
*  subject: CN=test.openquantumsafe.org
*  start date: Mar  6 15:35:26 2025 GMT
*  expire date: Jun  4 15:35:25 2025 GMT
*  issuer: C=US; O=Let's Encrypt; CN=E6
*  SSL certificate verify result: unable to get local issuer certificate (20), continuing anyway.
*   Certificate level 0: Public key type EC/prime256v1 (256/128 Bits/secBits), signed using ecdsa-with-
SHA384
*   Certificate level 1: Public key type EC/secp384r1 (384/192 Bits/secBits), signed using sha256WithRS
AEncryption
* Connected to test.openquantumsafe.org (158.177.128.14) port 443
* using HTTP/1.x
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0} [5 bytes data]
```



```
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0} [5 bytes data]
> GET /CA.crt HTTP/1.1
> Host: test.openquantumsafe.org
> User-Agent: curl/8.13.0-DEV
> Accept: */*
>
* Request completely sent off
{ [5 bytes data]
* TLSv1.3 (IN), TLS handshake, Newsession Ticket (4):
{ [281 bytes data]
* TLSv1.3 (IN), TLS handshake, Newsession Ticket (4):
{ [281 bytes data]
< HTTP/1.1 200 OK
< Server: nginx/1.27.3
< Date: Mon, 10 Mar 2025 17:12:57 GMT
< Content-Type: application/x-x509-ca-cert
< Content-Length: 1809
< Last-Modified: Wed, 08 Jan 2025 12:00:01 GMT
< Connection: keep-alive
< ETag: "677e68c1-711"
< Accept-Ranges: bytes
<
{ [1809 bytes data]
100  1809  100  1809    0     0   1750      0  0:00:01  0:00:01 --:--:--  1751
* Connection #0 to host test.openquantumsafe.org left intact
```



```
tejesshree@tejesshree-Aspire-A715-76G:~/quantumsafe/curl$ $BUILD_DIR/bin/c
url -v --curves MLKEM768 --cacert $BUILD_DIR/ca.cert https://test.openquan
tumsafe.org:6367/
* Host test.openquantumsafe.org:6367 was resolved.
* IPv6: (none)
* IPv4: 158.177.128.14
*   Trying 158.177.128.14:6367...
* ALPN: curl offers http/1.1
* TLSv1.3 (OUT), TLS handshake, Client hello (1):
*  CAfile: /home/tejesshree/quantumsafe/build/ca.cert
*  CApath: /etc/ssl/certs
* TLSv1.3 (IN), TLS handshake, Server hello (2):
* TLSv1.3 (IN), TLS change cipher, Change cipher spec (1):
* TLSv1.3 (IN), TLS handshake, Encrypted Extensions (8):
* TLSv1.3 (IN), TLS handshake, Certificate (11):
* TLSv1.3 (IN), TLS handshake, CERT verify (15):
* TLSv1.3 (IN), TLS handshake, Finished (20):
* TLSv1.3 (OUT), TLS change cipher, Change cipher spec (1):
* TLSv1.3 (OUT), TLS handshake, Finished (20):
* SSL connection using TLSv1.3 / TLS_AES_256_GCM_SHA384 / MLKEM768 / id-ml
-dsa-65
* ALPN: server accepted http/1.1
* Server certificate:
*  subject: CN=test.openquantumsafe.org
*  start date: Jan  8 11:59:46 2025 GMT
*  expire date: Jan  8 11:59:46 2026 GMT
*  subjectAltName: host "test.openquantumsafe.org" matched cert's "test.op
enquantumsafe.org"
*  issuer: CN=oqstest_intermediate_mldsa65
*  SSL certificate verify ok.
*   Certificate level 0: Public key type ML-DSA-65 (15616/1536 Bits/secBit
s), signed using ML-DSA-65
*   Certificate level 1: Public key type ML-DSA-65 (15616/1536 Bits/secBit
s), signed using sha256WithRSAEncryption
*   Certificate level 2: Public key type RSA (4096/152 Bits/secBits), sign
ed using sha256WithRSAEncryption
* Connected to test.openquantumsafe.org (158.177.128.14) port 6367
* using HTTP/1.x
> GET / HTTP/1.1
> Host: test.openquantumsafe.org:6367
> User-Agent: curl/8.13.0-DEV
```



```
s), signed using ML-DSA-65
*   Certificate level 1: Public key type ML-DSA-65 (15616/1536 Bits/secBit
s), signed using sha256WithRSAEncryption
*   Certificate level 2: Public key type RSA (4096/152 Bits/secBits), sign
ed using sha256WithRSAEncryption
* Connected to test.openquantumsafe.org (158.177.128.14) port 6367
* using HTTP/1.x
> GET / HTTP/1.1
> Host: test.openquantumsafe.org:6367
> User-Agent: curl/8.13.0-DEV
> Accept: */*
>
* Request completely sent off
* TLSv1.3 (IN), TLS handshake, Newsession Ticket (4):
* TLSv1.3 (IN), TLS handshake, Newsession Ticket (4):
< HTTP/1.1 200 OK
< Server: nginx/1.27.3
< Date: Tue, 11 Mar 2025 07:37:17 GMT
< Content-Type: text/html
< Transfer-Encoding: chunked
< Connection: close
<
<!DOCTYPE html>
<html>
<head>
<title>Open Quantum Safe interop test server for quantum-safe cryptography
</title>
</head>
<body>
<h1 align=center>
Successfully connected using
mldsa65-mlkem768!
</h1>

Client-side KEM algorithm(s) indicated:
mlkem768
</body>
</html>

* shutting down connection #0
tejesshree@tejesshree-Aspire-A715-76G:~/quantumsafe/curl$
```

Pure PQC in TLS with Test server

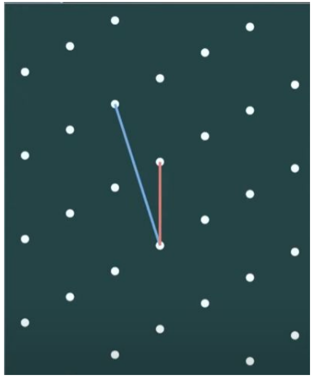
Lattice-based cryptography

- ❑ Lattice-based Cryptography uses the mathematical properties of lattices to create secure encryption schemes.
- ❑ Idea: To create problems that are computationally hard to solve, by leveraging geometric complexity and high-dimensional nature of lattices.

Key features includes:

- Post-quantum security: secure against quantum attacks.
- Efficiency: encryption and decryption operation more flexible than traditional cryptographic systems.
- Hardness Assumption: Difficult to solve even with quantum computers.

Lattice-based cryptography

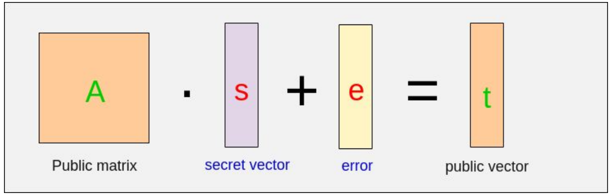


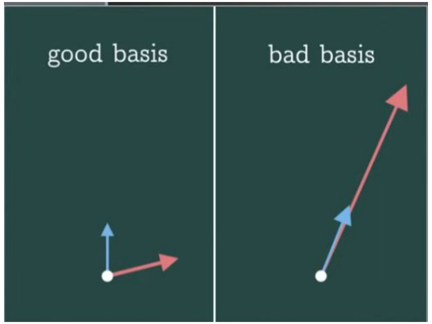
Hard Problem in Lattice Based cryptography:

❑ **Shortest Vector Problem (SVP)**: To find the lattice point closest to but not equal to the origin, Secure foundation for LWE, Attacks on cryptanalysis protocols often involves SVP.

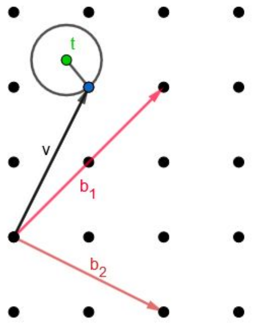
❑ **Learning with Error (LWE):** solves a system of linear equation where small errors are added, difficult to determine the original noise.

$$A\,.\,s + e = t$$





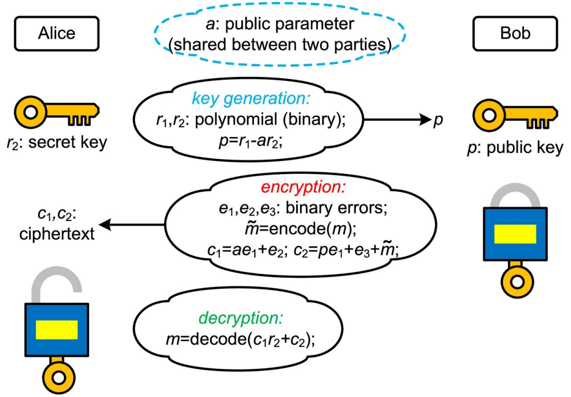
Lattice-based cryptography



Hard Problem in Lattice Based cryptography:

- ❑ **Closest Vector Problem (CVP)**: To find the nearest lattice point to a given target point in space, Break encryption by finding the close lattice point
- ❑ **Ring Learning with Error (Ring-LWE)**: More Structured version of LWE that operates over polynomial rings instead of vectors.

$$R_q = Z[x] / (f(x)) \bmod q$$



Alice

a: public parameter (shared between two parties)

Bob

key generation: r_1,r_2: polynomial (binary); $p=r_1-ar_2$; → *p*

r_2: secret key

p: public key

encryption: e_1,e_2,e_3: binary errors; $\tilde{m}$=encode(*m*); $c_1=ae_1+e_2$; $c_2=pe_1+e_3+\tilde{m}$;

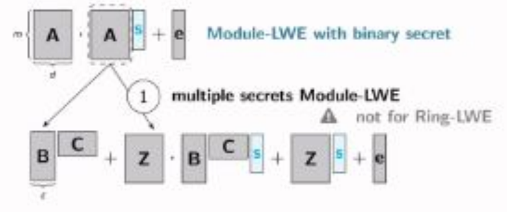
c_1,c_2: ciphertext

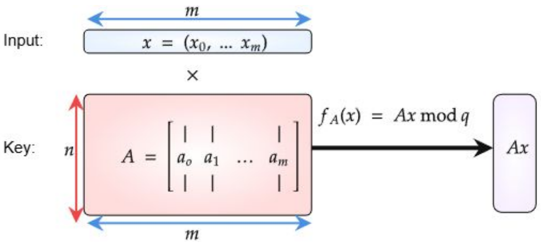
decryption: *m*=decode($c_1r_2+c_2$);

Lattice-based cryptography

Hard Problem in Lattice Based cryptography:

- ❑ **Module Learning with Error (Module-LWE)**: Generalization of RLWE, balancing efficient and flexibility, trade-off between security and performance compared to RLWE.
- ❑ **Short Integer Solution (SIS)**: to find a short nonzero vector that satisfies a linear equation modulo q, Bais for lattice-based digital signatures, Hash-and-sign cryptographic constructions. It is closely related to SVP.





m
Input:
$x = (x_0, \dots x_m)$
×
Key:
n
$A = \begin{bmatrix} | & | & & | \\ a_o & a_1 & \dots & a_m \\ | & | & & | \end{bmatrix}$
m
$f_A(x) = Ax \bmod q$
Ax

FALCON

- ❑ Falcon designed for high security and efficiency, selected by NIST due to its compact signature and fast verification.

- ❑ It is based on NTRU (N-th Degree Truncated Polynomial Ring) problem and Gentry-peikert Vaikuntanathan (GVP) framework. It is very difficult to solve, even by Quantum computers, making Falcon secure against Quantum Attacks.

- ❑ Digital Signature Working:
  - Key Generation: Create public and private keys.
  - Signing: Use the private key to generate a signature for a message.
  - Verification: Anyone can verify the signature using the public key.

- ❑ It is very fast in verification, smallest signature size than Dilithium, and Mathematically efficient.

SPHINCS+

❑ Sphincs+ is based on hash-based cryptography, NIST selected it as a backup option for post-quantum digital signatures, ensuring a alternative in case lattice-based methods like Dilithium, kyber face unexpected vulnerabilities.

❑ Digital Signature working:

- Key generation: Create Public and private key using Cryptographic hash functions.
- Signing: Generate a signature using a tree-based structure.
- Verification: Anyone can check the signature using the public key.

❑ Stateless Design, does not require state information between signatures, simplifying implementation and reducing vulnerabilities.

❑ It has Larger signature sizes and slower signing speeds compared to lattice-based algorithm.

References

[1] R. Rios, J. A. Montenegro, A. Muñoz and D. Ferraris (2025), "Towards the Quantum-Safe Web: Benchmarking Post-Quantum TLS." in IEEE Network

[2] E. Kupcova, J. Simko, M. Pleva and M. Drutarovsky (2024), "Experimental Framework for Secure Post-Quantum TLS Client-Server Communication", 2024 International Symposium ELMAR, Zadar, Croatia, 2024.

[3] R. Döring and M. Geitz (2022), "Post-Quantum Cryptography in Use: Empirical Analysis of the TLS Handshake Performance", NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium, Budapest, Hungary, 2022.

[4] A. C. H. Chen (2024), "Post-Quantum Cryptography X.509 Certificate," 2024 International Conference on Smart Systems for applications in Electrical Sciences (ICSSES), Tumakuru, India, 2024.

[5] Fan, Jinnan & Willems, Fabian & Zahed, Jafar & Gray, John & Mister, Serge & Ounsworth, Mike & Adams, Carlisle. (2021). “Impact of post-quantum hybrid certificates on PKI, common libraries, and protocols.” International Journal of Security and Networks.

References

[6] Giron, Alexandre & Custódio, Ricardo & Rodríguez-Henríquez, Francisco. (2022). “Post-quantum hybrid key exchange: a systematic mapping study.” Journal of Cryptographic Engineering.

[7] Nouri Alnahawi, Johannes Müller, Jan Oupický, and Alexander Wiesmaier (2024), “A Comprehensive Survey on Post-Quantum TLS” IACR Communications in Cryptology, vol. 1, no. 2, Jul 08, 2024.

[8] J. Sowa et al. (2024), "Post-Quantum Cryptography (PQC) Network Instrument: Measuring PQC Adoption Rates and Identifying Migration Pathways," 2024 IEEE International Conference on Quantum Computing and Engineering (QCE), Montreal, QC, Canada, 2024.

[9] Panos Kampanakis and Michael Kallitsis (2022), “Speeding Up Post-Quantum TLS handshakes by Suppressing Intermediate CA Certificates”, In Cyber Security, Cryptology, and Machine Learning: 6th International Symposium, CSCML 2022, Be'er Sheva, Israel, June 30 – July 1, 2022, Proceedings. Springer-Verlag, Berlin, Heidelberg, 337–355.

References

[10] M. Raavi, P. Chandramouli, S. Wuthier, X. Zhou and S. -Y. Chang (2021), "Performance Characterization of Post-Quantum Digital Certificates," 2021 International Conference on Computer Communications and Networks (ICCCN), Athens, Greece, 2021.

[11] OpenSSL github: https://github.com/openssl/openssl.git.

[12] libOQS github: https://github.com/open-quantum-safe/liboqs.git.

[13] Integration and testing github: https://gist.github.com/ajbozarth/65aee2084f7bc089704b4e7afb54801e.

[14] Test Server by Open Quantum Safe Provider (OQS-Provider) reference: https://test.openquantumsafe.org/?utm_source=ibm_developer&utm_content=in_content_link&utm_id=tutorials_awb-quantum-safe-openssl

Thank You