Applied Cryptography – CMPS 297AD/396AI

QuantumSafe-SMB-Link

A Post-Quantum Hybrid Tunnel for SMB

Georges Ghanem

Department of Computer Science
American University of Beirut

January 4, 2026

Abstract

The advent of large-scale quantum computers threatens the public-key cryptography currently used to secure network protocols and tunnels. In particular, elliptic-curve Diffie–Hellman (ECDH) and RSA, which protect many VPNs and SMB tunnels, can be broken by Shor's algorithm. To address this risk, NIST has standardized lattice-based post-quantum primitives, including ML-KEM-768 for key encapsulation and ML-DSA-2 for digital signatures.

This project presents *QuantumSafe-SMB-Link*, a Rust implementation of a hybrid, post-quantum-aware tunnel for the Server Message Block (SMB) protocol. On top of TCP, QuantumSafe-SMB-Link runs a custom handshake that negotiates capabilities, performs a hybrid key establishment using ML-KEM-768 and optional X25519, and derives symmetric keys via an HKDF-based key schedule. SMB traffic is then carried inside a simple encrypted framing layer protected by the ChaCha20-Poly1305 authenticated-encryption scheme. Detached ML-DSA-2 signatures are implemented and can be used to authenticate handshake transcripts.

The report documents the protocol design, the Rust implementation architecture, and the migration rationale from classical to hybrid post-quantum security. It also analyzes performance and security trade-offs, discusses downgrade resistance and hybrid security guarantees, and outlines how this design can guide real-world migration of SMB-based infrastructures to post-quantum cryptography.

Acknowledgments

I would like to thank the course instructor, Dr. Nadim Kobeissi, for designing a project that bridges theoretical post-quantum cryptography with practical software engineering, and for the guidance provided throughout the semester.

Contents

Abstract **i**

Acknowledgments **ii**

Contents **iii**

List of Figures **vi**

List of Tables **vii**

1 Introduction **1**

1.1 Motivation 1

1.2 Project Overview 1

1.3 Contributions 2

1.4 Document Structure 2

2 Background **4**

2.1 Threat Model and Quantum Attacks 4

2.2 Post-Quantum Primitives 4

2.2.1 ML-KEM-768 (ML-KEM-768) 4

2.2.2 ML-DSA-2 (ML-DSA-2) 5

2.3 Classical Primitives 5

2.3.1 X25519 (X25519) 5

2.3.2 Authenticated Encryption (ChaCha20-Poly1305) 5

2.3.3 HKDF over SHA-256 5

3 System and Protocol Requirements **6**

3.1 Functional Requirements 6

3.2 Security Requirements 6

3.3 Non-Functional Requirements 7

4 Protocol Design **8**
4.1 High-Level Flow 8
4.2 Message Types and Magic Values 8
4.3 Capabilities and Algorithm Negotiation 9
4.4 ClientHello and ServerHello 9
4.4.1 ClientHello Structure 9
4.4.2 ServerHello Structure 10
4.5 Shared Secrets and Transcript Hash 11

5 Implementation in Rust **12**
5.1 Project Structure 12
5.2 Command-Line Interface (main.rs) 12
5.3 Networking and Tunnel Logic (net.rs) 13
5.3.1 Server Mode 13
5.3.2 Client Mode 14
5.3.3 Handshake Functions 14
5.3.4 Framed Pumping 14
5.4 Authenticated Encryption and Key Schedule (aead.rs) 15
5.4.1 SessionKeys and Roles 15
5.4.2 NonceCounter 15
5.4.3 AEAD Wrapper Functions 16
5.5 Framing (frame.rs) 16
5.6 Dilithium Signature Helpers (sig.rs) 16
5.7 Legacy Crypto Helpers (crypto.rs) 17

6 Performance Considerations **18**
6.1 Benchmarking Methodology 18
6.2 Example Result Tables 19
6.2.1 Primitive-Level Benchmarks 19
6.2.2 Handshake and SMB Throughput 19
6.3 Qualitative Discussion 19

7 Security Analysis **21**
7.1 Security Goals Recap 21
7.2 Hybrid Security 21
7.3 Transcript Binding and Downgrade Resistance 22
7.4 Forward Secrecy 22
7.5 Implementation Considerations 22

7.5.1 Randomness 22
7.5.2 Error Handling and Side Channels 23
7.5.3 Denial-of-Service 23

8 Conclusion and Future Work **24**

Bibliography **25**

A Wire Formats **26**
A.1 ClientHello 26
A.2 ServerHello 27
A.3 Encrypted Data Frame 27

B CLI Usage **28**
B.1 Server Mode Example 28
B.2 Client Mode Example 28

List of Figures

4.1 High-level handshake flow between client and server. 8

List of Tables

6.1 Example primitive-level benchmark structure (fill with measured values). 19
6.2 Example handshake and throughput benchmark structure. 19

Chapter 1

Introduction

1.1 Motivation

Server Message Block (SMB) is widely used in organizational networks for file sharing, printer sharing, and networked storage. In many deployments, SMB traffic is either implicitly trusted within a local network or protected using classical VPNs that rely on elliptic-curve Diffie–Hellman (ECDH) or RSA. These classical public-key schemes are vulnerable to quantum attacks: a sufficiently powerful quantum computer running Shor's algorithm could recover private keys and decrypt recorded traffic.

The threat is particularly relevant for *record-now, decrypt-later* adversaries, who capture encrypted data today with the expectation of decrypting it once quantum computers mature. SMB traffic often contains sensitive files and credentials with long-term confidentiality requirements, making it an important candidate for post-quantum migration.

At the same time, replacing widely deployed protocols is challenging. Full replacement of TLS or SMB stacks is not always feasible. Instead, many organizations rely on protocol-agnostic tunnels: programs that accept cleartext connections (e.g., SMB) and forward them through an encrypted channel. Migrating such tunnels to post-quantum security requires integrating new primitives, designing hybrid modes, and maintaining backward compatibility.

1.2 Project Overview

This project implements *QuantumSafe-SMB-Link*, a proof-of-concept SMB tunnel that uses NIST-standardized post-quantum algorithms in a hybrid design. The tunnel is built in Rust and consists of:

- A **hybrid handshake** that negotiates supported KEM and signature algorithms, performs ML-KEM-768 encapsulation, and optionally performs an X25519 ECDH exchange.
- A **key schedule** that combines classical and post-quantum shared secrets via HKDF over SHA-256 to derive bidirectional traffic keys.
- An **authenticated encryption layer** based on ChaCha20-Poly1305, using per-frame nonces and associated data (AAD).
- A **length-prefixed framing layer** that carries encrypted SMB payloads over an established TCP stream.
- A set of **Rust modules** that clearly separate networking, cryptographic operations, and protocol logic.

The codebase is organized to be educational: it exposes the protocol machinery explicitly, rather than hiding it behind generic TLS libraries. At the same time, the design borrows concepts from TLS 1.3 such as transcripts and HKDF-based key schedules.

1.3 Contributions

The contributions of this project are:

1. **Protocol Design:** A concrete, documented wire protocol for a post-quantum / classical hybrid handshake aimed at SMB tunneling.
2. **Implementation:** A full Rust implementation using production-quality crates for ML-KEM-768, ML-DSA-2, X25519, ChaCha20-Poly1305, and HKDF.
3. **Documentation & Analysis:** A structured explanation of the migration rationale, implementation details, performance considerations, and a security analysis of the design.

1.4 Document Structure

The rest of this report is organized as follows:

- Chapter 2 covers the cryptographic background and the primitives used.
- Chapter 3 defines system and security requirements.

- Chapter 4 describes the protocol, handshake messages, and key schedule in detail.
- Chapter 5 documents the Rust modules and key functions.
- Chapter 6 discusses performance and provides a structure for presenting benchmark results.
- Chapter 7 analyzes the security of the hybrid design and discusses downgrade resistance and implementation concerns.
- Chapter 8 concludes and outlines future work.
- Appendices provide wire-format details and CLI usage examples.

Chapter 2

Background

2.1 Threat Model and Quantum Attacks

We consider a standard Dolev–Yao network attacker with full control over the network: the adversary can intercept, delay, drop, modify, and inject packets. Additionally, we consider a *record-now, decrypt-later* adversary who stores ciphertexts and later obtains access to a quantum computer.

Classical public-key schemes such as RSA, Diffie–Hellman, and elliptic-curve Diffie–Hellman rely on problems (factoring, discrete logarithms) that are efficiently solvable by quantum computers running Shor's algorithm. This breaks both confidentiality and authentication.

For SMB tunnels that protect long-lived or sensitive data, we therefore require key establishment mechanisms that remain secure in the presence of quantum attackers.

2.2 Post-Quantum Primitives

2.2.1 ML-KEM-768 (ML-KEM-768)

ML-KEM-768 is a lattice-based KEM selected by NIST for standardization. It provides three parameter sets; this project uses the `kyber768` variant via the `pqcrypto-kyber` crate. The main algorithms are:

- KeyGen: $(pk, sk) \leftarrow \mathsf{KeyGen}()$.
- Encaps: Given pk, output (ct, ss), where ss is the shared secret.
- Decaps: Given ct and sk, recover ss.

The shared secret ss is used as part of the input keying material to HKDF.

2.2.2 ML-DSA-2 (ML-DSA-2)

ML-DSA-2 is a lattice-based signature scheme. This project uses the `dilithium2` variant via the `pqcrypto-dilithium` crate. The API (as wrapped by `sig.rs`) is:

- `generate_keys()` $\rightarrow$ (pk, sk)
- `sign_detached(msg, sk)` $\rightarrow$ sig
- `verify_detached(msg, sigma, pk)` $\rightarrow$ success/failure success/failure

The current codebase includes these helpers and tests; actual integration into the handshake for authentication is summarized in Chapter 7 as future work.

2.3 Classical Primitives

2.3.1 X25519 (X25519)

For hybrid operation, the project uses X25519 via the `x25519-dalek` crate. Each side generates an ephemeral secret and public key:

$$\mathsf{pk} = \mathsf{X25519}(sk, G),$$

and the shared secret is:

$$ss_{\text{classical}} = \mathsf{X25519}(sk_{\text{local}}, pk_{\text{peer}}).$$

2.3.2 Authenticated Encryption (ChaCha20-Poly1305)

Tunnel data is protected with ChaCha20-Poly1305 using the `chacha20poly1305` crate. It provides authenticated encryption with a 256-bit key and 96-bit nonce. In `aead.rs`, the functions are:

- `aead_seal(key, nonce, aad, plaintext)`.
- `aead_open(key, nonce, aad, ciphertext)`.

2.3.3 HKDF over SHA-256

Key derivation is based on HKDF with SHA-256, implemented via the `hkdf` crate. The transcript hash is used as the salt; the concatenation of classical and PQ shared secrets is used as input keying material.

Chapter 3

System and Protocol Requirements

3.1 Functional Requirements

The QuantumSafe-SMB-Link system is designed to satisfy the following functional requirements:

- Act as an SMB tunnel: accept incoming SMB connections on a local address and forward them to a configured remote SMB server through an encrypted channel.
- Perform a key-establishment handshake before any SMB data is exchanged.
- Support both hybrid (X25519 + ML-KEM-768) and pure post-quantum (ML-KEM-768-only) modes.
- Provide a simple command-line interface to configure server/client roles, addresses, and cryptographic options.

3.2 Security Requirements

The security requirements are:

- **Confidentiality:** SMB payloads must remain confidential against network attackers.
- **Integrity:** Modifications to tunnel traffic must be detected and rejected.

- **Forward secrecy:** Compromise of long-term keys must not reveal past session keys.
- **Post-quantum security:** Session keys should remain secure against quantum attacks due to the ML-KEM-768 component.
- **Downgrade resistance:** When both parties support PQ/hybrid suites, the protocol should not silently fall back to weaker classical-only modes.

3.3 Non-Functional Requirements

- **Performance:** The handshake overhead must be acceptable, and steady-state tunnel throughput should approach that of raw TCP.
- **Portability:** The Rust code should compile and run on major operating systems supported by the Rust toolchain.
- **Robustness:** Invalid or malformed handshake messages must be handled gracefully, with precise error messages for debugging.

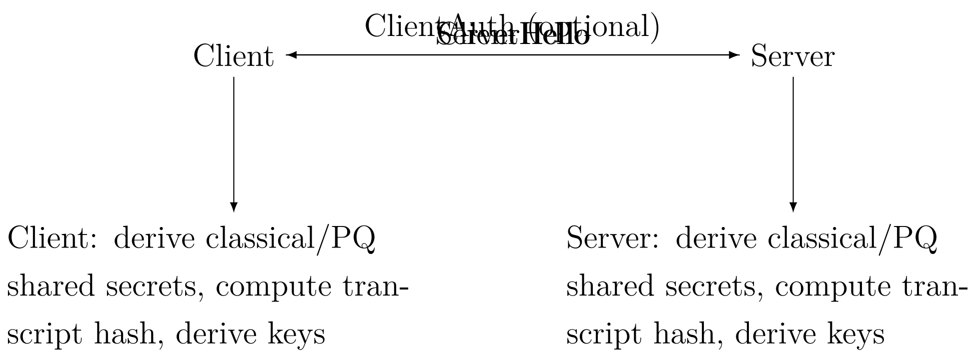
Chapter 4

Protocol Design

4.1 High-Level Flow

The QuantumSafe-SMB-Link protocol operates over a single TCP connection, with the following phases:

1. **Handshake phase:** Client and server exchange `ClientHello` and `ServerHello` messages, negotiate a cryptographic suite, perform hybrid key establishment, and derive symmetric keys.
2. **Data phase:** SMB payloads are encrypted, framed, and relayed between tunnel endpoints.



ClientAuth (optional)
ServerHello
ClientHello
Client
Server
Client: derive classical/PQ shared secrets, compute transcript hash, derive keys
Server: derive classical/PQ shared secrets, compute transcript hash, derive keys

Figure 4.1: High-level handshake flow between client and server.

4.2 Message Types and Magic Values

`kem.rs` defines constant magic values and protocol version:

- `CLIENT_HELLO_MAGIC = "QSLH"`.

- `SERVER_HELLO_MAGIC = "QSLS"`.
- `CLIENT_AUTH_MAGIC = "QSLA"`.
- `PROTOCOL_VERSION = 1`.

Each handshake message starts with a magic constant and protocol version, followed by a length-prefixed payload. Chapter A provides the binary layout.

4.3 Capabilities and Algorithm Negotiation

Algorithm capabilities are represented by the `Capabilities` struct:

- A list of supported KEM algorithms (ML-KEM-768, future variants).
- A list of supported signature algorithms (ML-DSA-2, etc.).
- A boolean `hybrid` flag.
- A boolean `pq_required` flag.

These are encoded as TLV (type–length–value) pairs, with tags:

- `TLV_KEM_LIST = 0x0001`.
- `TLV_SIG_LIST = 0x0002`.
- `TLV_HYBRID = 0x0003`.
- `TLV_PQ_REQUIRED = 0x0004`.

The decoding logic checks alignment and length, rejecting malformed messages (e.g., KEM list not aligned to 2-byte IDs).

The server selects a `ChosenSuite` that belongs to the intersection of client and server capabilities. The `pq_required` flag prevents the server from selecting a non-PQ suite when the client insists on post-quantum security.

4.4 ClientHello and ServerHello

4.4.1 ClientHello Structure

The `ClientHello` struct (simplified) is:

```
pub struct ClientHello {
    pub random: [u8; 32],
    pub x_pub: [u8; 32],
    pub kyber_pub: Vec<u8>,
    pub caps: Capabilities,
    pub caps_encoded: Vec<u8>,
    // internal: optional EphemeralSecret and Kyber secret
}
```

- random is filled with 32 bytes from OsRng.
- x_pub is the client's X25519 public key (zeros if hybrid is disabled).
- kyber_pub is the serialized ML-KEM-768 public key.
- caps and caps_encoded represent capabilities and their binary encoding.

4.4.2 ServerHello Structure

The ServerHello struct is:

```
pub struct ServerHello {
    pub random: [u8; 32],
    pub x_pub: [u8; 32],
    pub kyber_ct: Vec<u8>,
    pub chosen_suite: ChosenSuite,
    pub chosen_suite_encoded: Vec<u8>,
    pub signature: Vec<u8>,
}
```

- random is server-generated.
- x_pub is server X25519 public key (zeros when not hybrid).
- kyber_ct is the Kyber ciphertext encapsulated to the client's public key.
- chosen_suite describes the selected KEM/signature suite and hybrid flag.
- signature is reserved for future Dilithium signatures over the transcript.

4.5 Shared Secrets and Transcript Hash

The `Shared` struct contains:

```
pub struct Shared {
    pub classical: Option<[u8; 32]>,
    pub pq: Vec<u8>,
}
```

- `classical` is `Some(shared_x25519)` in hybrid mode and `None` otherwise.
- `pq` is the Kyber shared secret.

The handshake transcript consists of a deterministic concatenation of:

1. Client random.
2. Server random.
3. Client and server X25519 public keys.
4. Client Kyber public key and server Kyber ciphertext.
5. Encoded capabilities and chosen suite.

This transcript is hashed with SHA-256 and used as the salt for HKDF. The input keying material is constructed by concatenating the classical shared secret (if present) and the PQ shared secret.

Chapter 5

Implementation in Rust

5.1 Project Structure

The repository contains the following top-level elements:

- `Cargo.toml`: crate metadata and dependencies.
- `src/main.rs`: CLI and entry point.
- `src/net.rs`: networking logic and tunnel orchestration.
- `src/kem.rs`: handshake and hybrid KEM implementation.
- `src/aead.rs`: HKDF-based session keys and AEAD helpers.
- `src/frame.rs`: frame format for encrypted records.
- `src/sig.rs`: Dilithium signature wrapper.
- `src/crypto.rs`: legacy helpers (kept for reference).
- `docs/protocol.md`: informal documentation of the protocol.

5.2 Command-Line Interface (main.rs)

The CLI uses the `clap` crate with subcommands for client and server roles:

```
#[derive(Parser)]
#[command(name = "QuantumSafe-SMB-Link", version,
          about = "Post-quantum SMB tunnel")]
struct Cli {
    #[command(subcommand)]
    command: Command,
}

#[derive(Subcommand)]
enum Command {
    Server(ServerOpts),
    Client(ClientOpts),
}
```

Both `ServerOpts` and `ClientOpts` (defined with `#[derive(Args)]`) include:

- A local bind/dial address (`listen` or `dial`).
- A remote SMB server address (`forward`).
- Paths to Dilithium keys (for future authenticated mode).
- Flags for hybrid mode and PQ requirement.

The function `init_tracing()` configures `tracing-subscriber` with an `EnvFilter`, enabling log-level control via environment variables. Keys are loaded with:

```
fn load_key(path: &PathBuf) -> Result<Arc<[u8]>> {
    let bytes = fs::read(path)
        .with_context(|| format!("failed to read key {:?}",
           path))?;
    Ok(Arc::from(bytes))
}
```

5.3 Networking and Tunnel Logic (net.rs)

5.3.1 Server Mode

In server mode, `run_server` binds a `TcpListener` and accepts incoming tunnel connections:

- For each accepted connection:
    1. Perform the server-side handshake via `server_handshake`.
    2. Connect to the target SMB server.
    3. Start bidirectional encrypted pumping of data using `pump_streams`.

5.3.2 Client Mode

In client mode, `run_client` listens for local SMB connections, and for each:

1. Connects to the remote tunnel peer.
2. Executes `client_handshake`.
3. Pipes SMB data between the local client and the remote tunnel using the derived keys.

5.3.3 Handshake Functions

The networking layer calls into `kem.rs` as follows:

- `client_handshake(stream, config)`:
  1. Build `ClientHello` using `ClientHello::new`.
  2. Send encoded `ClientHello`.
  3. Receive and decode `ServerHello`.
  4. Compute `Shared` secrets and derive `SessionKeys`.
- `server_handshake(stream, config)`:
  1. Read and decode `ClientHello`.
  2. Generate Kyber and X25519 components.
  3. Build `ServerHello` and send it.
  4. Derive `Shared` and `SessionKeys`.

The handshake functions also compute associated data (AAD) for AEAD, typically including a hash of the transcript to bind the data channel to the negotiated keys.

5.3.4 Framed Pumping

The function `pump_streams` splits the tunnel and target TCP streams into read-/write halves and concurrently:

- Reads plaintext from the SMB side, encrypts it into a frame, and writes it to the tunnel.

- Reads encrypted frames from the tunnel, decrypts them, and writes plaintext to the SMB side.

Nonce management is handled by `NonceCounter` from `aead.rs`, ensuring that each frame uses a unique nonce.

5.4 Authenticated Encryption and Key Schedule (aead.rs)

5.4.1 SessionKeys and Roles

The `SessionKeys` struct stores transmit (tx) and receive (rx) keys:

```
#[derive(Clone)]
pub struct SessionKeys {
    pub tx: [u8; 32],
    pub rx: [u8; 32],
}

#[derive(Clone, Copy, Debug, Eq, PartialEq)]
pub enum SessionRole {
    Client,
    Server,
}
```

The function `derive_session_keys(shared, transcript_salt, role)`:

- Builds input keying material by concatenating classical and PQ secrets.
- Initializes HKDF with the transcript salt and IKM.
- Expands to two 256-bit keys labeled `"QuantumSafe tx"` and `"QuantumSafe rx"`.
- Swaps roles so that the client's `tx` matches the server's `rx`, and vice versa.

5.4.2 NonceCounter

`NonceCounter` wraps a 64-bit counter and exposes:

- `next()` → 96-bit nonce built from the counter and fixed prefix.

This yields a unique nonce per frame until the counter wraps (an event considered beyond practical usage limits).

5.4.3 AEAD Wrapper Functions

The functions:

```
pub fn aead_seal(key: &[u8; 32], nonce12: &[u8; 12],
                 plaintext: &[u8], aad: &[u8]) -> Result<Vec<
                    u8>>

pub fn aead_open(key: &[u8; 32], nonce12: &[u8; 12],
                 ciphertext: &[u8], aad: &[u8]) -> Result<Vec
                    <u8>>
```

wrap the underlying `ChaCha20Poly1305` implementation, mapping errors into `anyhow::Error` with clear messages.

5.5 Framing (frame.rs)

The framing layer defines a record format:

Field	Type	Description
`len`	u32 LE	Total length of `nonce` + `ct`
`nonce`	u64 LE	Per-frame nonce (counter)
`ct`	bytes	Ciphertext from AEAD

`write_frame` and `read_frame` are implemented in terms of `AsyncWriteExt` and `AsyncReadExt` and bail out on short reads or invalid lengths.

5.6 Dilithium Signature Helpers (sig.rs)

`sig.rs` provides a thin abstraction over the `dilithium2` crate:

```
pub fn sign_detached(msg: &[u8], sk_bytes: &[u8]) -> Result<
   Vec<u8>> { ... }

pub fn verify_detached(msg: &[u8], sig: &[u8], pk_bytes: &[u8
   ]) -> Result<()> { ... }
```

A unit test checks round-trip correctness, verifying that a signature on `"QuantumSafe-SMB-Link"` under a freshly generated keypair will be accepted.

In future iterations, these functions can be used to sign the handshake transcript, providing mutual authentication.

5.7 Legacy Crypto Helpers (crypto.rs)

`crypto.rs` contains earlier AEAD and HKDF helper functions that are superseded by `aead.rs`. They serve as reference for the evolution of the design and are kept for completeness.

Chapter 6

Performance Considerations

6.1 Benchmarking Methodology

To evaluate the performance of QuantumSafe-SMB-Link, the following methodology is suggested:

- Run micro-benchmarks for:
  - ML-KEM-768 key generation, encapsulation, and decapsulation.
  - X25519 key generation and shared-secret computation.
  - AEAD sealing and opening for typical SMB frame sizes (e.g., 4 KiB, 16 KiB).
- Measure end-to-end handshake latency between tunnel endpoints.
- Measure SMB file transfer throughput over:
  1. Raw TCP (baseline).
  2. Classical-only (if implemented).
  3. Hybrid mode (X25519 + ML-KEM-768).
  4. Pure ML-KEM-768 mode.

6.2 Example Result Tables

6.2.1 Primitive-Level Benchmarks

Table 6.1: Example primitive-level benchmark structure (fill with measured values).

Operation	Mean time (µs)	Std. dev. (µs)
Kyber-768 KeyGen	–	–
Kyber-768 Encaps	–	–
Kyber-768 Decaps	–	–
X25519 KeyGen	–	–
X25519 SharedSecret	–	–
ChaCha20-Poly1305 Seal 4 KiB	–	–
ChaCha20-Poly1305 Open 4 KiB	–	–

6.2.2 Handshake and SMB Throughput

Table 6.2: Example handshake and throughput benchmark structure.

Mode	Handshake latency (ms)	Throughput (MB/s)
Raw TCP	0	–
Classical-only tunnel	–	–
Hybrid (X25519 + ML-KEM-768)	–	–
Pure ML-KEM-768	–	–

You can fill these tables with actual values obtained from experiments.

6.3 Qualitative Discussion

Even without precise numbers, we can predict several trends:

- **Handshake overhead:** Post-quantum key establishment adds computational cost and increases handshake message sizes due to larger keys and ciphertexts. However, this cost is paid only once per connection.
- **Steady-state performance:** After the handshake, performance is dominated by ChaCha20-Poly1305 AEAD and I/O. The PQ component does not affect steady-state throughput.

- **Hybrid vs PQ-only:** Hybrid mode performs both X25519 and ML-KEM-768 operations, roughly doubling the cost of key establishment. This is acceptable for connection-oriented SMB usage but should be acknowledged.

Chapter 7

Security Analysis

7.1 Security Goals Recap

The handshake aims to provide:

- Confidentiality and integrity of SMB data via AEAD.
- Post-quantum security of session keys (due to ML-KEM-768).
- Forward secrecy via ephemeral keys.
- Resistance to downgrade attacks.

7.2 Hybrid Security

In hybrid mode, the session key is derived from both classical and post-quantum shared secrets:

$$\mathsf{IKM} = ss_{\text{classical}} \parallel ss_{\text{pq}}.$$

HKDF with a transcript-based salt extracts a pseudorandom key used to derive traffic keys. Assuming HKDF behaves as a secure key derivation function and at least one of the shared secrets remains unpredictable to the adversary, the derived keys remain secure.

Thus, hybrid mode provides:

- Classical security if ML-KEM-768 is later broken but X25519 remains secure.
- Post-quantum security if X25519 is broken (e.g., by quantum computers) but ML-KEM-768 remains secure.

7.3 Transcript Binding and Downgrade Resistance

Because the transcript hash includes:

- Both parties' random nonces.
- All public keys and ciphertexts.
- The encoded capabilities and chosen suite.

any modification to these elements by an active attacker will lead to mismatched transcripts and thus mismatched session keys. As a result, AEAD decryption will fail and the connection will not succeed.

The `pq_required` flag further limits downgrade risk: if a client requires PQ, the server must choose a suite with a PQ-capable KEM, or abort.

When Dilithium signatures are incorporated over the handshake transcript, an active attacker will be cryptographically prevented from altering the transcript without detection. This is discussed as future work.

7.4 Forward Secrecy

Forward secrecy is achieved through:

- Fresh X25519 ephemeral secrets per handshake.
- Fresh ML-KEM-768 keypairs per handshake (as implemented in `kem.rs`).

Compromise of long-term keys (e.g., Dilithium signing keys) does not reveal past session keys, assuming ephemeral secrets are erased after use.

7.5 Implementation Considerations

7.5.1 Randomness

All keys and nonces use `OsRng`. The security of the protocol relies on the OS-provided CSPRNG. In production, entropy sources and failure modes should be carefully audited.

7.5.2 Error Handling and Side Channels

The use of `anyhow` yields user-friendly error messages. Care should be taken not to expose internal state or sensitive values in error messages in production. Also, while the underlying PQ libraries aim to be side-channel resistant, a full side-channel analysis is outside the scope of this course project.

7.5.3 Denial-of-Service

Because the server allocates buffers and performs expensive operations after reading the handshake, large or malformed messages could be used for DoS attacks. The implementation mitigates this with `MAX_HANDSHAKE_LEN`, rejecting handshakes exceeding a reasonable size.

Chapter 8

Conclusion and Future Work

This report has presented QuantumSafe-SMB-Link, a Rust-based prototype of a post-quantum hybrid tunnel for SMB. The project demonstrates how NIST-standardized primitives such as ML-KEM-768 and ML-DSA-2 can be integrated into a practical handshake and key schedule, complementing classical X25519 to provide hybrid security.

From a migration perspective, the design shows that:

- Hybrid key establishment can be cleanly added around existing application protocols like SMB without modifying SMB itself.
- Capability negotiation and a transcript-based key schedule offer a path to incremental deployment and downgrade resistance.
- The implementation overhead is modest, especially in Rust, where strong type-safety and crate ecosystems simplify cryptographic plumbing.

Future work includes:

- Integrating Dilithium signatures into the handshake for full authentication.
- Extending the capability negotiation to support multiple PQ parameter sets and other KEM/signature families.
- Running extensive benchmarks across platforms and SMB workloads, and optimizing TCP buffering and framing.
- Hardening the codebase with configuration files, logging policies, and tests against malformed inputs.

Bibliography

[1] NIST. *FIPS 203: Module-Lattice-Based Key-Encapsulation Mechanism Standard.* 2024.

[2] NIST. *FIPS 204: Module-Lattice-Based Digital Signature Standard.* 2024.

[3] D. J. Bernstein. *The ChaCha20-Poly1305 Authenticated Encryption.* 2015.

[4] Y. Nir and A. Langley. *ChaCha20 and Poly1305 for IETF Protocols.* RFC 8439, 2018.

[5] D. J. Bernstein. *Curve25519: high-speed elliptic-curve Diffie-Hellman.* 2006.

Appendix A

Wire Formats

This appendix specifies the on-the-wire layout of all handshake messages.

A.1 ClientHello

Field	Type	Description
Magic	4 bytes	`"QSLH"`
Version	u16 (BE)	Protocol version (1)
Length	u16 (BE)	Payload length
Random	32 bytes	Client random nonce
X25519 pk	32 bytes	Client public key (all zeros if not hybrid)
Kyber pk len	u16 (BE)	Length of Kyber public key
Kyber pk	variable	Kyber public key bytes
Caps len	u16 (BE)	Length of capabilities TLV block
Caps (TLVs)	variable	TLVs with KEM list, SIG list, hybrid, pq_required

A.2 ServerHello

Field	Type	Description
Magic	4 bytes	`"QSLS"`
Version	u16 (BE)	Protocol version (1)
Length	u16 (BE)	Payload length
Random	32 bytes	Server random nonce
X25519 pk	32 bytes	Server X25519 public key (zeros if not hybrid)
Kyber ct len	u16 (BE)	Length of Kyber ciphertext
Kyber ct	variable	Kyber ciphertext
Suite len	u16 (BE)	Length of chosen-suite encoding
Chosen suite	variable	Encoded KEM+SIG+hybrid selection
Sig len	u16 (BE)	Length of signature (0 if unused)
Signature	variable	Dilithium signature over transcript (future)

A.3 Encrypted Data Frame

Data frames follow the format described in Chapter 5: `len` (u32 LE), `nonce` (u64 LE), and ciphertext bytes.

Appendix B

CLI Usage

B.1 Server Mode Example

```
QuantumSafe-SMB-Link server \
    --listen 0.0.0.0:4444 \
    --forward 10.0.0.5:445 \
    --hybrid \
    --pq-required
```

B.2 Client Mode Example

```
QuantumSafe-SMB-Link client \
    --listen 127.0.0.1:1445 \
    --dial 192.0.2.10:4444 \
    --hybrid \
    --pq-required
```

In this configuration, SMB clients can connect to `127.0.0.1:1445`, and their traffic will be tunneled securely to the SMB server at `10.0.0.5:445` via the post-quantum-aware tunnel at `192.0.2.10:4444`.