Published by :
https://www.ijert.org/
An International Peer-Reviewed Journal

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 10 , October - 2026

Quantum-Resistant Secure Communication System

Mr. Srinivasamurthy K T
Assistant Professor,
Department of CSE
Acharya Institute of Technology
Bengaluru, Karnataka

Bhuvan S
Department of CSE
Acharya Institute of Technology
Bengaluru, Karnataka

Vignesh T
Department of CSE
Acharya Institute of Technology
Bengaluru, Karnataka

Supreetha H H
Assistant Professor,
Department of CSE
Acharya Institute of Technology
Bengaluru, Karnataka

Trishal Raju
Department of CSE
Acharya Institute of Technology
Bengaluru, Karnataka

Vinay Kumar
Department of CSE
Acharya Institute of Technology
Bengaluru, Karnataka

***Abstract* - Most secure messaging today relies on RSA or elliptic-curve Diffie–Hellman, which a sufficiently large quantum computer could break, and traffic recorded today can be stored and decrypted later. This paper describes a multi-client secure communication system that uses post-quantum cryptography for session-key establishment. Key encapsulation uses ML-KEM-768 (FIPS 203) by default, identities and key-distribution packets are signed with ML-DSA-65 (FIPS 204), and message and file contents are protected with AES-256-GCM. The relay server routes traffic and metadata but is not a cryptographic authority: a receiver verifies a packet's signature against its own locally held key before decapsulating or installing any key, and a peer is trusted only after explicit out-of-band confirmation. We evaluated the system with 200-iteration micro-benchmarks, 2,188 automated tests in 15 batches, 33 UI-driven acceptance tests and six attack and fault scenarios. In this implementation, a full MLKEM-768 session-key round trip takes 8.16ms against 0.51ms for RSA-2048, a gap that mainly reflects a pure-Python MLKEM library compared with native RSA. ML-KEM-768 public keys and ciphertexts are 4.6 and 4.3 times larger. ML-DSA65 signing averages 0.62ms. Every forged, stripped, malformed or tampered input was rejected before any key was installed. Manual acceptance testing and real-hardware audio checks remain open.**

Keywords - Post-quantum cryptography, ML-KEM, MLDSA, key encapsulation, secure messaging, end-to-end encryption, AES-256-GCM

I. INTRODUCTION

Confidential messaging depends on public-key cryptography so that parties who have never met can agree on keys. The widely deployed schemes, RSA and elliptic-curve Diffie– Hellman, rest on the hardness of factoring and discrete logarithms, and Shor's algorithm solves both in polynomial time on a sufficiently large quantum computer. Cryptographically relevant quantum computers remain a future risk, but the threat is already relevant for data with long confidentiality lifetimes. An adversary who records today's traffic can decrypt it once a capable machine is available, a pattern usually called "harvest now, decrypt later". In August 2024 NIST published its first post-quantum standards: ML-KEM for key encapsulation (FIPS 203) and ML-DSA for digital signatures (FIPS 204) [1], [2].

Replacing the key-exchange primitive is not enough on its own. A key-encapsulation mechanism gives confidentiality but no proof of who produced a key, so a relay that carries key material must not be able to substitute its own. A messaging system must also support group membership, several devices per account, attachments and reconnection, and each of these changes how keys are distributed. The project's Phase 1 proposal set out to build such a system and to measure what post-quantum security costs in time and bandwidth compared with RSA.

This paper makes four contributions.

1) A design in which the server is untrusted for cryptography. Every key-establishment packet carries an MLDSA-65 signature that the receiver checks against its own trusted key before decapsulation, and peer trust is a local decision made by a person (Section IV).
2) A working implementation across a desktop client, a browser client and a PostgreSQL-backed server, including group chat, encrypted attachments, multi-device key sync and approval-based group invitations.
3) Measured comparisons of ML-KEM-768 with RSA2048 and of ML-DSA-65 operations, reporting sizes that hold for any conforming implementation and timings together with their caveats.
4) Automated, acceptance and adversarial test evidence, with an explicit account of what has not yet been verified.

Section II gives background, Section III reviews existing systems and their limitations, Section IV describes the proposed system, Section V reviews related work, Section VI reports the evaluation and results, Section VII discusses limitations and future work, and Section VIII concludes.

II. BACKGROUND

A. Traditional Key Exchange

Public-key key agreement in deployed messaging and web services uses RSA key transport or elliptic-curve Diffie–Hellman, and classical TLS 1.3 protects the transport. Both

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 10 , October - 2026

families rest on problems that Shor's algorithm breaks, so their security depends on the absence of a cryptographically relevant quantum computer.

B. Post-Quantum Standardization

NIST's standards are built on lattice schemes selected through its post-quantum competition, namely CRYSTALSKyber and CRYSTALS-Dilithium, published as ML-KEM and ML-DSA [1], [2]. ML-KEM is a key encapsulation mechanism: the sender obtains a shared secret and a ciphertext from the receiver's public key, and only the holder of the private key can recover the secret.

C. Origin Authentication

Signatures are a separate primitive and must be added explicitly for origin authentication. Each key packet in this system carries an ML-DSA-65 signature of 3,309 bytes [2].

D. Threat Model

We assume an adversary who can record ciphertext and handshake traffic today and who may hold a sufficiently large quantum computer later. The same adversary can also intercept, delay, reorder, rewrite and inject relayed packets, which is the relay model used in Section IV.

E. Integrated Secure Messaging

A secure messenger combines key agreement, origin authentication, local key storage and a trust decision by users. The design in this paper combines these in one architecture: a KEM for confidentiality, ML-DSA for origin, explicit human verification for trust, and a server that relays ciphertext only.

III. EXISTING SYSTEM AND LIMITATIONS

Most deployed secure messengers and web services establish keys with RSA or elliptic-curve Diffie–Hellman, and protect transport with classical TLS. The closest deployed design to ours is Signal, which combines a ratchet with a postquantum extension of its initial key agreement (Section V). The Phase 1 prototype of this project used a Kyber-512 demonstration over plain TCP with a console client. It had no storage and no signatures, so nothing in that prototype authenticated the origin of a key.

The Phase 1 plan and the implemented system differ as summarized in Table I. The Phase 1 timing figures came from the earlier synopsis and are superseded by the measurements in Section VI.

The limitations of the existing approaches that motivate this work are as follows.

- Classical key agreement is broken by a sufficiently large quantum computer, so data recorded today stays exposed.
- A key-encapsulation mechanism alone gives no proof of who produced a key, so a relay can substitute keys unless packets are signed.
- The Phase 1 prototype used plain TCP and had no persistence, so it could not support the group, multidevice and reconnection behaviour that real messaging needs.
- Published post-quantum measurements cover isolated primitives or handshakes, not complete applications.

TABLE I
PHASE 1 PLAN VERSUS PHASE 2 IMPLEMENTATION

Area	Phase 1 plan	Phase 2 implementation
Key exchange	Kyber-512 prototype	ML-KEM-768 (FIPS 203) default; RSA-2048-OAEP as comparison mode
Signatures	Dilithium planned	ML-DSA-65 (FIPS 204) signs identities, messages and key packets
Payload cipher	AES-256, CBC	AES-256-GCM (12-byte nonce, 16-byte tag)
Transport	Plain TCP	TLS 1.2+ with JWT login sessions
Application	Console client	Desktop app, browser client via gateway, Android prototype
Storage	None	PostgreSQL state; encrypted blob store for files

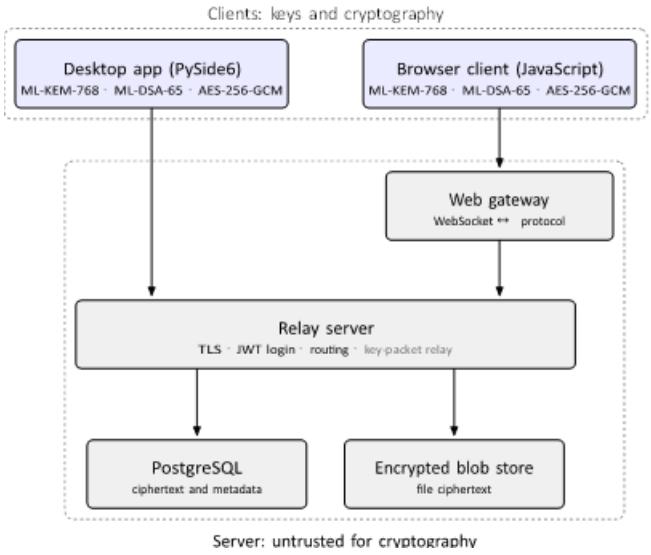


Fig. 1. System architecture. Cryptography runs only in the clients; the server routes ciphertext and metadata, and the browser client reaches the relay through the WebSocket gateway.

Therefore, there is a need for a system that unifies key establishment, origin authentication, local key protection and the user's trust decision in one design, while keeping the server out of the trust chain. The proposed system does this with ML-KEM-768 for key establishment, ML-DSA-65 for origin authentication and AES-256-GCM for content, and it reports the measured cost of doing so.

IV. PROPOSED SYSTEM AND METHODOLOGY

The proposed system takes each user's identity, contacts, group membership and messages as input. The clients perform all cryptography: they announce identities, establish keys, encrypt and sign packets, and verify what they receive. The server authenticates connections, routes packets, persists conversation state and stores file ciphertext. Figure 1 shows the components.

The server can observe who communicates with whom, when, message sizes, file names and group membership. It

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 10 , October - 2026

TABLE II
CRYPTOGRAPHIC PRIMITIVES AND IMPLEMENTING MODULES

Purpose	Primitive and standard	Module
Key establishment (default)	ML-KEM-768, FIPS 203	crypto/kyber.py
Key establishment (comparison)	RSA-2048, OAEP with SHA-256	crypto/rsa.py
Origin authentication	ML-DSA-65, FIPS 204	crypto/ml_dsa.py
Message and file encryption	AES-256-GCM	crypto/aes.py, crypto/ payload_cipher.py
Local key-store KDF	Argon2id [3]	storage/ secure_key_store.py
Password hashing	Argon2 [3]	security/ password_handler.py
Login sessions	JSON Web Tokens	security/ jwt_handler.py
Transport	TLS 1.2+	security/tls.py

can drop, delay, reorder or rewrite relayed packets. It cannot read message bodies or file contents, obtain AES session keys or private keys, forge a signed packet, replace a signing key, or install a key at a receiver. Trust state is stored locally on each client.

Table II lists the primitives and the modules that implement them. The browser client uses vendored JavaScript builds of the same algorithms, ML-KEM-768 and ML-DSA-65 from @noble/post-quantum v0.7.1, and its interoperability with the Python client is covered by the automated tests.

A. Key Establishment and Origin Authentication

Every key-establishment path follows the same mandatory order, shown here for group key distribution; direct conversations use the same steps with one recipient.

1) Announce. Each client publishes its ML-KEM-768 encapsulation key and its ML-DSA-65 verification key in a self-signed identity announcement. A receiver records the identity as UNVERIFIED.
2) Verify the person. Users compare key fingerprints out of band. Only an explicit confirmation sets a peer to VERIFIED; no packet can do so.
3) Encapsulate. The sender runs Encaps(ek_R) to obtain a fresh 32-byte secret K and a 1,088-byte ciphertext c. It then encrypts the conversation key under AES-256GCM with K.
4) Sign. The sender signs a canonical, length-framed encoding of the sender, conversation identifier, recipient, c, the wrapped key and the epoch, using ML-DSA-65 under a fixed domain-separation context.
5) Relay. The server forwards the packet. It can drop or alter it, but it cannot produce a valid signature.
6) Verify, then decapsulate. The receiver resolves the sender's verification key from its own local peer state, never from the packet, and checks the signature. Only after that check does it decapsulate, decrypt and store the key. A forged packet never reaches decapsulation or key storage.

Signing the ciphertext and wrapped key binds the encapsulation to a specific sender, recipient and epoch, so a valid signature for one purpose cannot be replayed for another. In the current implementation, the 32-byte ML-KEM shared secret is used directly as the AES-256-GCM wrapping key, without a separate key-derivation step. Section VII returns to this point.

B. Security and Privacy Measures

Message bodies and attachments are encrypted on the client with AES-256-GCM using a fresh random 12-byte nonce and a 16-byte tag. Keys are scoped by conversation and epoch, and epoch reservation is coordinated through the server. Files are encrypted before upload, and the server stores and returns only ciphertext.

Each client persists its ML-KEM-768 key pair, its MLDSA-65 signing seed, conversation keys and peer-verification records in one file. The file is encrypted with AES-256-GCM under a key derived from the account password using Argon2id with a random 16-byte salt. A wrong password, corrupted file or tampered file fails closed. RSA keys are not persisted and are regenerated each session. An attacker who obtains both the file and the password can read that user's key material.

Peer identities move through three states: UNVERIFIED (observed, the default), VERIFIED (confirmed by a person) and KEY_CHANGED (a verified peer later presents a different key). A changed key is never silently accepted. The old key stays in place, the user is warned, and no key is accepted from the new identity until it is re-verified. Group invitations are routed through each invitee's inbox. Only members who approve are added, the approval checks the inviter against an identity the invitee has already observed, and invitations expire after 14 days.

C. Deployment and Monitoring

The server and core libraries are written in Python 3.12 and total about 52,000 lines of Python, excluding tests, scripts and migrations. The desktop client uses PySide6 6.9.1 and is packaged as a Windows executable. The browser client is about 9,200 lines of JavaScript that talks to the server through a WebSocket gateway. The server uses SQLAlchemy and Alembic migrations on PostgreSQL. An Android prototype exists, but it is paused and has not been validated on hardware, so it is excluded from the results.

V. RELATED WORK

Several studies have measured post-quantum algorithms inside TLS. Paquin, Stebila and Tamvada benchmarked postquantum key exchange and signature algorithms in TLS 1.3 using the Open Quantum Safe project. Sosnowski et al. studied how post-quantum choices affect TLS 1.3 performance [4]. These works evaluate primitives or handshakes in isolation.

In messaging, the Signal protocol's Double Ratchet provides forward secrecy and post-compromise recovery through per-message keys. Signal's PQXDH extends its initial key agreement with a post-quantum KEM [5]. Our system does not ratchet, so its forward secrecy is limited to what perconversation keys provide. It differs from both protocols in being server-mediated

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 10 , October - 2026

with multi-device and group distribution, and in signing every key packet with ML-DSA-65 so that the relay cannot substitute keys.

The proposed system combines the key-establishment and origin-authentication techniques surveyed above with a relay architecture and an explicit verification state machine. It reports measured costs for a complete application rather than for a handshake or primitive in isolation.

VI. RESULTS AND EVALUATION PROTOCOL

The system was implemented and tested as a complete application, with the automated suites and UI acceptance tests run against the built desktop executable. The measured costs and the adversarial outcomes are reported below.

Evaluation Protocol. Micro-benchmarks. Measurements ran on Windows 11 (AMD64, Intel64 Family 6 Model 151) with Python 3.12.0 (key exchange) and 3.12.10 (MLDSA). RSA and ML-DSA used the cryptography package 49.0.0 with native Rust/OpenSSL bindings. ML-KEM used kyber-py 1.2.0, a pure-Python library. The code, benchmark output and test results are in the project repository. Key exchange was timed over 200 iterations after 20 warmup runs, and key generation over 25 runs for KEM and RSA and 100 for ML-DSA. Each ML-DSA sample was checked against the fixed key and signature sizes, and each signature was independently verified.

Automated tests. The regression suite runs as 15 batches over 194 test files (scripts/ run_regression_batches.py) against PostgreSQL 17 and Google Chrome. The final run on 22 October 2026 took 33.1 minutes. The acceptance suite drives the built Desktop executable and Chrome to exercise real multi-user scenarios.

Adversarial and confidentiality checks. Six scenarios covered forged, stripped, malformed and tampered inputs, together with one duplicate-delivery defect (Table V). After the acceptance run, a confidentiality scan checked 713,873 bytes of logs, 436 database rows and two blobs against 439 secret patterns.

Key Exchange Performance. Table III and Fig. 2 give the mean times. ML-KEM-768 key generation is 13.7 times faster than RSA-2048 key generation, which is the one operation where it wins. Every session-key operation is slower. The full round trip, establishment plus recovery, is 16.1 times slower: 8.16ms against 0.51ms. The group-key wrap and unwrap follow the same pattern.

The timing gap is not a property of the algorithms. The pure-Python ML-KEM implementation is measured against compiled RSA, and both timings come from one machine and one run. A native ML-KEM backend would change the ratio, though we did not measure one. The size results below do not depend on the implementation.

TABLE III
KEY-ESTABLISHMENT TIMING, MEAN OVER 200 ITERATIONS (KEY GENERATION: 25)

Operation	RSA-2048 (ms)	ML-KEM-768 (ms)	Ratio
Key generation (per login)	33.21	2.43	0.07
Session key, establish	0.025	3.38	133
Session key, recover	0.480	4.78	10.0
Session key, full round trip	0.506	8.16	16.1
Group key, wrap	0.022	3.87	172
Group key, unwrap	0.478	5.05	10.6

Ratio = ML-KEM-768 ÷ RSA-2048; values below 1 mean ML-KEM-768 is faster. RSA runs natively and ML-KEM in pure Python, so these ratios reflect the implementations, not the algorithms.

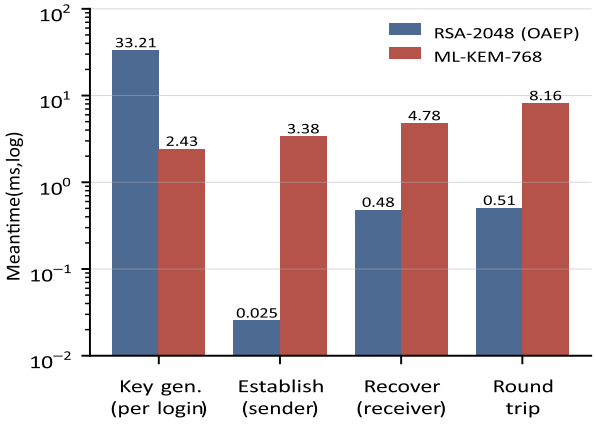


Fig. 2. Mean time per key-establishment operation on a log scale (200 iterations; key generation 25). ML-KEM-768 is faster only for key generation.

TABLE IV
SERIALIZED SIZES IN BYTES

Item	RSA-2048	ML-KEM-768	Ratio
Public key (raw)	256	1,184	4.6
Ciphertext or encapsulation (raw)	256	1,088	4.3
Ciphertext or encapsulation (wire, base64)	344	1,452	4.2
Group key material (encapsulation + wrapped key)	344	1,548	4.5
Shared secret	32	32	1.0
ML-DSA-65, identical in both modes Public key			952
Signature			3,309
Private seed			32

Size Overhead. Table IV gives the sizes. ML-KEM-768 public keys are 4.6 times larger than RSA-2048 keys, and ciphertexts are 4.3 times larger. The shared secret is 32 bytes in both cases. For a group key distribution, the key material grows from 344 to 1,548 bytes. Because the 3,309-byte ML-DSA65 signature is present in both modes, the complete signed packet grows by about 1.2KB, a factor of about 1.3. Signatures therefore dominate bandwidth, and the price of post-quantum key exchange is modest once they are included.

Signature Performance. ML-DSA-65 key generation averages 0.14ms over 100 runs, verification 0.14ms and signing

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 10 , October - 2026

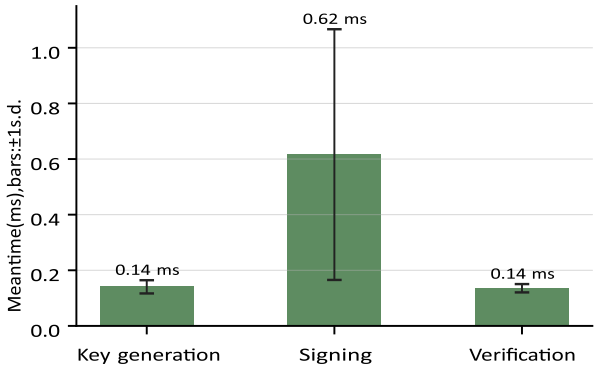


Fig. 3. ML-DSA-65 operation times on the native backend. Bars show means and error bars one standard deviation (key generation 100 runs; signing and verification 200 runs).

TABLE V
ADVERSARIAL AND FAULT SCENARIOS

Scenario	Method	Outcome
Forged key packet	Relay rewrites the wrapped key in a signed group-key packet	Rejected: signature fails; no key installed
Signature stripping	Relay deletes signing fields from an identity announcement	Blocked: first contact stays UNVERIFIED
Legacy session-key forgery	Forged packet on the old Kyber session-key path	Rejected unconditionally
Malformed public key	Invalid Base64 or wrong length (not 1,184 bytes)	Refused at entry
Ciphertext tampering	Bytes of an AES-GCM payload modified	Rejected: tag check fails
Duplicate send	A message is sent twice	Fixed: delivered once (defect)

0.62ms over 200 runs (Fig. 3). Signing varies most (standard deviation 0.45ms) because the algorithm uses rejection sampling. The verification cost is small, which matters because every received key packet is verified before use.

Adversarial Scenarios. Table V summarizes the six scenarios. Each forged, stripped, malformed or tampered input was rejected or blocked before a key was installed, and the peer trust state did not change. One scenario, duplicate delivery, was a functional defect rather than an attack. The acceptance tests found it, and it was fixed and retested. These are scenario tests, not a formal proof or an external penetration test.

Automated and Acceptance Testing. The final run passed 2,188 of 2,188 automated tests across 194 files in 15 batches (Fig. 4), with 33 of 33 UI acceptance tests passing against the real executable. The suite grew from 1,919 to 2,188 tests (+14.0%) in three days. The first full run had one failure and one error, which were attributed to transient page-load stalls. Neither reproduced, and both batches passed when rerun. Passing tests show the implemented behaviour matches the test specifications; they do not show that the specifications are complete. The test suite was written by the same developers, so shared blind spots are possible.

Discussion. Three points follow from the results. First, the

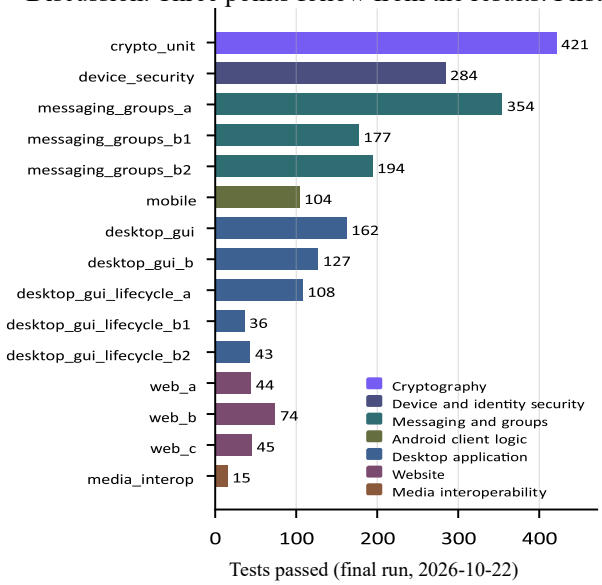


Fig. 4. Tests passed per regression batch in the final run (22 October 2026), coloured by area of the system. Totals: 2,188 tests, all passing.

post-quantum cost is mostly a size cost. ML-KEM-768 keys and ciphertexts are four to five times larger, and this does not depend on the implementation. Second, the computational gap is real in this build but is an artifact of the pure-Python library, so it should be re-measured with a native backend before any claim about speed is made. Third, verifying the signature before decapsulation adds one ML-DSA verification to each key packet, about 0.14ms in this build. We consider that a modest price for rejecting forged key material before any decapsulation or storage takes place.

VII. FUTURE WORK

Limitations. The results carry the following limits, which the paper does not hide.

- Single environment. All timings come from one Windows machine and one run. They are not portable performance claims.
- Unequal implementations. RSA is native code and MLKEM is pure Python.
- Manual acceptance is incomplete. Round 1 of the team's manual acceptance checklist passed 14 of 20 items and failed 6. Fixes were made, but re-testing of those items is pending.
- Audio path not verified on hardware. Automated tests use synthetic signals and fake devices. An audio self-test script exists, but it has not been run on a real microphone or camera.
- Transport is classical. TLS 1.2+ is not post-quantum. Message confidentiality against a future quantum adversary rests on the ML-KEM-protected key establishment and AES-256-GCM, but metadata protected only by TLS is exposed to harvest-now-decrypt-later collection. The RSA

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 10 , October - 2026

comparison mode is not post-quantum for key establishment, although its signatures are ML-DSA-65.

- No ratchet or traffic privacy. Forward secrecy is limited to per-conversation keys. The server sees metadata. Deleting blobs is not yet implemented.
- Key derivation. The ML-KEM shared secret is used directly as an AES key. NIST SP 800-56C recommends a key-derivation function with protocol context, and adding one is planned.
- Deployment settings. Development TLS certificates are used, the executable is not code-signed, and the development JWT secret is a placeholder that must be overridden in any real deployment.
- No formal analysis or external audit. The protocol has not been checked with a symbolic verifier or by an independent security review.

Future work will proceed in the following order. (1) Rebenchmark ML-KEM with a native backend, on more than one machine, with like-for-like RSA timing. (2) Complete manual acceptance round 2 and run the audio self-test on real hardware. (3) Add a key-derivation step with protocol context to the key-establishment path. (4) Replace the TLS transport with a post-quantum or hybrid TLS 1.3 key exchange and production certificates. (5) Add a ratchet for forward secrecy and verify the protocol with a symbolic tool such as ProVerif or Tamarin. (6) Validate the Android client on hardware.

VIII. CONCLUSION

This paper presented a multi-client secure messaging system that uses ML-KEM-768 for key establishment and ML-DSA65 to authenticate every key packet and identity. The server routes packets but cannot substitute a key, because a receiver verifies each signature against its own trusted state before decapsulating anything. Measured against RSA-2048, postquantum key establishment costs about 4.5 times more key material and, in this pure-Python build, a slower session setup. Its benefit is resistance to a quantum adversary.

IX. REFERENCES

[1] National Institute of Standards and Technology, “Module-lattice-based key-encapsulation mechanism standard,” U.S. Department of Commerce, Federal Information Processing Standard (FIPS) 203, Aug. 2024.

[2] ——, “Module-lattice-based digital signature standard,” U.S. Department of Commerce, Federal Information Processing Standard (FIPS) 204, Aug. 2024.

[3] A. Biryukov, D. Dinu, D. Khovratovich, and S. Josefsson, “The memoryhard Argon2 password hashing and proof-of-work function,” RFC 9106, 2021.

[4] M. Sosnowski, F. Wiedner, E. Hauser, and G. Carle, “The performance of post-quantum TLS 1.3,” in *Proc. ACM CoNEXT ’23 Companion*, Dec. 2023.

[5] Signal Messenger, “The PQXDH key agreement protocol,” Signal Technical Specification, https://signal.org/docs/specifications/pqxdh/, 2023.

(This work is licensed under a Creative Commons Attribution 4.0 International License.)