The Design and Analysis of the Post Quantum Shell Protocol

John G. Underhill

Quantum Resistant Cryptographic Solutions Corporation

Abstract. The Post Quantum Shell (PQS) protocol is a Simplex, server authenticated, post-quantum secure channel that establishes a shared set of symmetric keys through an authenticated key-exchange flow based on ML-KEM, ML-DSA, and Keccak derived functions. The protocol binds configuration parameters, key identifiers, and the server's public verification key into a session cookie that is used with the encapsulated shared secret to key a cSHAKE based key-derivation function. The resulting transmit and receive keys support an authenticated encryption channel that uses either RCS or AES-GCM with the complete PQS header encoded as associated data. This paper

presents a formal security analysis of PQS in a server-only authenticated key-exchange model that reflects the design of the reference implementation. The analysis defines the security goals for server authentication, session key indistinguishability, forward secrecy for the client, and channel integrity with replay resistance. The handshake and data-channel behavior are specified in a Simplex AKE framework, and the main results are given through reductions to the IND-CCA security of ML-KEM, the EUF-CMA security of ML-DSA, and the pseudo-random properties of cSHAKE. The treatment includes explicit bounds, an examination of data-plane authenticity, and a discussion of implementation considerations such as state validation, key erasure, and constant-time behavior. The results show that, under standard assumptions for

the underlying post-quantum primitives, the PQS handshake and encrypted channel achieve the intended security properties and align with the operational behavior of the published PQS specification and codebase.

1 Introduction

1.1 Background and Motivation

The Post Quantum Shell (PQS) protocol is designed to provide a secure channel between a client and a server using post-quantum primitives for authentication and key establishment. The protocol follows a Simplex exchange structure in which the client authenticates the server through a long-term verification key, and the key-exchange phase is server-only authenticated. This separates the concerns of channel establishment from higher level user authentication, which may take place over the encrypted channel using application specific mechanisms.

The protocol uses ML-KEM for encapsulation, ML-DSA for server authentication, and Keccak based hashing and key-derivation functions. After the handshake, communication proceeds over an authenticated encryption channel instantiated with either RCS or AES-GCM, with each packet carrying a fixed-format header that is bound to the ciphertext through the AEAD associated-data mechanism. The simplicity of the exchange and the deterministic state structure make PQS practical for systems that require efficient connection establishment and secure packet processing under a post-quantum threat model.

1.2 Contributions

This work provides a complete provable security treatment of the PQS protocol, aligned with both the specification and the reference implementation. The main contributions are as follows.

First, the paper defines a Simplex server-only authenticated key-exchange (SOAKE) model that reflects the trust pattern of PQS and captures the acceptance conditions for client sessions. Second, it presents an engineering level description of the protocol that consolidates the wire format, handshake flow, packet structure, and state transitions as implemented in the codebase. Third, the work introduces formal security definitions tailored to PQS, including server authentication, session key indistinguishability, forward secrecy for the client, and channel integrity with replay resistance. Fourth, the analysis develops a sequence of hybrid games and reductions to the IND-CCA security of ML-KEM, the EUF-CMA security of ML-DSA, and the pseudo-random properties of cSHAKE. Fifth, the treatment includes implementation and deployment considerations, such as validation rules, key erasure, constant-time behavior, and the impact of packet timing constraints.

1.3 Organization of the Paper

The paper begins with an engineering description of PQS that outlines entities, state structures, packet formats, and the complete handshake and data-channel flow. This is followed by the preliminaries and model section, which introduces notation, cryptographic assumptions, and the Simplex server-only authenticated key-exchange framework. The formal specification then describes the handshake transcript, session cookie, KEM based shared secret, key derivation procedure, and packet acceptance conditions. The next section presents the security definitions for authentication, key indistinguishability, forward secrecy, and channel integrity. The provable security analysis establishes the main theorems using game based reductions and concrete bounds. Subsequent sections address data-channel security, attack analysis, implementation conformance, and performance considerations. The paper concludes with a summary of results and avenues for future work.

2 Engineering Description of the PQS Protocol

2.1 Entities and Long Term Keys

PQS runs between a client and a server that share a pinned server identity but do not mutually authenticate at the key exchange layer. On the client, the server is represented by a pinned verification record

$$(\mathsf{cfg}, \mathsf{kid}, \mathsf{pvk}, \mathsf{expiration}),$$

where cfg is a configuration identifier, kid is a 128 bit key identifier, pvk is the server's ML-DSA verification key, and expiration is an absolute expiry time. This record is stored as trusted local state and is not transmitted in the protocol.
The server holds the corresponding ML-DSA signing key pair $(\mathsf{ssk}, \mathsf{pvk})$ and maintains a keyring indexed by kid. Each entry carries the same configuration string and expiration time as the client pin. The private signing key ssk is used only to sign the Connect Response in the Simplex handshake.

2.2 Packet Format and Header Fields

All PQS messages share a common packet structure. Each packet begins with a fixed length header of 21 bytes followed by a variable length message body. The header fields are:

- Packet Flag (1 byte) identifying the message type.
- Packet Sequence (8 byte unsigned counter).
- Message Size (4 byte unsigned payload length).
- UTC Time (8 byte timestamp sampled at send time).

On the wire the fields are serialized in a canonical order as

$$\mathsf{hdr} = \mathsf{flag} \,\|\, \mathsf{seq} \,\|\, \mathsf{mlen} \,\|\, \mathsf{time},$$

with platform independent encoding for the integer fields. The message body is a byte string of length mlen and may be empty.
Before any cryptographic processing, the receiver reconstructs the header, checks that:

- the flag is allowed in the current connection state,
- the sequence number equals the expected receive counter or lies within the configured window,
- the message size does not exceed the protocol maximum and matches the expected structure for the flag,
- the timestamp lies within the allowed freshness window relative to local time.

If any of these checks fail the implementation signals an error condition, such as `pqs_error_packet_unsequenced` or `pqs_error_message_time_invalid`, and tears down the connection.

2.3 State Structures

The internal connection state structure maintains all long lived protocol state for a single PQS session. Conceptually it contains:

- a transport target and connection instance handle,
- a Cipher Send State and a Cipher Receive State, each holding an RCS or AES-GCM context with key and nonce,
- a transmit sequence counter and a receive sequence counter,
- a KEX flag tracking the current handshake or established state,
- a ratchet key reserved for future asymmetric or symmetric ratcheting,
- a session token and cookie fields used to bind the handshake transcript,
- an index or pointer into the client's pinned key store.

During the handshake each side also maintains a Simplex KEX state. The client KEX state holds:

- the key identifier and configuration from the pinned record,
- the server verification key pvk and its expiration,
- a freshly sampled session token,
- the session cookie sch,
- the KEM shared secret sec once derived.

The server KEX state holds the same public fields together with:

- a freshly sampled session token (512 bits),
- an ephemeral ML-KEM key pair $(\mathsf{pk}, \mathsf{sk})$,
- the shared secret sec obtained by decapsulation.

The session token and KEX flag couple the handshake and application layers, while sch and sec feed the key schedule.

2.4 Simplex Handshake Walkthrough

The Simplex handshake is a four message exchange carried in PQS packets:

$$\begin{array}{c}\text{Client} \rightarrow \text{Server: Connect Request, Exchange Request.}\\ \text{Server} \rightarrow \text{Client: Connect Response, Exchange Response.}\end{array}$$

We describe the sequence in implementation order.

Client Connect Request.

1. The client selects a pinned record $(\mathsf{cfg}, \mathsf{kid}, \mathsf{pvk}, \mathsf{expiration})$ for the target server and checks that $\mathsf{expiration}$ has not passed.
2. It initializes its KEX state with these values and samples a fresh session token.
3. It constructs a message body
$$m_1 = \mathsf{kid} \,\|\, \mathsf{cfg}.$$

4. It sets the transmit sequence counter to its initial value (for example zero), samples the current UTC time, and builds the header

$$\mathsf{hdr}_1 = (\mathsf{flag} = \mathsf{Connect\ Request}, \mathsf{seq}, \mathsf{mlen} = |m_1|, \mathsf{time}).$$

5. It serializes hdr_1 and sends the packet $\mathsf{hdr}_1 \,\|\, m_1$ over the transport.

Server processing of Connect Request and Connect Response.

1. The server receives a packet, parses hdr_1 and m_1, and validates:
   - $\mathsf{flag} = \mathsf{Connect\ Request}$,
   - seq matches the expected receive counter,
   - time is within the freshness window,
   - mlen equals the size of a $\mathsf{kid} \,\|\, \mathsf{cfg}$ payload.

   On failure it emits an appropriate Error Condition packet and aborts.
2. It parses m_1 into $(\mathsf{kid}, \mathsf{cfg})$ and uses kid to locate the corresponding verification key record $(\mathsf{cfg}', \mathsf{pvk}, \mathsf{expiration})$ in its keyring.
3. It checks that $\mathsf{cfg}' = \mathsf{cfg}$ and that $\mathsf{expiration}$ has not passed. If either check fails it sets the error code to `pqs_error_key_unrecognized` or `pqs_error_key_expired` and aborts.
4. It computes the session cookie

$$\mathsf{sch} = \mathsf{H}(\mathsf{cfg} \,\|\, \mathsf{kid} \,\|\, \mathsf{pvk}),$$

   using SHA3 256, and stores sch and the pinned public values in its server KEX state.
5. It samples a fresh session token and writes it to the server KEX state.
6. It generates an ephemeral ML-KEM key pair $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KEM.KeyGen}()$ and stores both keys in the server KEX state.
7. It prepares the Connect Response header hdr_2 using the next sequence number and current UTC time, with flag set to Connect Response and message length equal to the size of $\mathsf{spkh} \,\|\, \mathsf{pk}$, where spkh is defined below.
8. It serializes hdr_2 to a byte string, computes

$$h = \mathsf{SHA3}(\mathsf{hdr}_2 \,\|\, \mathsf{pk}),$$

   and signs h with its long term signing key $\sigma \leftarrow \mathsf{SIG.Sign}(\mathsf{ssk}, h)$.
9. It serializes σ into spkh and sets the response body

$$m_2 = \mathsf{spkh} \,\|\, \mathsf{pk}.$$

10. It transmits the packet $\mathsf{hdr}_2 \,\|\, m_2$ to the client.

Client processing of Connect Response.

1. The client receives $\mathsf{hdr}_2 \,\|\, m_2$ and validates:
   - the flag equals Connect Response in the correct state,
   - the sequence number equals the next expected value,

- the timestamp is within the acceptance window,
- the message length matches the expected size of $\mathsf{spkh} \,\|\, \mathsf{pk}$.

Failure produces an error such as `pqs_error_connect_failure`.

2. It parses m_2 into $(\mathsf{spkh}, \mathsf{pk})$ and recomputes
$$h = \mathsf{SHA3}(\mathsf{hdr}_2 \,\|\, \mathsf{pk}).$$
3. Using the pinned verification key pvk it checks $\mathsf{SIG.Verify}(\mathsf{pvk}, h, \sigma) = 1$, where σ is recovered from spkh. If verification fails the client signals `pqs_error_hash_invalid` or `pqs_error_verify_failure` and aborts.
4. It computes its own copy of the session cookie
$$\mathsf{sch} = \mathsf{H}(\mathsf{cfg} \,\|\, \mathsf{kid} \,\|\, \mathsf{pvk})$$
and stores sch and pk in the client KEX state.

Exchange Request and Exchange Response.

1. The client performs ML-KEM encapsulation
$$(\mathsf{cpt}, \mathsf{sec}) \leftarrow \mathsf{KEM.Encaps}(\mathsf{pk}),$$
stores sec in its KEX state, and erases any temporary KEM material.
2. It constructs the Exchange Request header hdr_3 with flag set to Exchange Request, the next sequence number, message length equal to $|\mathsf{cpt}|$, and a fresh UTC time. It sets the body $m_3 = \mathsf{cpt}$ and transmits $\mathsf{hdr}_3 \,\|\, m_3$.
3. The server receives $\mathsf{hdr}_3 \,\|\, m_3$, validates the header in the same way as before but with expected flag Exchange Request and expected length $|\mathsf{cpt}|$. On failure it sets `pqs_error_exchange_failure` and aborts.
4. It decapsulates
$$\mathsf{sec} \leftarrow \mathsf{KEM.Decaps}(\mathsf{sk}, \mathsf{cpt}),$$
stores sec in the server KEX state, and erases sk from memory to obtain forward secrecy for the client.
5. It builds the Exchange Response header hdr_4 with flag set to Exchange Response, the next sequence number, message length zero, and a fresh UTC time. The body m_4 is empty. It sends $\mathsf{hdr}_4 \,\|\, m_4$ to the client.
6. The client receives hdr_4, validates the header with expected flag Exchange Response and zero message length, and on success marks the handshake as complete.

At this point both endpoints hold the same shared secret sec and cookie sch in their KEX state and transition to key schedule and cipher initialization. Any failure in the steps above triggers an Error Condition packet with the appropriate error code and immediate connection termination.

2.5 Key Schedule and Cipher State

The PQS key schedule derives directional AEAD keys and nonces from sec and sch and loads them into RCS or AES-GCM cipher states.

1. Both sides recompute or retrieve the session cookie $\mathsf{sch} = \mathsf{H}(\mathsf{cfg} \,\|\, \mathsf{kid} \,\|\, \mathsf{pvk})$.

2. They invoke a Keccak based XOF such as cSHAKE with input sec and customization string sch to obtain a pseudo-random stream

$$\mathsf{prnd} \leftarrow \mathsf{cSHAKE}(\mathsf{sec}, \mathsf{sch}, \mathsf{outlen}),$$

where outlen is sufficient to cover two AEAD keys and two nonces.

3. They parse prnd into

$$(k_1, n_1, k_2, n_2),$$

where each k_i has the length required by the chosen AEAD (RCS or AES-GCM) and each n_i has the required nonce length.

4. Directional assignment is asymmetric:
   - the client uses (k_1, n_1) for its Cipher Send State and (k_2, n_2) for its Cipher Receive State,
   - the server uses (k_2, n_2) for its Cipher Send State and (k_1, n_1) for its Cipher Receive State.

   This guarantees that each key nonce pair is used in exactly one direction.

5. Each side initializes its chosen AEAD instance:
   - for RCS, by running the cSHAKE based key expansion with key k_i, customization identifying PQS, and nonce n_i to derive round keys and a MAC key,
   - for AES-GCM, by loading k_i as the AES key and n_i as the initial nonce.

6. The initialized cipher contexts are stored in the Cipher Send State and Cipher Receive State fields of the connection state and the KEX state is cleared.

Because sec is IND-CCA secure, sch is derived from pinned public parameters, and cSHAKE behaves as a pseudo-random function, the resulting keys and nonces can be treated as independent pseudo-random values conditioned on the handshake being authentic.

2.6 Encrypted Data Phase

Once the key schedule completes and the connection flag records that the session is established, all application data is sent in packets with flag set to Encrypted Message.

Sending encrypted data. For each outbound message m:

1. The sender increments its transmit sequence counter and samples the current UTC time.

2. It constructs a header

$$\mathsf{hdr} = (\mathsf{flag} = \mathsf{Encrypted\ Message}, \mathsf{seq}, \mathsf{mlen}, \mathsf{time}),$$

where mlen will be set to the length of the ciphertext plus authentication tag.

3. It serializes hdr and passes the header bytes as associated data to the Cipher Send State using `pqs_cipher_set_associated`.

4. It invokes `pqs_cipher_transform` on the plaintext m to obtain ciphertext c and tag t. For RCS, this uses Rijndael 256 in counter mode with key and nonce from the send state and computes a KMAC tag over the associated data, nonce, ciphertext, and encoded lengths. For AES-GCM, this uses the standard GCM encryption and tag generation with the same associated data.

5. It sets $\mathsf{mlen} = |c \,\|\, t|$ in the header, writes $c \,\|\, t$ as the message body, and sends the packet.

Receiving encrypted data. On receipt of a packet with flag Encrypted Message:

1. The receiver parses and validates the header:
   - the flag must equal Encrypted Message,
   - the sequence number must equal or advance the receive counter within the allowed window,
   - the message length must be within bounds and at least the tag length,
   - the timestamp must be within the acceptance window.

   Failure yields errors such as `pqs_error_packet_unsequenced`, `pqs_error_message_time_invalid`, or `pqs_error_invalid_input`, and triggers teardown.
2. It splits the body into ciphertext c and tag t, reconstructs and serializes the header, and supplies the header bytes as associated data to the Cipher Receive State.
3. It invokes `pqs_cipher_transform` in decrypt mode. For RCS, decryption first recomputes the KMAC tag over the associated data, nonce, ciphertext, and lengths and compares it to t. For AES-GCM, it verifies the GCM tag. Only if the tag matches does it recover the plaintext.
4. If tag verification fails the implementation raises `pqs_error_authentication_failure` or `pqs_error_decryption_failure` and erases the connection state.

The inclusion of the serialized header in the associated data binds the flag, sequence number, message length, and timestamp into the AEAD transcript. Under the INT-CTXT and IND-CCA security of RCS or AES-GCM, any new forged triple (hdr, c, t) that passes header validation is accepted only with negligible probability.

2.7 Control Packets, Keepalive and Errors

In addition to handshake and encrypted data packets, PQS defines control packets with dedicated flags, including:

- Keep Alive Request and Keep Alive Response for liveness probing,
- Remote Connected and Remote Terminated for connection state notifications,
- Connection Terminated and Error Condition for explicit shutdown and error reporting.

Before the handshake completes these packets are processed in clear but still pass through header validation and sequence and time checks. After the session is established, any control packet that carries a body is sent through the same AEAD path as application data, with the control flag set in the header and the header serialized as associated data. Error codes from the `pqs_errors` enumeration identify the precise cause of failure and are mapped to Error Condition packets where appropriate. In all fatal cases the implementation zeroizes keys and cipher state before closing the underlying transport.

3 Preliminaries and Model

3.1 Notation

We use standard notation for bit strings, probability spaces, and cryptographic algorithms. A bit string of length ℓ is written as $x \in \{0,1\}^\ell$. Concatenation is denoted by $x \parallel y$. Sampling a uniformly random value from a set $\mathcal{X}$ is written $x \leftarrow \mathcal{X}$, and sampling from a probabilistic algorithm $\mathcal{A}$ on input u is written $x \leftarrow \mathcal{A}(u)$. Deterministic algorithms are written $x = \mathcal{A}(u)$.
Headers are treated both as structured tuples and as canonical serialized byte strings. If

$$\mathsf{hdr} = (\mathsf{flag}, \mathsf{seq}, \mathsf{mlen}, \mathsf{time}),$$

then $\mathsf{encode}(\mathsf{hdr})$ denotes its 21-byte serialization in network order exactly as produced by `pqs_packet_header_serialize`. The fields seq and time are 64-bit integers, and mlen is a 32-bit integer.
We model the KEM, signature scheme, hash function, XOF, and AEAD as abstract interfaces:

- $\mathsf{KEM.KeyGen}(), \mathsf{KEM.Encaps}(pk), \mathsf{KEM.Decaps}(sk, c)$,
- $\mathsf{SIG.Sign}(ssk, m), \mathsf{SIG.Verify}(pvk, m, \sigma)$,
- $\mathsf{H} : \{0,1\}^* \to \{0,1\}^{256}$ representing SHA3-256,
- $\mathsf{XOF}(s, \mathsf{cust})$ representing cSHAKE with input string s and customization string cust,
- $\mathsf{AEAD.Enc}(k, n, a, m)$ and $\mathsf{AEAD.Dec}(k, n, a, c, t)$, modeling either RCS or AES-GCM.

The associated data a is always the serialized header $\mathsf{encode}(\mathsf{hdr})$. Nonce n is derived during the key schedule and is never reused.
All events and probabilities are taken over the random coins of the adversary and all randomized protocol components.

3.2 Cryptographic Primitives

ML-KEM. We assume that ML-KEM is an IND-CCA secure key encapsulation mechanism. For any probabilistic polynomial-time adversary $\mathcal{A}$,

$$\mathsf{Adv}^{\text{IND-CCA}}_{\text{ML-KEM}}(\mathcal{A})$$

denotes the standard indistinguishability advantage in the post-quantum setting.

ML-DSA. The signature scheme is ML-DSA, modeled as an EUF-CMA secure signature scheme. For any adversary $\mathcal{A}$,

$$\mathsf{Adv}^{\text{EUF-CMA}}_{\text{ML-DSA}}(\mathcal{A})$$

denotes its existential unforgeability advantage.

SHA3-256 and cSHAKE. SHA3-256 is modeled as a collision-resistant and second-preimage resistant hash function. cSHAKE is modeled as a pseudo-random function keyed by its input string, customization string, and domain separation parameters. For any adversary $\mathcal{A}$,

$$\mathsf{Adv}^{\text{PRF}}_{\text{cSHAKE}}(\mathcal{A})$$

denotes its advantage in distinguishing cSHAKE output from uniform randomness.

AEAD: RCS and AES-GCM. PQS uses an AEAD scheme instantiated by either:

- RCS, which uses Rijndael-256 in counter mode for confidentiality and KMAC for integrity, or
- AES-GCM, which uses AES-128 or AES-256 in counter mode together with GMAC.

Both are modeled as providing IND-CCA security for confidentiality and INT-CTXT security for integrity. For any adversary $\mathcal{A}$,

$$\mathsf{Adv}^{\text{IND-CCA}}_{\mathsf{AEAD}}(\mathcal{A}), \qquad \mathsf{Adv}^{\text{INT-CTXT}}_{\mathsf{AEAD}}(\mathcal{A})$$

denote the corresponding advantages. The formal properties of RCS follow the analysis in the RCS specification: the KMAC tag covers the associated data, nonce, ciphertext, and encoded lengths, ensuring that any modification to the header or ciphertext leads to a failed verification.

3.3 Communication and Trust Model

The protocol is executed between a single client C and a single server S connected by an adversarial network. The adversary controls packet delivery, reordering, dropping, and injection, as well as timing.

Key distribution occurs before the protocol begins. The client is provisioned with a pinned server verification key $(\mathsf{cfg}, \mathsf{kid}, \mathsf{pvk}, \mathsf{expiration})$. The server holds the corresponding signing key. No assumptions are made about the secrecy of pvk; it is public.

PQS follows a Simplex trust pattern: the client authenticates the server using pvk, but the server does not authenticate the client during the key-exchange phase. Any client authentication occurs at the application layer after the encrypted channel is established.

3.4 Adversarial Capabilities

The adversary $\mathcal{A}$ is a probabilistic polynomial-time machine with the following capabilities:

- **Full network control:** $\mathcal{A}$ may read, drop, modify, inject, delay, or reorder packets.
- **KEM oracle access:** in the IND-CCA experiment, $\mathcal{A}$ may query decapsulation oracles except on the challenge ciphertext.
- **Signature oracle access:** $\mathcal{A}$ may request signatures on arbitrary messages under the server's long term key.
- **AEAD oracle access:** $\mathcal{A}$ may submit chosen messages, ciphertexts, and associated data to encryption and decryption oracles, except on challenge inputs.
- **Adaptive compromise:** $\mathcal{A}$ may corrupt:
  - the client's session state,
  - the server's session state,
  - long term keys of either party,

  subject to freshness conditions in the security definitions.

These capabilities model an active man-in-the-middle adversary with access to side-channel free cryptographic oracles.

3.5 Security Objectives

The PQS protocol aims to satisfy the following security goals in the Simplex setting:

- **Server authentication:** the client accepts a session only if the Connect Response signature verifies under its pinned pvk.
- **Session key indistinguishability:** for a fresh client session that authenticated the correct server, the derived symmetric keys are indistinguishable from random even under active attack.
- **Forward secrecy for the client:** compromise of long term signing keys after a session completes does not reveal the session keys, due to use of an ephemeral ML-KEM exchange.
- **Basic post compromise guarantees:** compromise of the session keys affects only the compromised session; the ephemeral KEM ensures that later sessions remain secure after rekeying.
- **Channel integrity:** modification of headers or ciphertexts is detected with negligible probability due to AEAD INT-CTXT security and binding of the entire packet header as associated data.
- **Replay and reordering resistance:** packets outside the valid sequence window or timestamp window are rejected regardless of cryptographic validity.
- **Configuration binding and downgrade protection:** the session cookie

  $$\mathsf{sch} = \mathsf{H}(\mathsf{cfg} \parallel \mathsf{kid} \parallel \mathsf{pvk})$$

  binds the session to a unique configuration and verification key, preventing downgrade attacks that alter cfg or substitute an alternate server key.

These inform the security definitions of the following sections: the Simplex server-only authenticated key-exchange model, the session key indistinguishability experiment, the forward secrecy experiment, and the packet integrity and replay games.

4 Formal Specification of PQS

4.1 Simplex AKE Syntax

We specify PQS as a Simplex authenticated key-exchange protocol between a client C and a server S. The protocol exposes two algorithmic interfaces:

- $\mathsf{InitClient}(\mathsf{cfg}, \mathsf{kid}, \mathsf{pvk}, \mathsf{expiration})$: Initializes a client instance with pinned verification key metadata.
- $\mathsf{InitServer}(\mathsf{ssk}, \mathsf{pvk}, \mathsf{cfg}, \mathsf{kid}, \mathsf{expiration})$: Initializes a server instance with long term signing keys and metadata.

Each instance maintains a local state st containing its session token, session cookie, ephemeral or long term keys, and the negotiated shared secret.

A session is defined to complete on the client when it receives a valid Connect Response and valid Exchange Response, and on the server when it has successfully received a verified Connect Request and decapsulated the KEM ciphertext from the Exchange Request.

At completion, both parties output a session key vector

$$\mathsf{SK} = (k_1, n_1, k_2, n_2),$$

where the ordering reflects directional assignment as defined below. If any validation or cryptographic check fails, the party outputs $\perp$.

4.2 Handshake Transcript and Session State

The handshake is a four-message transcript consisting of:

$$\mathsf{CRQ}, \quad \mathsf{CRP}, \quad \mathsf{XRQ}, \quad \mathsf{XRP},$$

corresponding to Connect Request, Connect Response, Exchange Request, and Exchange Response.
Each transcript element is a PQS packet

$$\mathsf{pkt} = \mathsf{hdr} \parallel m,$$

where the header is the tuple

$$\mathsf{hdr} = (\mathsf{flag}, \mathsf{seq}, \mathsf{mlen}, \mathsf{time})$$

and $\mathsf{encode}(\mathsf{hdr})$ is its canonical serialization.
The message bodies are:

$$\begin{aligned} m_{\mathsf{CRQ}} &= \mathsf{kid} \parallel \mathsf{cfg}, \\ m_{\mathsf{CRP}} &= \mathsf{spkh} \parallel \mathsf{pk}, \\ m_{\mathsf{XRQ}} &= \mathsf{cpt}, \\ m_{\mathsf{XRP}} &= \epsilon, \end{aligned}$$

where pk and sk are the server's ephemeral ML-KEM keys, spkh is the signature of the hash $\mathsf{SHA3}(\mathsf{encode}(\mathsf{hdr}_{\mathsf{CRP}}) \parallel \mathsf{pk})$, and cpt is the client's ML-KEM ciphertext.
The local state is updated as follows:

- On sending CRQ, the client stores $(\mathsf{cfg}, \mathsf{kid}, \mathsf{pvk}, \mathsf{expiration})$ and computes its session cookie.
- On receiving CRQ, the server loads its pinned verification key record and computes the session cookie.
- On receiving CRP, the client verifies the signature, stores pk, and updates its state.
- On receiving XRQ, the server decapsulates cpt to obtain sec.

The handshake completes on both sides after XRP is validated.

4.3 Session Cookie and Key Derivation Function

The session cookie binds the configuration, key identifier, and server verification key:

$$\mathsf{sch} = \mathsf{H}(\mathsf{cfg} \parallel \mathsf{kid} \parallel \mathsf{pvk}).$$

The shared secret sec is the output of the ML-KEM exchange:

$$(\mathsf{cpt}, \mathsf{sec}) \leftarrow \mathsf{KEM.Encaps}(\mathsf{pk}), \qquad \mathsf{sec} \leftarrow \mathsf{KEM.Decaps}(\mathsf{sk}, \mathsf{cpt}).$$

The PQS key schedule applies cSHAKE to derive sufficient pseudo-random material:

$$\mathsf{prnd} \leftarrow \mathsf{XOF}(\mathsf{sec}, \mathsf{sch}),$$

and parses it deterministically into:

$$\mathsf{prnd} = k_1 \parallel n_1 \parallel k_2 \parallel n_2,$$

where k_i are AEAD keys and n_i are nonces of appropriate length.

4.4 Directional Keys and Nonces

PQS assigns directional keys asymmetrically:

$$\begin{aligned} \text{Client:} \quad & \mathsf{Send} = (k_1, n_1), \quad \mathsf{Receive} = (k_2, n_2), \\ \text{Server:} \quad & \mathsf{Send} = (k_2, n_2), \quad \mathsf{Receive} = (k_1, n_1). \end{aligned}$$

No key or nonce pair is used in both directions. The separation prevents reflection or loopback attacks and enforces unidirectional AEAD semantics per direction.

4.5 Authenticated Encryption and Header Binding

For any PQS packet with header hdr and body m, the sender computes:

$$(c, t) \leftarrow \mathsf{AEAD.Enc}(k, n, \mathsf{encode}(\mathsf{hdr}), m),$$

and transmits $\mathsf{hdr} \parallel c \parallel t$.
The associated data is exactly the serialized header and includes:

$$(\mathsf{flag}, \mathsf{seq}, \mathsf{mlen}, \mathsf{time}).$$

At the receiver side, upon reconstructing the header and splitting the body into ciphertext c and tag t, decryption proceeds as:

$$m \leftarrow \mathsf{AEAD.Dec}(k, n, \mathsf{encode}(\mathsf{hdr}), c, t).$$

If the tag fails, decryption returns $\perp$. The binding of the entire header ensures that any change to flag, sequence number, message length, or timestamp invalidates the tag.

4.6 Packet Acceptance Predicate

A packet $\mathsf{pkt} = \mathsf{hdr} \parallel x$ is accepted by a party with local state st if and only if all of the following hold:

1. **Sequence validity:**

   $$\mathsf{hdr.seq} = \mathsf{st.rxseq} + 1 \quad \text{or} \quad \mathsf{hdr.seq} \in \text{valid window}.$$

   Upon acceptance, $\mathsf{st.rxseq} \leftarrow \mathsf{hdr.seq}$.

2. **Timestamp validity:**

   $$|\mathsf{hdr.time} - \mathsf{localTime}| \leq \Delta,$$

   where Δ is the configured freshness bound.

3. **Message length validity:**

   $$\mathsf{hdr.mlen} = |x| \quad \text{and} \quad \mathsf{hdr.mlen} \leq \mathsf{PQS\_MESSAGE\_MAX}.$$

4. **AEAD tag verification:** If hdr.flag corresponds to an encrypted message or authenticated control packet, then

$$\mathsf{AEAD.Dec}(k, n, \mathsf{encode}(\mathsf{hdr}), c, t) \neq \bot.$$

If all checks succeed, the packet is accepted and the plaintext is returned. If any check fails, the acceptance predicate yields $\bot$, the connection is terminated, and all keys and cipher states are erased. A session is considered established when both parties have validated the full handshake transcript and initialized their cipher states.

5 Simplex AKE Security Definitions

5.1 Server Only Authenticated Key Exchange Experiment

We define a server only authenticated key exchange experiment

$$\mathsf{Exp}^{\mathsf{PQS\text{-}SOAKE}}(\mathcal{A})$$

that captures the Simplex trust model of PQS. The experiment models one client instance C and one server instance S interacting with an adversary $\mathcal{A}$ that controls all network communication.

Setup. The challenger samples the server's long term ML-DSA key pair

$$(\mathsf{ssk}, \mathsf{pvk}) \leftarrow \mathsf{SIG.KeyGen}()$$

and gives pvk to the client instance. The challenger also publishes pvk and all public parameters to $\mathcal{A}$. The client stores the pinned record $(\mathsf{cfg}, \mathsf{kid}, \mathsf{pvk}, \mathsf{expiration})$.

Oracle interface. $\mathcal{A}$ interacts through the following oracles:

- StartClient(): Returns a client instance identifier and initializes a fresh client state.
- StartServer(): Returns a server instance identifier and initializes a fresh server state with $(\mathsf{ssk}, \mathsf{pvk})$.
- Send(U, pkt): Delivers a packet pkt to instance U and returns the response packet produced by the protocol state machine of U, or $\bot$ if U rejects.
- CorruptState(U): Returns the full local state of instance U.
- CorruptServer(): Returns the long term signing key ssk.

Client acceptance. A client instance C accepts a session if:

1. it has validated a Connect Response whose signature verifies under its pinned pvk,
2. it has validated the Exchange Response,
3. it has derived a session key vector $\mathsf{SK} = (k_1, n_1, k_2, n_2)$.

The server is not required to authenticate the client and may accept any well formed Connect Request and Exchange Request.

Server only authentication condition. A client acceptance event is considered *authenticated* if:

The Connect Response used values signed under ssk.

More precisely, the client must have validated a signature on

$$h = \mathsf{SHA3}(\mathsf{encode}(\mathsf{hdr}_{\mathsf{CRP}}) \parallel \mathsf{pk})$$

using its pinned verification key, and no adversary is allowed to forge such a signature unless it breaks ML-DSA.
The experiment returns 1 if the adversary causes a client acceptance event without an authenticated server signature. Otherwise it returns 0.
The adversary's advantage in breaking server authentication is

$$\mathsf{Adv}^{\mathsf{auth}}_{\mathsf{PQS}}(\mathcal{A}) = \Pr\left[\mathsf{Exp}^{\mathsf{PQS\text{-}SOAKE}}(\mathcal{A}) = 1\right].$$

5.2 Session Key Indistinguishability Game

We define the session key indistinguishability experiment

$$\mathsf{Exp}^{\mathsf{ind}}_{\mathsf{PQS}}(\mathcal{A})$$

for the Simplex setting.

Test session. A client session is *fresh* if:

- it has accepted with server authentication as above,
- neither its state nor the server's state has been corrupted,
- the long term signing key has not been corrupted before acceptance,
- no ciphertext or tag forged under its keys has been decrypted successfully.

Challenge. When $\mathcal{A}$ makes a *Test* query on a fresh client session with key vector SK, the challenger flips a random bit $b \leftarrow \{0, 1\}$.

$$K^* = \begin{cases} \mathsf{SK}, & b = 0, \\ U, & b = 1, \end{cases}$$

where U is a uniformly random key vector of correct length.
The challenger returns K^* to $\mathcal{A}$. $\mathcal{A}$ may continue interacting with oracles subject to freshness constraints.

Output. $\mathcal{A}$ outputs a bit b'. The experiment outputs 1 if $b' = b$.
The adversary's advantage is:

$$\mathsf{Adv}^{\mathsf{ind}}_{\mathsf{PQS}}(\mathcal{A}) = \left|\Pr[b' = b] - \tfrac{1}{2}\right|.$$

5.3 Forward Secrecy in the Simplex Model

Forward secrecy for PQS concerns only the confidentiality of past session keys against compromise of the server's long term key ssk after session completion. Define:

$$\mathsf{Exp}^{\mathsf{fs}}_{\mathsf{PQS}}(\mathcal{A}).$$

Experiment.

1. The adversary interacts normally with the protocol.
2. A client session completes with key vector SK and is deemed fresh.
3. The adversary issues $\mathsf{CorruptServer}()$, thus learning ssk *after* the session has completed.
4. The adversary issues a *Test* query on the completed client session, receiving either SK or a random vector depending on a hidden bit b.
5. The adversary outputs a guess b'.

The session has forward secrecy if:

$$\mathsf{Adv}^{\mathsf{fs}}_{\mathsf{PQS}}(\mathcal{A}) = \left|\Pr[b' = b] - \tfrac{1}{2}\right|$$

is negligible. The security follows from the ephemeral ML-KEM exchange, because ssk is used only for authentication and never contributes to sec.

5.4 Post Compromise and KCI Style Properties

We define two additional notions for the Simplex setting.

Session key compromise. If an adversary learns the key vector of one session through $\mathsf{CorruptState}$, PQS guarantees that no other session is compromised. KEM freshness and unique cSHAKE derivation ensure that later sessions derive independent sec values and keys.

Long term key compromise. If an adversary corrupts ssk before a session begins, authentication is lost but confidentiality remains dependent on ML-KEM. If ssk is corrupted after session completion, previously derived keys remain hidden due to the IND-CCA security of ML-KEM.

KCI style resilience. Because the server does not authenticate the client in the Simplex model, KCI resilience applies only in the direction that compromise of SK or ssk does not allow impersonation of the server in sessions where the adversary cannot forge a signature on the Connect Response. This corresponds exactly to the unforgeability of ML-DSA.

5.5 Channel Integrity and Replay Resistance

We define two additional experiments to model transport-layer security.

AEAD authenticity. The experiment

$$\mathsf{Exp}^{\mathsf{int}}_{\mathsf{PQS}}(\mathcal{A})$$

asks $\mathcal{A}$ to produce a new packet

$$\mathsf{hdr} \parallel c \parallel t$$

that passes:

- all PQS header validation rules, and
- $\mathsf{AEAD.Dec}(k, n, \mathsf{encode}(\mathsf{hdr}), c, t) \neq \perp$.

The advantage

$$\mathsf{Adv}^{\mathsf{int}}_{\mathsf{PQS}}(\mathcal{A})$$

is bounded by the INT-CTXT advantage of the underlying AEAD.

Replay resistance. Using the PQS acceptance predicate $\mathsf{Acc}(\mathsf{hdr}, x)$, define the replay experiment:

$$\mathsf{Exp}^{\mathsf{replay}}_{\mathsf{PQS}}(\mathcal{A})$$

where $\mathcal{A}$ outputs a previously received and authenticated packet

$$(\mathsf{hdr}, c, t)$$

and wins if:

$$\mathsf{Acc}(\mathsf{hdr}, c \parallel t) = 1$$

after it has already been accepted once.
Because timestamps and sequence numbers are bound into the AEAD associated data, replays are rejected with overwhelming probability.

6 Provable Security Analysis

6.1 Overview of Assumptions and Reductions

The security of PQS follows from the hardness assumptions underlying its components and the binding of the handshake transcript to the session key. The reduction strategy proceeds in four stages.

First, we relate security in the Simplex authenticated key-exchange model to the IND-CCA security of ML-KEM. If an adversary can distinguish the session key of a fresh client session from random, then the reduction constructs an adversary that distinguishes the ML-KEM shared secret from random in the standard IND-CCA experiment.

Second, we reduce server authentication to the EUF-CMA security of ML-DSA. In order to cause an unauthenticated client acceptance or to modify the Connect Response while maintaining a valid signature, an adversary must forge a signature on a value of the form

$$h = \mathsf{SHA3}(\mathsf{encode}(\mathsf{hdr}) \parallel \mathsf{pk}),$$

which contradicts unforgeability.

Third, we reduce the pseudo-randomness of the derived keys to the PRF style security of cSHAKE. Replacing the output of cSHAKE with a uniform string is indistinguishable to any adversary that cannot distinguish the XOF from a random oracle with respect to the session cookie and shared secret.

Fourth, we rely on the IND-CCA and INT-CTXT security of the AEAD scheme (RCS or AES-GCM), which ensures confidentiality and integrity of encrypted packets and prevents adversarial manipulation of header fields bound as associated data. Together these reductions yield the final bound for session key indistinguishability.

6.2 Hybrid Argument for Session Key Indistinguishability

We define a sequence of hybrid experiments transforming the real experiment into one in which the session key is independent of the adversary's view.

Hybrid H_0: real experiment. H_0 is the original Session Key Indistinguishability experiment $\mathsf{Exp}^{\mathsf{ind}}_{\mathsf{PQS}}$. The adversary receives either the real session key SK or a random vector U, depending on a challenge bit b.

$$\Pr[b' = b] = \tfrac{1}{2} + \mathsf{Adv}^{\mathsf{ind}}_{\mathsf{PQS}}(\mathcal{A}).$$

Hybrid H_1: replace ML-KEM shared secrets with random. In H_1, whenever the client encapsulates $(\mathsf{cpt}, \mathsf{sec}) \leftarrow \mathsf{KEM.Encaps}(\mathsf{pk})$ for a fresh session, the challenger replaces sec with a uniformly random value sec^* of the same length, unless the adversary queries a prohibited decapsulation oracle.

By the IND-CCA security of ML-KEM, any adversary distinguishing H_0 from H_1 breaks ML-KEM:

$$|\Pr[H_0] - \Pr[H_1]| \leq \mathsf{Adv}^{\text{IND-CCA}}_{\text{ML-KEM}}(\mathcal{A}).$$

Hybrid H_2: simulate signatures. In H_2, the challenger simulates the server's ML-DSA signatures by programming the signature oracle. For each Connect Response, rather than computing $\sigma = \mathsf{SIG.Sign}(\mathsf{ssk}, h)$, the challenger samples σ uniformly subject to the verification equation.
A distinguishing advantage between H_1 and H_2 yields a forger:

$$|\Pr[H_1] - \Pr[H_2]| \leq \mathsf{Adv}^{\text{EUF-CMA}}_{\text{ML-DSA}}(\mathcal{A}).$$

Hybrid H_3: replace cSHAKE output with random. In H_3, the challenger replaces

$$\mathsf{prnd} \leftarrow \mathsf{XOF}(\mathsf{sec}^*, \mathsf{sch})$$

with a string prnd^* sampled uniformly from the same domain.
Because sch is derived from public pinning data and sec^* is uniform in H_1, cSHAKE behaves as a PRF on combined input. Any adversary distinguishing H_2 from H_3 breaks PRF security:

$$|\Pr[H_2] - \Pr[H_3]| \leq \mathsf{Adv}^{\text{PRF}}_{\text{cSHAKE}}(\mathcal{A}).$$

Hybrid H_4: final game with independent keys. In H_4, because prnd^* is uniform, the derived key vector

$$(k_1, n_1, k_2, n_2)$$

is independent of all adversarial observations. Therefore:

$$\Pr[b' = b] = \tfrac{1}{2}.$$

Conclusion. By telescoping the hybrids,

$$\mathsf{Adv}^{\mathsf{ind}}_{\mathsf{PQS}}(\mathcal{A}) \leq \mathsf{Adv}^{\text{IND-CCA}}_{\text{ML-KEM}} + \mathsf{Adv}^{\text{EUF-CMA}}_{\text{ML-DSA}} + \mathsf{Adv}^{\text{PRF}}_{\text{cSHAKE}} + \varepsilon_{\text{AEAD}} + \varepsilon_{\text{misc}},$$

where $\varepsilon_{\text{AEAD}}$ and $\varepsilon_{\text{misc}}$ capture negligible AEAD forgery probability, birthday bounds on header fields, and timestamp window constraints.

6.3 Reduction to IND–CCA Security of ML–KEM

Assume an adversary $\mathcal{A}$ distinguishes the session key in H_0 from random. We construct an adversary $\mathcal{B}$ attacking ML-KEM in the IND-CCA experiment.

Embedding the challenge. $\mathcal{B}$ receives a challenge public key pk^* from the KEM experiment. For the targeted client session, $\mathcal{B}$ embeds pk^* in the Connect Response and uses the challenge ciphertext in place of cpt. $\mathcal{B}$ simulates all other sessions honestly.
If $\mathcal{A}$ distinguishes the session key, it recovers information about the challenge shared secret. Thus:

$$\mathsf{Adv}^{\mathsf{ind}}_{\mathsf{PQS}}(\mathcal{A}) \leq \mathsf{Adv}^{\text{IND-CCA}}_{\text{ML-KEM}}(\mathcal{B}) + \text{negl.}$$

6.4 Reduction to EUF–CMA Security of ML–DSA

Assume $\mathcal{A}$ causes a client acceptance event without server authentication, or causes a transcript to verify under pvk with a modified header or public key.
We build a forger $\mathcal{B}$ for ML-DSA:

- $\mathcal{B}$ forwards signature queries from $\mathcal{A}$ to the signing oracle.
- When $\mathcal{A}$ outputs a forged Connect Response, $\mathcal{B}$ outputs the signature σ^* as its forgery on message h^*.

Because PQS requires exact matching of

$$h = \mathsf{SHA3}(\mathsf{encode}(\mathsf{hdr}) \parallel \mathsf{pk}),$$

any transcript deviation corresponds to an ML-DSA forgery:

$$\mathsf{Adv}^{\mathsf{auth}}_{\mathsf{PQS}}(\mathcal{A}) \leq \mathsf{Adv}^{\text{EUF-CMA}}_{\text{ML-DSA}}(\mathcal{B}).$$

6.5 Reduction to PRF Style Security of cSHAKE

Let $\mathcal{A}$ distinguish the derived AEAD keys from random. Because the session cookie is fixed public data and sec is uniform by the ML-KEM reduction, the XOF input is pseudo-random. We define a distinguisher $\mathcal{B}$ for cSHAKE by:

- receiving either true cSHAKE output or a uniform string,
- using this output as the PQS key schedule material,
- simulating all protocol operations for $\mathcal{A}$.

A successful distinction implies:

$$\mathsf{Adv}^{\mathsf{ind}}_{\mathsf{PQS}}(\mathcal{A}) \leq \mathsf{Adv}^{\text{PRF}}_{\text{cSHAKE}}(\mathcal{B}) + \text{negl.}$$

6.6 Concrete Security Bounds

Collecting terms from all reductions, for any probabilistic polynomial-time adversary $\mathcal{A}$,

$$\boxed{\mathsf{Adv}^{\mathsf{ind}}_{\mathsf{PQS}}(\mathcal{A}) \leq \mathsf{Adv}^{\text{IND-CCA}}_{\text{ML-KEM}} + \mathsf{Adv}^{\text{EUF-CMA}}_{\text{ML-DSA}} + \mathsf{Adv}^{\text{PRF}}_{\text{cSHAKE}} + \varepsilon_{\text{AEAD}} + \varepsilon_{\text{seq}} + \varepsilon_{\text{time}}}$$

where:

- $\varepsilon_{\text{AEAD}}$ is the AEAD forgery probability,
- ε_{seq} is the birthday bound on sequence numbers,
- $\varepsilon_{\text{time}}$ is the probability of timestamp collision inside the validity window.

All remaining terms are negligible in the security parameter.

7 Security Analysis of the Data Channel

7.1 Confidentiality of Application Data

Once the handshake completes, all application data is encrypted using an AEAD scheme instantiated by RCS or AES-GCM with keys derived from the KEM shared secret and session cookie. The directional key assignment ensures that the ciphertext for each direction is produced under a unique (k, n) pair.
Let $(k_{\mathsf{send}}, n_{\mathsf{send}})$ denote the sender's AEAD key and nonce, and let $\mathsf{encode}(\mathsf{hdr})$ denote the associated data for the packet header. For any plaintext m, the ciphertext and tag are generated as:

$$(c, t) \leftarrow \mathsf{AEAD.Enc}(k_{\mathsf{send}}, n_{\mathsf{send}}, \mathsf{encode}(\mathsf{hdr}), m).$$

Under the PQS acceptance predicate, the receiver only accepts packets whose header fields satisfy the sequence, timestamp, and length checks, after which the ciphertext must pass AEAD tag verification.
By the IND-CCA security of the AEAD and the SK-IND property of the handshake, we obtain that any adversary with full network control cannot distinguish the plaintext of an accepted ciphertext from random, except with probability bounded by:

$$\mathsf{Adv}^{\mathrm{conf}}_{\mathsf{PQS}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathsf{ind}}_{\mathsf{PQS}}(\mathcal{A}) + \mathsf{Adv}^{\text{IND-CCA}}_{\mathrm{AEAD}}(\mathcal{A}) + \mathrm{negl}.$$

Thus the confidentiality of application data is guaranteed whenever the handshake was authenticated and the session keys remain fresh.

7.2 Integrity and Non Malleability

Integrity of the data channel follows from the INT-CTXT security of the AEAD and the binding of the entire PQS header as associated data. For each packet, the sender commits to:

$$(\mathsf{flag}, \mathsf{seq}, \mathsf{mlen}, \mathsf{time})$$

by including $\mathsf{encode}(\mathsf{hdr})$ as associated data in the AEAD computation.
Because RCS and AES-GCM authenticate the associated data, any adversarial attempt to modify a header field or alter the ciphertext without recomputing a valid tag under the same key requires a successful AEAD forgery. For example:

- altering the packet flag changes $\mathsf{encode}(\mathsf{hdr})$,
- replaying or reordering packets changes seq,
- truncating or extending the message changes mlen,
- shifting timestamps changes time.

All such modifications yield incorrect AEAD inputs and cause tag verification to fail.
By the INT-CTXT property:

$$\mathsf{Adv}^{\mathrm{int}}_{\mathsf{PQS}}(\mathcal{A}) \leq \mathsf{Adv}^{\text{INT-CTXT}}_{\mathrm{AEAD}}(\mathcal{A}) + \mathrm{negl}.$$

In particular, for RCS, the KMAC tag computed over the associated data, nonce, ciphertext, and encoded lengths (as proven in the RCS formal cryptanalysis) ensures strict integrity of both header and payload.

7.3 Replay and Reordering Resistance

PQS enforces two independent mechanisms for freshness:

1. **Sequence monotonicity.** The receive counter must equal the packet's sequence number or fall within an allowed window. Any reuse or backward movement in the sequence number results in rejection.
2. **Timestamp freshness.** The header timestamp must lie within a configured acceptance window relative to the local clock. Attempts to replay packets outside this window fail the timestamp validation.

Even if an adversary replays a previously accepted ciphertext with the same valid tag, at least one of these checks will fail. Since the header is authenticated by the AEAD, the adversary cannot modify sequence numbers or timestamps without producing an AEAD forgery, which is negligible.
Therefore:

$$\mathsf{Adv}^{\text{replay}}_{\mathsf{PQS}}(\mathcal{A}) \leq \mathsf{Adv}^{\text{INT-CTXT}}_{\text{AEAD}}(\mathcal{A}) + \varepsilon_{\text{seq}} + \varepsilon_{\text{time}},$$

where ε_{seq} and $\varepsilon_{\text{time}}$ reflect the negligible probabilities of accidental sequence or time collisions.

7.4 Directional Separation and Reflection Resistance

PQS assigns unique AEAD keys and nonces to each communication direction:

$$\text{Client: } (k_1, n_1)_{\mathsf{send}},\ (k_2, n_2)_{\mathsf{receive}}, \qquad \text{Server: } (k_2, n_2)_{\mathsf{send}},\ (k_1, n_1)_{\mathsf{receive}}.$$

Thus no ciphertext produced by one party is valid under the AEAD key of the opposite direction. This prevents all reflection attacks, including:

- reusing a client-produced ciphertext as if produced by the server,
- reusing a server-produced ciphertext as if produced by the client,
- symmetric-loopback attacks where an adversary mirrors ciphertexts back to the sender.

Because each direction uses non-overlapping key and nonce domains, any such attempt requires AEAD forgery and is therefore negligible.

7.5 Downgrade and Configuration Binding

The session cookie

$$\mathsf{sch} = \mathsf{H}(\mathsf{cfg} \parallel \mathsf{kid} \parallel \mathsf{pvk})$$

binds the handshake to a specific configuration string, key identifier, and pinned verification key. Both sides compute sch independently from pinned data, not from transmitted data, which prevents adversarial manipulation of configuration values.

In particular:

- altering cfg or kid during the Connect Request results in mismatched sch, causing the derived keys k_1, k_2, n_1, n_2 to differ across endpoints.

- substituting an alternate server key pvk' produces a different cookie and key schedule, unless the adversary forges a signature, which contradicts EUF-CMA security of ML-DSA.

Thus downgrade attempts or misbinding attacks fail either by causing handshake validation failure or by producing mismatched key schedules, which leads to AEAD tag failure at the earliest packet.
This establishes configuration binding and downgrade protection for the PQS data channel.

8 Attack Analysis and Limitations

8.1 Generic and Brute Force Attacks

PQS derives its security from the hardness of the underlying post-quantum primitives and from the structure of the authenticated encryption channel. Generic attacks that attempt to recover the session key by exhaustive search must contend with several barriers.

First, recovering the KEM shared secret sec requires breaking ML-KEM, which for the selected parameter sets provides at least 128 bits of post-quantum security against classical and quantum exhaustive search. Second, given sec, the adversary must invert the cSHAKE based key-derivation function to recover (k_1, n_1, k_2, n_2). The XOF expansion provides pseudo-random output with min-entropy equal to the security level of the underlying Keccak permutation. Third, recovering AEAD keys directly from observing ciphertexts and tags is no easier than breaking the IND-CCA security of the selected AEAD scheme (RCS or AES-GCM). The tag space in both AEAD modes is 128 bits or greater, so brute force tag forgery is infeasible.

Because the session cookie sch is derived from public data and serves only as a customization string, brute force attacks on sch do not yield useful information about the KEM secret or the derived symmetric keys.

8.2 Attacks on Underlying Primitives

The security of PQS reduces cleanly to the security of its cryptographic components.

ML-KEM. An attack on ML-KEM that exposes the shared secret sec allows the adversary to reconstruct all derived symmetric keys for a session. This breaks session confidentiality but does not compromise other sessions because each handshake uses a fresh ephemeral ML-KEM key pair.

ML-DSA. A successful forgery against ML-DSA allows an adversary to produce a valid Connect Response for any chosen public KEM key and header values. This defeats server authentication in the Simplex model, allowing the attacker to impersonate the server. The communication channel after such a handshake is still protected by ML-KEM, but the attacker may cause the client to derive keys in a session of their own construction.

SHA3 and cSHAKE. A break in collision resistance or second-preimage resistance of SHA3 could allow manipulation of the session cookie sch, enabling downgrade or configuration confusion attacks that would otherwise be detected. Breaking the PRF style properties of cSHAKE compromises the key-derivation function and allows the adversary to distinguish derived keys from random, undermining the SK-IND guarantee.

AEAD (RCS or AES-GCM). Breaking IND-CCA confidentiality of the AEAD reveals plaintext application data for a compromised direction. Breaking INT-CTXT allows the adversary to inject or modify ciphertext that passes tag verification, undermining data authenticity and the integrity of PQS headers bound as associated data.

8.3 Transcript and Downgrade Attacks

Manipulation of handshake messages must contend with two independent mechanisms: signature authentication and KDF binding. The Connect Response carries a signature on

$$h = \mathsf{SHA3}(\mathsf{encode}(\mathsf{hdr}) \parallel \mathsf{pk}),$$

so modifying the ephemeral public key, sequence number, timestamp, or any other header field requires forging an ML-DSA signature. This rules out all transcript modification attacks not predicated on a signature forgery.
Downgrade attacks are mitigated through the session cookie:

$$\mathsf{sch} = \mathsf{H}(\mathsf{cfg} \parallel \mathsf{kid} \parallel \mathsf{pvk}),$$

which ensures that any change in configuration parameters or verification key yields a different key-derivation path. Because both endpoints compute sch locally from their trusted records rather than from transmitted values, any mismatch in cfg, kid, or pvk produces a different set of symmetric keys and causes AEAD tag failure on the first encrypted message.
Thus transcript attacks are constrained by the signature scheme, and downgrade attacks fail due to KDF mismatch or tag invalidation.

8.4 Implementation Level and Side Channel Attacks

The formal model assumes idealized cryptographic oracles and does not account for side-channel leakage or implementation faults. In practice, several attack vectors must be considered:

Timing and microarchitectural leakage. Non-constant-time implementations of ML-KEM, ML-DSA, SHA3, cSHAKE, RCS, or AES-GCM may leak secret information through timing variations, cache access patterns, or other microarchitectural channels. PQS relies on constant-time implementations of these primitives.

Memory disclosure. Failure to erase ephemeral secrets, particularly the ML-KEM private key sk and the KEM shared secret sec, can compromise forward secrecy and allow adversaries to recover past or current symmetric keys.

State desynchronization. Incorrect handling of sequence counters or improper enforcement of timestamp windows may cause the receiver to accept stale or reordered packets that would otherwise be rejected by the formal acceptance predicate.

Error handling. Unexpected implementation behavior in error paths may reveal partial information through error codes or timing differences. PQS mitigates this by erasing sensitive material and closing the connection upon any fatal error.

These considerations lie outside the symbolic model but are relevant for robust deployment of PQS.

8.5 Limitations of the Simplex Model

The Simplex trust model used by PQS introduces inherent limitations that must be acknowledged.

Absence of client authentication at the key-exchange layer. PQS authenticates only the server during the handshake. The client is anonymous at the key-exchange layer, and any client authentication must be performed at an upper layer over the encrypted channel. This limits the applicability of the model to deployments where unilateral authentication is sufficient.

Post-compromise security without ratcheting. PQS does not include a built-in ratchet mechanism. If session keys for an active session are compromised, the remainder of that session is exposed. Future sessions remain secure due to fresh ML-KEM contributions, but PQS does not provide continuous healing of state as in double-ratchet systems.

KCI style considerations. Because the server does not authenticate the client, compromise of the client's session state or keys allows adversaries to impersonate the client to the server at the application layer. This is expected in the Simplex model and must be handled by upper-layer authentication.

Deployment requirements. PQS relies on a valid UTC time source for enforcing timestamp windows. Environments without reliable clocks must broaden the acceptance window or disable time checks, which weakens replay resistance. PQS also requires secure management of pinned verification keys on clients to prevent substitution or deletion attacks.
These limitations are structural properties of the Simplex model and should be understood when deploying PQS in larger systems.

9 Implementation Conformance and System Considerations

9.1 Mapping Between Model and Reference Implementation

The formal model reflects the behavior of the reference implementation in `pqs.c`, `pqs.h`, and the supporting `kex.c` and cipher modules. We briefly summarize the correspondence between the abstract protocol and concrete code paths.

The client and server state structures in the implementation contain fields for transmit and receive sequence counters, cipher send and receive contexts, session tokens, pinned key metadata, and the KEX state containing sch and sec. These directly match the abstract state components defined in the formal specification.

Packet processing follows the order assumed by the proofs:

1. The packet header is deserialized and validated for flag correctness, sequence monotonicity, timestamp freshness, and message length constraints.
2. Only after successful header validation is the AEAD associated data set to encode(hdr).
3. The AEAD decryption function is invoked and may accept only if the tag verifies.

This ordering is central to the integrity and replay arguments and is consistent with the implementation, in which `pqs_packet_validate` performs all validation prior to decryption and immediately aborts on failure.

Similarly, the handshake processing code in `kex.c` and `pqs.c` performs these steps in the exact order assumed in the security proof: parsing of kid and cfg, lookup of the verification key, computation of the session cookie, signing of the Connect Response header concatenated with the ephemeral KEM public key, KEM encapsulation or decapsulation, erasure of the ephemeral private key after use, and final key schedule computation.

9.2 Constant Time Behavior and Side Channels

The formal model assumes idealized black box access to the cryptographic primitives without side-channel leakage. For implementation level security, PQS requires that ML-KEM, ML-DSA, SHA3, cSHAKE, and the AEAD schemes (RCS or AES-GCM) run in constant time and do not exhibit data dependent branching or memory access.

In particular:

- decapsulation of ML-KEM must avoid branching on ciphertext validity,
- ML-DSA signing must avoid timing leakage associated with rejection sampling operations,
- SHA3 and cSHAKE must perform all permutation rounds in a constant number of operations,
- RCS and AES-GCM must ensure that MAC computation and block processing are constant time with respect to secret inputs.

The protocol logic itself avoids secret dependent branches. All validation errors are handled uniformly by setting an error code and tearing down the session, without selective error messages or timing differences that would reveal internal state.

9.3 Random Number Generation Requirements

The security of PQS depends on high quality randomness for:

- generating the ephemeral ML-KEM key pair $(\mathsf{pk}, \mathsf{sk})$,
- sampling the KEM encapsulation randomness during $\mathsf{KEM.Encaps}$,
- generating ML-DSA signatures,
- generating session tokens and any nonce material required for internal use.

Implementations should use a NIST style DRBG or an equivalent system random generator with forward and backward security guarantees. Entropy sources must be properly seeded before cryptographic operations begin. Weak randomness directly affects the security of ML-KEM and ML-DSA and, by extension, the security of the PQS handshake and data channel.

9.4 Key and State Erasure

Correct erasure of cryptographic material is necessary to preserve the security properties proven in the formal model.

- The ephemeral ML-KEM private key sk must be securely erased immediately after computing sec through $\mathsf{KEM.Decaps}$. This ensures forward secrecy for the client.
- The shared secret sec, session cookie sch, and derived keys (k_1, n_1, k_2, n_2) must be erased when a session completes or when an error condition triggers teardown.
- Cipher send and receive contexts must overwrite internal key schedule state during teardown.
- All KEX state structures must be cleared once key derivation completes.

The implementation conforms to these requirements by zeroizing temporary buffers and sensitive fields in the `pqs_state` and `pqs_kex_state` structures upon failure or explicit session termination.

9.5 Clock Synchronization and Deployment Constraints

PQS enforces replay protection using the timestamp field in each packet header. The acceptance predicate requires that:

$$|\mathsf{hdr.time} - \mathsf{localTime}| \leq \Delta,$$

where Δ is the configured acceptance window. This relies on the client and server clocks being synchronized to a coarse level.
Systems deploying PQS are expected to maintain an approximate UTC reference. If clocks drift beyond the configured tolerance, legitimate packets may be rejected. Conversely, widening Δ increases the window in which replayed packets remain valid, weakening the replay guarantees.
In constrained environments without reliable clocks, deployments may disable or relax timestamp checks, in which case the burden shifts to stricter sequence number enforcement and shorter session lifetimes. These adjustments must be made consciously, as they reduce the strength of replay and ordering protections established in the formal analysis.

10 Performance and Practicality

10.1 Asymptotic Cost of the Handshake

The PQS handshake requires one ML-DSA signature by the server and a single ML-KEM encapsulation and decapsulation pair. Its asymptotic cost is therefore dominated by:

- one ML-DSA signing operation;
- one ML-DSA verification performed by the client;
- one ML-KEM key generation by the server;
- one ML-KEM encapsulation by the client;
- one ML-KEM decapsulation by the server.

The remaining operations consist of SHA3-256 and cSHAKE calls to compute the session cookie and expand the shared secret. These hash-based steps have linear cost in the size of the Keccak rate and are negligible compared to the cost of the KEM and signature operations.

The handshake therefore scales asymptotically as

$$T_{\mathsf{handshake}} = T_{\mathsf{sign}} + T_{\mathsf{verify}} + T_{\mathsf{keygen}} + T_{\mathsf{encaps}} + T_{\mathsf{decaps}} + O(r),$$

where r is the rate of the Keccak permutation. For parameter sets targeting 128-bit security, these operations complete in a small number of milliseconds on modern CPUs.

Because PQS is a Simplex authenticated key-exchange protocol, the server performs no verification of client data at the handshake layer, reducing cost relative to fully authenticated two-sided protocols.

10.2 Data Channel Throughput and Overheads

PQS uses a fixed 21 byte header for all packets, regardless of message type. Encrypted packets append AEAD ciphertext and a 16 byte tag (for RCS or AES-GCM). The per-packet overhead is therefore:

$$\mathsf{overhead} = 21 \text{ bytes (header)} + 16 \text{ bytes (tag)} = 37 \text{ bytes},$$

plus any padding or alignment applied by the chosen AEAD mode. This constant overhead is independent of the plaintext length.

Throughput on the data channel is determined by:

- the per-block performance of RCS or AES-GCM,
- the availability of hardware acceleration for AES or SHA3 style permutations,
- the cost of serializing the header and performing minimal packet validation.

RCS uses Rijndael-256 in counter mode and KMAC for authentication. When implemented with optimized 64-bit instructions, its throughput is comparable to AES-GCM on systems without AES-NI acceleration and slower on systems with hardware AES support. AES-GCM benefits significantly from AES-NI and CLMUL instructions, often reaching multiple gigabytes per second.

Latency is dominated by the round trip time of the four handshake messages. Once established, the data channel adds only negligible computation time per packet beyond the cost of AEAD encryption or decryption.

10.3 Comparison with Classical and Hybrid Channels

Compared to classical SSH style channels, PQS replaces the Diffie Hellman exchange and classical signature choices with post-quantum primitives. The structure of the handshake is simpler because the trust model is asymmetric: the client authenticates the server, but the server does not authenticate the client at the key-exchange layer. As a result, PQS requires fewer cryptographic operations per handshake than typical mutually authenticated protocols.

Relative to hybrid post-quantum modes that combine classical key exchange with PQ KEMs, PQS avoids duplicating work or performing parallel operations. Hybrid designs often require two KEM encapsulations (one classical, one PQ) or a classical signature plus PQ signature to achieve robust dual security. PQS performs a single ML-KEM exchange and a single ML-DSA signature, giving it lower cryptographic load at the cost of relying entirely on PQ security assumptions.

Because PQS uses a single cSHAKE based key schedule for both directions and a fixed format header with AEAD protection, its data channel has performance similar to high performance authenticated encryption channels in classical protocols. The overhead of the additional header fields (sequence, timestamp) is minimal relative to the payload and provides strong replay protection.

Overall, PQS is practical for deployment in settings that require unilateral server authentication and post-quantum confidentiality and integrity, without the complexity or large performance overheads associated with hybrid or fully authenticated protocols.

11 Conclusion

11.1 Summary of Results

This paper presented a formal security analysis of the PQS protocol within a Simplex server only authenticated key-exchange model. We defined a dedicated SOAKE framework tailored to the unilateral trust structure of PQS, where only the server is authenticated during the handshake and the client is authenticated, if needed, at a higher application layer.

We provided formal definitions for server authentication, session key indistinguishability, forward secrecy in the Simplex setting, basic post compromise guarantees, and the integrity and replay properties of the data channel. The main theorems establish that PQS achieves these goals under standard post-quantum assumptions. In particular:

- server authentication reduces to the EUF-CMA security of ML-DSA,
- session key indistinguishability reduces to IND-CCA security of ML-KEM, PRF style security of cSHAKE, and AEAD confidentiality,
- channel integrity and non malleability reduce to AEAD INT-CTXT security,
- replay and ordering guarantees rely on sequence number and timestamp validation, both bound to the AEAD associated data.

The resulting concrete bound shows that the adversary's advantage is dominated by the hardness of ML-KEM, ML-DSA, and the cryptographic strength of cSHAKE and the AEAD scheme.

11.2 Security Guarantees and Caveats

PQS provides unilateral authentication of the server, confidentiality and integrity of application data, protection against transcript modification, configuration binding, and robust replay resistance. Each session uses a fresh ML-KEM exchange that ensures forward secrecy for the client and independence of session keys across executions.

Several caveats follow naturally from the Simplex model. PQS does not authenticate the client at the key-exchange layer, so client authentication must occur at the application layer. PQS does not include a ratcheting mechanism, which limits its post compromise guarantees to the independence of future sessions. Implementation level concerns such as constant time behavior, side-channel resistance, secure erasure, and high quality randomness remain essential for preserving the guarantees proven in the formal model.

The replay protection mechanism depends on correct handling of timestamps and sequence numbers, which in turn assumes approximate clock synchronization between endpoints.

Configuration pinning on the client is also critical to prevent downgrade or substitution attacks.

11.3 Directions for Future Work

Future work may extend PQS with asymmetric or symmetric ratcheting, allowing continuous key evolution and stronger post compromise security. Another direction is the application of formal verification techniques, such as symbolic verification with automatic proof tools or formal semantics for message parsing and state transitions, to validate the reference implementation against the formal model.

Comparative evaluation of PQS against other post-quantum and hybrid secure channel designs would clarify the relative performance and security margins of Simplex style protocols. Finally, exploring multi party or multi server extensions of Simplex authenticated channels may broaden the applicability of PQS in distributed post-quantum systems.

12 References

References

1. Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., Stehlé, D. *CRYSTALS-Kyber: A CCA-Secure Module-Lattice-Based KEM.* NIST Post-Quantum Cryptography Project.
2. Lepoint, T., Lyubashevsky, V., Seiler, G., et al. *CRYSTALS-Dilithium: Digital Signatures from Module Lattices.* NIST Post-Quantum Cryptography Project.
3. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G. *The Keccak Reference.* NIST SHA-3 Standardization, 2011.
4. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G. *Duplexing the Sponge: Single-Pass Authenticated Encryption and Other Applications.* Selected Areas in Cryptography (SAC), 2011.
5. Bellare, M., Namprempre, C. *Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm.* Advances in Cryptology – ASIACRYPT 2000.
6. Underhill, J.G. *PQS: Post Quantum Shell, Protocol Specification.* Quantum Resistant Cryptographic Solutions Corporation, 2025. Available at: https://www.qrcscorp.ca/documents/pqs_specification.pdf
7. Underhill, J.G. *PQS: Post Quantum Shell, Implementation Analysis.* Quantum Resistant Cryptographic Solutions Corporation, 2025. Available at: https://www.qrcscorp.ca/documents/pqs_implementation.pdf
8. QRCS Corporation. *PQS Reference Implementation (Source Code Repository).* Available at: https://github.com/QRCS-CORP/PQS