The Design and Formal Analysis of the Quantum Secure Tunneling Protocol

John G. Underhill

Quantum Resistant Cryptographic Solutions Corporation

Abstract. QSTP is a post quantum tunneling protocol that composes a lattice based key encapsulation mechanism, a post quantum signature scheme, and the RCS authenticated encryption scheme into a simple linear handshake and encrypted channel. This paper gives a formal description of QSTP, specifies the security model, and proves that the protocol achieves authenticated key exchange and channel confidentiality under standard cryptographic assumptions. The analysis is carried out in a game based framework that treats QSTP as a composition of well defined primitive interfaces, and derives explicit bounds in terms of the underlying KEM, signature, key derivation, and AEAD advantages. The work also evaluates replay protection, denial of service considerations, and design trade offs relative to existing tunneling frameworks such as TLS 1.3 and WireGuard.

1 Introduction

1.1 Context and Motivation

Secure tunneling protocols are used to establish protected communication channels between authenticated endpoints in environments where adversaries may intercept, modify, delay, or reorder network traffic. Modern deployments rely heavily on TLS 1.3 and on lightweight virtual private network mechanisms such as WireGuard. These systems provide strong guarantees but also carry significant structural complexity, negotiation logic, and multi branch state machines that complicate formal analysis, particularly in post quantum settings.

QSTP is a post quantum tunneling protocol designed for deployments that require a fixed cryptographic posture, a linear state machine, and deterministic handshake behavior. The protocol combines a lattice based key encapsulation mechanism, a post quantum signature scheme, and the RCS authenticated encryption function into a simple two party exchange that avoids negotiation, avoids optional branches, and permits a clear reduction based security analysis. The goal is not to replace TLS or WireGuard in general purpose settings but to provide a protocol suitable for controlled environments where predictable behavior, minimal attack surface, and alignment with post quantum assumptions are required.

The design of QSTP eliminates cipher suite negotiation and reduces the handshake to a small number of strictly ordered messages. This structure removes downgrade vectors and simplifies the partnering and transcript consistency arguments needed in a formal analysis. The resulting protocol is straightforward to implement and more amenable to end to end reasoning than multi mode frameworks that allow dynamic capability selection.

1.2 Contributions

This work provides a formal treatment of QSTP and establishes its security under standard cryptographic assumptions. The main contributions are as follows.

- A complete and precise description of the QSTP protocol, aligned with the public specification and reference implementation.
- A formal authenticated key exchange and channel security model tailored to the structure of QSTP, covering entity authentication, key indistinguishability, replay restrictions, and correct channel binding.
- A sequence of game based proofs that reduce the security of QSTP to the hardness assumptions of its components, including the IND-CCA security of the key encapsulation mechanism, the EUF-CMA security of the signature scheme, the pseudo-randomness of the key derivation process in the random oracle model, and the AEAD security of the RCS channel.
- A cryptanalytic evaluation of the protocol design, including analysis of the replay window, the decision to provide server only authentication, and the potential denial of service surface created by post quantum key exchange.
- A comparative study that situates QSTP with respect to TLS 1.3 and WireGuard, emphasizing the differences in state machine structure, negotiation behavior, and deployment assumptions.

1.3 Organization of the Paper

Section 2 gives a high level overview of the protocol and its cryptographic components. Section 3 introduces the notation and preliminary concepts used throughout the analysis.

Section 4 formalizes the QSTP execution model, including session identifiers, partnering, and transcript structure. Section 5 states the security goals for both the handshake and the encrypted channel. Section 6 gives the adversarial experiments and the security definitions. Section 7 states the primitive assumptions. Section 8 contains the game based security proofs for authenticated key exchange and channel confidentiality. Section 9 provides a cryptanalytic discussion of design choices and operational considerations. Section 10 compares QSTP with TLS 1.3 and WireGuard. Section 11 discusses implementation considerations and Section 12 concludes the paper.

2 Protocol Overview

2.1 Roles and Trust Assumptions

QSTP involves two communicating parties, a client and a server, that establish a secure channel over an adversarial network. The server holds a long term post quantum signature key pair and presents its public verification key to clients in the form of a certificate issued by a trusted root authority. The client is assumed to possess the root public key needed to validate the server certificate. Client authentication is not part of the protocol and no client specific long term key material is required.
Both parties rely on a key encapsulation mechanism for ephemeral key establishment. The server provides its long term signature key for authentication, and both parties use the derived ephemeral keying material to establish symmetric channel keys. No additional trust anchors or negotiation mechanisms are introduced. The trust assumptions are therefore limited to the authenticity of the server certificate, the correctness of the root key distribution, and the security of the underlying post quantum primitives.

2.2 Cryptographic Components

QSTP composes four cryptographic components, each selected to provide post quantum security properties.

- **Key Encapsulation Mechanism.** A lattice based IND-CCA secure KEM is used for establishing an ephemeral shared secret (Kyber or McEliece). The client encapsulates to the server's KEM public key, and the server decapsulates to recover the shared secret. This material forms the core input to the key derivation step.
- **Digital Signature Scheme.** A post quantum signature scheme, ML-DSA (Dilithium), provides server authentication. The server signs its handshake messages and the client verifies them using the certificate chain rooted in a trusted authority. The protocol provides unilateral authentication only.
- **RCS Authenticated Encryption.** RCS provides a symmetric authenticated encryption channel. It is used for all application data after the handshake completes. The header fields of QSTP packets are bound through the associated data mechanisms of RCS, ensuring integrity of sequence numbers and timestamps.
- **Key Derivation Function.** A hash based KDF maps the KEM shared secret and the final transcript hash to the session key material. In the reference implementation the derived pseudorandom stream is computed as $\mathsf{prnd} \leftarrow \mathsf{cSHAKE}(\mathsf{sec}, \mathsf{sch}_2)$, where sch_2 is used as the customization string. The output is partitioned into the send and receive keys and the initial nonces for the RCS channels in each transmission direction.

2.3 Message Flow

The QSTP handshake consists of a fixed sequence of four packets exchanged in a single linear path. No negotiation or optional branches occur and each packet has a single well defined purpose. After the ExchangeResponse is received, the client performs a local transcript verification step to establish the session; this verification is not an additional network message. The packets are:

1. **ConnectRequest.** The client sends an initial message indicating protocol parameters and supported configuration.
2. **ConnectResponse.** The server replies with its certificate, signature key, configuration values, and session cookie. This message is signed under the server's long term signature key and authenticated by the client.
3. **ExchangeRequest.** The client encapsulates to the server's KEM public key and sends the KEM ciphertext along with values needed to bind the transcript. This message initiates shared secret generation.
4. **ExchangeResponse.** The server decapsulates the ciphertext to recover the ephemeral shared secret. The server then derives its side of the channel keys and acknowledges completion of the exchange.
5. **EstablishVerify.** The client verifies that its derived keying material and transcript state match the server's response. If validation succeeds, both parties transition into the encrypted channel state.

After the final message, both parties hold identical symmetric channel keys and nonces and may begin exchanging encrypted packets under RCS.

2.4 Session Keys and Channel Structure

The KEM shared secret and the final transcript hash are processed by the KDF to produce four independent values:

$$(k_{c\to s},\, n_{c\to s},\, k_{s\to c},\, n_{s\to c}),$$

which form the send and receive keys and the initial nonces for the client-to-server and server-to-client transmission directions.

These keys form the base keying material for the session. QSTP defines two optional re-keying mechanisms that may be invoked over an established channel to update session keys without a full handshake.

Symmetric ratchet. The symmetric ratchet periodically refreshes session keys using a randomly generated token transmitted over the encrypted channel. Each ratchet step advances a persistent ratchet key state and derives a new set of session keys, providing forward secrecy at the symmetric layer: compromise of the current session state does not expose keying material from prior ratchet epochs.

The symmetric ratchet is optional and is signaled by the dedicated packet flag `qstp_flag_symmetric_ratchet_request`. Its formal security properties are analyzed in Section 11.

Application data is transmitted in packets that include a small fixed header consisting of a message flag, a sequence number, and a timestamp. The sequence number enforces strict monotonicity and prevents reordering beyond protocol bounds. The timestamp provides a coarse freshness check aligned with a configured acceptance window. The header is treated as associated data to RCS, ensuring that any modification to sequence or timestamp causes the packet to be rejected without revealing plaintext.

Each party maintains an independent sequence counter that advances for every transmitted packet. The RCS AEAD construction ensures that ciphertext integrity and confidentiality hold for all messages under the derived keys. Upon failure of header validation or RCS tag verification, the packet is discarded and the session may be terminated depending on the implementation policy.

3 Engineering Description of QSTP

This chapter gives a precise engineering description of QSTP based directly on the reference implementation. The goal is to complement the formal model by describing the exact operational behavior of the protocol as realized in the code modules `qstp.c`, `qstp.h`, `kex.c`, `client.c`, `server.c`, and `root.c` of the QSTP library. The presentation is implementation agnostic: it abstracts from C level details such as memory management and explicit error codes, while preserving the logical structure, message sequencing, and state transitions exactly as implemented. All cryptographic operations, packet constructions, and validation rules are derived from the concrete semantics of the source code.

3.1 Overview of the Engineering Model

The QSTP implementation is organized into a small set of modules that together realize the handshake, key establishment, channel creation, and packet protection mechanisms. Each module performs a clearly defined role, and their composition yields the end to end behavior of the protocol. The engineering model presented in this chapter mirrors this organization and describes the functional boundaries without referring to implementation specific artifacts such as pointers, structs, or return codes.

Root Module. The root module provides the trusted root certificate and the functions needed to validate server certificates. It exports the root public key, the certificate fingerprinting function, and the logic used to confirm that a presented server certificate matches the expected root lineage. This module defines the static trust anchor used by the client.

Client Module. The client module drives the handshake from the initiator side. It constructs the initial connection request, validates the server certificate and configuration, performs encapsulation to the server KEM public key, derives the client side channel keys, and transitions into the encrypted state. The client module relies on the QSTP core for message formatting and parsing, and on the kex engine for encapsulation operations.

Server Module. The server module responds to client requests, constructs signed configuration messages, decapsulates the client's KEM ciphertext, derives the server side channel keys, and finalizes the handshake. It maintains the long term server signature key and an ephemeral KEM decapsulation key. The KEM key pair is generated inside the kex engine when forming the server's connect response, stored in the kex server state for that handshake, and erased when the session state is destroyed. The server exposes functions for processing inbound messages and producing the corresponding outbound messages in each handshake stage.

QSTP Core. The QSTP core defines the packet formats, handshake message encodings, state machine transitions, and channel level sequence number and timestamp rules. It contains the logic for deriving the session cookie, checking transcript consistency, constructing the channel keys through the KDF, and initializing the encryption and decryption contexts. It also defines the authenticated data for RCS and the rules for accepting or rejecting packets based on header fields.

KEX Engine. The kex engine implements the post quantum key encapsulation mechanism. It provides the encapsulation and decapsulation routines used during the handshake and exposes a simple interface for generating the shared secret from a client ciphertext. The kex module is deterministic with respect to the code path: server decapsulation always produces a shared secret or signals failure, and client encapsulation always succeeds given a valid server public key.

Intermodule Interaction. The client and server modules call into the QSTP core to serialize and parse messages, validate signatures, and maintain transcript state. The QSTP core calls the kex engine for encapsulation and decapsulation, and calls the RCS AEAD interface for channel encryption and decryption. The root module is consulted once during certificate validation. The overall engineering model is therefore a linear and hierarchical structure with clear responsibilities and minimal coupling.

The remainder of this chapter describes these components in detail and provides an exact, implementation aligned description of the handshake mechanics, message flow, key derivation, packet formatting, and authenticated encryption behavior.

3.2 Data Structures and Constants

The QSTP implementation relies on fixed size buffers and statically defined constants that determine packet bounds, key lengths, certificate sizes, and internal state layout. These constants define the operational limits of the protocol and ensure that all packet structures and cryptographic values have well defined lengths throughout the handshake and channel phases.

3.2.1 Global Constants and Limits

The implementation defines a small set of global constants that constrain all packet formats and channel structures.

- **Session Cookie Length.** The session cookie is a fixed length hash value derived from the server certificate chain. Its size is determined by the hash function used in the implementation and is treated as an opaque byte string of constant length.
- **KEM Ciphertext and Shared Secret Sizes.** The ciphertext produced by client encapsulation and consumed by server decapsulation has a fixed length determined by the underlying lattice based KEM. The shared secret recovered from decapsulation also has a fixed length and is used directly as input to the key derivation function.
- **Key Derivation Outputs.** The KDF produces four fixed length values: a client to server key, a server to client key, and two initial nonces for the corresponding transmission directions. These values are of length suitable for the RCS AEAD primitive.
- **Packet Header Fields.** Each encrypted packet carries a header consisting of a one byte flag field, a four byte message length field, an eight byte sequence number, and an eight byte timestamp. These header components have fixed widths and are authenticated through RCS associated data.
- **Maximum Packet Sizes.** The implementation sets explicit upper bounds on plaintext and ciphertext sizes to ensure that the AEAD layer processes packets within a controlled range. These bounds include the size of the header, ciphertext, and AEAD tag.

These constants collectively define the dimensional constraints under which all protocol messages are constructed and validated. No dynamic resizing or negotiation of sizes occurs at runtime.

3.2.2 Certificate and Configuration Structures

The certificate and configuration structures define the server identity and handshake parameters.

- **Root Certificate.** The root module stores the long term trusted certificate used to validate server certificates. Its public key is used to verify the signature on the server certificate. The root certificate is static and known to the client in advance.
- **Server Certificate.** The server certificate contains the server's public signature key, configuration metadata, and certificate validity information. It is signed by the root authority. The certificate is provisioned in advance on the client and stored in the local configuration together with the root certificate. The client validates the server certificate under the root authority when configuration is loaded, and uses only derived values (such as the session cookie and certificate serial) during the handshake. The ConnectResponse does not transmit the full certificate again.
- **Server Configuration.** The server configuration includes protocol and implementation parameters and a session cookie value computed over the certificate state. The KEM public key used in the handshake is generated by the kex engine when forming the server's connect response and is not stored as a long term configuration value. These values define the cryptographic capabilities of the server for the duration of the handshake.
- **Constraints and Consistency Rules.** The implementation enforces that the server certificate matches the root authority, that the server configuration is consistent with the certificate, and that the session cookie is reproducible by both parties. Any inconsistency causes immediate handshake termination.

These structures collectively define the authenticated state of the server and are used by the client to validate the handshake transcript.

3.2.3 Session State Structures

The implementation maintains explicit state objects for each active handshake or channel session. These structures capture all information required to process inbound and outbound messages.

Client State. The client state includes:

- the validated server certificate and configuration,
- the session cookie derived from the server certificate chain,
- the KEM ciphertext and shared secret produced during encapsulation,
- the derived client to server and server to client channel keys,
- the initial nonces for both transmission directions,
- the client's outbound sequence counter and inbound sequence tracking state,
- the timestamp used for replay and freshness checks,
- the current handshake stage.

The client state is created at handshake initiation and transitions deterministically through the handshake phases.

Server State. The server state includes:

- the long term server signature key and KEM decapsulation key,
- the server certificate and configuration values sent to the client,
- the session cookie used for transcript binding,
- the shared secret recovered during decapsulation,
- the derived server to client and client to server channel keys,
- the initial nonces for both transmission directions,
- the server's outbound sequence counter and inbound sequence tracking state,
- the local timestamp used for freshness checks,
- the handshake stage associated with the inbound message.

The server allocates a new session state for each client handshake and transitions it based on inbound messages.

KEX State. The KEX engine maintains:

- the server's long term KEM public and secret keys,
- temporary buffers used during encapsulation and decapsulation,
- deterministic output regions for ciphertext and shared secret material.

The KEX state is stateless between handshakes aside from the long term KEM keys.

Channel State. Once the handshake completes, the channel state includes:

- the RCS encryption context for outbound packets,
- the RCS decryption context for inbound packets,
- the client to server and server to client nonces,
- directional sequence counters,
- timestamp freshness parameters,
- replay and reordering tracking state.

Channel state persists until session termination and is destroyed when the connection ends.
These session state structures correspond exactly to the fields manipulated by the implementation and form the operational substrate upon which the protocol executes.

3.3 Handshake and State Machine

The QSTP handshake is implemented as a deterministic sequence of message exchanges between a client and a server. The state machine is linear and contains no negotiation branches or optional transitions. Each state transition is triggered by the reception of a specific inbound message type, and each transition results in the construction of the corresponding outbound message. The client and server each maintain a local handshake state that ensures correct sequencing and transcript consistency.

3.3.1 State Machine Overview

The QSTP implementation defines a finite sequence of handshake stages that both parties traverse in lockstep. The high level state flow is as follows.

- **Client State 0: Idle.** The client begins with no session state. When a connection request is initiated, the client transitions to State 1 and constructs a ConnectRequest message.
- **Server State 0: Idle.** The server listens for inbound ConnectRequest messages. For each request, it allocates a new session state and transitions to State 1.
- **State 1: ConnectRequest Sent / Received.** The client sends a ConnectRequest. The server receives it, validates the request parameters, and constructs a ConnectResponse containing its certificate, configuration, and session cookie.
- **State 2: ConnectResponse Sent / Received.** After sending the ConnectResponse, the server enters State 2 awaiting the client's key exchange initiation. The client receives the ConnectResponse, validates the certificate chain, parses configuration, recomputes the session cookie, and transitions to State 3.
- **State 3: ExchangeRequest Sent / Received.** The client performs KEM encapsulation to the server's ephemeral public KEM key, advances the transcript hash to sch_2, derives session keys, and sends the ExchangeRequest containing the KEM ciphertext. The server receives this message in State 2, advances its own transcript to sch_2, performs KEM decapsulation, derives its session keys, and transitions to State 4.
- **State 4: ExchangeResponse Sent / Received.** The server constructs the ExchangeResponse containing the final transcript hash sch_2 as explicit key confirmation and transmits it to the client. The client receives this message and transitions to State 5.
- **State 5: EstablishVerify.** The client extracts the server's sch_2 from the ExchangeResponse body and performs a constant-time comparison against its own locally computed sch_2. If the values are equal, the session is declared established. If they differ, the client immediately tears down the connection and reports an authentication failure. No further message is sent; the EstablishVerify step is a local client verification operation.
- **State 6: Channel Established.** Both parties have identical session keys, nonces, and replay state. All subsequent communication proceeds through RCS authenticated encryption.

At any point, if a message type is unexpected, if certificate or configuration validation fails, if the session cookie does not match, or if encapsulation or decapsulation fails, the session is immediately aborted and the outgoing message is not generated.

3.3.2 Handshake Message Types

The implementation defines all handshake messages explicitly, each with a fixed structure and purpose. Below is a complete enumeration that reflects actual behavior in the code.

ConnectRequest. The client constructs the ConnectRequest message to initiate the handshake. It contains:

- protocol version and feature flags,
- client specific session identifier information,

- a request for server configuration parameters.

This message contains no cryptographic material and is accepted by any server session in the Idle state.

ConnectResponse. The server responds with a ConnectResponse containing:

- the server's ephemeral public encapsulation key $\mathsf{pk}_{\mathsf{kem}}$,
- the signed hash $\mathsf{phash} = \mathsf{Hash}(\mathsf{phdr} \parallel \mathsf{sch}_0 \parallel \mathsf{pk}_{\mathsf{kem}})$, which binds the response header, the initial transcript state, and the ephemeral key under the server's long-term signature key.

The full server certificate is provisioned on the client in advance and is not retransmitted during the handshake.

The client must validate the server certificate, confirm the signature, and recompute the session cookie. If any check fails, the session is aborted.

ExchangeRequest. The client constructs this message after validating the server configuration and the signed ephemeral encapsulation key. It contains:

- a KEM ciphertext produced by encapsulating to the server's ephemeral public key.

Before sending, the client advances the local transcript hash to sch_2 by hashing the KEM ciphertext into sch_1, then derives the session keys from $(\mathsf{sec}, \mathsf{sch}_2)$. This message transitions the server into the key derivation phase.

ExchangeResponse. The server decapsulates the ciphertext, advances its transcript to sch_2, and derives its session keys. The ExchangeResponse message contains:

- the final transcript hash sch_2 as explicit key confirmation.

After sending this message, the server transitions into the channel established state. The ephemeral KEM secret key is securely erased.

EstablishVerify. Upon receiving the ExchangeResponse, the client extracts the server's sch_2 value and performs a constant-time comparison against its own locally computed sch_2. If the values are equal, the session is confirmed established. If they differ, the client immediately tears down the connection and reports an authentication failure to the application layer. No outbound message is generated in this step; EstablishVerify is a local confirmation operation.

Established Channel. Once Verify is received, both parties have identical derived keys:

$$(k_{c\to s},\, k_{s\to c},\, n_{c\to s},\, n_{s\to c}),$$

and both parties initialize their RCS encryption and decryption contexts. Sequence numbers and timestamps begin at their initial values, and the session enters the protected data exchange phase.
All handshake messages have fixed formats and lengths determined by the implementation, and no part of the handshake supports negotiation or optional fields. This ensures that both code paths and state transitions remain deterministic and predictable for formal reasoning.

3.4 Session Cookie and Transcript Binding

The session cookie is a fixed length cryptographic value that binds the server certificate chain and configuration parameters into all subsequent handshake messages. Its purpose is to ensure that both the client and server derive identical transcript state without requiring the client to store or revalidate every field individually during later stages of the handshake.
The session transcript is maintained as a running hash that is extended at each handshake step. This rolling transcript ensures that every derived key is bound to the complete sequence of authenticated messages exchanged during the handshake.
The transcript chain begins from an implementation defined transcript seed that is initialized on both endpoints prior to the exchange of handshake packets. In the reference implementation this value is stored as sch_0 in the local key exchange state and is not derived as part of the wire protocol. The server enforces configuration consistency by verifying the received cfg and serial values against its local configuration and certificate state before generating a ConnectResponse. Any construction of sch_0 from certificate and configuration inputs, if used, is performed during configuration loading and is outside the scope of the handshake message processing.
After the server generates its ephemeral KEM key pair and constructs the ConnectResponse, it computes a per-message hash

$$\mathsf{phash} = \mathsf{Hash}(\mathsf{phdr} \parallel \mathsf{sch}_0 \parallel \mathsf{pk}_{\mathsf{kem}}),$$

where phdr is the serialized ConnectResponse packet header and $\mathsf{pk}_{\mathsf{kem}}$ is the ephemeral public encapsulation key. The server signs phash and advances the transcript:

$$\mathsf{sch}_1 = \mathsf{Hash}(\mathsf{sch}_0 \parallel \mathsf{phash}).$$

Upon receiving the ConnectResponse, the client verifies the signature, independently recomputes phash, and advances its local transcript to sch_1 identically.
After encapsulation, the client commits the KEM ciphertext cpta to the transcript to produce the final transcript hash:

$$\mathsf{sch}_2 = \mathsf{Hash}(\mathsf{sch}_1 \parallel \mathsf{cpta}).$$

The server performs the same computation upon receiving the ExchangeRequest. Both parties then derive session keys and nonces using

$$\mathsf{prnd} \leftarrow \mathsf{cSHAKE}(\mathsf{sec}, \mathsf{sch}_2),$$

where sec is the KEM shared secret and sch_2 is the final transcript hash used as the customization string. The transcript chain ensures that any adversarial substitution or modification of any handshake message causes sch_2 to diverge, producing mismatched session keys and an immediate handshake failure.

3.5 Server Certificate Validation

The client must validate the server certificate before processing any key exchange messages. Certificate validation is performed using the trusted root key embedded in the root module. This validation step is mandatory and executed in the client immediately after receiving the ConnectResponse message.
The validation process consists of three checks:

- **Root Signature Verification.** The client verifies that the server certificate is signed using the root public key. Only certificates whose signature matches the root's verification key are accepted. If verification fails, the handshake aborts.

- **Configuration Consistency.** The ConnectResponse does not transmit a configuration block or a certificate. In the reference implementation, configuration consistency is enforced by validating the `cfg` and `serial` fields received in the ConnectRequest against locally configured protocol parameters and the server certificate state. The ConnectResponse contains the server signature over `phash` and the ephemeral KEM public key used for the exchange. Any mismatch in configuration or certificate identifiers causes the handshake to terminate.
- **Certificate Lifetime and Integrity.** The implementation checks that the certificate is structurally valid, has not expired, and contains all fields required for correct operation of QSTP. Fields missing from the certificate or values outside the permitted bounds result in immediate rejection. The certificate is treated as immutable once validated and is not modified by runtime logic.

After these checks succeed, the certificate is considered authentic and the server identity is established. The session cookie is then computed from the validated certificate chain, and the client transitions to the key exchange phase.

Certificate validation must succeed exactly once and is never repeated during the handshake or channel phases. Subsequent transcript and key derivation operations assume that the certificate and configuration have been authenticated and remain constant for the duration of the session.

3.6 Root Certificate Integrity and Self-Signing

The root certificate is the static trust anchor distributed to all clients before any session begins. By default, the root certificate is self-signed: a SHA3-256 hash of the certificate fields is computed in canonical order (verification key, issuer, serial number, expiration-from, expiration-to, algorithm identifier, version) and signed by the root's own private signing key. The resulting signature is stored in the certificate's signature field. Any recipient that holds the root public verification key may verify this self-signature to confirm the integrity of the root certificate's own fields without reliance on external infrastructure.

When the `QSTP_EXTERNAL_SIGNED_ROOT` compilation flag is defined, the root certificate is instead signed by an external authority. In this mode, additional fields for the external authority identity, key identifier, and signing scheme are included in the certificate and the external authority's signature replaces the self-signature. This mode supports hierarchical trust chains for deployments that require integration with an existing public key infrastructure.

In both modes, the server certificate is signed by the root's private signing key. The client uses the root public verification key to validate the server certificate's signature during the ConnectResponse processing step.

3.7 Key Encapsulation and Shared Secret Construction

QSTP uses a post quantum key encapsulation mechanism during the handshake to establish a shared secret. The server generates a fresh ephemeral KEM key pair for each handshake. This ephemeral public key is transmitted to the client in the ConnectResponse and signed under the server's long-term signature key. The client uses the public key to encapsulate a shared secret, and the server uses the corresponding ephemeral secret key to recover it. The secret key is erased immediately after decapsulation.

Client Encapsulation. Upon receiving and validating the ConnectResponse message, the client extracts the server's KEM public key from the configuration block. The client then performs encapsulation

$$(c, s_{\mathsf{cli}}) \leftarrow \mathsf{KEM.Enc}(\mathsf{pk}_{\mathsf{srv}}),$$

producing a ciphertext c and a shared secret s_{cli}. The ciphertext is transmitted in the ExchangeRequest message. The shared secret is retained locally by the client and is not exposed to the server until decapsulation succeeds.
Encapsulation is deterministic with respect to its cryptographic semantics: it always produces a valid ciphertext and shared secret pair unless the server's public key is malformed. Any such malformed public key would have been rejected earlier during the certificate and configuration validation phase.

Server Decapsulation. After receiving the ExchangeRequest message, the server performs decapsulation using its ephemeral KEM secret key:

$$s_{\mathsf{srv}} \leftarrow \mathsf{KEM.Dec}(\mathsf{sk}_{\mathsf{kem}}^{\mathsf{eph}}, c).$$

If the ciphertext c is valid, this produces a shared secret identical to s_{cli}. If c is malformed or inconsistent, decapsulation fails and the handshake is terminated immediately.
The KEX engine ensures that decapsulation either returns a valid shared secret or cleanly signals failure without revealing partial information. This behavior aligns with the IND-CCA security definition used in the formal analysis.

3.8 Key Derivation and Channel Parameter Construction

Once both parties hold the shared secret, the channel parameters are derived using a cSHAKE-based key derivation function. The shared secret sec is used as the input message and the final transcript hash sch_2 is used as the customization string:

$$\mathsf{prnd} \leftarrow \mathsf{cSHAKE}(\mathsf{sec},\ \mathsf{sch}_2).$$

The pseudo-random output block prnd is split into four independent values:

$$(k_{c\to s},\ n_{c\to s},\ k_{s\to c},\ n_{s\to c})\ \leftarrow\ \mathsf{prnd}.$$

Directional Keys. The first two outputs of the KDF become the client to server and server to client encryption keys for the RCS AEAD channel. These keys are distinct and never reused across directions, ensuring that encryption in each direction is independent.

Initial Nonces. The final two KDF outputs are used as initial nonces for each direction. Nonces are treated as fixed width integers incremented during the lifespan of the channel. Because the handshake is deterministic and includes no optional fields, both parties derive identical nonces for each direction.

Transcript Binding. Since sch_2 commits the protocol configuration, the server's certified identity, the ephemeral encapsulation key, and the client's ciphertext into the KDF customization string, the resulting keys are cryptographically bound to the complete authenticated transcript. Any attempt by an adversary to modify or replay any earlier handshake message produces a divergent sch_2 and causes immediate key derivation mismatch.

3.9 Channel Establishment and Cipher State

Once key derivation completes and the client confirms the EstablishVerify transcript hash comparison, both client and server transition into the established channel state.

RCS Initialization. Each party initializes two RCS contexts:

- an outbound context using the directional key and nonce for the sending direction,
- an inbound context using the directional key and nonce for the receiving direction.

Both contexts incorporate the header fields (flag byte, sequence number, timestamp) as associated data. No plaintext is released unless the RCS tag verifies correctly.

Sequence Counters. Each direction maintains a monotonic 64 bit sequence counter. For outbound packets, the counter increments with each encrypted message. For inbound packets, the counter must strictly increase relative to the last accepted packet. Packets with non increasing sequence numbers are rejected before decryption.

Timestamp Handling. Each packet contains a timestamp supplied by the sender and authenticated as associated data. Upon reception, the timestamp must fall within a configured freshness window Δ relative to the local system clock. Packets outside this window are rejected before decryption.

Channel Transition. The channel is considered established only when:

- key derivation succeeds locally,
- the final handshake confirmation (Verify) is exchanged,
- RCS contexts are initialized in both directions,
- sequence counters and nonces are set to their initial values.

At this point both parties share identical channel state and can exchange encrypted packets until termination. Any deviation from these rules, including timestamp violations, sequence number inconsistencies, or AEAD tag failures, results in packet rejection and may trigger session teardown depending on implementation policy.

3.10 Packet Formats and Header Semantics

QSTP packets transmitted after the handshake consist of a fixed format header followed by an RCS authenticated ciphertext. The header encodes information required for replay protection, directionality, and freshness checks. The RCS layer binds the header to the ciphertext through associated data, ensuring that any modification to the header invalidates the packet before decryption.

3.10.1 Header Fields

Each QSTP data packet begins with a compact header of constant length. The implementation defines and processes the following fields.

- **Flag Byte.** A single byte indicating the logical type of the packet. During the encrypted channel phase this value identifies ordinary data packets and any protocol specific control flags. The flag byte is public but must be authenticated to prevent reclassification attacks.
- **Sequence Number.** An unsigned 64 bit integer incremented by the sender for every outbound packet. It begins at the initial value derived at the end of the handshake and increments monotonically. The receiver must accept only packets whose sequence number is strictly greater than the maximum previously accepted value in that direction. Sequence number violations cause immediate rejection before AEAD decryption.

- **Timestamp.** An unsigned 64-bit low resolution timestamp representing the sender's notion of the current time in seconds from the epoch. The receiver enforces that the timestamp lies within a configured freshness window Δ of its own local clock (60 seconds by default). Packets whose timestamps fall outside this window are rejected without decryption.

The combined structure of the header is therefore:

$$\mathsf{Header} = \langle \mathsf{flag},\ \mathsf{mlen},\ \mathsf{seq},\ \mathsf{ts} \rangle,$$

with total length fixed across all packets.
The implementation neither negotiates nor adapts these header fields. Their format and length remain constant for the duration of the session.

3.10.2 Authenticated Data

The QSTP implementation uses RCS as an authenticated encryption scheme with associated data. The header described above is incorporated into the associated data parameter of RCS for every encrypted packet. Specifically, let H denote the serialized header, k the directional AEAD key, and n the current nonce. Encryption proceeds as:

$$C \leftarrow \mathsf{RCS.Enc}(k, n, H, P),$$

where P is the plaintext payload. The output C includes both ciphertext and authentication tag.
On reception, decryption proceeds as:

$$P \leftarrow \mathsf{RCS.Dec}(k, n', H, C),$$

where n' is the expected nonce value determined by the inbound sequence state. Decryption returns P only if the tag verifies and H matches the transmitted header exactly.
The use of associated data ensures that any modification to the flag field, message length, sequence number, or timestamp results in immediate rejection. Neither the ciphertext nor the tag is processed further in such cases. The AEAD layer therefore provides integrity for both packet payload and header, and the header values become inseparable from the encrypted packet structure.
Because the RCS tag binds the entire header, replay, reordering, and timestamp violations are detected before any plaintext is exposed. This design ensures that all channel security properties depend directly on the authenticated header semantics enforced by the AEAD mechanism.

3.11 Packet Acceptance and Replay Logic

After the handshake completes and the channel is established, each party enforces strict rules on inbound packets to prevent replay, reordering, and stale packet acceptance. These rules are implemented entirely in the header validation logic performed before invoking the RCS decryption operation. If any validation check fails, the packet is discarded without revealing any plaintext.

Sequence Number Monotonicity. Each direction maintains a local record of the sequence number $\mathsf{seq}_{\mathsf{cur}}$ that has been successfully accepted. When an inbound packet with sequence number seq is received, the packet is accepted for further processing only if the sequence number equals the current sequence number plus one:

$$\mathsf{seq} = \mathsf{seq}_{\mathsf{cur}} + 1.$$

If this condition fails, the packet is classified as a replay or reordering attempt and is rejected immediately. Upon successful acceptance, $\mathsf{seq}_{\mathsf{cur}}$ is updated to seq. No window or reordering buffer is used; the sequence number must increase monotonically for each received packet.

Timestamp Freshness Window. Each inbound packet includes a timestamp value ts authenticated as associated data. The receiving endpoint compares ts to its local clock value now and enforces a freshness bound Δ. The packet is accepted only if the packet time is withing the freshness window:

$$|\mathsf{ts} - \mathsf{now}| \le \Delta.$$

Packets whose timestamps lie outside this window are considered stale or potentially replayed and are discarded before invoking the AEAD decryption operation. The value of Δ is fixed for the duration of the session and does not adapt dynamically.

Consistent Direction and Header Semantics. The receiver verifies that the flag byte and header structure correspond to an expected channel packet and that the packet length falls within implementation defined bounds. The sequence number and timestamp fields must appear in the exact byte positions defined by the implementation. Any deviation indicates corruption or tampering.

AEAD Tag Validation. If the sequence and timestamp checks succeed, and the payload size is in the expected size range, the receiver invokes RCS decryption using the authenticated header H and the current nonce derived from inbound sequence state. The decryption operation succeeds only if the authentication tag matches the ciphertext. Failure of tag verification results in immediate rejection of the packet.
These rules ensure that the receiver processes packets only in a strictly increasing sequence, within a freshness window, and with correctly authenticated header and payload. No plaintext is exposed unless all checks succeed. This logic enforces the replay, ordering, and integrity guarantees required by the protocol.

3.12 Error Handling and Teardown

The implementation defines explicit abort conditions for both handshake and channel phases. These conditions cause immediate termination of the active session, preventing further processing of inbound or outbound messages. While the exact error codes and return values are implementation specific, the logical reasons for termination are consistent across all modules.

Handshake Abort Conditions. The handshake is terminated immediately under any of the following conditions.

- The certificate derived values in the ConnectResponse fail validation under the configured root and server certificate (for example signature verification fails or the session cookie and serial do not match the cached certificate).
- The server configuration fields conflict with the authenticated certificate chain.
- The session cookie computed by the client does not match the cookie transmitted by the server.
- Decapsulation of the client's KEM ciphertext fails on the server.
- Any signature in the handshake fails verification.

- Messages arrive out of order or with an unexpected type or size for the current state.
- Mandatory fields in handshake messages are malformed or outside the permissible bounds.

These conditions align with the requirement that transcript integrity must hold across all handshake stages.

Channel Abort Conditions. Once the channel is established, the following conditions cause immediate session teardown.

- Inbound packets fail sequence number monotonicity or timestamp freshness checks.
- The AEAD tag for a ciphertext does not verify successfully.
- The header fields are malformed or incompatible with the channel state.
- Nonce or sequence counter overflow occurs.
- An implementation configured timeout or idle interval expires.

The decryption operation is never attempted on packets that fail preliminary header validation.

Teardown Semantics. On teardown, all session state is destroyed, including:

- derived channel keys and nonces,
- KEM shared secret and intermediate buffers,
- sequence number and timestamp state,
- certificate and configuration references,
- inbound and outbound RCS contexts.

The implementation guarantees that no sensitive material persists beyond the lifetime of the session. Any subsequent communication requires a complete new handshake and revalidation of server credentials.
These teardown rules ensure that both handshake and channel failures result in consistent, secure termination without exposing cryptographic or protocol state.

3.13 Engineering Pseudocode Definitions

The pseudocode in this section follows the kex implementation at the level of individual message flows. Each algorithm represents the logic associated with one message type in the QSTP key exchange: connect request, connect response, exchange request, exchange response, and the final establish verify step. The algorithms abstract away C level details such as buffer allocation and error codes, but preserve the exact conditions, state flags, and cryptographic operations found in the source code.

3.13.1 Algorithm 1: ConnectRequest (Client)

This algorithm models the client side construction of the ConnectRequest. The client already holds a configured view of the server certificate, including the expected certificate serial, the certificate hash used as session cookie, and the protocol set string.

Algorithm 1 ConnectRequest

Require: Client kex state kcs with (serial, schash, verkey, expiration), connection state cns

Ensure: ConnectRequest packet M_1 or failure $\perp$

```
1: $tm \leftarrow$ current_time
2: if $tm >$ kcs.expiration then
3:     return $\perp$        // Local certificate expired
4: end if
5: cns.exflag $\leftarrow$ qstp_flag_none
6: Construct payload with
       certificate serial kcs.serial
       protocol set string QSTP_PROTOCOL_SET_STRING
7: Create header for $M_1$ with flag connect_request and length $KEX\_CONNECT\_REQUEST\_MESSAGE\_SIZE$
8: Write payload into $M_1$.pmessage
9: cns.exflag $\leftarrow$ qstp_flag_connect_request
10: return $M_1$
```

3.13.2 Algorithm 2: ConnectResponse (Server)

This algorithm models the server side processing of the ConnectRequest and construction of the ConnectResponse. The server verifies that the requested certificate serial and protocol set are supported, confirms that the local certificate is not expired, generates a fresh KEM key pair, and signs a hash that binds the response header, session cookie, and public encapsulation key.

Algorithm 2 CONNECTRESPONSE

```
Require: Server kex state kss with (serial, schash, sigkey, expiration), connection state cns,
    ConnectRequest $M_1$
Ensure: ConnectResponse packet $M_2$ or failure $\perp$
 1: if $M_1$.flag $\neq$ connect_request then
 2:     return $\perp$
 3: end if
 4: Parse $M_1$ payload to obtain requested serial $s_{\mathsf{req}}$ and protocol set $p_{\mathsf{req}}$
 5: if $s_{\mathsf{req}} \neq$ kss.serial or $p_{\mathsf{req}} \neq$ QSTP_PROTOCOL_SET_STRING then
 6:     return $\perp$
 7: end if
 8: $tm \leftarrow$ current_time
 9: if $tm >$ kss.expiration then
10:     return $\perp$
11: end if
12: Generate fresh KEM key pair (pk, sk) with key generator AG
13: Store pk and sk in kss for this session
14: Create header for $M_2$ with flag connect_response and length
    $KEX\_CONNECT\_RESPONSE\_MESSAGE\_SIZE$
15: Serialize header of $M_2$ into buffer $H$
16: Compute phash $\leftarrow$ Hash($H$ || kss.schash || pk)
    // kss.schash here is $\mathsf{sch}_0$ = Hash(cfg || serial || pvk)
17: Compute signature $\sigma \leftarrow$ ASsk(kss.sigkey, phash)
18: Advance transcript: kss.schash $\leftarrow$ Hash(kss.schash || phash)
    // kss.schash is now $\mathsf{sch}_1$
19: Write $\sigma$ and pk into $M_2$.pmessage in that order
20: cns.exflag $\leftarrow$ qstp_flag_connect_response
21: return $M_2$
```

3.13.3 Algorithm 3: ExchangeRequest (Client)

This algorithm models the client processing of the ConnectResponse and the construction of the ExchangeRequest. The client verifies the signed binding of header, cookie, and public key, then encapsulates a shared secret, derives keys and nonces, initializes its ciphers, and sends the ciphertext to the server.

Algorithm 3 ExchangeRequest

```
Require: Client kex state kcs with (schash, verkey), connection state cns, ConnectResponse $M_2$
Ensure: ExchangeRequest packet $M_3$ or failure $\perp$
 1: if cns.exflag $\neq$ qstp_flag_connect_request or $M_2$.flag $\neq$ connect_response then
 2:     return $\perp$
 3: end if
 4: Parse $M_2$.pmessage into $(\sigma, h_{\mathsf{cert}}, \mathsf{pk})$
 5: if $h_{\mathsf{cert}} \neq$ kcs.schash then
 6:     return $\perp$
 7: end if
 8: Serialize header of $M_2$ into buffer $H$
 9: phash $\leftarrow$ Hash($H$ || kcs.schash || pk)    // kcs.schash here is $\mathsf{sch}_0$
10: if AVpk(kcs.verkey, phash, $\sigma$) = false then
11:     return $\perp$
12: end if
13: Advance transcript: kcs.schash $\leftarrow$ Hash(kcs.schash || phash)
    // kcs.schash is now $\mathsf{sch}_1$
14: $(cpt, ssec) \leftarrow$ KEM_Encapsulate(pk)
15: Create header for $M_3$ with flag exchange_request and length
    $KEX\_EXCHANGE\_REQUEST\_MESSAGE\_SIZE$
16: Write $cpt$ into $M_3$.pmessage
17: Advance transcript: kcs.schash $\leftarrow$ Hash(kcs.schash || $cpt$)    // kcs.schash is now $\mathsf{sch}_2$
18: Use $ssec$ and kcs.schash as input to DeriveChannelKeys to initialize cns.txcpr and
    cns.rxcpr    // KDF input is (sec, $\mathsf{sch}_2$)
19: cns.exflag $\leftarrow$ qstp_flag_exchange_request
20: return $M_3$
```

3.13.4 Algorithm 4: ExchangeResponse (Server)

This algorithm models the server processing of the ExchangeRequest and the construction of the ExchangeResponse. The server decapsulates the client ciphertext using the private key generated during ConnectResponse, derives keys and nonces from the shared secret and session cookie, initializes its ciphers, and signals that the channel is ready.

Algorithm 4 ExchangeResponse

```
Require: Server kex state kss with (schash, pk, sk), connection state cns, ExchangeRequest $M_3$
Ensure: ExchangeResponse packet $M_4$ or failure $\perp$
 1: if cns.exflag $\neq$ qstp_flag_connect_response or $M_3$.flag $\neq$ exchange_request then
 2:     return $\perp$
 3: end if
 4: Extract ciphertext $cpt$ from $M_3$.pmessage
 5: $ssec \leftarrow$ KEM_Decapsulate(sk, $cpt$)
 6: if $ssec = \perp$ then
 7:     return $\perp$
 8: end if
 9: Advance transcript: kss.schash $\leftarrow$ Hash(kss.schash $\|$ $cpt$)
    // kss.schash is now $\mathsf{sch}_2$; server had already advanced to $\mathsf{sch}_1$ in Algorithm 2
10: Use $ssec$ and kss.schash as input to DeriveChannelKeys to initialize cns.txcpr and
    cns.rxcpr according to server role    // KDF input is (sec, $\mathsf{sch}_2$)
11: Securely erase sk and $ssec$ from kss
12: Create header for $M_4$ with flag exchange_response and length
    $KEX\_EXCHANGE\_RESPONSE\_MESSAGE\_SIZE$
13: Write the final transcript hash kss.schash ($\mathsf{sch}_2$) into $M_4$.pmessage
    // Explicit key confirmation: the client will verify this value against its own $\mathsf{sch}_2$
14: cns.exflag $\leftarrow$ qstp_flag_exchange_response
15: return $M_4$
```

3.13.5 Algorithm 5: EstablishVerify (Client)

This algorithm models the final client side state update after receiving the ExchangeResponse. It checks that the response flag matches the expected value and, if so, marks the channel as established. If the flag indicates an error or does not match the expected state, the session is torn down.

Algorithm 5 EstablishVerify

```
Require: Client kex state kcs, connection state cns, ExchangeResponse $M_4$
Ensure: Updated connection state cns and error code
 1: if cns.exflag $\neq$ qstp_flag_exchange_request then
 2:     return error_invalid_state
 3: end if
 4: if $M_4$.flag $\neq$ exchange_response then
 5:     if $M_4$.flag indicates an error then
 6:         return error_remote_failure
 7:     else
 8:         return error_invalid_request
 9:     end if
10: end if
11: Extract server transcript hash $\mathsf{sch}_2^{\mathsf{srv}}$ from $M_4$.pmessage
12: if $\neg$ ConstantTimeEqual($\mathsf{sch}_2^{\mathsf{srv}}$, kcs.schash) then    // kcs.schash is the client's locally computed $\mathsf{sch}_2$
13:     return error_verify_failure    // Explicit key confirmation failed; tear down session
14: end if
15: cns.exflag $\leftarrow$ qstp_flag_session_established
16: return error_none
```

3.13.6 Algorithm 6: KEM_Encapsulate

The encapsulation algorithm models the client side use of the asymmetric encryption primitive in the QSTP key exchange. The implementation calls into the KEX engine to generate a ciphertext of fixed length and a shared secret of length `QSTP_SECRET_SIZE`, using the server KEM public key and a cryptographically secure random number generator. The ciphertext is sent to the server as part of the ExchangeStart message, and the shared secret is retained locally for subsequent key derivation.

```
Algorithm 6 KEM_ENCAPSULATE
Require: Server KEM public key $\mathsf{pk}_{\mathsf{srv}}$
Ensure: Ciphertext $c$ and shared secret $ssec$
 1: Sample randomness $r$ from a cryptographically secure generator
 2: $(c, ssec) \leftarrow (\mathsf{pk}_{\mathsf{srv}}, r)$
 3: return $(c, ssec)$
```

Here denotes the asymmetric encapsulation function implemented by the KEX engine. The ciphertext c has fixed length equal to the asymmetric cipher text size and $ssec$ has fixed length equal to the secret size used by QSTP.

3.13.7 Algorithm 7: KEM_Decapsulate

The decapsulation algorithm models the server side use of the asymmetric decryption primitive. The implementation calls into the KEX engine with the server KEM secret key and the ciphertext extracted from the ExchangeStart message. If decapsulation succeeds, it returns a shared secret of the same length as on the client side. If the ciphertext is invalid or tampered, decapsulation fails and the handshake is aborted.

```
Algorithm 7 KEM_DECAPSULATE
Require: Server KEM secret key $\mathsf{sk}_{\mathsf{srv}}^{\mathsf{kem}}$, ciphertext $c$
Ensure: Shared secret $ssec$ or failure $\perp$
 1: if $c$ has invalid length or format then
 2:     return $\perp$
 3: end if
 4: $ssec \leftarrow_k (\mathsf{sk}_{\mathsf{srv}}^{\mathsf{kem}}, c)$
 5: if $ssec = \perp$ then
 6:     return $\perp$        // Decapsulation failure
 7: end if
 8: return $ssec$
```

Here $_k$ denotes the asymmetric decapsulation function implemented by the KEX engine. The output $ssec$ is a fixed length shared secret that matches the one produced by KEM_ENCAPSULATE when given the same server key pair and ciphertext. Any deviation in c from a valid ciphertext results in failure and no partial information about the secret is exposed.

3.13.8 Algorithm 8: DeriveChannelKeys

This algorithm models the concrete key derivation and cipher initialization performed in the QSTP implementation. Given a shared secret and the certificate based session cookie, it uses a cSHAKE based KDF to generate a block of pseudorandom bytes, splits this block into two symmetric keys and two nonces, then initializes the directional cipher contexts.

The mapping of keys to transmit and receive directions depends on the local role (client or server) to ensure consistent channel orientation.

Algorithm 8 DeriveChannelKeys

```
Require: Shared secret $ssec$, certificate hash schash, role role $\in$ {client, server}, connection
    state cns
Ensure: Initialized cipher contexts cns.txcpr and cns.rxcpr
 1: Let $L_k$ be the symmetric key length and $L_n$ the nonce length for RCS
 2: Let $L_r$ be the cSHAKE output block size
 3: Initialize cSHAKE state $K$
 4: input $\leftarrow$ $ssec$        // KEM shared secret
 5: custom $\leftarrow$ schash        // Final transcript hash $\mathsf{sch}_2$; caller must have advanced transcript before this call
 6: Initialize cSHAKE with $(K, \mathsf{input}, \mathsf{custom})$
 7: Generate $L_r$ bytes of pseudorandom output $R$ from $K$
 8: Overwrite $ssec$ and permute $K$ to avoid retaining the current key material
 9: $k_1 \leftarrow R[0..L_k - 1]$
10: $n_1 \leftarrow R[L_k..L_k + L_n - 1]$
11: $k_2 \leftarrow R[L_k + L_n..2L_k + L_n - 1]$
12: $n_2 \leftarrow R[2L_k + L_n..2L_k + 2L_n - 1]$
13: if role = client then
14:     Initialize transmit cipher cns.txcpr with key $k_1$ and nonce $n_1$
15:     Initialize receive cipher cns.rxcpr with key $k_2$ and nonce $n_2$
16: else
17:     Initialize receive cipher cns.rxcpr with key $k_1$ and nonce $n_1$
18:     Initialize transmit cipher cns.txcpr with key $k_2$ and nonce $n_2$
19: end if
```

3.13.9 Algorithm 9: EncryptPacket

This algorithm models the packet encryption routine used by QSTP after the channel is established. It constructs the packet header, updates the transmit sequence number, serializes the header into a fixed length buffer, binds the header as associated data to the RCS AEAD context, and encrypts the plaintext payload into the packet body. The resulting network packet contains the authenticated header, ciphertext, and authentication tag.

Algorithm 9 ENCRYPTPACKET

```
Require: Established connection state cns, plaintext message $P$, length $|P|$
Ensure: Network packet pkt containing encrypted payload or error
 1: Initialize error code err $\leftarrow$ error_invalid_input
 2: if cns.exflag $\neq$ session_established or $|P| = 0$ then
 3:     return err
 4: end if
 5: Allocate header buffer $H$ of fixed size
 6: cns.txseq $\leftarrow$ cns.txseq $+ 1$
 7: Construct packet header in pkt with:
        flag $\leftarrow$ encrypted_message
        sequence $\leftarrow$ cns.txseq
        length $\leftarrow |P| +$ tag_size
        timestamp $\leftarrow$ current_time
 8: Serialize header fields from pkt into $H$
 9: Set associated data of transmit cipher cns.txcpr to $H$
10: Encrypt $P$ with cns.txcpr into pkt.pmessage, appending authentication tag
11: err $\leftarrow$ error_none
12: return err
```

3.13.10 Algorithm 10: DecryptPacket

This algorithm models the packet decryption routine used by QSTP in the established channel state. It validates the sequence number and timestamp, ensures the channel is active, serializes the header into a fixed length buffer, binds the header as associated data, and then invokes RCS decryption. On any failure, the packet is rejected and no plaintext is released.

Algorithm 10 DecryptPacket

```
Require: Established connection state cns, network packet pktin, output buffer message,
    length pointer msglen
Ensure: Updated msglen and error code err
 1: Initialize header buffer $H$ of size QSTP_PACKET_HEADER_SIZE
 2: err $\leftarrow$ error_invalid_input
 3: msglen $\leftarrow 0$
 4: if cns $= \perp$ or pktin $= \perp$ or message $= \perp$ or msglen $= \perp$ then
 5:     return err
 6: end if
 7: cns.rxseq $\leftarrow$ cns.rxseq $+ 1$
 8: if pktin.sequence $\neq$ cns.rxseq then
 9:     err $\leftarrow$ error_packet_unsequenced
10:     return err
11: end if
12: if cns.exflag $\neq$ session_established then
13:     err $\leftarrow$ error_channel_down
14:     return err
15: end if
16: if PacketTimeValid(pktin) $=$ false then
17:     err $\leftarrow$ error_message_time_invalid
18:     return err
19: end if
20: Serialize header fields of pktin into $H$
21: Set associated data of receive cipher cns.rxcpr to $H$
22: msglen $\leftarrow$ pktin.msglen $-$ QSTP_MACTAG_SIZE
23: if CipherTransform(cns.rxcpr, message, pktin.pmessage, msglen) $=$ true then
24:     err $\leftarrow$ error_none
25: else
26:     msglen $\leftarrow 0$
27:     err $\leftarrow$ error_authentication_failure
28: end if
29: return err
```

3.14 Summary of Engineering Behavior

The engineering model of QSTP described in this chapter matches the reference implementation at the level of message formats, state transitions, and cryptographic operations. The handshake is realized as a deterministic sequence of messages starting from a ConnectRequest and ending with a Verify confirmation. The server certificate and configuration are validated once using a fixed root key. A session cookie derived from the certificate chain binds the transcript to the authenticated server identity, and this cookie is fed into the key derivation function together with the KEM shared secret. The resulting channel keys and nonces are partitioned by direction according to the local role, and both sides initialize their RCS contexts from the same derived values.

Packet formats use a fixed header consisting of a flag byte, sequence number, and timestamp. The implementation treats these header fields as associated data for RCS, so any modification to them causes decryption to fail before plaintext is exposed. Sequence numbers enforce strict monotonicity and provide replay protection, while timestamps are checked against a freshness window to reject stale packets.

The packet acceptance logic and teardown rules ensure that any violation, including malformed messages, failed certificate checks, KEM decapsulation errors, timestamp

violations, sequence anomalies, or AEAD tag failures, leads to immediate session abort and destruction of sensitive state. This behavior corresponds directly to the abstract security model in which handshake failures and channel errors are modeled as local aborts with no information leakage.
Taken together, these engineering properties implement the formal QSTP protocol model with a linear state machine, fixed cryptographic posture, and explicit binding between the authenticated certificate chain, the key exchange, and the RCS protected channel.

4 Notation and Preliminaries

4.1 Notation

We use standard notation for bit strings, sampling, concatenation, and protocol state. All bit strings are elements of $\{0,1\}^*$. Concatenation of strings X and Y is written $X \parallel Y$. Sampling x uniformly from a finite set S is written $x \xleftarrow{\$} S$. If F is a function, we write $y \leftarrow F(x)$ for deterministic evaluation. Tuples are written $\langle x_1, x_2, \ldots, x_n \rangle$.
For a party P, local sessions are indexed by integers i and are denoted P^i. Each session maintains a session identifier sid determined by its transcript, and a peer identifier pid indicating the intended partner. When a session derives a key, we denote its session key by sk. The symbol $\perp$ denotes failure. Negligible functions are denoted $\mathsf{negl}(\cdot)$.
For random oracles modeled in the analysis, we write $\mathcal{H}(\cdot)$ for the hash based primitive and assume that $\mathcal{A}$ may issue queries to $\mathcal{H}$ during the experiment. All random choices made by the oracles or by honest parties are independent unless specified otherwise.

4.2 Cryptographic Primitives

QSTP is constructed from four cryptographic primitives. We define each abstractly and rely on their standard security notions.

Key Encapsulation Mechanism. A KEM consists of algorithms

$$(\mathsf{KEM.KeyGen},\ \mathsf{KEM.Enc},\ \mathsf{KEM.Dec})$$

where $\mathsf{KEM.KeyGen}$ outputs a public key and secret key pair $(\mathsf{pk}, \mathsf{sk})$, $\mathsf{KEM.Enc}(\mathsf{pk})$ outputs a ciphertext c and a shared secret s, and $\mathsf{KEM.Dec}(\mathsf{sk}, c)$ outputs s or $\perp$. We assume the KEM is IND-CCA secure, meaning no probabilistic polynomial time adversary can distinguish its shared secret from uniform in the standard challenge experiment.

Digital Signature Scheme. A signature scheme consists of $(\mathsf{SIG.KeyGen},\ \mathsf{SIG.Sign},\ \mathsf{SIG.Verify})$. The key generation algorithm outputs $(\mathsf{vk}, \mathsf{sk})$, the signing algorithm computes $\sigma \leftarrow \mathsf{SIG.Sign}(\mathsf{sk}, m)$, and the verification algorithm outputs 1 or 0. We assume existential unforgeability under chosen message attack (EUF-CMA).

Authenticated Encryption with Associated Data. An AEAD scheme provides $(\mathsf{AEAD.Enc},\ \mathsf{AEAD.Dec})$ with associated data. Encryption takes a key k, associated data A, and plaintext P, and outputs ciphertext C and tag T. Decryption returns P or $\perp$. QSTP uses RCS as its AEAD primitive. We assume standard AEAD security: confidentiality under chosen plaintext attack and ciphertext integrity under chosen ciphertext attack.

Key Derivation Function. A key derivation function KDF maps the KEM shared secret and a session specific cookie to four outputs used as symmetric channel keys and nonces. In the analysis the KDF is modeled as a random oracle or as a pseudorandom function with appropriate domain separation.

4.3 Adversarial Model and Network

The protocol runs in an adversarial network controlled by a probabilistic polynomial time adversary $\mathcal{A}$. The adversary delivers all messages between parties, and may delay, drop, duplicate, or reorder them arbitrarily. The adversary has full visibility of ciphertexts, associated data, and header fields.

The adversary may create multiple concurrent sessions, interact with them through the Send oracle, corrupt long term keys when permitted by the security model, and attempt to distinguish the session key of a fresh session from a random value. The adversary cannot compromise the root key that validates server certificates, and cannot violate the assumed hardness of the underlying cryptographic primitives.

The network is synchronous only in the sense that messages are delivered through the adversary interface. Parties maintain local state and accept or reject incoming messages based on signature validation, session consistency, and replay and freshness checks. No assumptions are made about reliable or ordered delivery beyond what the protocol enforces.

5 Formal QSTP Protocol Model

5.1 Participants, Sessions, and Identifiers

Let $\mathcal{P}$ be the set of protocol participants. In QSTP there are two roles: a client C and a server S. Each party may spawn multiple local sessions indexed by integers, written C^i and S^j. A session identifier sid is associated with each local session and is defined as a function of the ordered transcript of messages exchanged during the handshake together with the identities of the parties. Two sessions are said to have the same sid if and only if they have identical transcripts and opposite roles.

Each session stores a peer identifier pid, which is the identity of the intended partner. For client sessions this is set to the server identity extracted from the server certificate and configuration. For server sessions this is set to the public key or connection context used to create the session. Once set, the peer identifier does not change.

A session is *complete* when it finishes the five message handshake and derives symmetric keys for both transmission directions. Before completion the session is *in progress.* A session may be *orphaned* if it receives no matching partner in the execution.

5.2 Local State and Transitions

Each local session maintains the following state components.

- **Role.** A bit indicating whether the session is a client or server instance. This determines which handshake messages it sends and accepts.
- **Long term keys.** The server maintains a long term signature key pair $(\mathsf{vk}_S, \mathsf{sk}_S)$ that authenticates its handshake messages. The client maintains the root verification key used to validate the server certificate.
- **Ephemeral KEM state.** The server generates a fresh KEM key pair $(\mathsf{pk}_S^{\mathsf{kem}}, \mathsf{sk}_S^{\mathsf{kem}})$ for each handshake instance. This key pair is created when constructing the ConnectResponse and the secret key is erased upon completion of decapsulation. The client produces a KEM ciphertext c and shared secret s via $\mathsf{KEM.Enc}(\mathsf{pk}_S^{\mathsf{kem}})$.
- **Transcript and session cookie.**
- **Transcript hash chain.** Sessions maintain the rolling transcript hash $(\mathsf{sch}_0, \mathsf{sch}_1, \mathsf{sch}_2)$ computed from the protocol configuration, the server certificate fields, the signed ephemeral key, and the KEM ciphertext as defined in Section 11

and §4.4. These values are computed independently by each party and determine the transcript consistency checks performed during the handshake.

- **Derived channel keys.** Upon successful exchange, each session derives

$$(k_{c\to s},\ n_{c\to s},\ k_{s\to c},\ n_{s\to c})$$

using the key derivation function $\mathsf{cSHAKE}(\mathsf{sec}, \mathsf{sch}_2)$ applied to the KEM shared secret and the final transcript hash.

- **Sequence counters.** Each session maintains a non decreasing sequence counter for its sending direction and monitors the sequence counter for the receiving direction. Sequence values are authenticated through RCS associated data.

State transitions occur when a session sends or receives one of the five handshake messages. At each transition the session verifies that:

- the message type matches the expected state,
- the peer identifier is consistent,
- all signatures validate,
- the transcript values match previously recorded configuration,
- and, at the final step, the derived keying material matches the session expectations.

Any deviation causes the session to abort and output $\perp$.

5.3 Network Execution and Scheduling

The protocol execution is modeled through an experiment controlled by a probabilistic polynomial time adversary $\mathcal{A}$. The adversary interacts with session instances through a Send oracle. A call $\mathsf{Send}(P^i, m)$ delivers message m to session P^i, which processes the message according to the local transition rules and may produce an outgoing message m'. If m' is produced, the adversary receives m' and may deliver it to any other session or withhold it.

The adversary controls the ordering and delivery of all messages. Messages may be delayed, duplicated, or dropped arbitrarily. The adversary sees all ciphertexts, associated data, and headers. The only constraints are the cryptographic correctness of primitives and the local state transition logic.

A session that reaches the end of the handshake derives a session key vector $(k_{c\to s}, k_{s\to c})$ and enters the channel phase. The adversary may continue issuing Send queries to deliver encrypted packets, but decryption occurs only after RCS tag verification and header validation. The adversary never sees plaintext unless it can break RCS confidentiality.

5.4 Session Cookie and Channel Binding

The transcript hash chain $(\mathsf{sch}_0, \mathsf{sch}_1, \mathsf{sch}_2)$ binds the protocol configuration, the server's certified identity, the server's ephemeral encapsulation key, and the client's KEM ciphertext into the key derivation process. Formally, the chain is defined as:

$$\begin{aligned}
\mathsf{sch}_0 &= \mathsf{Hash}(\mathsf{cfg} \parallel \mathsf{serial} \parallel \mathsf{pvk}),\\
\mathsf{sch}_1 &= \mathsf{Hash}(\mathsf{sch}_0 \parallel \mathsf{phash}), \quad \mathsf{phash} = \mathsf{Hash}(\mathsf{phdr} \parallel \mathsf{sch}_0 \parallel \mathsf{pk}_{\mathsf{kem}}),\\
\mathsf{sch}_2 &= \mathsf{Hash}(\mathsf{sch}_1 \parallel \mathsf{cpta}),
\end{aligned}$$

where cfg is the protocol configuration string, serial is the server certificate serial number, pvk is the server's public signature verification key, phdr is the serialized ConnectResponse

header, $\mathsf{pk}_{\mathsf{kem}}$ is the server's ephemeral public encapsulation key, and cpta is the KEM ciphertext produced by the client.
The KDF uses the pair $(\mathsf{sec}, \mathsf{sch}_2)$ as input, where sec is the KEM shared secret. Because sch_2 commits every handshake message into the derivation input, any attempt by an adversary to substitute configuration values, modify the ephemeral key, or tamper with the ciphertext yields a different sch_2 and causes the derived keys to mismatch. This binding property ensures that the resulting channel keys are linked to the complete authenticated handshake transcript.

6 Security Goals

6.1 Authenticated Key Exchange

The primary objective of the QSTP handshake is to establish a shared secret key between a client and a server in such a way that the resulting session key is known only to the two matching parties. The protocol provides unilateral authentication: the server proves possession of its certified signature key, and the client verifies this before accepting any derived keying material.

An authenticated key exchange execution of QSTP satisfies the following properties.

- **Agreement.** If a client session completes with peer identifier S and derives a session key vector, then there exists a unique matching server session that has derived the same key vector and has the client as its peer.
- **Authenticity.** A client session accepts only if all server handshake messages are verified under the certified server verification key. Any adversarial attempt to forge, modify, or replay handshake messages outside the permitted transcript structure causes the session to abort.
- **Key Indistinguishability.** For any fresh client or server session, the derived session keys are computationally indistinguishable from random under the assumed hardness of the KEM, the security of the signature scheme, the pseudo-randomness of the key derivation function, and the AEAD security of the channel.

6.2 Channel Security

Once the handshake completes, QSTP provides a symmetric channel encrypted under RCS. The security goals of the channel phase are as follows.

- **Confidentiality.** Encrypted packets reveal no information about the plaintext beyond its length. This follows from the confidentiality properties of RCS applied to the keys derived during the handshake.
- **Integrity.** A receiver accepts a ciphertext only if the RCS tag verifies successfully using the derived channel key. Any modification to ciphertext, associated data, sequence number, or timestamp results in rejection without release of plaintext.
- **Channel Binding.** The handshake transcript, certificate identity, and session cookie are incorporated into the key derivation input, ensuring that the symmetric keys are bound to the authenticated server identity.

Between ratchet steps, session keys remain fixed. The integrity and confidentiality guarantees therefore rely on the uniqueness of sequence numbers and on the soundness of the derived symmetric keys within each ratchet epoch. When the optional symmetric ratchet is active, these guarantees extend across epochs by the composition established in Section 11.

6.3 Replay and Reordering Properties

Each direction of a QSTP channel maintains a monotonically increasing sequence number that is authenticated as part of the associated data during encryption. The receiver tracks the highest accepted sequence number and rejects any packet whose sequence number is not strictly greater than the maximum previously accepted value.
A timestamp is included to provide a coarse freshness check and to mitigate replay attempts across larger time windows. The timestamp must be within an acceptable window Δ of the current time when the packet is processed. A packet is rejected if the timestamp is outside this window or if the sequence number violates the monotonicity rule. Neither the ciphertext nor the header is processed further in such cases.
These rules enforce that packets cannot be replayed, significantly reordered, or delivered with artificially inflated timestamps. All such attempts cause immediate rejection in the channel layer before any plaintext is exposed.

6.4 Denial of Service Considerations

Post quantum key exchange operations, particularly KEM decapsulation, can be computationally expensive and present an attack surface for adversarial clients attempting to exhaust server resources. QSTP aims to limit this exposure through a predictable handshake structure and minimal server side work prior to validation of the initial client request.
The security goal is to ensure that the server performs no costly cryptographic operations until the client has provided syntactically valid and context consistent messages. The server verifies the client's initial request, the configuration fields, and the certificate chain of its own messages before decapsulation occurs. Implementations may incorporate additional measures such as stateless retry tokens or early rejection logic to strengthen this property.
The analysis in this paper does not attempt to quantify denial of service resilience formally but treats it as an engineering goal that complements the cryptographic guarantees of the protocol.

7 Security Model and Experiments

7.1 Oracles and Adversary Capabilities

The adversary $\mathcal{A}$ is a probabilistic polynomial time algorithm that interacts with session instances through a set of oracle queries. These oracles model the ability of $\mathcal{A}$ to activate participants, deliver messages, corrupt keys, and attempt to distinguish session keys from random. The following interface is provided to $\mathcal{A}$.

- $\mathsf{Send}(P^i, m)$**.** Delivers message m to session P^i. The session processes the message according to the QSTP transition rules and returns any outgoing message. The adversary may deliver this output to any other session or withhold it.
- $\mathsf{Reveal}(P^i)$**.** If session P^i has completed the handshake and derived a session key, this oracle returns the key vector $(k_{c\to s}, k_{s\to c})$. The session is then marked *revealed* and cannot be used in a freshness test.
- $\mathsf{Corrupt}(S)$**.** Corrupts the long term signature key of the server S, returning sk_S to $\mathcal{A}$. In keeping with QSTP design goals and typical deployment assumptions, the model does not allow corruption of the root verification key. Client long term keys do not exist.
- $\mathsf{Test}(P^i)$**.** May be issued once. If P^i is a fresh session, the experiment samples a random bit b. If $b = 0$ the oracle returns the real session key vector. If $b = 1$ it returns

a uniformly random string of equal length. The adversary outputs a bit b' at the end of the experiment and succeeds if $b' = b$.

A session is *fresh* if it has completed the handshake, its peer has not been corrupted, and neither it nor its matching partner has been revealed. This ensures that revealed or corrupted sessions cannot be used in the indistinguishability test.

7.2 Partnering and Matching Conversations

QSTP has a linear and deterministic handshake, so partnering can be defined directly in terms of message transcripts and peer identities.

Definition 1 (Matching Conversations)**.** Two sessions P^i and Q^j have matching conversations if:

- P^i is a client session and Q^j is a server session, or vice versa,
- the ordered transcripts of the five handshake messages are identical,
- the peer identifiers satisfy $\mathsf{pid}_{P^i} = Q$ and $\mathsf{pid}_{Q^j} = P$.

We denote this relation by $P^i \leftrightarrow Q^j$.

Definition 2 (Partnering)**.** A session P^i is partnered if there exists exactly one session Q^j such that $P^i \leftrightarrow Q^j$. In this case, Q^j is said to be the partner of P^i. If no such session exists the session is unpartnered.

Partnering ensures that key agreement and explicit authentication are treated consistently in the security experiment.

7.3 AKE Security Experiment

The authenticated key exchange security of QSTP is defined through an experiment

$$\mathsf{Exp}^{\mathsf{AKE}}_{\mathsf{QSTP}}(\mathcal{A})$$

based on the Bellare Rogaway model. The experiment runs all session instances of client and server parties and provides $\mathcal{A}$ with the oracle interface described above. The experiment proceeds as follows.

1. The adversary adaptively activates sessions, delivers handshake messages through Send queries, and may issue Reveal or $\mathsf{Corrupt}$ queries subject to freshness constraints.
2. At any time, $\mathcal{A}$ may issue a single $\mathsf{Test}(P^i)$ query to a fresh session. The experiment flips a random bit b. If $b = 0$ it returns the true session key of P^i, and if $b = 1$ it returns a uniformly random string.
3. The adversary continues interacting with sessions through Send queries but may not issue further Reveal or $\mathsf{Corrupt}$ queries that violate freshness.
4. The adversary outputs a bit b' and succeeds if $b' = b$.

The advantage of $\mathcal{A}$ in the AKE experiment is defined as

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

QSTP achieves AKE security if this advantage is negligible for all probabilistic polynomial time adversaries.

7.4 Channel Security Experiment

The QSTP channel is an AEAD protected, sequence numbered stream. After the handshake completes and session keys are established, the adversary may continue to deliver ciphertexts through Send queries. Decryption occurs only if the ciphertext verifies under RCS and satisfies the replay and freshness constraints defined earlier.
Channel security is captured by a confidentiality and integrity experiment that runs in parallel with the AKE experiment.

Let:

$$\mathsf{Exp}^{\mathsf{CHAN}}_{\mathsf{QSTP}}(\mathcal{A})$$

be the channel experiment.

- **Confidentiality.** The adversary chooses two equal length plaintexts for a fresh session and receives an encryption of one of them under a hidden bit. The adversary may continue scheduling packets and issuing Send queries. It wins if it distinguishes which plaintext was encrypted.
- **Integrity.** The adversary succeeds if it causes a fresh session to accept a ciphertext not produced by the honest partner, or if it causes acceptance of a packet whose associated data or header has been modified.

These goals correspond to an ACCE style definition in which the handshake provides authenticated key exchange and the AEAD layer provides confidentiality and integrity of the established channel. QSTP achieves channel security if both the confidentiality and integrity advantages of $\mathcal{A}$ are negligible under the assumed hardness of the underlying primitives.

8 Primitive Security Assumptions

8.1 KEM Security

QSTP uses a lattice based key encapsulation mechanism for ephemeral key establishment. We assume the KEM satisfies IND-CCA security. The corresponding adversarial experiment defines the advantage of an adversary $\mathcal{B}$ as

$$\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{KEM}}(\mathcal{B}) = \left|\Pr[b' = b] - \frac{1}{2}\right|,$$

where b is the challenge bit used to select between a real shared secret and a random value, and b' is the guess produced by $\mathcal{B}$. The hardness of recovering the real shared secret from the challenge ciphertext is based on the computational difficulty of the underlying module lattice problem.
IND-CCA security implies that replacing the shared secret derived from a challenge ciphertext by a uniformly random element is indistinguishable to any probabilistic polynomial time adversary. This property is used in the first game transition of the QSTP security analysis.

8.2 Signature Security

QSTP authenticates server handshake messages using a post quantum signature scheme. We assume the signature scheme is existentially unforgeable under chosen message attack. For an adversary $\mathcal{B}$ making signing queries and eventually outputting a candidate forgery (m^*, σ^*), the EUF-CMA advantage is defined as:

$$\mathsf{Adv}_{\mathsf{SIG}}^{\mathsf{EUF\text{-}CMA}}(\mathcal{B}) = \Pr\Big[\mathsf{SIG.Verify}(\mathsf{vk}, m^*, \sigma^*) =$$

$$1 \ \wedge \ (m^*, \sigma^*) \text{ was not obtained from } \mathsf{SIG.Sign}\Big].$$

This assumption ensures that an adversary cannot forge the server's signature on any handshake message and therefore cannot impersonate the server or manipulate transcript values. The reduction in the QSTP proof uses this property in the final game transition, where forging an authenticated transcript is shown to imply a signature forgery.

8.3 Key Derivation and Random Oracle Assumption

The QSTP key derivation step maps the KEM shared secret and the session cookie into four independent values used as channel keys and nonces. We model the key derivation function KDF as a random oracle $\mathcal{H}$. The adversary may issue arbitrary queries to $\mathcal{H}$ during the experiment.

In the random oracle model, each distinct query x yields an independent uniformly random output $\mathcal{H}(x)$, unless the value has been defined by previous oracle programming. The random oracle assumption implies that replacing the derived channel keys in the challenge session by uniform random variables is indistinguishable to any adversary that does not query the oracle at the exact input used in that session.

This property supports the transition in which the adversary's view is decoupled from the real key derivation input $(\mathsf{sec}, \mathsf{sch}_2)$ and replaced by uniform values in the AKE analysis.

Domain Separation Across Sessions. A subtle but important property of the QSTP KDF is that the transcript hash sch_2 acts as a cSHAKE customization string, providing domain separation across sessions automatically. Even in the negligible probability event that two sessions share the same KEM shared secret sec (which would require a KEM collision), their respective values of sch_2 differ with overwhelming probability because sch_2 commits the ephemeral public key $\mathsf{pk}_{\mathsf{kem}}$ and the client ciphertext cpta, both of which are freshly and independently sampled per session. Two sessions with distinct sch_2 values produce statistically independent random oracle outputs, so the derived session keys of any two distinct sessions are independent even conditioned on one session key being known to the adversary.

Multi-Session Security. Theorem 1 is stated for a single challenge session, following the standard Bellare-Rogaway formulation. In a deployment with n_s concurrent sessions the adversary may attempt to exploit correlations across sessions. We argue that QSTP is secure in the multi-session setting with a tightness loss of at most n_s in the reduction.

The argument proceeds as follows. Each session uses an independently generated ephemeral KEM key pair, so the n_s KEM challenge distributions are independent. The adversary selects one session as the Test target. In the reduction, $\mathcal{B}_1$ must guess which session the adversary will test in order to embed its KEM challenge; the correct guess occurs with probability $1/n_s$, introducing a factor of n_s in the reduction. This gives the multi-session bound:

$$\mathsf{Adv}_{\mathsf{QSTP},\, n_s}^{\mathsf{AKE}}(\mathcal{A}) \ \le \ n_s \cdot \mathsf{Adv}_{\mathsf{KEM}}^{\mathsf{IND\text{-}CCA}}(\mathcal{B}_1) \ + \ \frac{q_{\mathcal{H}}}{|\mathcal{S}|} \ + \ n_s \cdot \mathsf{Adv}_{\mathsf{SIG}}^{\mathsf{EUF\text{-}CMA}}(\mathcal{B}_4) \ + \ \mathsf{negl}(\lambda).$$

The factor of n_s is acceptable in practice because the KEM and signature security levels are set at 256-bit post-quantum security, providing substantial headroom even for large n_s. Domain separation via sch_2 ensures that breaking one session's derived keys provides no advantage in attacking any other session's keys.

8.4 AEAD Security of RCS

QSTP uses the RCS authenticated encryption scheme for its symmetric channel. RCS is assumed to satisfy the standard AEAD security notion: confidentiality under chosen plaintext attack and ciphertext integrity under chosen ciphertext attack.

The confidentiality advantage of an adversary $\mathcal{B}$ attacking RCS is defined as

$$\mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{RCS}}(\mathcal{B}) = \left| \Pr[b' = b] - \frac{1}{2} \right|,$$

where b determines whether the adversary receives an encryption of one of two chosen plaintexts.

The integrity advantage is defined as

$$\mathsf{Adv}^{\mathsf{INT\text{-}CTXT}}_{\mathsf{RCS}}(\mathcal{B}) = \Pr\big[\mathsf{AEAD.Dec}(k, A, C, T) \neq \bot$$
$$\text{on a ciphertext not produced by } \mathsf{AEAD.Enc}(k, A, \cdot)\big].$$

In the QSTP security analysis, AEAD security ensures that an adversary cannot create a ciphertext that is accepted by the partner session unless it originates from the honest sender. It also ensures that the adversary learns nothing from observing ciphertexts encrypted under the derived channel keys. These bounds are used when transitioning from the real channel behavior to an idealized authenticated channel in the game sequence.

9 Security Theorems and Game Based Analysis

9.1 Proof Overview

The security of QSTP is established through a sequence of game transformations that gradually replace components of the protocol with idealized versions. The strategy is to show that any adversary that distinguishes the session key from random in the AKE experiment can be transformed into an adversary that breaks the underlying KEM, signature scheme, key derivation function, or RCS AEAD primitive.

The proof begins with the real experiment of QSTP, where the adversary interacts with honest parties running the actual protocol. In the first transition the KEM shared secret used in the challenge session is replaced by a uniformly random value. The second transition replaces the outputs of the key derivation function with random oracle responses. The third transition idealizes the RCS channel so that decryption is replaced by an oracle that returns $\perp$ for any forged ciphertext. The final transition simplifies the authentication logic and shows that any session key mismatch or unauthorized acceptance implies a signature forgery.

Each transition changes the adversary's success probability by at most the advantage of an adversary against one of the underlying primitives. Summing these differences yields a bound on the AKE advantage of QSTP.

9.2 Main AKE Security Theorem

Theorem 1 (AKE Security of QSTP)**.** *For any probabilistic polynomial time adversary $\mathcal{A}$ attacking the QSTP authenticated key exchange experiment,*

$$\mathsf{Adv}^{\mathsf{AKE}}_{\mathsf{QSTP}}(\mathcal{A}) \;\leq\; \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{KEM}}(\mathcal{B}_1) \;+\; \frac{q_{\mathcal{H}}}{|\mathcal{S}|} \;+\; \mathsf{Adv}^{\mathsf{AEAD}}_{\mathsf{RCS}}(\mathcal{B}_3) \;+\; \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{SIG}}(\mathcal{B}_4) \;+\; \mathsf{negl}(\lambda),$$

where λ is the security parameter, $q_{\mathcal{H}}$ is the number of random oracle queries issued by $\mathcal{A}$, $|\mathcal{S}|$ is the cardinality of the KEM shared secret space (2^{256} for ML-KEM at security level

5), and each $\mathcal{B}_i$ is an efficient adversary against the corresponding primitive constructed from $\mathcal{A}$. The term $q_{\mathcal{H}}/|\mathcal{S}|$ is negligible for any feasible $q_{\mathcal{H}}$, and the additive $q_{\mathcal{H}}^2/2^\ell$ term in Lemma 4 is negligible for $\ell = 256$. In particular, QSTP achieves key indistinguishability and explicit server authentication under the IND-CCA hardness of the KEM and the EUF-CMA security of the signature scheme, with the channel protected by the AEAD security of RCS.

9.3 Sequence of Games

Game 0 (Real Protocol). Game 0 corresponds exactly to the AKE experiment $\mathsf{Exp}^{\mathsf{AKE}}_{\mathsf{QSTP}}(\mathcal{A})$. The adversary interacts with honest parties running the full QSTP protocol, including real KEM encapsulation and decapsulation, real key derivation, and the RCS AEAD channel. The adversary selects a fresh session and issues a Test query to obtain either the real session key or a random value.

Let $\Pr[\mathsf{Game0} = 1]$ denote the probability that $\mathcal{A}$ outputs 1 in Game 0.

Game 1 (Replace KEM Shared Secret). In Game 1 we modify the challenge session as follows. When the adversary issues a Test query to a fresh session, the shared secret s obtained from the KEM is replaced by a uniformly random element s^* sampled from the KEM key space. All other behavior remains unchanged and all other sessions use their real KEM shared secrets.
The IND-CCA security of the KEM implies that the adversary cannot distinguish whether s or s^* is used in the challenge session.

Lemma 1.
$$|\Pr[\mathsf{Game0} = 1] - \Pr[\mathsf{Game1} = 1]| \le \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{KEM}}(\mathcal{B}_1).$$

Game 2 (Random Oracle Replacement of Session Keys). In Game 2 we replace the session keys of the challenge session with uniformly random and independent values. Concretely, the keys $(k_{c\to s}, n_{c\to s}, k_{s\to c}, n_{s\to c})$ used in the challenge session are sampled fresh from the uniform distribution over their respective domains rather than being derived via $\mathsf{cSHAKE}(\mathsf{sec}^*, \mathsf{sch}_2)$. All other sessions continue to use real derived keys.
The justification is a standard random oracle argument. Model the key derivation function $\mathsf{cSHAKE}(\cdot,\cdot)$ as a random oracle $\mathcal{H}$. In Game 1, the shared secret sec^* used in the challenge session is uniformly random over the KEM shared secret space $\mathcal{S}$, independent of the adversary's view and of all other protocol values. Let $q_{\mathcal{H}}$ denote the total number of queries $\mathcal{A}$ submits to $\mathcal{H}$ during the experiment. Each query is a pair $(x, y) \in \mathcal{S} \times \{0,1\}^*$. Since sec^* is uniformly random and is never given directly to $\mathcal{A}$, the probability that any single query satisfies $x = \mathsf{sec}^*$ is exactly $|\mathcal{S}|^{-1}$. By a union bound over all $q_{\mathcal{H}}$ queries,

$$\Pr[\exists \, \text{query } (x,y) : \; x = \mathsf{sec}^*] \;\le\; \frac{q_{\mathcal{H}}}{|\mathcal{S}|}.$$

If $\mathcal{A}$ does not query $\mathcal{H}$ at $(\mathsf{sec}^*, \mathsf{sch}_2)$, then the random oracle output at that point is information-theoretically independent of everything $\mathcal{A}$ has observed. Replacing the derived session keys with uniformly random values therefore changes $\mathcal{A}$'s view by a statistical distance of at most $q_{\mathcal{H}}/|\mathcal{S}|$.

Remark 1. The customization string sch_2 used in the cSHAKE call additionally provides domain separation across sessions. Even in the negligible probability event that two sessions share the same ephemeral shared secret sec, their transcript hashes sch_2 differ with overwhelming probability because sch_2 commits the ephemeral public key and KEM ciphertext, which are independently sampled. Distinct sch_2 values produce independent random oracle outputs, so session keys across distinct sessions are independent even conditioned on any single session key being known.

For ML-KEM (Kyber) at security level 5, $|\mathcal{S}| = 2^{256}$, giving a bound of $q_{\mathcal{H}}/2^{256}$, which is negligible in the security parameter for any feasible $q_{\mathcal{H}}$.

Lemma 2.

$$|\Pr[\mathsf{Game1} = 1] - \Pr[\mathsf{Game2} = 1]| \leq \frac{q_{\mathcal{H}}}{|\mathcal{S}|},$$

where $q_{\mathcal{H}}$ is the number of random oracle queries issued by $\mathcal{A}$ and $|\mathcal{S}|$ is the cardinality of the KEM shared secret space. This quantity is negligible in the security parameter λ.

Proof. Define event E as the event that $\mathcal{A}$ queries $\mathcal{H}$ at any pair (x, y) with $x = \mathsf{sec}^*$. By the union bound, $\Pr[E] \leq q_{\mathcal{H}}/|\mathcal{S}|$. Conditioned on $\neg E$, the value $\mathcal{H}(\mathsf{sec}^*, \mathsf{sch}_2)$ is never revealed to $\mathcal{A}$ and is uniformly random over the output space. The distributions of all adversary-visible values in Game 1 and Game 2 are therefore identical conditioned on $\neg E$. Consequently,

$$|\Pr[\mathsf{Game1} = 1] - \Pr[\mathsf{Game2} = 1]| \leq \Pr[E] \leq \frac{q_{\mathcal{H}}}{|\mathcal{S}|}. \qquad \square$$

Game 3 (AEAD Idealization). Game 3 replaces the behavior of the RCS AEAD channel in the challenge session with an ideal authenticated channel. Specifically, encryption is simulated without reference to the real channel key, and decryption accepts only ciphertexts that were generated by the honest partner. All forged ciphertexts immediately return $\perp$ without leaking any information.
This modification affects only adversarial forgeries or attempts to observe differences in ciphertext processing. The AEAD security of RCS bounds the adversary's distinguishing advantage.

Lemma 3.

$$|\Pr[\mathsf{Game2} = 1] - \Pr[\mathsf{Game3} = 1]| \leq \mathsf{Adv}^{\mathsf{AEAD}}_{\mathsf{RCS}}(\mathcal{B}_3).$$

Game 4 (Authentication and Partnering). In Game 4 we enforce perfect explicit authentication: any client session that completes the handshake is guaranteed to have a uniquely matching server session with an identical transcript. We show that any violation of this guarantee implies a break of the EUF-CMA security of the signature scheme.

Adversary strategies and their reductions. The adversary $\mathcal{A}$ controlling the network may attempt to break the authentication guarantee through the following distinct strategies. We analyze each in turn.

Strategy 1: Modified ephemeral public key. Suppose $\mathcal{A}$ intercepts the ConnectResponse from an honest server S^j and replaces the ephemeral encapsulation key $\mathsf{pk}_{\mathsf{kem}}$ with an adversarially chosen value pk'. The server's signed value is $\sigma = \mathsf{SIG.Sign}(\mathsf{sk}_S, \mathsf{phash})$ where $\mathsf{phash} = \mathcal{H}(\mathsf{phdr} \parallel \mathsf{sch}_0 \parallel \mathsf{pk}_{\mathsf{kem}})$. The client recomputes $\mathsf{phash}' = \mathcal{H}(\mathsf{phdr} \parallel \mathsf{sch}_0 \parallel \mathsf{pk}')$ and verifies $\mathsf{SIG.Verify}(\mathsf{vk}_S, \mathsf{phash}', \sigma)$. Since $\mathsf{pk}' \neq \mathsf{pk}_{\mathsf{kem}}$ and $\mathcal{H}$ is collision resistant (modeled as a random oracle), $\mathsf{phash}' \neq \mathsf{phash}$ except with negligible probability, so verification fails and the session aborts. For the client to accept a modified key, $\mathcal{A}$ must produce a valid signature σ' on phash' under vk_S, which constitutes an EUF-CMA forgery since phash' was never submitted to the signing oracle.

Strategy 2: Replayed ConnectResponse. Suppose $\mathcal{A}$ replays an old ConnectResponse from a prior session. The replayed message carries an old serialized packet header phdr containing a timestamp and sequence number from the earlier session. The client validates the packet header before signature verification, and the timestamp freshness check rejects any packet whose timestamp falls outside the Δ-second acceptance window. Even within the window, the sequence number in phdr is checked against the current session counter, so a packet from a different session is rejected on sequence mismatch. Replay therefore fails independently of the signature check.

Strategy 3: Transcript divergence via ciphertext substitution. Suppose $\mathcal{A}$ intercepts the client's KEM ciphertext cpta in the ExchangeRequest and delivers a different ciphertext cpta' to the server. Both parties compute $\mathsf{sch}_2 = \mathcal{H}(\mathsf{sch}_1 \parallel \mathsf{cpta})$ and $\mathsf{sch}'_2 = \mathcal{H}(\mathsf{sch}_1 \parallel \mathsf{cpta}')$ respectively, which are distinct with overwhelming probability. The server transmits its sch'_2 in the ExchangeResponse body. The client performs a constant-time comparison of the received sch'_2 against its locally computed sch_2, which fails, and the client tears down the session. The EstablishVerify step therefore detects this attack independently of the signature and KEM layers.

Strategy 4: Server impersonation. Suppose $\mathcal{A}$ attempts to impersonate an honest server S entirely, generating its own ConnectResponse without access to sk_S. The client will verify the signature σ under the certified verification key vk_S embedded in the server certificate. Producing a valid σ on any phash without access to sk_S is exactly the EUF-CMA forgery problem. A successful impersonation in this game yields an adversary $\mathcal{B}_4$ against the EUF-CMA security of the signature scheme: $\mathcal{B}_4$ uses the honest signing oracle to answer all signing queries from $\mathcal{A}$, and when $\mathcal{A}$ succeeds in impersonating the server, $\mathcal{B}_4$ extracts the forgery $(\mathsf{phash}^*, \sigma^*)$ and outputs it. The message phash^* was never queried to the signing oracle because it binds the fresh session's packet header and ephemeral key, which are independently chosen in this session.

Strategy 5: Key mismatch without visible divergence. All other paths by which the client and server could derive different session keys without triggering an abort require either a collision in the random oracle (negligible by the random oracle model) or producing a fresh valid ciphertext that decapsulates to a different shared secret under the same server private key (impossible by correctness of the KEM).

Combining all five cases, the only non-negligible path to breaking authentication and partnering in Game 3 is a forgery against the signature scheme. The construction of $\mathcal{B}_4$ from $\mathcal{A}$ is efficient: it simulates all sessions faithfully using the signing oracle, and the forgery it extracts is a fresh, never-queried message-signature pair.

Lemma 4.

$$|\Pr[\mathsf{Game3} = 1] - \Pr[\mathsf{Game4} = 1]| \ \le\ \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{SIG}}(\mathcal{B}_4) \ +\ \frac{q^2_{\mathcal{H}}}{2^\ell},$$

where ℓ is the output length of $\mathcal{H}$ in bits and the additive term accounts for random oracle collision probability across $q_{\mathcal{H}}$ queries. For $\ell = 256$ this term is negligible.

Proof. Construct $\mathcal{B}_4$ as follows. $\mathcal{B}_4$ receives a verification key vk^* and access to a signing oracle $\mathcal{O}_{\mathsf{sign}}(\cdot)$. $\mathcal{B}_4$ embeds vk^* as the target server's verification key in the simulated certificate and runs $\mathcal{A}$ in Game 3, answering all signing queries by forwarding them to $\mathcal{O}_{\mathsf{sign}}$. If $\mathcal{A}$ causes a client session to accept a ConnectResponse message that no honest server session produced (Strategy 4 above), then $\mathcal{A}$ has induced the client to accept a valid signature σ^* on a message phash^* that was never submitted to $\mathcal{O}_{\mathsf{sign}}$. $\mathcal{B}_4$ outputs $(\mathsf{phash}^*, \sigma^*)$ as its forgery. The binding of phash^* to the session's packet header and ephemeral key ensures it is distinct from all signing oracle queries except with probability bounded by the random oracle collision term. Strategies 1 through 3 and 5 either cause the session to abort or reduce to negligible probability events as argued above. □

9.4 Proof of Theorem 1

Proof. Let W_i denote the event that the adversary outputs 1 in Game i. By the triangle inequality and the lemmas above,

$$|\Pr[W_0] - \Pr[W_4]| \ \le\ \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{KEM}}(\mathcal{B}_1) \ +\ \frac{q_{\mathcal{H}}}{|\mathcal{S}|} \ +\ \mathsf{Adv}^{\mathsf{AEAD}}_{\mathsf{RCS}}(\mathcal{B}_3) \ +\ \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{SIG}}(\mathcal{B}_4) \ +\ \frac{q^2_{\mathcal{H}}}{2^\ell},$$

where the final term accounts for the random oracle collision probability introduced in Lemma 4 and is negligible for $\ell = 256$.
In Game 4 the session key used in the challenge session is uniformly random and independent of all adversarial observations, so

$$\Pr[W_4] = \frac{1}{2}.$$

Therefore,

$$\mathsf{Adv}^{\mathsf{AKE}}_{\mathsf{QSTP}}(\mathcal{A}) \;=\; \left|\Pr[W_0] - \frac{1}{2}\right| \;\le\; \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{KEM}}(\mathcal{B}_1) + \frac{q_{\mathcal{H}}}{|\mathcal{S}|} + \mathsf{Adv}^{\mathsf{AEAD}}_{\mathsf{RCS}}(\mathcal{B}_3) + \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{SIG}}(\mathcal{B}_4) + \mathsf{negl}(\lambda).$$

This completes the proof. □

9.5 Forward Secrecy

QSTP provides forward secrecy with respect to the server's long-term signature key. The key property that enables this is the ephemerality of the KEM key pair: the server generates a fresh $(\mathsf{pk}_{\mathsf{kem}}, \mathsf{sk}_{\mathsf{kem}})$ for each handshake and erases $\mathsf{sk}_{\mathsf{kem}}$ immediately upon completing decapsulation in Algorithm 4 (ExchangeResponse).
We extend the security model to capture forward secrecy by permitting the adversary to corrupt the server's long-term signing key sk_S after a target session has completed and its ephemeral key material has been erased.

Definition 3 (Post-Session Corruption)**.** A session S^j is *post-erased* if it has completed the handshake, $\mathsf{sk}_{\mathsf{kem}}$ has been overwritten, and all intermediate buffers holding sec have been cleared. A $\mathsf{Corrupt}(S)$ query issued after session S^j is post-erased is called a *post-erasure corruption.*

In the forward secrecy experiment, the adversary is permitted to issue a post-erasure corruption of the server's long-term signing key sk_S after the challenge session completes, and we ask whether it can distinguish the session key of the challenge session from a random value.

Theorem 2 (Forward Secrecy of QSTP)**.** *Let C^i be a fresh client session that has completed the handshake with server S, and let S^j be its matching partner session. Suppose both sessions are post-erased. Then for any probabilistic polynomial time adversary $\mathcal{A}$ that may corrupt sk_S after erasure and subsequently issues a $\mathsf{Test}(C^i)$ query,*

$$\mathsf{Adv}^{\mathsf{FS}}_{\mathsf{QSTP}}(\mathcal{A}) \;\le\; \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{KEM}}(\mathcal{B}_1) \;+\; \frac{q_{\mathcal{H}}}{|\mathcal{S}|} \;+\; \mathsf{negl}(\lambda).$$

Proof. We trace through the session key derivation and identify what information the adversary gains from a post-erasure corruption.

What corruption reveals. After a post-erasure corruption, the adversary obtains sk_S. This signing key was used in the challenge session only to produce the signature $\sigma = \mathsf{SIG.Sign}(\mathsf{sk}_S, \mathsf{phash})$ in the ConnectResponse, where $\mathsf{phash} = \mathcal{H}(\mathsf{phdr} \parallel \mathsf{sch}_0 \parallel \mathsf{pk}_{\mathsf{kem}})$. Knowledge of sk_S allows the adversary to forge future signatures and impersonate the server in future sessions, but it does not reveal $\mathsf{sk}_{\mathsf{kem}}$, which is an independently generated ephemeral key. The ephemeral secret key $\mathsf{sk}_{\mathsf{kem}}$ was used once, during decapsulation to recover sec from cpta, and was then erased from memory.

What the adversary must recover. The session key is derived as

$$(k_{c\to s}, n_{c\to s}, k_{s\to c}, n_{s\to c}) \;\leftarrow\; \mathcal{H}(\mathsf{sec}, \mathsf{sch}_2).$$

The adversary observes cpta (transmitted in clear over the network), sch_2 (transmitted in the ExchangeResponse), and after corruption, sk_S. To compute the session key, it must determine sec.

Reduction to KEM IND-CCA. The adversary's task is to recover sec from the ciphertext cpta under the public key $\mathsf{pk}_{\mathsf{kem}}$, without access to the corresponding $\mathsf{sk}_{\mathsf{kem}}$ (which has been erased). This is precisely the KEM decapsulation problem. Under IND-CCA security of the KEM, no probabilistic polynomial time adversary can distinguish sec from a uniformly random element of $\mathcal{S}$ given only $(\mathsf{pk}_{\mathsf{kem}}, \mathsf{cpta})$. We construct $\mathcal{B}_1$ against the KEM as follows: $\mathcal{B}_1$ receives a challenge public key pk^* and challenge ciphertext cpta^* from the KEM IND-CCA experiment and embeds them as the server's ephemeral key and the client's ciphertext in the challenge session. It runs $\mathcal{A}$ in the forward secrecy experiment, simulating all other sessions faithfully using independent KEM key pairs. The long-term signing key sk_S is generated and held by $\mathcal{B}_1$, so the post-erasure corruption query is answered directly. When $\mathcal{A}$ issues the Test query, $\mathcal{B}_1$ uses the KEM challenge bit b to decide whether to return the real session key or a random value. $\mathcal{A}$'s distinguishing advantage in the forward secrecy experiment is directly transferred to $\mathcal{B}_1$'s advantage in the KEM experiment.

Random oracle term. If $\mathcal{A}$ happens to query $\mathcal{H}$ at $(\mathsf{sec}^*, \mathsf{sch}_2)$ before the Test query, it learns the session key directly. By the same argument as Lemma 2, this occurs with probability at most $q_{\mathcal{H}}/|\mathcal{S}|$.

Combining both cases,

$$\mathsf{Adv}^{\mathsf{FS}}_{\mathsf{QSTP}}(\mathcal{A}) \;\le\; \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{KEM}}(\mathcal{B}_1) \;+\; \frac{q_{\mathcal{H}}}{|\mathcal{S}|}. \qquad \square$$

Remark 2 (Scope of Forward Secrecy). Theorem 2 establishes forward secrecy against corruption of the server's *signature* key. QSTP does not use a long-term KEM key for the channel: the KEM key pair is freshly generated per session and erased immediately. A corruption of a long-term KEM key (were one to exist) would therefore provide no advantage either, since no such key is retained. This distinguishes QSTP from protocols such as RSA-based TLS in which the server's static private key is used directly for key transport; in QSTP the static signing key is used only for authentication of the ephemeral key, not for derivation of the session secret.

Remark 3 (Post-Compromise Security). Forward secrecy protects past sessions against future key compromise. The converse property, post-compromise security (PCS), would protect future sessions against past compromise. QSTP does not provide PCS in the base protocol because the server's long-term signing key, once compromised, allows an adversary to impersonate the server in all future handshakes. The optional symmetric ratchet mechanism provides forward secrecy across session epochs (Theorem 4) but does not provide PCS in the forward direction. An adversary who compromises the current session state can decrypt the ratchet token transmitted over the channel and propagate the ratchet key chain forward to all future epochs indefinitely. Post-compromise security in the forward direction requires injection of computationally hidden entropy via a fresh asymmetric operation, which is outside the scope of the current QSTP specification. Deployments requiring PCS should enforce periodic full re-handshakes.

9.6 Explicit Key Confirmation

The QSTP handshake provides *explicit key confirmation*: at the conclusion of the exchange, the client is not merely assured that it shares a key with an authenticated server, but has positive evidence that the server has derived an *identical* key from an *identical* transcript.

Definition 4 (Explicit Key Confirmation)**.** A session C^i achieves explicit key confirmation if its session key acceptance is conditioned on receipt of a value that is derivable only by a party holding the correct KEM shared secret and having observed the same handshake transcript.

In QSTP this is realized by the EstablishVerify step. The server transmits sch_2 in the ExchangeResponse body. Since $\mathsf{sch}_2 = \mathcal{H}(\mathsf{sch}_1 \parallel \mathsf{cpta})$ and both sch_1 and cpta are determined by the preceding messages, a server that has processed a different cpta (due to adversarial substitution, as in Strategy 3 of Lemma 4) will transmit a different sch_2'. The client's constant-time comparison therefore detects any transcript divergence before the session is declared established.

Corollary 1 (Explicit Key Confirmation)**.** *In any execution of QSTP in which the client session transitions to the* $\mathsf{session\_established}$ *state, the following hold simultaneously with probability* $1 - \mathsf{negl}(\lambda)$*:*

1. *There exists a unique matching server session* S^j *with an identical transcript* $(\mathsf{sch}_0, \mathsf{sch}_1, \mathsf{sch}_2)$*.*
2. *Both sessions have derived identical session keys from identical KDF inputs* $(\mathsf{sec}, \mathsf{sch}_2)$*.*
3. *The server has performed decapsulation and successfully recovered* sec *prior to transmitting* sch_2*.*

Proof. Point 1 follows from Lemma 4: any session that accepts has a matching server partner except with probability bounded by the EUF-CMA advantage and the random oracle collision term. Point 2 follows because both parties use the same deterministic KDF on the same inputs; since their transcripts are identical by point 1 and sec is uniquely determined by the ciphertext under the correct decapsulation key, their derived keys are identical. Point 3 follows because the server computes sch_2 only after decapsulation succeeds; a server whose decapsulation failed would have transmitted an error and the session would have aborted before EstablishVerify. □

Remark 4. Explicit key confirmation is a stronger property than implicit key confirmation (in which each party merely derives the same key under correct conditions but does not verify that its partner has done so). Without the EstablishVerify comparison, a client could declare a session established while its partner has derived a different key due to transcript divergence caused by an active network adversary. The EstablishVerify step closes this gap at the cost of one additional message in the exchange response, which carries no keying material and is protected only by the AEAD layer.

9.7 Channel Confidentiality and Integrity

The channel security of QSTP follows from the combination of authenticated key exchange and the AEAD security of RCS. The handshake produces independent keys for each direction, and the idealization in Game 3 ensures that forged ciphertexts are rejected.

Theorem 3 (Channel Security of QSTP)**.** *For any probabilistic polynomial time adversary* $\mathcal{A}$ *attacking the QSTP channel experiment* $\mathsf{Exp}^{\mathsf{CHAN}}_{\mathsf{QSTP}}(\mathcal{A})$ *after a successful handshake,*

$$\mathsf{Adv}^{\mathsf{CHAN}}_{\mathsf{QSTP}}(\mathcal{A}) \;\leq\; \mathsf{Adv}^{\mathsf{AKE}}_{\mathsf{QSTP}}(\mathcal{B}_5) \;+\; 2 \cdot \mathsf{Adv}^{\mathsf{AEAD}}_{\mathsf{RCS}}(\mathcal{B}_6) \;+\; \mathsf{negl}(\lambda).$$

Proof. We address confidentiality and integrity separately and then combine.

Confidentiality. The channel confidentiality experiment asks whether $\mathcal{A}$ can distinguish an encryption of plaintext P_0 from an encryption of P_1 under the session key. We construct $\mathcal{B}_5$ against AKE security as follows. $\mathcal{B}_5$ runs the AKE experiment and, when $\mathcal{A}$ submits the

challenge plaintexts (P_0, P_1), queries Test on the challenge session. If Test returns the real session key, $\mathcal{B}_5$ constructs a real RCS encryption; if it returns a random key, $\mathcal{B}_5$ encrypts under the random key. In the case where $\mathcal{B}_5$ received the real key, $\mathcal{A}$'s distinguishing advantage directly translates to $\mathcal{B}_5$'s AKE advantage.
If the session key is already uniformly random from $\mathcal{A}$'s view (which occurs in the AKE game after Game 2), we construct $\mathcal{B}_6$ against RCS IND-CPA as follows. $\mathcal{B}_6$ receives an RCS challenge ciphertext and delivers it to $\mathcal{A}$ as the encrypted channel packet. $\mathcal{A}$'s ability to distinguish plaintext is directly $\mathcal{B}_6$'s IND-CPA advantage against RCS. Combining both reductions,

$$\mathsf{Adv}^{\mathsf{CHAN\text{-}CONF}}_{\mathsf{QSTP}}(\mathcal{A}) \;\le\; \mathsf{Adv}^{\mathsf{AKE}}_{\mathsf{QSTP}}(\mathcal{B}_5) \;+\; \mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{RCS}}(\mathcal{B}_6).$$

Integrity. The channel integrity experiment asks whether $\mathcal{A}$ can cause a session to accept a ciphertext not produced by the honest partner, or accept a packet with modified associated data. By Corollary 1, any session that has completed the handshake has a unique matching partner with an identical session key. Suppose $\mathcal{A}$ delivers a packet (H^*, C^*, T^*) to an honest session, where (H^*, C^*, T^*) was not produced by the matching partner. For the session to accept, RCS must verify the tag T^* under the derived session key with associated data H^*. We construct $\mathcal{B}'_6$ against RCS INT-CTXT as follows. $\mathcal{B}'_6$ simulates the QSTP execution, derives the session key using the AKE subroutine, initializes an RCS instance with that key, and forwards all honest encryption queries to the RCS encryption oracle. When $\mathcal{A}$ produces a forgery (H^*, C^*, T^*) that is accepted, $\mathcal{B}'_6$ outputs this as its INT-CTXT forgery. The associated data bound to the header ensures that modifying any header field (sequence number, timestamp, flag) also constitutes a forgery. Therefore,

$$\mathsf{Adv}^{\mathsf{CHAN\text{-}INT}}_{\mathsf{QSTP}}(\mathcal{A}) \;\le\; \mathsf{Adv}^{\mathsf{AKE}}_{\mathsf{QSTP}}(\mathcal{B}_5) \;+\; \mathsf{Adv}^{\mathsf{INT\text{-}CTXT}}_{\mathsf{RCS}}(\mathcal{B}'_6).$$

Combination. The channel advantage is the sum of confidentiality and integrity advantages. Since RCS satisfies both IND-CPA and INT-CTXT simultaneously under the same key (standard AEAD security), we may bound both by a single adversary $\mathcal{B}_6$ against the combined AEAD notion, paying a factor of 2. This gives

$$\mathsf{Adv}^{\mathsf{CHAN}}_{\mathsf{QSTP}}(\mathcal{A}) \;\le\; \mathsf{Adv}^{\mathsf{AKE}}_{\mathsf{QSTP}}(\mathcal{B}_5) \;+\; 2 \cdot \mathsf{Adv}^{\mathsf{AEAD}}_{\mathsf{RCS}}(\mathcal{B}_6) \;+\; \mathsf{negl}(\lambda),$$

completing the proof. □

10 Cryptanalytic Evaluation of QSTP Design Choices

10.1 Server Only Authentication

QSTP provides unilateral authentication: only the server presents a certified public key and proves possession of its corresponding signature key during the handshake. This design reflects scenarios in which clients are devices or applications without long term identity keys, and where authentication of the server endpoint is the primary requirement.
The security model captures this by requiring explicit authentication only for client initiated sessions. An adversary cannot impersonate the server without forging a signature or violating the certificate chain. However, QSTP does not prevent an adversary from impersonating a client, nor does it attempt to bind a client identity to the session key. Client authentication is therefore outside the scope of the AKE security theorem.
Deployments that require mutual authentication can add a client authentication layer above QSTP. Because the QSTP channel is integrity protected, such an additional layer can safely transport client credentials or client specific signatures. This separation keeps the core protocol simple while allowing stronger authentication when needed.

10.2 Replay and Freshness Window

QSTP uses two complementary mechanisms to enforce freshness of application data packets: a monotonically increasing sequence number and a timestamp included within a fixed acceptance window Δ. The sequence number ensures strict ordering and prevents reuse of previously accepted packets. The timestamp provides an additional coarse filter, allowing the receiver to reject packets that are excessively delayed or originate from an earlier execution.

The security model incorporates the sequence number directly, since it appears in the associated data for RCS and is authenticated as part of the ciphertext. Any modification to the sequence number causes the AEAD tag to fail and the packet to be discarded. Timestamps, however, rely on the correctness of local clocks. While they protect against long term replay attempts, they are susceptible to clock skew or adversarial manipulation of time sources.

The cryptanalytic implication is that QSTP's replay protection is sound within a bounded drift model. If an adversary can significantly skew the receiver's notion of time, the acceptance window may widen or narrow unpredictably. Implementations should therefore combine timestamp checks with stable monotonic counters or limit reliance on timestamps to coarse window enforcement. The formal model treats timestamp rejection as a local policy decision that complements but does not replace the cryptographic guarantees of the protocol.

10.3 Denial of Service Considerations

A key concern for post quantum protocols is the computational cost of key encapsulation and decapsulation. QSTP reduces this surface by requiring the server to perform only signature verification and configuration checks before decapsulating the client's KEM ciphertext. These operations are significantly cheaper than decapsulation, limiting the worst case cost that can be imposed by unauthenticated clients.

However, a determined adversary can still send a large number of apparently valid ExchangeRequest messages that force the server to perform KEM decapsulation. To mitigate this, implementations may introduce a lightweight stateless mechanism, such as a retry token or cookie, that must be validated by the server before performing any heavy KEM operations. The protocol description allows such mechanisms as implementation level defenses without altering the formal handshake.

The security model in this paper does not quantify resource usage but assumes that servers enforce reasonable limits on the number of active incomplete sessions. The cryptanalytic evaluation highlights that such measures are recommended to preserve availability in adversarial environments.

10.4 RCS AEAD as Channel Primitive

RCS is used as the symmetric channel primitive in QSTP due to its clear security proof, simple structure, and efficient implementation. RCS treats the packet header, including sequence number and timestamp, as associated data, ensuring that any modification to these fields results in rejection before decryption. This property is central to QSTP's replay and ordering guarantees.

The choice of RCS aligns with the design goals of QSTP. Its deterministic layout, small state footprint, and constant time operations reduce side channel exposure. Its security analysis establishes indistinguishability under chosen plaintext attack and ciphertext integrity under chosen ciphertext attack, which are precisely the properties required for a post handshake encrypted channel.

Using RCS also simplifies the reduction proofs, since its behavior matches the idealized authenticated channel used in the game-based analysis. The formal security of QSTP therefore relies directly on the proven AEAD security of RCS rather than on complex or multi-variant AEAD constructions.

AES-256-GCM as an Alternative Symmetric Primitive. The QSTP specification provides AES-256-GCM as a compile-time alternative to RCS, selected by leaving the `QSTP_USE_RCS_ENCRYPTION` flag undefined. The security theorems in this paper (Theorems 1, 2, and 3) are stated in terms of a generic AEAD primitive. They apply equally to the AES-256-GCM configuration with the following substitution: replace each occurrence of $\mathsf{Adv}^{\mathsf{AEAD}}_{\mathsf{RCS}}$ with $\mathsf{Adv}^{\mathsf{AEAD}}_{\mathsf{AES\text{-}GCM}}$, where the latter denotes the standard AEAD advantage against AES-256-GCM as specified in NIST SP 800-38D.
Three practical differences between the two configurations are worth noting for security evaluation purposes.

- **Nonce size and collision bound.** AES-256-GCM uses a 96-bit nonce, giving a birthday bound collision threshold of approximately 2^{48} encrypted packets under a single key before the confidentiality guarantee degrades. RCS uses a 256-bit nonce, making nonce collision negligible for any feasible session lifetime. Deployments using AES-256-GCM should enforce session rekeying or connection limits well below 2^{32} packets per session key to remain comfortably within the security margin.
- **Post-quantum symmetric security.** Both AES-256 and RCS operate on 256-bit keys. Grover's algorithm reduces the effective security of a 256-bit key to 128 bits against a quantum adversary. The post-quantum security of the symmetric channel therefore rests on 128-bit quantum security for either cipher, which is considered sufficient under current NIST post-quantum guidelines. RCS offers no material advantage over AES-256-GCM in this respect.
- **Standards compliance.** AES-256-GCM is the appropriate choice for deployments that require conformance to NIST SP 800-38D or interoperability with existing GCM-based infrastructure. The security reduction for the AES-GCM configuration is cleaner in the sense that AES-256-GCM has a longer public cryptanalytic record than RCS. Deployments prioritizing conservative standardized cryptography should prefer the AES-256-GCM configuration; deployments prioritizing post-quantum symmetric security margins and performance should prefer RCS.

Both cipher options use an identical key derivation path through $\mathsf{cSHAKE}(\mathsf{sec}, \mathsf{sch}_2)$ and produce output of the same length, so the handshake security theorems are unaffected by this choice. The selection affects only the channel phase security and is fully captured by the generic AEAD term in the channel theorem.

11 Symmetric Ratchet and Post-Session Forward Secrecy

11.1 Overview

QSTP defines one optional key update mechanism: the symmetric ratchet. This mechanism operates over an established encrypted channel and provides forward secrecy across session epochs — compromise of the current session state does not expose keying material from prior ratchet epochs. The mechanism is optional and is signaled by the dedicated packet flag `qstp_flag_symmetric_ratchet_request`.

The base handshake (Theorem 2) establishes forward secrecy with respect to the server's long-term signing key. The symmetric ratchet extends this guarantee to provide forward secrecy with respect to the symmetric session state itself: an adversary who compromises the live session keys at epoch t gains no information about keys from epochs $t' < t$.

11.2 Mechanism

Let epoch $t = 0$ denote the state immediately after the base handshake completes. At epoch $t = 0$ the session state includes:

- session keys $(k^{(0)}_{c\to s}, n^{(0)}_{c\to s}, k^{(0)}_{s\to c}, n^{(0)}_{s\to c})$ derived from the handshake KDF,
- a persistent ratchet key $\mathsf{rk}^{(0)}$ initialized as a hash of the full KDF output block from the handshake: $\mathsf{rk}^{(0)} = \mathcal{H}(\mathsf{prnd})$.

A symmetric ratchet step at epoch t proceeds as follows.

1. The initiating party samples a fresh random token $\tau^{(t)} \xleftarrow{\$} \{0,1\}^n$, where $n = 256$ bits in the QSTP instantiation.
2. The initiator encrypts and transmits $\tau^{(t)}$ to the peer over the established channel, authenticated under the current session keys.
3. Upon receiving and decrypting $\tau^{(t)}$, both parties advance the ratchet key:
$$\mathsf{rk}^{(t+1)} \;=\; \mathcal{H}\Big(\mathsf{rk}^{(t)} \parallel \tau^{(t)}\Big) .$$
4. Both parties derive new session keys from the updated ratchet key:
$$\big(k^{(t+1)}_{c\to s}, n^{(t+1)}_{c\to s}, k^{(t+1)}_{s\to c}, n^{(t+1)}_{s\to c}\big) \;\leftarrow\; \mathsf{cSHAKE}\Big(\mathsf{rk}^{(t+1)}, \mathsf{sch}_2\Big) .$$
5. The cipher instances for both directions are re-initialized with the new keys and nonces. All previous session keys and the token $\tau^{(t)}$ are securely erased from memory.

The ratchet key chain is therefore:

$$\mathsf{rk}^{(0)} \xrightarrow{\tau^{(0)}} \mathsf{rk}^{(1)} \xrightarrow{\tau^{(1)}} \mathsf{rk}^{(2)} \xrightarrow{\tau^{(2)}} \cdots$$

where each arrow denotes the application of $\mathcal{H}(\cdot \parallel \tau^{(t)})$.

11.3 Security Model for the Ratchet

Definition 5 (Ratchet Epoch State)**.** The ratchet state at epoch t is the tuple

$$\Sigma^{(t)} \;=\; \big(\mathsf{rk}^{(t)}, k^{(t)}_{c\to s}, k^{(t)}_{s\to c}\big).$$

An adversary that *compromises* epoch t is one that obtains $\Sigma^{(t)}$ in full, either through a $\mathsf{Corrupt}$ query or by any other means.

We extend the channel security experiment to include epoch-indexed $\mathsf{Corrupt}$ queries. The adversary may issue $\mathsf{Corrupt}(t)$ to obtain $\Sigma^{(t)}$ for any epoch t it chooses. A session is *epoch-fresh* at epoch t' if no $\mathsf{Corrupt}(t)$ query was issued for any $t \geq t'$. The ratchet forward secrecy experiment asks whether the adversary can distinguish the session keys of an epoch-fresh epoch from random, given corruption of a later epoch.

11.4 Forward Secrecy of the Symmetric Ratchet

Theorem 4 (Forward Secrecy of the Symmetric Ratchet)**.** *Let $\mathcal{A}$ be a probabilistic polynomial time adversary that compromises the ratchet state at epoch t and attempts to distinguish the session keys of any prior epoch $t' < t$ from random. Let $\delta = t - t'$ be the epoch gap. Then*

$$\mathsf{Adv}^{\mathsf{FS\text{-}RAT}}_{\mathsf{QSTP}}(\mathcal{A}) \;\leq\; \delta \cdot \mathsf{Adv}^{\mathsf{OW}}_{\mathcal{H}}(\mathcal{B}) \;+\; \mathsf{negl}(\lambda),$$

where $\mathsf{Adv}^{\mathsf{OW}}_{\mathcal{H}}(\mathcal{B})$ denotes the one-wayness advantage of an efficient adversary $\mathcal{B}$ against $\mathcal{H}$ on inputs of the relevant length.

Proof. We proceed by induction on $\delta = t - t'$.

Base case ($\delta = 1$). The adversary holds $\Sigma^{(t)}$ and wishes to distinguish the session keys at epoch $t-1$ from random. Session keys at epoch $t-1$ are derived deterministically from $\mathsf{rk}^{(t-1)}$, so it suffices to show that $\mathsf{rk}^{(t-1)}$ is computationally hidden given $\mathsf{rk}^{(t)}$.
Since $\mathsf{rk}^{(t)} = \mathcal{H}(\mathsf{rk}^{(t-1)} \,\|\, \tau^{(t-1)})$, recovering $\mathsf{rk}^{(t-1)}$ requires inverting $\mathcal{H}$ at the output $\mathsf{rk}^{(t)}$ over inputs of the form $\mathsf{rk}^{(t-1)} \,\|\, \tau^{(t-1)}$.
We construct $\mathcal{B}$ as follows. Given a one-wayness challenge image $y = \mathcal{H}(x^*)$, $\mathcal{B}$ embeds y as $\mathsf{rk}^{(t)}$ in the simulation and runs $\mathcal{A}$. $\mathcal{B}$ simulates all channel traffic for epoch t consistently using $\Sigma^{(t)}$ derived from y. If $\mathcal{A}$ successfully recovers $\mathsf{rk}^{(t-1)}$, then $\mathcal{B}$ has found a preimage of y under $\mathcal{H}$. The probability that $\mathcal{A}$ succeeds in the base case is therefore at most $\mathsf{Adv}^{\mathsf{OW}}_{\mathcal{H}}(\mathcal{B})$.
Note that $\tau^{(t-1)}$ is erased after the ratchet step and was encrypted under $k^{(t-1)}$, which $\mathcal{A}$ does not hold, so $\mathcal{A}$ cannot recover $\tau^{(t-1)}$ to assist in the inversion.

Inductive step. Assume the theorem holds for gap $\delta - 1$, i.e., recovering keys at epoch $t - (\delta - 1)$ from $\Sigma^{(t)}$ has advantage at most $(\delta - 1) \cdot \mathsf{Adv}^{\mathsf{OW}}_{\mathcal{H}}(\mathcal{B})$.
To recover keys at epoch $t - \delta$, $\mathcal{A}$ must first recover $\mathsf{rk}^{(t-\delta+1)}$ (which by the inductive hypothesis requires inverting $\mathcal{H}$ with advantage at most $(\delta - 1) \cdot \mathsf{Adv}^{\mathsf{OW}}_{\mathcal{H}}(\mathcal{B})$) and then invert one further hash step to reach $\mathsf{rk}^{(t-\delta)}$ (cost $\mathsf{Adv}^{\mathsf{OW}}_{\mathcal{H}}(\mathcal{B})$ by the base case). By the union bound, the total advantage is at most $\delta \cdot \mathsf{Adv}^{\mathsf{OW}}_{\mathcal{H}}(\mathcal{B})$.

Session keys at epoch t' are derived deterministically from $\mathsf{rk}^{(t')}$ via cSHAKE, so computational indistinguishability of $\mathsf{rk}^{(t')}$ implies computational indistinguishability of the session keys. □

Remark 5 (Concrete Bound)*.* For SHA3-256 instantiating $\mathcal{H}$, the one-wayness advantage of any feasible adversary is at most $q/2^{256}$ where q is the number of hash queries, by a standard preimage resistance argument in the random oracle model. Even for $\delta = 2^{24}$ ratchet epochs (the maximum ratchet interval referenced in the specification), the bound $\delta/2^{256}$ is negligible.

11.5 Limitations of the Symmetric Ratchet

Remark 6 (No Post-Compromise Security)*.* The symmetric ratchet provides forward secrecy only in the backward direction. It does *not* provide post-compromise security (PCS) in the forward direction.
An adversary who compromises $\Sigma^{(t)}$ and subsequently observes the encrypted channel can decrypt the ratchet token $\tau^{(t)}$ from the ciphertext (since they hold $k^{(t)}$), compute $\mathsf{rk}^{(t+1)} = \mathcal{H}(\mathsf{rk}^{(t)} \,\|\, \tau^{(t)})$, and propagate forward to derive all future epoch keys indefinitely.
This is a fundamental limitation of purely symmetric ratchet constructions: the token $\tau^{(t)}$ introduces fresh randomness from the initiator's perspective, but an adversary who holds the decryption key for the channel carrying $\tau^{(t)}$ observes it in the clear and can replicate the entire ratchet computation. PCS requires injection of entropy that is computationally

hidden from the compromised adversary, which in turn requires a fresh asymmetric operation such as a new KEM encapsulation over a public key the adversary has not seen the corresponding secret for.
Deployments requiring PCS should limit session lifetimes, enforce periodic full re-handshakes, or extend the protocol with an asymmetric re-keying mechanism in a future revision.

Remark 7 (Composition with Base Handshake Forward Secrecy). Theorem 2 establishes that the base session key is forward secret with respect to corruption of the server's long-term signing key. Theorem 4 establishes that ratchet epoch keys are forward secret with respect to corruption of later epoch symmetric state. These two guarantees are orthogonal and compose: an adversary who corrupts both the long-term signing key (after session completion) and the epoch t symmetric state learns neither the base session keys (by Theorem 2) nor any epoch $t' < t$ keys (by Theorem 4).

12 Comparative Analysis

12.1 Comparison with TLS 1.3

TLS 1.3 is the most widely deployed authenticated key exchange protocol and forms the basis of many secure channel frameworks. It provides a flexible negotiation framework, multiple authentication modes, session resumption, and hybrid post quantum key exchange. These features support a wide range of deployment environments but introduce structural complexity that is not present in QSTP.

- **Handshake RTT and message count.** TLS 1.3 uses a two round trip handshake. QSTP also completes its five message exchange across two round trips.
- **State machine and negotiation.** TLS 1.3 includes numerous branches, including cipher suite negotiation, certificate selection, optional client authentication, and session resumption. These introduce potential downgrade vectors and complicate formal reasoning. QSTP eliminates negotiation entirely and uses a fixed cryptographic suite, making the transcript deterministic and easier to analyze.
- **Post quantum readiness.** TLS 1.3 supports hybrid post quantum key exchange but does so through extensions layered on top of a classical protocol. QSTP uses a post quantum KEM and signature scheme natively, without fallback paths or hybrid composition. This simplifies the modeling but reduces flexibility.

As a result, QSTP should not be viewed as a general purpose replacement for TLS 1.3. Instead, it is a more specialized protocol intended for deployments where negotiation and flexibility are not required and where simplicity and predictability are paramount.

12.2 Comparison with WireGuard and Noise Based Tunnels

WireGuard is a streamlined virtual private network protocol based on the Noise framework. Like QSTP, it prioritizes minimalism and predictable state transitions. However, the Noise framework allows re-keying, optional messages, and multiple handshake patterns, whereas QSTP defines a single handshake structure without alternatives.

- **Simplicity of state machine.** WireGuard relies on the Noise IK pattern, which includes identity hiding and symmetric chaining. QSTP instead uses a fixed sequence of signed messages and a KEM for ephemeral key exchange. QSTP's state machine has fewer branches and is fully deterministic.

- **Key update and re-key mechanisms.** WireGuard performs periodic re-keying and supports continuous key updates with post-compromise security properties derived from its use of Diffie-Hellman ratcheting. QSTP provides an optional symmetric ratchet that advances session keys at configurable intervals, offering backward forward secrecy: knowledge of the current epoch state reveals nothing about prior epoch keys. However, the QSTP symmetric ratchet does not provide post-compromise security in the forward direction, since a compromised party can observe the ratchet token and propagate the key chain forward. WireGuard's re-keying mechanism, by contrast, incorporates fresh Diffie-Hellman contributions that an adversary holding the current state cannot predict, providing stronger recovery properties. Long-lived QSTP deployments requiring forward-direction key recovery guarantees should enforce periodic full re-handshakes rather than relying solely on the symmetric ratchet.

- **Replay handling and DoS considerations.** WireGuard uses a sliding window bitmap for replay protection and employs stateless cookies to mitigate denial of service. QSTP uses sequence numbers combined with a freshness window and may incorporate implementation level retry mechanisms. Both approaches resist replay and reordering, but QSTP depends on local clock stability for timestamp checks.

QSTP and WireGuard therefore share an emphasis on simplicity and performance but differ significantly in key update behavior, authentication goals, and replay handling strategies.

12.3 Summary of Differences

QSTP occupies a design point between highly flexible protocols such as TLS 1.3 and minimalist tunnels such as WireGuard. Its distinguishing characteristics are:

- **Deterministic state machine.** QSTP defines a single handshake path with no negotiation. This greatly simplifies partnering and transcript analysis.

- **Native post quantum construction.** All primitives in QSTP are post quantum secure, removing the need for hybrid modes or multi variant negotiation.

- **Unilateral authentication.** Unlike TLS 1.3 or WireGuard, QSTP authenticates only the server by default. This matches controlled deployment environments but is not intended as a general purpose VPN mechanism.

- **Optional symmetric ratchet; no post-compromise re-keying.** QSTP provides an optional symmetric ratchet for advancing session keys without a full re-handshake. This mechanism provides forward secrecy across ratchet epochs (Theorem 4) but does not provide post-compromise security in the forward direction. For sessions requiring PCS or very long lifetimes, periodic full re-handshakes are recommended.

- **Replay and freshness rules.** QSTP uses a combination of sequence numbers and timestamps. This is simpler than Noise window management but relies on the accuracy of system clocks.

Overall, QSTP provides a simple and analyzable post quantum tunnel suitable for deployments where configuration is fixed and server authentication is sufficient, while protocols like TLS 1.3 and WireGuard remain preferable when flexibility, mutual authentication, or frequent re-keying are required.

13 Implementation and Engineering Considerations

13.1 Alignment with Specification and Code

The formal model presented in this paper was constructed to match the QSTP specification and the behavior of the reference implementation. The message formats, transcript structure, and session cookie computation follow the layout defined in the implementation header files. The state transitions in the handshake reflect the same ordering and error handling logic used in the C code, including verification of configuration values, signature checks, and transcript consistency.

The derivation of channel keys and nonces uses exactly the same inputs as the implementation: the KEM shared secret and the certificate bound session cookie. The authenticated encryption layer models the RCS usage precisely, including the placement of the sequence number and timestamp within the associated data. The model also incorporates the implementation's rule that packets with invalid tags or out of range timestamps are rejected before any decryption is performed.

A small number of engineering details that do not affect the security analysis are abstracted away in the formal model. These include buffer handling, explicit error codes, and session teardown policies. The model captures only the cryptographic semantics required for the reduction without imposing constraints on error reporting or diagnostic behavior.

13.2 Side Channel and Constant Time Properties

The security model assumes that the implementation of QSTP and its underlying primitives does not leak secret information through timing, branching, or memory access patterns. This assumption is typical for protocol analyses but must be addressed explicitly at the implementation level.

The reference implementation performs signature checks, KEM decapsulation, and RCS operations in constant time with respect to secret data. In particular:

- RCS encryption and decryption execute in constant time for all valid input lengths, and tag verification is performed without early exit behavior.
- The KEM decapsulation routine does not branch on values derived from secret material and ensures that decapsulation failure does not leak information beyond the standard IND-CCA interface.
- Sequence number and timestamp comparisons are performed on public values and therefore do not affect the security of the symmetric key material.
- Certificate parsing and signature verification operate on public inputs and do not require constant time behavior for correctness of the cryptographic model.

To preserve the assumptions used in the formal model, implementations should ensure that all operations on secret keys and derived session keys follow constant time programming practices. This includes avoiding data dependent branches, preventing instruction level leakage, and ensuring that buffers holding sensitive material are cleared when sessions terminate.

These engineering considerations strengthen the alignment between the theoretical analysis and deployed behavior of the protocol.

14 Limitations and Future Work

The analysis presented in this paper establishes authenticated key exchange and channel security for QSTP under standard post quantum hardness assumptions. While the security model captures the core cryptographic behavior of the protocol, several limitations remain that define the scope of the results and suggest directions for future work.

Absence of Client Authentication. QSTP provides unilateral authentication only. The formal model does not consider client credentials or client specific identity binding. In deployments requiring mutual authentication, an additional layer would need to be composed above the QSTP channel, and a complete analysis of such a composition is left for future work.

Idealized Random Oracle. The key derivation function is modeled as a random oracle, which simplifies the analysis and allows clean reductions. A proof that avoids the random oracle model, or that analyzes cSHAKE concretely as a pseudo-random function in the standard model, would strengthen the theoretical guarantees. In particular, the use of sch_2 as the cSHAKE customization string rather than as a concatenated input means that the domain separation argument relies on the specifics of the cSHAKE construction rather than on generic PRF properties; a standard model treatment would need to account for this carefully.

Formal Machine Verification. The handshake structure of QSTP is well suited to verification in the Tamarin prover. The linear state machine, absence of negotiation branches, and fixed transcript structure reduce the lemma complexity significantly compared to protocols such as TLS 1.3. The three core properties that a Tamarin model should verify are:

1. **Auth_S**: If a client session accepts, the server has sent the corresponding ConnectResponse (server authentication).
2. **SK_Secrecy**: The session key k is not derivable by the adversary unless the KEM or signature scheme is broken.
3. **Forward_Secrecy**: After erasure of the ephemeral KEM secret key, corruption of the long-term signing key does not yield the session key.

Developing and publishing a complete Tamarin model that verifies these three lemmas would provide machine-checked evidence complementing the game-based reductions in this paper, and is a concrete direction for future work. Any such model should instantiate the KEM and signature scheme as abstract function symbols with appropriate axioms for correctness and security, and should model the transcript hash chain $(\mathsf{sch}_0, \mathsf{sch}_1, \mathsf{sch}_2)$ explicitly as persistent facts to capture the iterative binding property.

Restricted Corruption Model. The model does not allow corruption of the root authority, and server key compromise is limited to explicit adversarial queries. The symmetric ratchet is analyzed in Section 11 within an extended corruption model that permits epoch-indexed compromise queries. A richer model could extend this analysis to cover composition with the base handshake forward secrecy guarantee under a unified corruption oracle, and could quantify the interaction between long-term key compromise and ratchet epoch compromise more precisely. Post-compromise security in the forward direction is not achieved by the symmetric ratchet and would require a future protocol extension incorporating a fresh asymmetric operation per re-keying epoch.

Replay and Freshness Assumptions. Replay detection relies on a combination of sequence numbers and a coarse timestamp window. While captured in the model, this approach depends on correct timekeeping and bounded clock drift. A more robust analysis could consider alternative replay mechanisms based solely on monotonic counters.

Denial of Service Analysis. The treatment of denial of service is qualitative. Although the protocol structure limits costly operations on unauthenticated traffic, a full model that quantifies adversarial resource expenditure and incorporates stateless retry mechanisms could provide stronger availability guarantees.

ACCE and Continuous Channel Properties. The analysis focuses on authenticated key exchange followed by a protected channel with fixed keys. Protocols such as TLS 1.3 and WireGuard support re-keying and continuous channel security. Extending QSTP with similar capabilities and analyzing it under an ACCE style model is a natural direction for future investigation.

Overall, while the present results establish strong foundational guarantees for QSTP, additional work can expand both the theoretical model and the protocol design to support broader deployment scenarios and stronger long term security properties.

15 Conclusion

QSTP provides a streamlined post quantum tunneling protocol built from a fixed suite of cryptographic primitives and a deterministic handshake structure. Its design avoids negotiation, reduces state machine complexity, and enables a clean reduction based analysis. In this work we presented a formal execution model for QSTP, specified authenticated key exchange and channel security goals, and proved that the protocol achieves these goals under standard assumptions on its underlying KEM, signature scheme, key derivation function, and RCS authenticated encryption.

The sequence of game based reductions demonstrates that any adversary capable of distinguishing the QSTP session key from random or forging a valid handshake transcript can be transformed into an adversary that breaks one of the supporting primitives. The resulting bounds show that the security of QSTP inherits directly from the hardness of the module lattice problems underlying the KEM and signature scheme, as well as the AEAD security of RCS. The channel phase provides confidentiality and integrity for all application data packets, with replay protection enforced through authenticated sequence numbers and a configured freshness window.

Beyond these provable guarantees, the cryptanalytic evaluation highlights design tradeoffs such as server-only authentication, timestamp-based freshness, and the cost of post quantum operations on unauthenticated traffic. The symmetric ratchet provides an optional mechanism for advancing session keys within a connection, offering forward secrecy across ratchet epochs at the cost of one additional encrypted message per epoch transition. Session keys are otherwise fixed for the lifetime of the connection between ratchet steps, and the ratchet does not provide post-compromise security in the forward direction.

Overall, the analysis shows that QSTP achieves strong post quantum authenticated key exchange and secure channel properties in a simple and predictable framework. The protocol is suitable for environments that prioritize deterministic behavior, minimal configuration, and native post quantum security, and it provides a solid foundation for future extensions that incorporate mutual authentication, re-keying, or enhanced denial of service resilience.

References

1. National Institute of Standards and Technology (NIST). *FIPS 202: SHA3 Standard, Permutation Based Hash and Extendable Output Functions.* U.S. Department of Commerce, 2015. Available at: https://csrc.nist.gov/publications/detail/fips/202/final
2. National Institute of Standards and Technology (NIST). *FIPS 203: Module-Lattice-Based Key-Encapsulation Mechanism Standard.* U.S. Department of Commerce, 2024. Available at: https://doi.org/10.6028/NIST.FIPS.203
3. National Institute of Standards and Technology (NIST). *SP 800 185: SHA3 Derived Functions, cSHAKE, KMAC, TupleHash and ParallelHash.* U.S. Department of Commerce, 2016. Available at: https://csrc.nist.gov/publications/detail/sp/800-185/final
4. Bernstein, D., Chuengsatiansup, C., Lange, T., and van Vredendaal, C. *NTRU Prime, Round 3 Submission.* National Institute of Standards and Technology, 2020. Available at: https://ntruprime.cr.yp.to
5. Bernstein, D., Hoffstein, J., Pipher, J., Silverman, J., Whyte, W., and Zhang, Z. *NTRU: A Ring Based Public Key Cryptosystem.* Algorithmica, 1998. Available at: https://link.springer.com/article/10.1007/PL00009108
6. Joux, A. *Authentication Failures in NIST Post Quantum Candidates.* Cryptology ePrint Archive, 2020. Available at: https://eprint.iacr.org
7. Bellare, M., and Rogaway, P. *Entity Authentication and Key Distribution.* Advances in Cryptology, CRYPTO 1993. Available at: https://cseweb.ucsd.edu/~mihir/papers/eakd.pdf
8. Bellare, M., and Rogaway, P. *The Security of Triple Encryption and a Framework for Code Based Game Playing.* Advances in Cryptology, EUROCRYPT 2006. Available at: https://cseweb.ucsd.edu/~mihir/papers/triple.pdf
9. Krawczyk, H. *SIGMA: The Cryptographic Protocol that You Can Explain.* IACR ePrint Archive, 2003. Available at: https://eprint.iacr.org
10. Abdalla, M., Fouque, P., and Pointcheval, D. *Password Based Authenticated Key Exchange in the Three Party Setting.* PKC 2005. Available at: https://www.iacr.org
11. Rogaway, P. *Nonce Based Symmetric Encryption.* Proceedings of FSE 2004. Available at: https://web.cs.ucdavis.edu/~rogaway/papers/nonce.pdf
12. McGrew, D., and Viega, J. *The Galois Counter Mode of Operation.* NIST Modes of Operation Submission, 2004. Available at: https://csrc.nist.gov/projects/block-cipher-techniques/gcm
13. Apon, D., Liu, Y., Shirvanian, M., and Wessels, D. *Authenticated Encryption under Misuse.* NIST Lightweight Cryptography Workshop, 2020. Available at: https://csrc.nist.gov
14. Alkim, E., Ducas, L., Poppelmann, T., and Schwabe, P. *Post Quantum Key Exchange, NewHope.* USENIX Security 2016. Available at: https://newhopecrypto.org
15. QRCS Corporation. *QSTP Technical Specification* Quantum Resistant Cryptographic Solutions Corporation, 2024. Available at: https://www.qrcscorp.ca/documents/qstp_specification.pdf
16. QRCS Corporation. *QSTP Implementation Analysis* Quantum Resistant Cryptographic Solutions Corporation, 2025. Available at: https://www.qrcscorp.ca/documents/qstp_analysis.pdf
17. QRCS Corporation. QSTP Library Implementation (GitHub Source Code Repository). https://github.com/QRCS-CORP/QSTP