**Review 3 (16.04.2025) Team A12**

**CS6611 - CREATIVE AND INNOVATIVE PROJECT**

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

**Team:** Tejesshree S (2022503524), Janani A (2022503502), Kathirvel M (2022503060).

**Mentor:** Dr. R. Gunasekaran

**DOMAIN:** Post-Quantum Cryptography

**PROBLEM STATEMENT:**

Classical cryptographic algorithms like RSA, ECC, and DHMKEX are used in network protocols such as TLS and SSH. However, quantum algorithms like Shor’s and Grover’s can break these classical cryptographic methods. The computational hardness that secures classical cryptography is vulnerable to quantum computing advancements, creating a need for cryptographic algorithms based on hard mathematical problems that even quantum computers cannot solve. TLS (Transport Layer Security), the backbone security protocol of the Internet, must adopt quantum-secure cryptography to remain resilient. Additionally, “Harvest now, Decrypt later” threats make post-quantum cryptography urgent to protect sensitive data from future quantum decryption.

**OBJECTIVE:**

* To implement Pure Post-Quantum Cryptography in TLS 1.3 by integrating Post-Quantum Cryptographic primitives for Key Exchange and Authentication
* To develop a test environment that simulates real-world network conditions for benchmarking PQC TLS performance.
* To assess the security benefits, computational overhead and compatibility issues of Pure PQC adoption in TLS.

**MODULES:**

1. **Study of PQC Algorithms and TLS Handshake:**

* Conduct a study on PQC algorithms - ML-KEM and ML-DSA, along with an analysis of TLS handshake flow.

1. **Classical cryptography implementation with OpenSSL:**

* Experiment with OpenSSL by implementing classical cryptographic algorithms such as RSA, AES, ECDH and TLS.

1. **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.

1. **Building Custom test environment:**

* Develop a custom test server with NGINX with Root CA signed using MLDSA algorithm.

1. **Performance evaluation and benchmarking:**

* Measure Handshake time, Round trip time, Certificate sizes and Key exchange length during TLS with various modes.

**Literature Survey:**

| **S.no** | **Paper title, Author and Publication** | **Methodology** | **Limitations** |
| --- | --- | --- | --- |
| 1 | Towards the Quantum-Safe Web:  Benchmarking Post-Quantum TLS  *IEEE Network, 2025* | - Framework setup uses OpenSSL for TLS connections, libOQS and oqs-provider for PQC primitives  - Implements dockerized environment for controlled testing  - Handshake efficiency and data transmission overhead are evaluated  - ML-KEM exhibits comparable performance to traditional algorithms, FALCON is the most efficient signature scheme and SPHINCS is slower due to its hash structure | - Focus is only on lab-based testing rather than real-world web traffic scenarios  - Does not consider any hybrid TLS configurations |
| 2 | Experimental Framework for Secure Post-Quantum TLS Client-Server Communication  International Symposium ELMAR,2024 | - Stated that integrating the new algorithm requires adjustments in identification numbers, and their support by certification authorities necessitates internal system changes.  - Successfully tested the client connection on a public testing server that is part of the Open Quantum Safe project.  - Certificate verification times are comparable. | - The problem lies specifically in the key size.  - Differences across computers with varying computational power underscore the challenge of integrating PQC algorithms into the TLS protocol, highlighting the need for security analysis and speed optimization. |
| 3 | Post-Quantum Cryptography in Use: Empirical Analysis of the TLS Handshake Performance  IEEE/IFIP Network Operations and Management Symposium, 2022 | - The Security levels are compared to the difficulty to break classical encryption or hashing schemes with appropriate key length.  - Saber or CRYSTAL\_KYBER for key exchange together with FALCON signature scheme seems to be a performant choice for further prototyping with Post-Quantum Cryptography.  - Saber has the highest decapsulation and encapsulation rate followed by CRYSTALS-KYBER and HQC | - stated that the significant difference between classical and PQC algorithms is not visible.  - No FALCON variant for level 3.  - Only the performance results are given, no information on experiment setup. |
| 4 | [Post-Quantum Cryptography X.509 Certificate](https://drive.google.com/file/d/16XPQrbKbnp4y5TxVJ_YBynFw4G7IA1XM/view?usp=sharing)    2024 International Conference on Smart Systems for applications in Electrical Sciences (ICSSES) | Analyzes RSA/ECC vulnerabilities to quantum threats, integrating PQC into X.509 certificates. Case studies on Falcon, Dilithium, and Sphincs assess certificate size, generation time, and verification efficiency. | TLS integration untested; simulations may not reflect real-world constraints. |

**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. |

**Module 1: Study of PQC Algorithms and TLS Handshake:**

Conduct a study on PQC algorithms - ML-KEM and ML-DSA, along with an analysis of TLS handshake flow.

**TLS 1.3:**

* 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

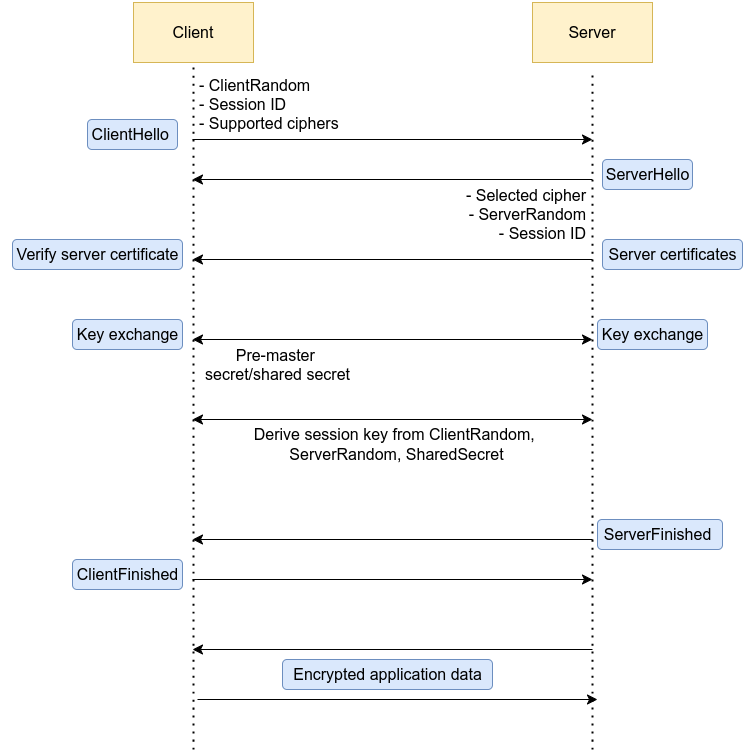


Fig 1. TLS Handshake

**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

**Step 3: Decapsulation (Bob)**

Bob recovers the shared secret from ciphertext sent by Alice.

* Using Bob’s secret key S, he computes:
  + V’ = ST.U mod 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:

**Key Generation:**

Alice does:

1. Select , , and .
2. Compute:
3. Alice’s encryption (public) key is ; her decryption (private) key is .

Note**:** Computing from is an instance of MLWE.

**Encryption:**

To encrypt a message for Alice, Bob does:

1. Obtain an authentic copy of Alice’s encryption key .
2. Select , , and .
3. Compute:
4. Output the ciphertext:

Note: .

**Decryption:**

To decrypt , Alice does:

1. Compute:

Note: Alice uses her decryption key .

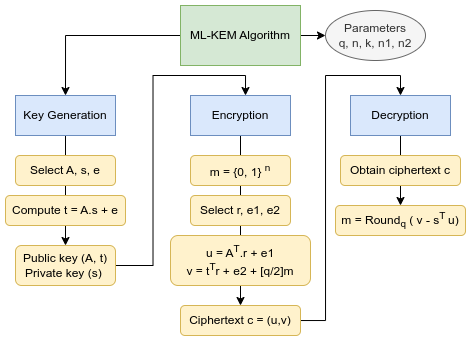


Fig 2. ML-KEM Algorithm

**ML-DSA:**

**Domain Parameters:**

* Field modulus: *q* = 223 − 213 + 1
* Dimension parameter: *n* = 256
* Matrix size: (*k, ℓ*) = (8*,* 7)
* Secret key bounds: *η* = 2
* Commitment bound: *γ*1 = 219
* Challenge length: *τ* = 60
* Response bound: *β* = *τη* = 120
* Hash function: *H* : {0*,* 1}∗ → *Bτ*

**Key Generation (Alice):**

Alice generates her keys as follows:

* Select:
  + A uniformly random public matrix .
  + A small secret vector .
  + A small secret vector .
* Compute:
* Alice’s verification (public) key is , and her signing (private) key is .

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

**Signature Generation:**

To sign a message , Alice performs:

* Select a random masking vector:
* Compute commitment:
* Compute challenge:
* Compute response:
* Output the signature:

**Signature Verification:**

Bob verifies Alice’s signature on message as follows:

* Obtain an authentic copy of Alice’s public key .
* Compute the commitment reconstruction:
* Since , we have:
* Since has small coefficients, the LowBits of remain small:
* Therefore, Bob computes:
* Verify that:

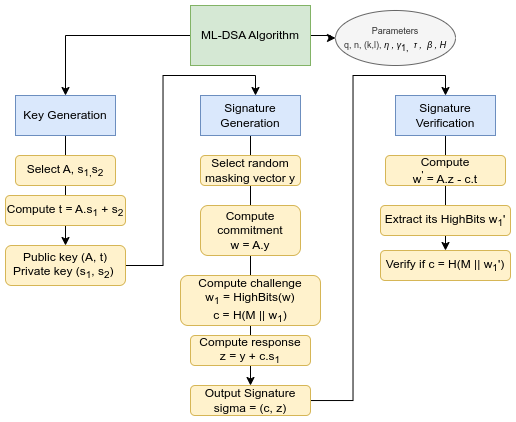


Fig 3. ML-DSA Algorithm

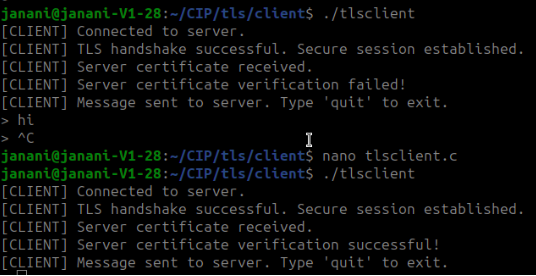
**Module 2: Classical cryptography implementation with OpenSSL:**

**Goal:** Experiment with OpenSSL by implementing classical cryptographic algorithms such as RSA, AES, ECDH and TLS.

Implemented RSA, AES, and ECDH using OpenSSL, including key generation, CSR creation, and self-signing.

**TLS:**

* 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.



**Module 3: 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.

**3.1. 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

**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

**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

**Step 4: Install OQS Provider**

Clone and build the Open Quantum Safe Provider:

cd $WORKSPACE

git clone https://github.com/open-quantum-safe/oqs-provider.git

cd oqs-provider

liboqs\_DIR=$BUILD\_DIR cmake -DCMAKE\_INSTALL\_PREFIX=$WORKSPACE/oqs-provider -DOPENSSL\_ROOT\_DIR=$BUILD\_DIR -DCMAKE\_BUILD\_TYPE=Release -S . -B \_build

cmake --build \_build

# Copy provider files to OpenSSL build

cp \_build/lib/\* $BUILD\_DIR/lib/

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

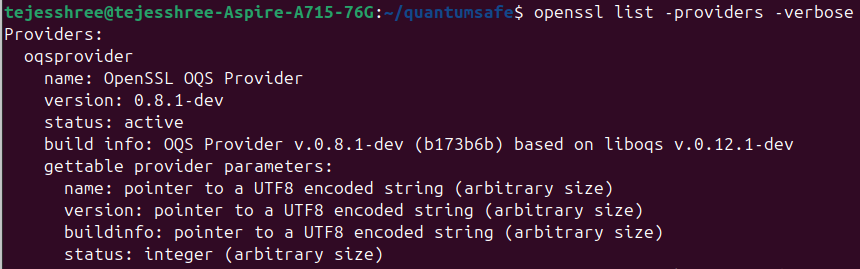
Set environmental variables:

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



**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

make -j $(nproc)

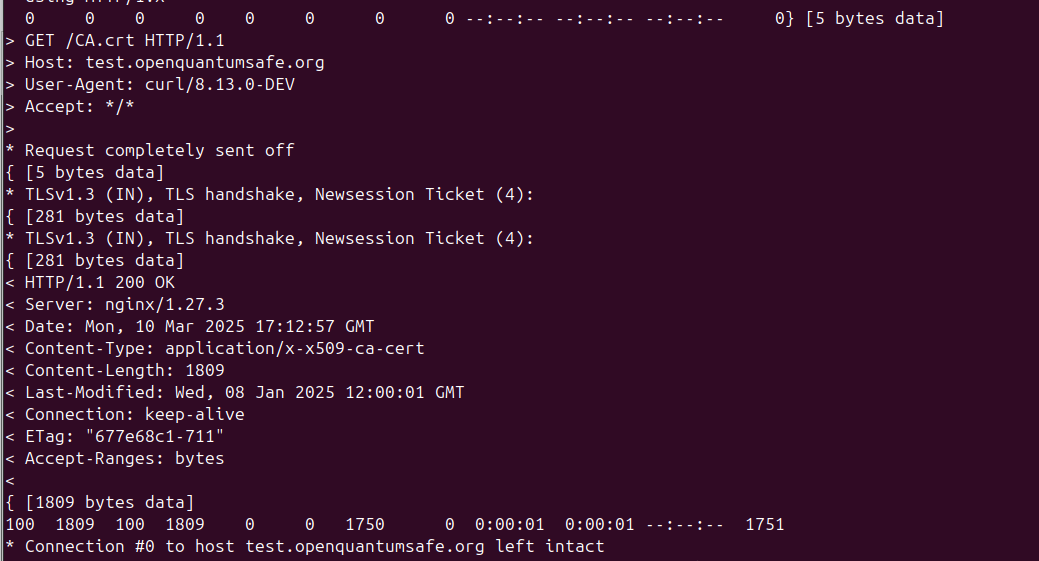
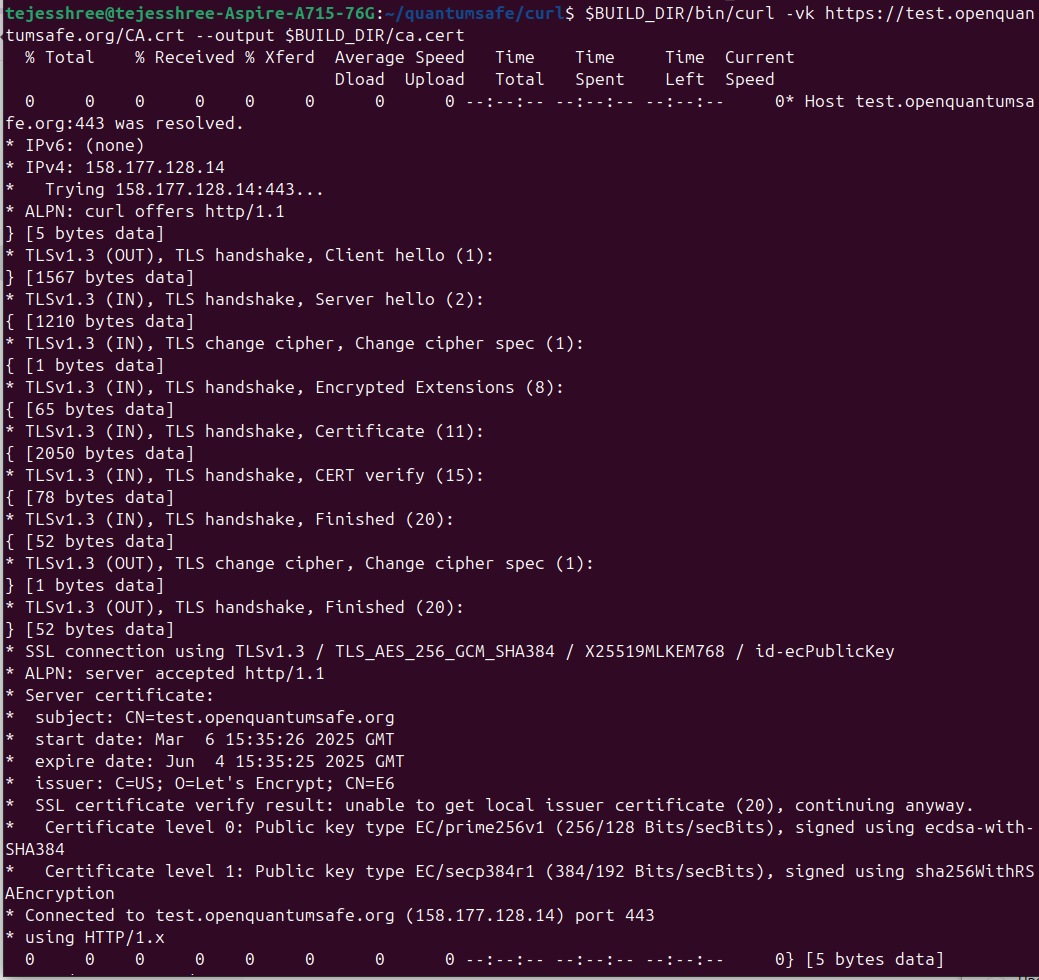
make -j $(nproc) install

**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



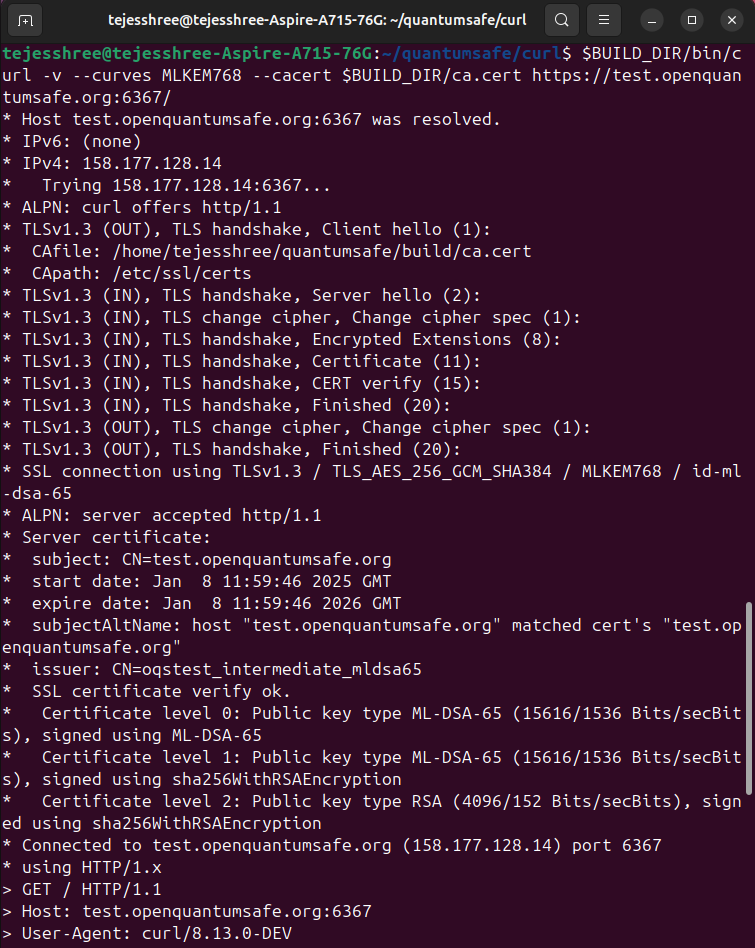
--output $BUILD\_DIR/ca.cert →Saves the certificate locally as ca.cert in the build directory.

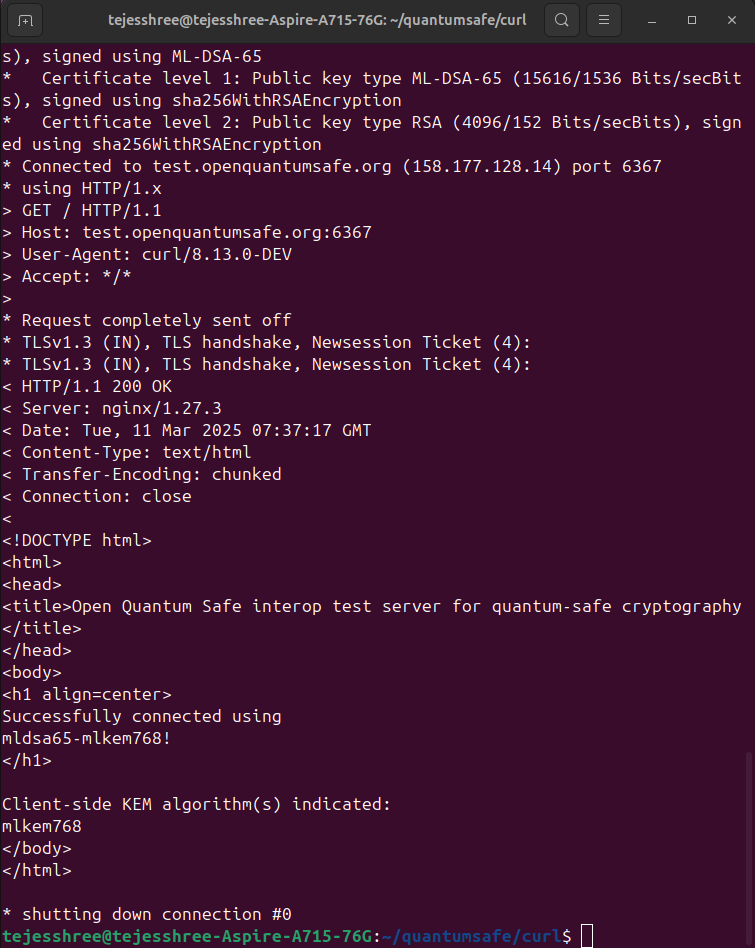
Test a quantum-safe TLS handshake with a server of specified port number using post-quantum key exchange and signature algorithm:

$BUILD\_DIR/bin/curl -v --curves MLKEM512 --cacert $BUILD\_DIR/ca.cert [https://test.openquantumsafe.org:6346/](https://test.openquantumsafe.org:6162/)

--curves MLKEM512 → Specifies the key exchange algorithm.

[https://test.openquantumsafe.org:6346/](https://test.openquantumsafe.org:6162/) →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)



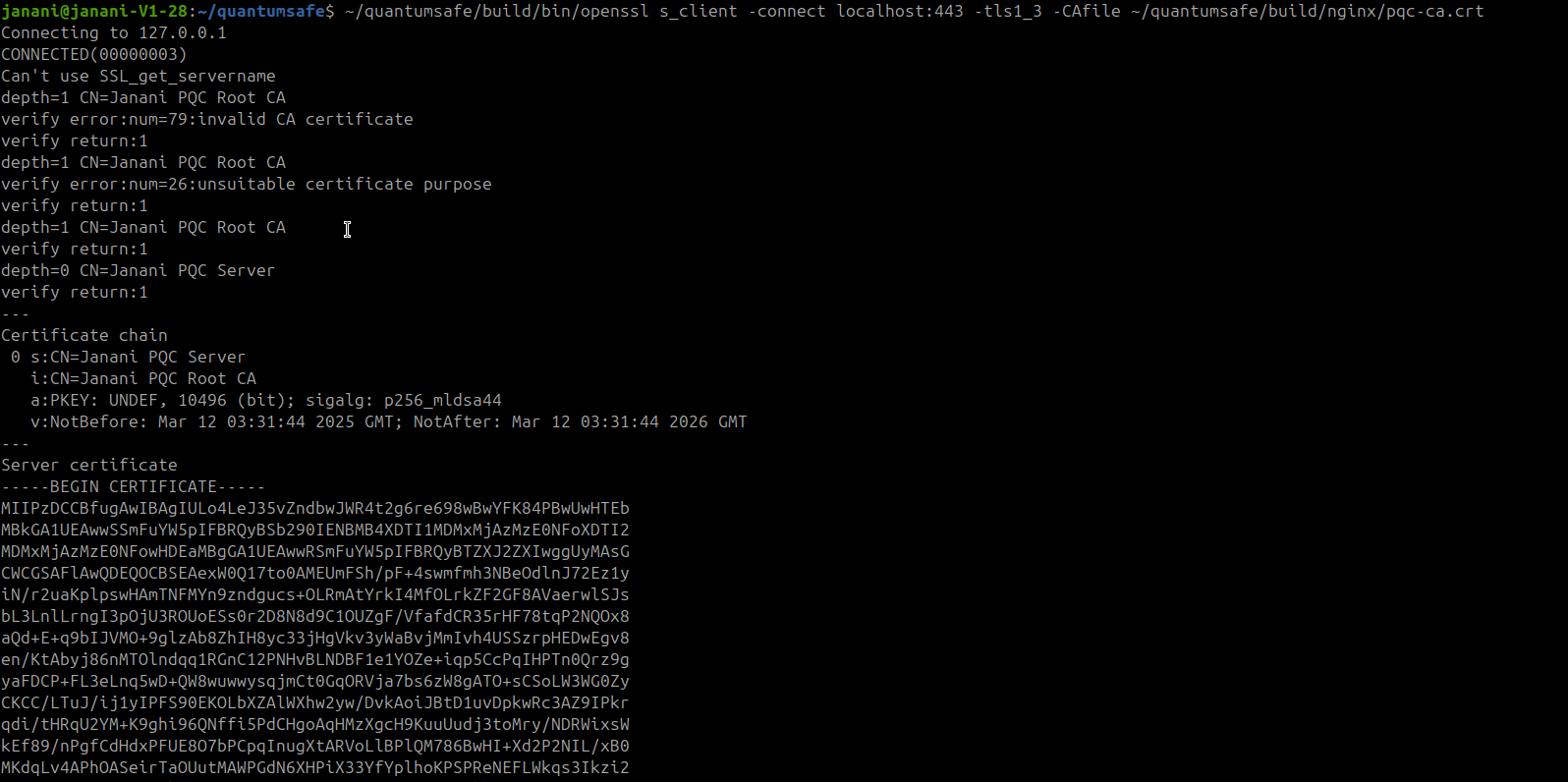


**Module 4: Custom test environment**

**Goal:** Develop a custom test server with NGINX that supports PQC algorithms in TLS with aid of OpenSSL and libOQS.

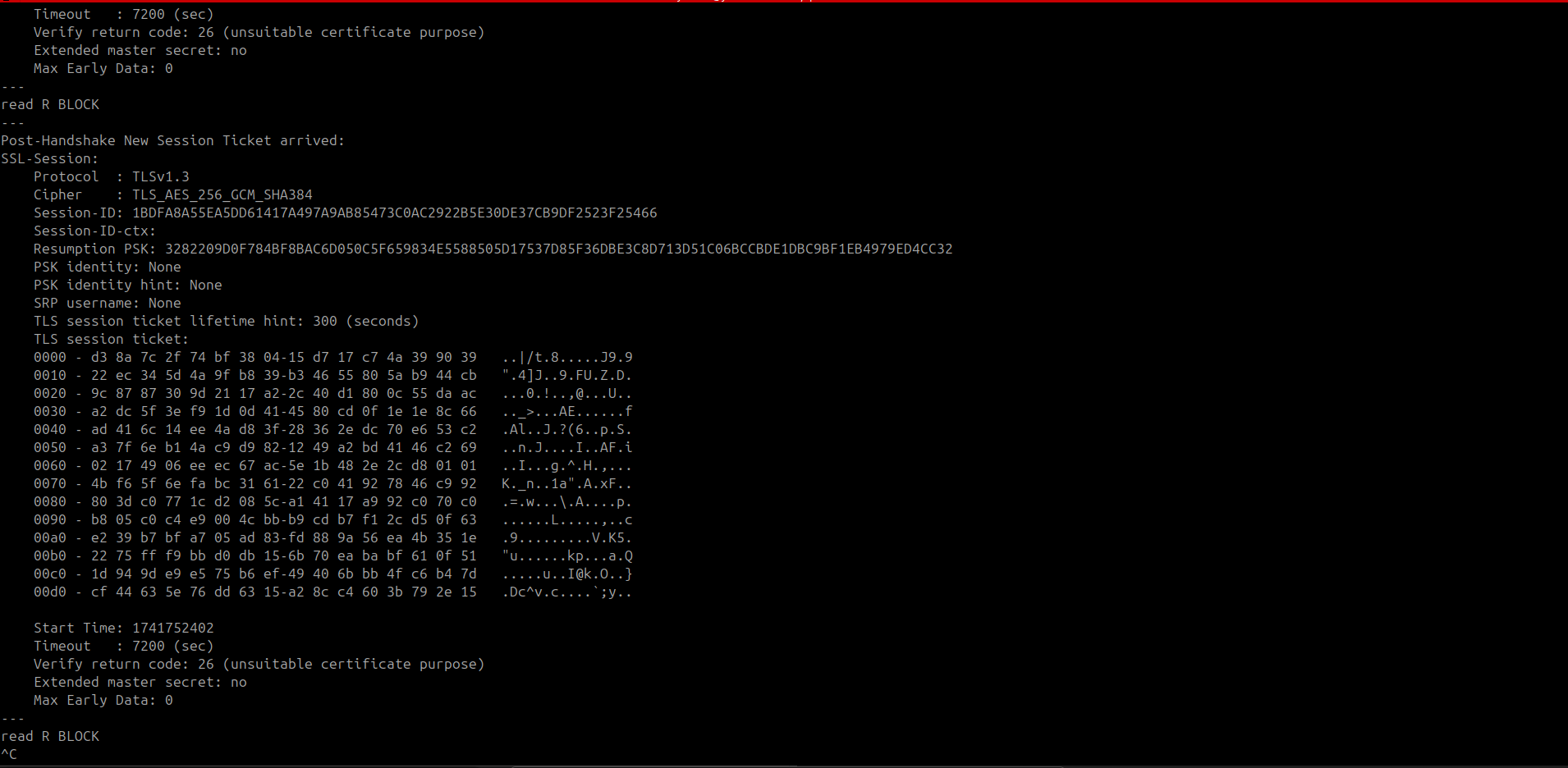
**4.1. NGINX Test server:**

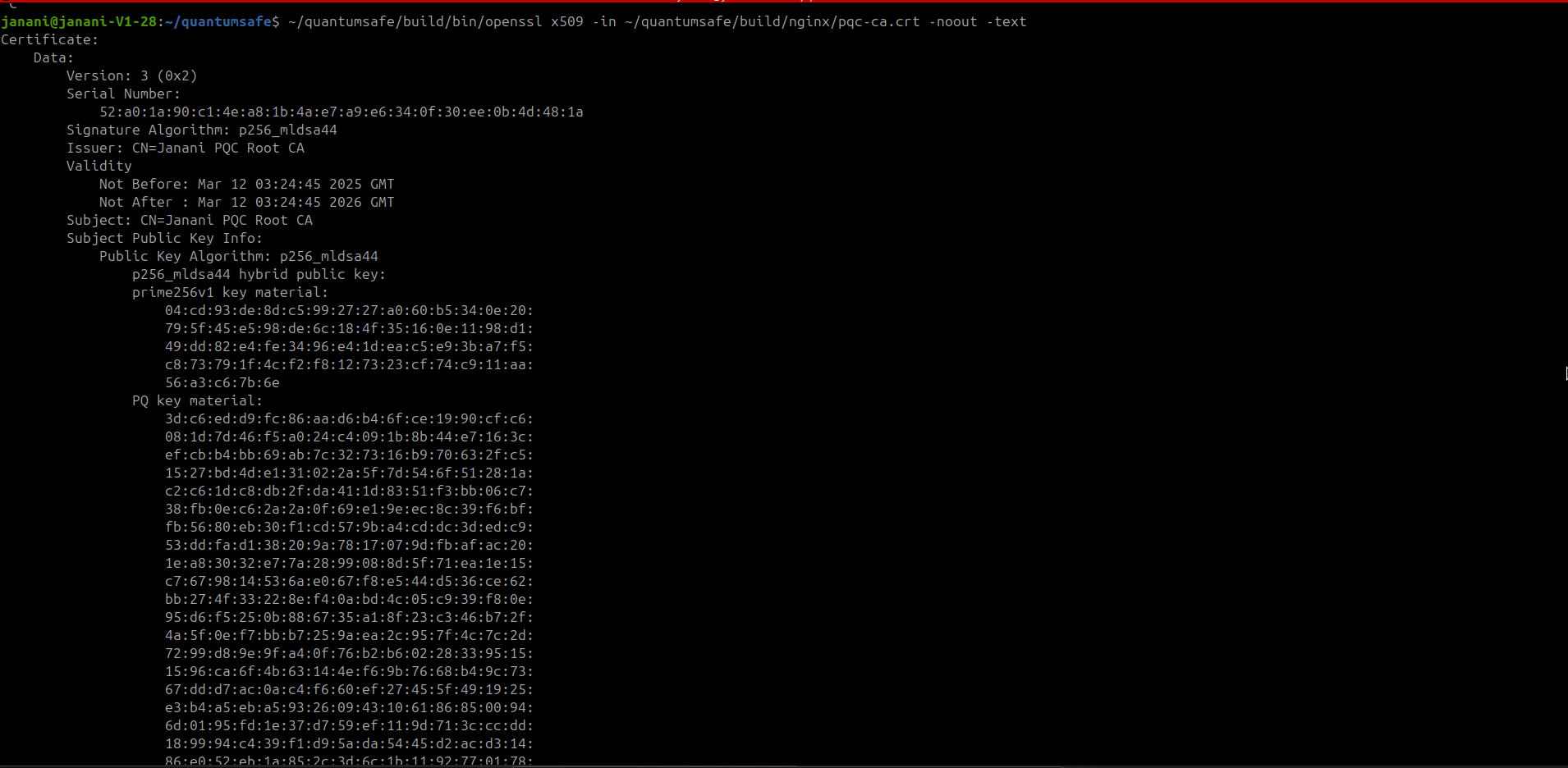
* Downloaded and compiled NGINX with OQS-provider as the TLS library.
* Created Root CA with PQC Algorithm (MLDSA44) (Current methods use RSA for Root CA).
* Modified ngnix.conf to use th epQC certificate.
* Run the custom Nginx server.
* Test PQC TLS connection using cURL and OpenSSL.



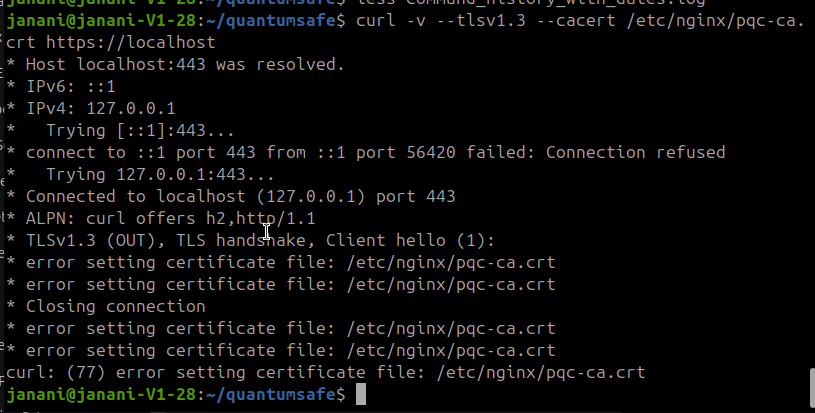






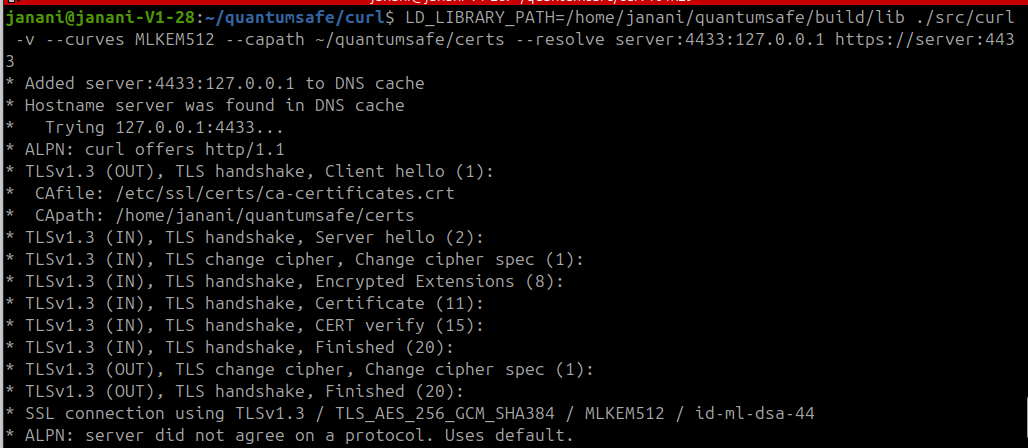


**NGINX test server - Issues Faced**

****

**Issues faced:** Root CA was being recognized as Invalid CA certificate and TLS handshake was not initiated

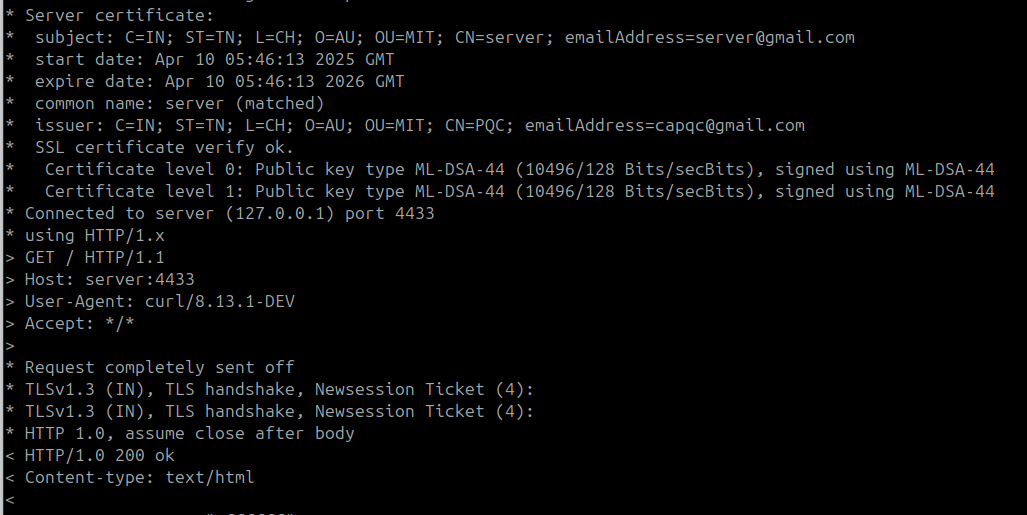
**Issue Resolved:** The error faced in the above section was rectified. PQC based TLS connection is established successfully with Root CA certificate signed using MLDSA44 algorithm and the same for server certificate.

****

PQC based successful TLS connection was made. The Root CA certificate which has been signed using RSA (Classical cryptography) algorithm is changed to ML-DSA (PQC-based) Algorithm.

**Server Certificate:** MLDSA44

**Root CA Certificate:** MLDSA44

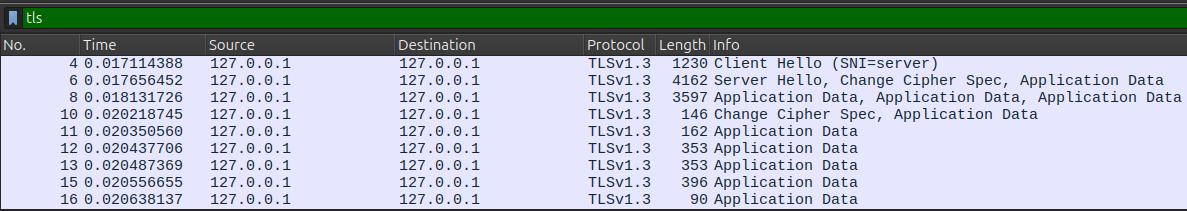
****

**Module 5: Performance evaluation and benchmarking**

**5.1. Evaluation metrics:**

**5.1.1. Handshake Time:**

Time taken from initiating the TLS handshake (Client Hello) to its successful completion (Finished message), including key exchange, certificate verification, etc.

****

**5.1.2. Certificate Size:**

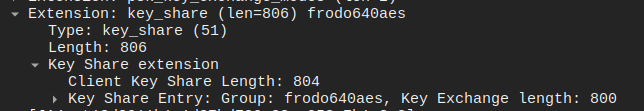
Tle file size of the X.509 Root CA certificate during TLS handshake, typically in bytes or Kilobytes.





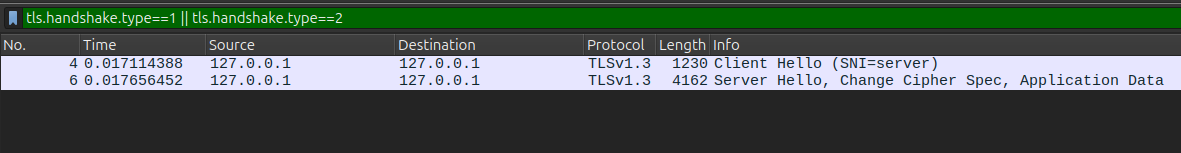
**5.1.3. Key Exchange Size:**

Size of the key exchange message sent during the handshake. In PQC, these keys can be large depending on the algorithm.



**5.1.4. RTT (Round Trip Time):**

The time it takes for a signal to travel from the client to the server and back. It reflects network latency and affects how fast the handshake completes.



**5.2. Performance Analysis:**

**5.2.1. Pure- PQC:**

Performance analysis across Pure-PQC security levels

| **Root CA Algorithm** | **Key Exchange Algorithm** | **Signature Algorithm** | **Handshake Time**  **(ms)** | **Certificate size**  **(bytes)** | | **Key Exchange Length**  **(bytes)** | **RTT (Round Trip Time)**  **(ms)** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Root CA** | **Server** |
| MLDSA44 | MLKEM512 | MLDSA44 | 2.5415523 | 5697 | 5677 | 800 | 0.3458945 |
| MLDSA65 | MLKEM768 | MLDSA65 | 2.620661 | 7769 | 7651 | 1184 | 0.357551 |
| MLDSA87 | MLKEM1024 | MLDSA87 | 2.8391154 | 10422 | 10304 | 1568 | 0.3620452 |

**5.2.2. Hybrid (Partially-Pure) PQC:**

Performance analysis across Hybrid (Partially Pure) PQC security levels

| **Root CA Algorithm** | **Key Exchange Algorithm** | **Signature Algorithm** | **Handshake Time**  **(ms)** | **Certificate size**  **(bytes)** | | **Key Exchange Length**  **(bytes)** | **RTT (Round Trip Time)**  **(ms)** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Root CA** | **Server** |
| RSA | MLKEM512 | MLDSA44 | 3.5840696 | 1367 | 2659 | 800 | 1.1983031 |
| RSA | MLKEM768 | MLDSA65 | 3.6592075 | 1367 | 3524 | 1184 | 1.2344027 |
| RSA | MLKEM1024 | MLDSA87 | 4.1899918 | 1367 | 4389 | 1568 | 1.3166472 |

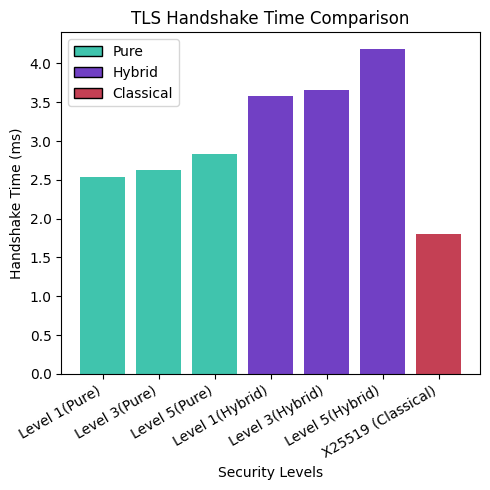
**5.2.3. Classical Cryptography Algorithm**

Performance analysis across Classical Cryptography Algorithm

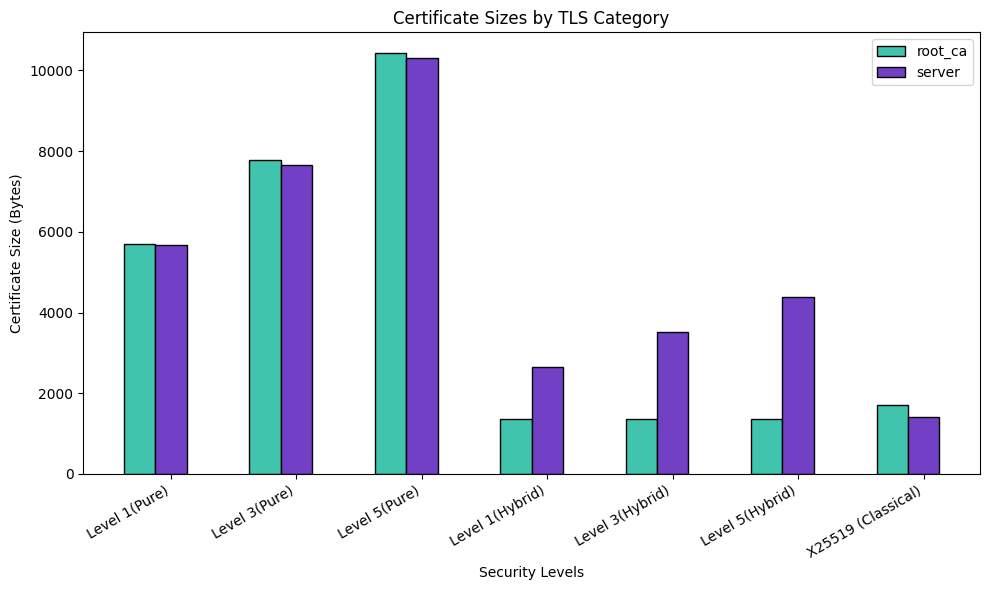
| **Root CA Algorithm** | **Key Exchange Algorithm** | **Signature Algorithm** | **Handshake Time**  **(ms)** | **Certificate size**  **(bytes)** | | **Key Exchange Length**  **(bytes)** | **RTT (Round Trip Time)**  **(ms)** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Root CA** | **Server** |
| RSA | X25519 | RSA | 1.8071568 | 1712 | 1424 | 32 | 0.6668389 |

**5.3 Graphs and Inference:**

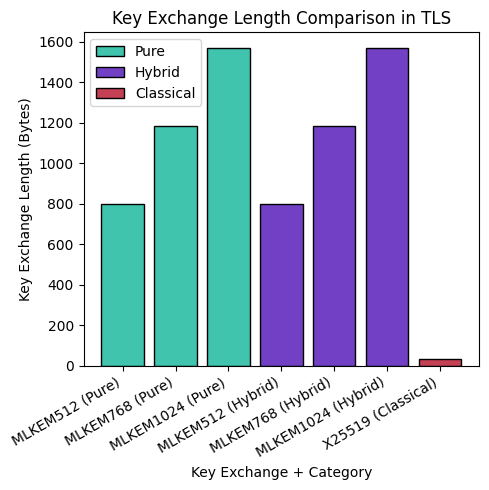
**5.3.1 Handshake time variation between Pure PQC, Hybrid PQC and Classical approaches:**

****

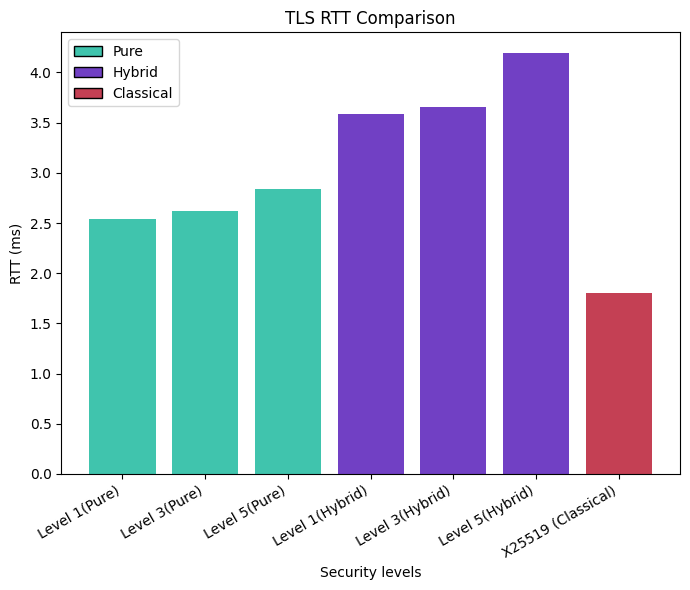
**5.3.2 Certificate size variation between Pure PQC, Hybrid PQC and Classical approaches:**

****

**5.3.3 Key exchange length variation between Pure PQC, Hybrid PQC and Classical approaches:**

****

**5.3.4 Round trip time variation between Pure PQC, Hybrid PQC and Classical approaches:**

****

**5.4 Performance and Trend Analysis**

**5.4.1 Overview of Security Levels**

The performance metrics in this study are evaluated across three post-quantum security levels:

* MLDSA44 / MLKEM512 - NIST Level 1
* MLDSA65 / MLKEM768 - NIST Level 3
* MLDSA87 / MLKEM1024 - NIST Level 5

As the security level increases, cryptographic key sizes, signature sizes, and computational overhead naturally rise to provide enhanced resistance against both classical and quantum attacks.

**5.4.2 Parameter-wise Comparative Analysis**

**5.4.2.1. Handshake Time**

Pure-PQC exhibits moderately increasing handshake times as the security level rises, ranging from 2.54 ms at Level 1 to 2.83 ms at Level 5.

In contrast, Hybrid-PQC incurs higher handshake times, starting at 3.58 ms and increasing to 4.19 ms due to the additional overhead of combining classical RSA operations with post-quantum key exchange and signatures.

Classical cryptography remains the fastest, with a handshake time of 1.80 ms.

**Trend:**

A clear positive correlation is observed between handshake time and security level in both Pure-PQC and Hybrid-PQC configurations, attributed to the increased size and complexity of key materials and cryptographic computations.

**5.4.2.2. Certificate Size**

In Pure-PQC, certificate sizes increase substantially as security levels rise - from 5697 bytes (Level 1) to 10422 bytes (Level 5).

In Hybrid-PQC, the Root CA certificate remains constant at 1367 bytes (RSA-based), while the server certificate size grows proportionally with the security level, from 2659 bytes to 4389 bytes.

Classical certificates are relatively small, with a fixed size of 1712 bytes.

**Trend:**

There is a significant growth in certificate size with increasing security levels in post-quantum schemes, driven by larger public keys and signatures embedded within the certificates.

**5.4.3.3. Key Exchange Length**

The key exchange payload size for both Pure-PQC and Hybrid-PQC increases with security level, ranging from 800 bytes (Level 1) to 1568 bytes (Level 5).

Classical X25519 key exchange is minimal, requiring only 32 bytes.

**Trend:**

Key exchange lengths scale consistently with security levels in post-quantum configurations, as stronger cryptographic assurances necessitate larger encapsulated key materials.

**5.4.3.4. Round-Trip Time (RTT)**

Interestingly, Pure-PQC achieves the lowest RTT values, starting at 0.345 ms and reaching 0.362 ms at higher security levels.

Hybrid-PQC demonstrates higher RTT values, ranging from 1.198 ms to 1.316 ms, reflecting the additional processing involved in hybrid cryptographic operations.

Classical cryptography maintains a constant RTT of 0.667 ms.

**Trend:**

RTT increases slightly with security level in both Pure-PQC and Hybrid-PQC settings, influenced by larger key and certificate sizes. However, Pure-PQC consistently outperforms Hybrid-PQC in terms of RTT, primarily due to the elimination of RSA-based processing overhead.

**CONCLUSION:**

While Pure-PQC schemes in TLS do not outperform classical cryptography in terms of handshake time, certificate size, and key exchange length, they deliver something far more critical: quantum-resilient security. Classical cryptographic algorithms, though highly efficient, are fundamentally vulnerable to quantum attacks, and their security cannot be guaranteed in the post-quantum era.

In essence, the slightly increased computational and communication costs of Pure-PQC in TLS are a necessary and worthwhile trade-off for achieving unconditional, long-term security against both classical and quantum adversaries - a level of protection that no classical system can offer.

**REFERENCES:**

[1] R. Rios, J. A. Montenegro, A. Muñoz and D. Ferraris, "Towards the Quantum-Safe Web: Benchmarking Post-Quantum TLS." in IEEE Network, doi: 10.1109/MNET.2025.3531116.

[2] Nouri Alnahawi, Johannes Müller, Jan Oupický, and Alexander Wiesmaier, “A Comprehensive Survey on Post-Quantum TLS” IACR Communications in Cryptology, vol. 1, no. 2, Jul 08, 2024, doi: 10.62056/ahee0iuc.

[3] R. Döring and M. Geitz, "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, pp. 1-5, doi: 10.1109/NOMS54207.2022.9789913.

[4] E. Kupcova, J. Simko, M. Pleva and M. Drutarovsky, "Experimental Framework for Secure Post-Quantum TLS Client-Server Communication", 2024 International Symposium ELMAR, Zadar, Croatia, 2024, pp. 213-216, doi: 10.1109/ELMAR62909.2024.10694092.

[5] OpenSSL github: <https://github.com/openssl/openssl.git>.

[6] libOQS github: <https://github.com/open-quantum-safe/liboqs.git>

[7] Integration and testing github: [https://gist.github.com/ajbozarth/ 65aee2084f7bc089704b4e7afb54801e](https://gist.github.com/ajbozarth/65aee2084f7bc089704b4e7afb54801e)

[8] Test Server by Open Qunatum 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>