

The Design and Formal Analysis of the Quantum Secure Messaging Protocol

John G. Underhill

Quantum Resistant Cryptographic Solutions Corporation

Abstract. The Quantum Secure Messaging Protocol (QSMP) is a post-quantum secure channel mechanism that supports two operational modes: a one-way authenticated *Simplex* exchange and a mutually authenticated *Duplex* exchange. Both modes combine a post-quantum key encapsulation mechanism, a signature scheme, and a SHA3-based key derivation function with a permutation-based authenticated encryption channel. This paper presents a formal cryptanalysis of QSMP in a multi-session adversarial model with active network control, adaptive compromise of long-term keys, and full observation of protocol transcripts.

We provide a complete engineering-level description of the protocol, derived directly from the reference implementation, and develop a symbolic protocol model suitable for cryptographic reduction proofs. For Simplex mode, we show that the client obtains unilateral authentication of the server and that the derived session keys achieve indistinguishability under compromise of the server's long-term signing key after the session. For Duplex mode, we establish mutual authentication and key indistinguishability under standard post-quantum hardness assumptions. In both modes, confidentiality and integrity of application data follow from the security of the underlying RCS-based authenticated channel, which binds packet headers and sequence information into the associated data.

Our analysis further examines replay and reordering resistance through the authentication of sequence numbers and timestamps, evaluates the impact of long-term key compromise and ratcheting on forward secrecy, and identifies precise boundaries where security depends on the IND-CCA security of the KEM, the EUF-CMA security of the signature scheme, and the indistinguishability of SHA3 used within cSHAKE. We conclude that QSMP meets its stated security goals under these assumptions and outline several refinements that strengthen the binding properties and formal treatment of key derivation and ratcheting.

1 Introduction

1.1 Background and Motivation

The development of post-quantum secure communication systems has accelerated due to the standardization of lattice based key encapsulation mechanisms and digital signatures, together with the growing availability of efficient permutation based symmetric constructions. Modern secure messaging systems increasingly rely on hybrid designs that combine asymmetric and symmetric components, apply domain separated hashing for session key derivation, and authenticate control data to prevent transcript manipulation. The Quantum Secure Messaging Protocol (QSMP) follows this design philosophy. It provides a compact post quantum handshake suitable for low latency applications and builds an authenticated and encrypted data channel on top of a Keccak based permutation. QSMP was designed to support deployments where endpoints must establish secure channels in environments that may include active adversaries with the ability to intercept, reorder, or drop packets. The protocol supports one way authenticated operation as well as full mutual authentication. It is intended to serve as a general secure transport mechanism rather than a domain specific protocol, so its security must be expressed in a model that captures many concurrent sessions and realistic adversarial capabilities.

1.2 High Level Description of QSMP

QSMP involves two roles, a client and a server, that exchange a sequence of structured packets containing both control information and cryptographic material. The protocol operates in two modes. The Simplex mode provides one way authentication, where the client authenticates the server, establishes a fresh session key through a post quantum key encapsulation mechanism, and initializes a one way encrypted channel. The Duplex mode provides mutual authentication through bidirectional exchange of signed ephemeral key material, establishment of two independent KEM based shared secrets, and verification of a session binding hash. Both modes employ SHA3 and cSHAKE for hashing and key derivation, and both use an RCS based authenticated encryption channel to protect application data. Packet headers include sequence numbers and timestamps that are authenticated as associated data.

Across both modes, QSMP maintains a stateful view of each connection that contains sequence counters, expiration times, session keys, and an optional ratchet value. These values determine how packets are interpreted and when a session must be terminated because of invalid timestamps or protocol errors.

1.3 Security Goals and Contributions

The primary security objectives of QSMP differ between Simplex and Duplex operation. In Simplex mode, the client obtains authentication of the server identity and derives a session key that remains confidential even if the server's long-term signing key is compromised after the handshake. There is no expectation of client identity authentication in this mode. In Duplex mode, both parties authenticate each other through signed ephemeral keys, and both derive session keys that achieve standard indistinguishability under passive and active adversaries. For both modes, channel confidentiality and integrity follow from the security of the RCS based authenticated encryption when used with packet headers as associated data.

This paper provides a formal cryptanalysis of QSMP. It develops a symbolic protocol model aligned with the reference implementation, defines authentication, key indistinguishability, forward secrecy, and channel integrity in a multi session setting, and proves that QSMP meets these goals under standard assumptions on the underlying cryptographic primitives.

The analysis clarifies the scope of Simplex authentication, characterizes forward secrecy with respect to long-term key compromise, and formalizes replay and reordering resistance through authenticated control data. The paper also examines the implications of ratcheting and evaluates the overall robustness of the protocol.

1.4 Document Roadmap

The next section gives an engineering level description of QSMP based directly on the reference implementation, stated in implementation independent terms. Section 3 presents the formal specification of the protocol, including roles, sessions, message flows, and adversarial capabilities. Section 4 defines authentication, key indistinguishability, forward secrecy, and channel security in the multi session model used throughout the proofs. Section 5 states the assumptions on the cryptographic primitives. Sections 6 and 7 contain the Simplex and Duplex security proofs. Section 8 discusses the channel binding properties of the authenticated encryption and the resistance to replay and reordering attacks. Section 9 provides a cryptanalytic evaluation of the protocol. Section 10 analyzes implementation conformance and side channel considerations. Section 11 gives concrete security estimates for recommended parameter choices. Section 12 concludes with limitations and directions for future work.

2 Engineering Description of QSMP

This section describes the behavior of QSMP as realized by the reference implementation, using implementation independent terminology. All details are derived from the public specification and code base and are expressed as abstract operations on state, messages, and cryptographic primitives, rather than as C level constructs.

2.1 Roles, Trust Model, and Deployment Context

QSMP operates between two roles, a *client* and a *server*. The client initiates the connection and drives the handshake, while the server responds and enforces policy on key usage and session creation. In Duplex deployments both peers may possess long-term signing keys, but for each connection one endpoint is treated as the client role and the other as the server role within the key exchange.

The trust model assumes that server long-term signing keys are distributed to clients through an external mechanism such as certificates or preconfigured key directories. In Duplex mode, each endpoint maintains a verification key for the other party and has its own signing key used to authenticate its ephemeral contributions. The protocol does not define how these long-term keys are provisioned or revoked, only that they are available and that their expiration times are honored.

QSMP assumes that the underlying post-quantum key encapsulation mechanism (KEM) provides IND-CCA security, that the signature scheme provides EUF-CMA security, and that SHA3 and cSHAKE behave as collision resistant hash and pseudo-random key derivation functions within their stated domains. The RCS construction is treated as an authenticated encryption with associated data (AEAD) scheme that provides confidentiality and integrity under chosen ciphertext attack when keyed with independent session keys. QSMP is designed to run over a reliable ordered transport. Packet delivery, loss, and reordering are considered under adversarial control, but the transport does not provide any cryptographic guarantees. Deployments may use TCP or another reliable channel that preserves byte ordering. The protocol itself enforces all cryptographic protections and imposes its own sequencing and time validation rules.

Simplex deployments use a long-term server signing key to authenticate the server to anonymous clients that do not possess their own long-term keys. Duplex deployments are

suitable to environments where both peers maintain signing keys and wish to obtain mutual authentication and stronger binding between session keys and long-term identities.

2.2 Global Parameters and Cryptographic Primitives

QSMP is parameterized by a configuration string `cfg` that identifies the set of cryptographic algorithms and parameter choices in use. The configuration string is a fixed length byte array that appears explicitly in handshake messages and is included in several hash computations that bind session keys to a particular algorithm suite.

Each long-term signing key is identified by a key identity `kid`, a 16 byte array that acts as a combined device and key identifier. The identity is carried in handshake messages and is used by the server to select the appropriate verification or signing key and by the client to associate a handshake with a particular expected server key.

Public verification keys and corresponding signing keys have an associated expiration time represented as a 64 bit Unix style UTC low resolution (seconds) timestamp. When the current time exceeds the expiration value the key is considered invalid and handshakes using that key must fail.

QSMP packets share a compact header format of fixed size 21 bytes. The header fields are:

- `flag` $\in \{0, \dots, 255\}$, a one byte message type that distinguishes handshake packets, error packets, and encrypted data packets.
- `msglen` $\in \{0, \dots, 2^{32} - 1\}$, a four byte unsigned integer giving the length of the message body in bytes.
- `sequence` $\in \{0, \dots, 2^{64} - 1\}$, an eight byte unsigned integer that increases monotonically for each packet sent on a connection.
- `utctime` $\in \{0, \dots, 2^{64} - 1\}$, an eight byte unsigned integer giving the sender's current UTC time in seconds.

The header is serialized in a deterministic byte order and is always treated as associated data for authenticated encryption operations. The maximum payload size is bounded by a global constant `QSMP_MESSAGE_MAX`, and implementations also enforce a maximum transmission unit for the combined header and ciphertext.

QSMP instantiates the KEM and signature scheme through abstract interfaces that provide key generation, encapsulation, decapsulation, signing, and verification. The specific algorithm sets (Dilithium with Kyber, Dilithium with McEliece, or McEliece with SPHINCS+) are fixed by the configuration string and not negotiated during the handshake, (while asymmetric primitive parameter sets are defined in the adjoining QSC cryptographic library in the implementation). SHA3 and cSHAKE are used for hashing and key derivation. The RCS construction is a duplex mode authenticated cipher built with wide-block Rijndael and the Keccak permutation, that takes a key, a nonce, and optional associated data and provides encryption and decryption with AEAD integrity protection (AES-GCM is an optional configuration).

2.3 Key Material and long-term State

The server maintains a signing key pair (`ssk`, `svk`) together with an associated configuration string `cfg`, key identity `kid`, and expiration time. In Duplex mode each endpoint may maintain both a local signing key pair and a stored verification key for its peer.

Client side long-term state consists of one or more verification keys `svk` or peer verification keys, the corresponding key identities, associated configuration strings, and expiration times. The protocol assumes that client software has verified these keys through some external mechanism prior to use.

For each handshake QSMP derives a session cookie hash sch that binds the session to long-term key material and the configuration string. In Simplex mode the session cookie is

$$\text{sch}_S = H(\text{cfg} \parallel \text{kid} \parallel \text{svk}),$$

where svk is the server verification key. Both client and server compute this value and store it as part of the handshake state. In Duplex mode the cookie includes both long-term verification keys,

$$\text{sch}_D = H(\text{cfg} \parallel \text{kid} \parallel \text{pvk}_C \parallel \text{pvk}_S),$$

where pvk_C and pvk_S are the client and server verification keys bound by the key identity kid . This value is used as input to key derivation and is also confirmed during the Duplex establish stage.

Each connection maintains a state record that stores at least:

- a transmit sequence counter txseq and receive sequence counter rxseq ,
- the current transmit cipher state and receive cipher state for the RCS channel,
- the current ratchet key state rtcs ,
- the last observed UTC time in seconds for the peer, used to enforce time windows,
- a flag indicating the last successful handshake stage.

This state persists for the lifetime of a session and is erased when the session is torn down.

2.4 Handshake and Session Establishment in Simplex Mode

The Simplex handshake consists of three stages: a connect request from the client, a connect response from the server, and an exchange stage that transfers a KEM ciphertext and establishes the data channel.

Connect request. The client first verifies that the stored server verification key corresponding to the chosen key identity kid is not expired. It then constructs a message body containing kid and cfg , and assembles a packet header with the connect request flag, the current transmit sequence number, the length of the message body, and the current UTC time. The client sets the packet sequence to the current txseq value without incrementing it yet. The Simplex session cookie sch_S is computed as $H(\text{cfg} \parallel \text{kid} \parallel \text{svk})$ and stored in the handshake state. The packet is then sent to the server.

Connect response. Upon receiving a packet, the server deserializes the header and validates that the flag is a connect request, that the message length matches the expected size, that the sequence number is rxseq , and that the timestamp lies inside an acceptable tolerance window relative to its local clock (implementation default is 60 seconds). It uses the received kid to select the appropriate server signing key and verifies that its own configuration string matches cfg . The server computes the same Simplex session cookie sch_S and stores it.

The server generates an ephemeral KEM key pair (pk, sk) and constructs a transcript hash over the serialized header and the public key. It signs this hash with its long-term signing key and returns a connect response message that carries the signature and the ephemeral public key. The header of the response is created with the connect response flag, the current server sequence number, the response length, and the current server time. The server increments its transmit sequence counter after sending the response.

Exchange stage. The client validates the connect response header with the same structural checks and time window. It verifies the server signature on the transcript hash and recomputes the hash over the received header and public key to check consistency. If verification succeeds, the client encapsulates a shared secret `sec` to the server’s ephemeral public key, obtaining a ciphertext `cpt` and secret `sec`.

The client uses cSHAKE as a key derivation function with input `sec` and customization string `schS`. The KDF output is parsed into two symmetric keys and two nonces,

$$(k_1, n_1, k_2, n_2) \leftarrow \text{KDF}(\text{sec}, \text{sch}_S),$$

and these values are used to initialize the transmit and receive RCS channels. The internal ratchet state `rtcs` is set to a slice of the permuted Keccak state after the key derivation, so that it does not equal any current encryption key.

The client assembles an exchange request packet whose message body contains the ciphertext `cpt`. The packet header is built with the exchange request flag, the current sequence number, the ciphertext length, and the current time. The client sends the packet and increments its transmit sequence counter.

When the server receives the exchange request, it validates the header and uses the ephemeral private key `sk` to decapsulate the ciphertext, recovering the same secret `sec`. It runs the same key derivation procedure with `sec` and `schS` to obtain k_1, n_1, k_2, n_2 , initializes its own RCS transmit and receive channels, and stores a ratchet key state derived from the post permutation Keccak state. The ephemeral KEM private key and shared secret are cleared from memory after the RCS keys and ratchet state have been initialized. No further handshake messages are needed in Simplex mode; once the exchange stage completes, the session transitions to the established state.

2.4.1 Simplex Key Exchange Pseudo-code

The Simplex key exchange is implemented by the functions `qsmp_kex_simplex_client_key_exchange` and `qsmp_kex_simplex_server_key_exchange` in `kex.c`, together with their internal helpers. The pseudo-code in this subsection follows the control flow of those functions and abstracts the code into Kyber style presentation, while preserving the exact cryptographic operations, header construction, and state updates.

We write `now` for the UTC timestamp function, `HeaderCreate` and `HeaderSerialize` for the packet header routines, `SHA3` for the SHA3 256-bit hash, `Encap` and `Decap` for the KEM encapsulation and decapsulation algorithms, `Sign` and `Verify` for the signature scheme, and `cSHAKE` for the KDF. RCS channel initialization is written as `RCS.Init` and follows the transmit and receive assignment in the implementation.

Client: Connect Request. In the Simplex connect request, the client first checks that the cached server key record is still valid with respect to its expiration time. If the key is valid, the client constructs a connect request packet that contains the server key identity and the fixed configuration string. At the same time the client computes the Simplex session cookie hash from the configuration, key identity, and stored server verification key. The packet header is created with the connect request flag, the current transmit sequence number, and the message length, and the exchange state flag is set to indicate that a connect request has been prepared.

Algorithm 1 Simplex.ClientConnectRequest

```

1: procedure CLIENTCONNECTREQUEST(kcs, cns, pkt)
2:    $t \leftarrow \text{now}()$ 
3:   if  $t > kcs.\text{expiration}$  then
4:      $cns.\text{exflag} \leftarrow \text{none}$ 
5:     return error_key_expired
6:   end if
7:    $\text{pkt.message} \leftarrow kcs.\text{keyid} \parallel \text{QSMP\_CONFIG\_STRING}$ 
8:    $\text{len} \leftarrow \text{KEX\_SIMPLEX\_CONNECT\_REQUEST\_MESSAGE\_SIZE}$ 
9:    $\text{pkt.header} \leftarrow \text{HeaderCreate}(\text{connect\_request}, cns.\text{txseq}, \text{len})$ 
10:   $kcs.\text{schash} \leftarrow \text{SHA3}(\text{QSMP\_CONFIG\_STRING} \parallel kcs.\text{keyid} \parallel kcs.\text{verkey})$ 
11:   $cns.\text{exflag} \leftarrow \text{connect\_request}$ 
12:  return error_none
13: end procedure

```

Server: Connect Response. On receipt of a Simplex connect request, the server verifies that the incoming key identity matches a configured record, that the configuration string equals the expected value, and that the server key has not expired. If these checks pass the server computes the same Simplex session cookie hash from its configuration string, key identity, and verification key. The server then generates an ephemeral KEM key pair, constructs a connect response header, hashes the serialized response header and the ephemeral public key, and signs this hash with its long-term signing key. The connect response message carries the signature and the ephemeral public key, and the exchange state flag is updated accordingly.

Algorithm 2 Simplex.ServerConnectResponse

```

1: procedure SERVERCONNECTRESPONSE(kss, cns, req, resp)
2:   confs  $\leftarrow$  extract configuration bytes from req.message
3:   if  $\text{KeyIdVerify}(kss.\text{keyid}, \text{req.message}) = \text{false}$  then
4:     return error_invalid_input
5:   end if
6:    $t \leftarrow \text{now}()$ 
7:   if  $t > kss.\text{expiration}$  then
8:     return error_key_expired
9:   end if
10:  if confs  $\neq$  QSMP_CONFIG_STRING then
11:    return error_unknown_protocol
12:  end if
13:   $kss.\text{schash} \leftarrow \text{SHA3}(\text{QSMP\_CONFIG\_STRING} \parallel kss.\text{keyid} \parallel kss.\text{verkey})$ 
14:   $(kss.\text{pubkey}, kss.\text{prikey}) \leftarrow \text{EncapKeyGen}()$ 
15:   $\text{len} \leftarrow \text{KEX\_SIMPLEX\_CONNECT\_RESPONSE\_MESSAGE\_SIZE}$ 
16:   $\text{resp.header} \leftarrow \text{HeaderCreate}(\text{connect\_response}, cns.\text{txseq}, \text{len})$ 
17:   $\text{shdr} \leftarrow \text{HeaderSerialize}(\text{resp.header})$ 
18:   $\text{phash} \leftarrow \text{SHA3}(\text{shdr} \parallel kss.\text{pubkey})$ 
19:   $\text{sig} \leftarrow \text{Sign}_{kss.\text{sigkey}}(\text{phash})$ 
20:   $\text{resp.message} \leftarrow \text{sig} \parallel kss.\text{pubkey}$ 
21:   $cns.\text{exflag} \leftarrow \text{connect\_response}$ 
22:  return error_none
23: end procedure

```

Client: Exchange Request. For the Simplex exchange request, the client first validates the connect response header, including flag, sequence number, timestamp window, and message size, as part of the surrounding key exchange driver. It then parses the response message into the server signature and ephemeral public key, recomputes the hash over the serialized response header and public key, and verifies the signature under the stored server verification key. If the signature and hash match, the client encapsulates a shared secret to the ephemeral public key, stores only the ciphertext in the exchange request message, and derives the transmit and receive keys and nonces from the combination of the shared secret and the previously computed session cookie hash using cSHAKE. The client initializes RCS for both directions and records the ratchet state extracted from the internal cSHAKE state.

Algorithm 3 Simplex.ClientExchangeRequest

```

1: procedure CLIENTEXCHANGEREQUEST(kcs, cns, resp, pkt)
2:   parse resp.message as (sig, pubk)
3:   shdr  $\leftarrow$  HeaderSerialize(resp.header)
4:   khash  $\leftarrow$  SHA3(shdr || pubk)
5:   if Verifykcs.verkey(sig, khash) = false then
6:     cns.exflag  $\leftarrow$  none
7:     return error_signature_invalid
8:   end if
9:   (ssec, cpt)  $\leftarrow$  Encap(pubk)
10:  pkt.message  $\leftarrow$  cpt
11:  len  $\leftarrow$  KEX_SIMPLEX_EXCHANGE_REQUEST_MESSAGE_SIZE
12:  pkt.header  $\leftarrow$  HeaderCreate(exchange_request, cns.txseq, len)
13:  prnd  $\leftarrow$  cSHAKE(ssec, custom = kcs.schash, outlen = 2 · QSC_KECCAK_256_RATE)
14:  derive (k1, n1, k2, n2) from prnd using the same byte layout as in kex.c
15:  cns.rtcs  $\leftarrow$  InternalState(cSHAKE)
16:  RCS.Init(cns.txcpr, k1, n1, true)
17:  RCS.Init(cns.rxcpr, k2, n2, false)
18:  zeroize ssec and prnd
19:  cns.exflag  $\leftarrow$  exchange_request
20:  return error_none
21: end procedure

```

Server: Exchange Response. When the server receives a Simplex exchange request, it first verifies the header fields outside this routine. It then decapsulates the KEM ciphertext from the message using the stored ephemeral private key. If decapsulation succeeds, the server runs cSHAKE on the shared secret with the Simplex session cookie hash as customization to derive the same pair of keys and nonces as the client, assigns them to its receive and transmit RCS instances in the opposite roles, and records a ratchet state derived from the internal cSHAKE state. The server zeroizes the shared secret and temporary buffers, and finally emits an exchange response packet with an empty body and the exchange response flag in the header to signal that the Simplex session is established.

Algorithm 4 Simplex.ServerExchangeResponse

```

1: procedure SERVEREXCHANGEREPLY(kss, cns, req, resp)
2:   cpt  $\leftarrow$  req.message
3:   ssec  $\leftarrow$  Decap(cpt, kss.prikey)
4:   if decapsulation fails then
5:     return error_decapsulation_failure
6:   end if
7:   prnd  $\leftarrow$  cSHAKE(ssec, custom = kss.schash, outlen = 2 · QSC_KECCAK_256_RATE)
8:   derive ( $k_1, n_1, k_2, n_2$ ) from prnd
9:   cns.rtcs  $\leftarrow$  InternalState(cSHAKE)
10:  RCS.Init(cns.rcpr,  $k_1, n_1$ , false)
11:  RCS.Init(cns.txcr,  $k_2, n_2$ , true)
12:  zeroize ssec and prnd
13:  resp.header  $\leftarrow$  HeaderCreate(exchange_response, cns.txseq, 0)    // empty body
14:  cns.exflag  $\leftarrow$  session_established
15:  return error_none
16: end procedure

```

Client: Establish Verify. The final Simplex step on the client side is to verify the exchange response. The client checks that the response header carries the expected exchange response flag, that the sequence number and timestamp are valid with respect to its local state and time window, and that the message length is zero as required. If these checks all succeed, the client marks the session as established in its connection state. Any failure in header validation or state checks leads to an error and aborts the key exchange.

Algorithm 5 Simplex.ClientEstablishVerify

```

1: procedure CLIENTESTABLISHVERIFY(kcs, cns, resp)
2:   if header validation on resp.header fails then
3:     cns.exflag  $\leftarrow$  none
4:     return error_invalid_input
5:   end if
6:   cns.exflag  $\leftarrow$  session_established
7:   return error_none
8: end procedure

```

2.5 Handshake and Session Establishment in Duplex Mode

The Duplex handshake extends Simplex by performing mutual authentication and by deriving session keys from two independent KEM secrets. It consists of a connect stage, an exchange stage, and an establish stage that confirms the session cookie. Duplex derives its channel keys from the concatenation of two independent asymmetric cipher shared secrets, each providing approximately 256-bits of classical security. These secrets are combined and expanded with cSHAKE_512, which yields RCS-512 keys with an effective 512-bit security margin against preimage attacks. This provides higher strength than the Simplex derivation, which uses only a single Kyber shared secret.

Connect stage. The client and server both possess long-term signing keys and verification keys. The client constructs a connect request that includes kid and cfg, an ephemeral KEM public key for the first shared secret, and a signature over a hash of the serialized header, the key identity, the configuration string, and the public key. The Duplex session cookie $\text{sch}_D = H(\text{cfg} \parallel \text{kid} \parallel \text{pvk}_C \parallel \text{pvk}_S)$ is computed and stored in the client state. The packet

header is created with the connect request flag, sequence number, message length, and current time. The client sends the packet and increments its transmit sequence counter. The server validates the header and checks that `kid` and `cfg` match a local key record. It verifies the client's signature using the stored client verification key and recomputes the transcript hash to bind the header and the client's public key. If verification succeeds, the server generates its own ephemeral KEM key pair for the second secret and computes the same Duplex session cookie `schD`. It constructs a connect response that carries the server's ephemeral public key and a signature over the hashed header and public key. The response header is filled with the connect response flag, current sequence number, message size, and time. The server sends the packet and increments its transmit sequence counter.

Exchange stage. After validating the connect response header, the client verifies the server's signature and recomputes the transcript hash. It encapsulates to the server's ephemeral public key to obtain a second shared secret `sec2`, while the server, upon receiving the subsequent exchange request, decapsulates the first shared secret `sec1` from the ciphertext carried in the client's exchange packet. In parallel, the client decapsulates `sec1` from the ciphertext received in the connect response, and the server encapsulates `sec2` to the client's public key. At the end of the exchange stage both parties hold the same pair `(sec1, sec2)`.

The KDF takes both secrets and the session cookie as input. Conceptually,

$$(k_1, n_1, k_2, n_2) \leftarrow \text{KDF}(\text{sec}_1, \text{sch}_D, \text{sec}_2),$$

and these values are used to initialize the bidirectional RCS channels. As in Simplex, a ratchet key state `rtcs` is stored from the post permutation Keccak state; the state is permuted specifically to generate an unrelated ratchet key. Ephemeral KEM private keys and shared secrets are cleared after use.

Establish stage. The Duplex establish stage confirms that both parties hold the same session cookie and that their derived keys are synchronized. The client constructs an establish request whose message body consists of `schD` encrypted under the transmit RCS channel. Before encryption, the client serializes the establish request header and sets it as associated data for the cipher. It then encrypts the cookie and sends the packet.

The server receives the establish request, serializes the header, and sets it as associated data on its receive channel. It decrypts the cookie candidate and compares it in constant time with its local value of `schD`. If they match, the server computes a hash of the cookie, for example $H(\text{sch}_D)$, and constructs an establish response whose message body is this hash encrypted under the transmit channel with the establish response header used as associated data. The establish response is sent with the appropriate flag and updated sequence number.

The client validates the establish response header, decrypts the message using the receive channel with the serialized header as associated data, and compares the recovered hash with $H(\text{sch}_D)$. If this check succeeds, both parties mark the session as established. Any failure in decryption, verification, or comparison causes the session to be torn down and all key material to be erased.

2.5.1 Duplex Key Exchange pseudo-code

The Duplex key exchange is implemented by `qsm_p_kex_duplex_client_key_exchange` and `qsm_p_kex_duplex_server_key_exchange` in `kex.c`, together with the internal helpers `kex_duplex_client_connect_request`, `kex_duplex_server_connect_response`,

`kex_duplex_client_exchange_request`, `kex_duplex_server_exchange_response`, `kex_duplex_client_establish_request`, `kex_duplex_server_establish_response`, and `kex_duplex_client_establish_verify`. The pseudo-code in this subsection follows the control flow and cryptographic operations of those functions as expressed in the reference implementation.

We write now for the UTC timestamp function, `HeaderCreate` and `HeaderSerialize` for the packet header routines, `SHA3` for SHA3 256-bit hashing, `Encap` and `Decap` for the KEM encapsulation and decapsulation primitives, `Sign` and `Verify` for the signature scheme, and `cSHAKE` for the KDF. RCS initialization is written as `RCS.Init`, matching the direction assignments and key layouts used in `kex.c`. In Duplex mode the client and server each hold a long-term signature key pair and an asymmetric encryption key pair, and the session cookie hash sch_D is computed from the configuration string, key identity, and both public verification keys.

Client: Connect Request. In the Duplex connect request, the client first checks that its local key record has not expired. It then constructs a session cookie hash from the configuration string and both public verification keys, which will be used later as input to the key derivation function. The client prepares a connect request message containing the server key identity and configuration string, computes a hash over the serialized connect request header and message, signs this hash with its own signing key, and appends the signature to the message. The header is created with the connect request flag, current transmit sequence number, and the total message length, and the connection state exchange flag is set to record that a Duplex connect request has been prepared.

Algorithm 6 Duplex.ClientConnectRequest

```

1: procedure DUPLEXCLIENTCONNECTREQUEST(kcs, cns, pkt)
2:    $t \leftarrow \text{now}()$ 
3:   if  $t > kcs.\text{expiration}$  then
4:      $cns.\text{exflag} \leftarrow \text{none}$ 
5:     return error_key_expired
6:   end if
7:    $\text{pkt.message} \leftarrow kcs.\text{keyid} \parallel \text{QSMP\_CONFIG\_STRING}$ 
8:    $\text{len} \leftarrow \text{KEX\_DUPLEX\_CONNECT\_REQUEST\_MESSAGE\_SIZE}$ 
9:    $\text{pkt.header} \leftarrow \text{HeaderCreate}(\text{connect\_request}, cns.\text{txseq}, \text{len})$ 
10:   $\text{shdr} \leftarrow \text{HeaderSerialize}(\text{pkt.header})$ 
11:   $\text{phash} \leftarrow \text{SHA3}(\text{shdr} \parallel \text{pkt.message})$ 
12:   $\text{slen} \leftarrow 0$ 
13:   $\text{siglen} \leftarrow \text{QSMP\_ASYMMETRIC\_SIGNATURE\_SIZE}$ 
14:   $\text{Sign}_{kcs.\text{sigkey}}(\text{phash}, \text{QSMP\_DUPLEX\_HASH\_SIZE}, \text{pkt.message} + |\text{pkt.message}|)$ 
15:   $kcs.\text{schash} \leftarrow \text{SHA3}(\text{QSMP\_CONFIG\_STRING} \parallel kcs.\text{keyid} \parallel kcs.\text{verkey} \parallel kcs.\text{rverkey})$ 
16:   $cns.\text{exflag} \leftarrow \text{connect\_request}$ 
17:  return error_none
18: end procedure

```

Server: Connect Response. On receipt of a Duplex connect request, the server validates the incoming header and timestamp in the key exchange driver. The helper procedure checks that the key identity in the message matches a configured record and that the configuration string equals the local configuration. It verifies the signature on the client hash by recomputing the hash over the serialized request header and message and comparing it in constant time to the signed hash. If the verification succeeds, the

server computes the Duplex session cookie hash from the configuration string, key identity, and both public verification keys. The server then generates an ephemeral KEM key pair for the Duplex exchange, constructs a connect response header, computes a hash of the serialized response header and ephemeral public key, signs this hash with its signing key, and appends both the signature and the public key to the response message. The exchange flag is updated to record a successful Duplex connect response.

Algorithm 7 Duplex.ServerConnectResponse

```

1: procedure DUPLEXSERVERCONNECTRESPONSE(kss, cns, req, resp)
2:   pkid  $\leftarrow$  key identity bytes from req.message
3:   cfg  $\leftarrow$  configuration bytes from req.message
4:   if KeyIdLookup(pkid) fails then
5:     cns.exflag  $\leftarrow$  none
6:     return error_invalid_input
7:   end if
8:   if cfg  $\neq$  QSMP_CONFIG_STRING then
9:     cns.exflag  $\leftarrow$  none
10:    return error_unknown_protocol
11:  end if
12:  milen  $\leftarrow$  QSMP_ASYMMETRIC_SIGNATURE_SIZE + QSMP_DUPLEX_HASH_SIZE
13:  slen  $\leftarrow$  0
14:  Verifykss.rverkey(req.message, milen, khash, slen)
15:  shdr  $\leftarrow$  HeaderSerialize(req.header)
16:  phash  $\leftarrow$  SHA3(shdr || req.message[0 : payloadLen])
17:  if ConstEq(khash, phash) = 0 then
18:    kss.schash  $\leftarrow$  SHA3(QSMP_CONFIG_STRING || kss.keyid || kss.rverkey || kss.verkey)
19:    (kss.pubkey, kss.prikey)  $\leftarrow$  EncapKeyGen()
20:    len  $\leftarrow$  KEX_DUPLEX_CONNECT_RESPONSE_MESSAGE_SIZE
21:    resp.header  $\leftarrow$  HeaderCreate(connect_response, cns.txseq, len)
22:    shdr  $\leftarrow$  HeaderSerialize(resp.header)
23:    phash  $\leftarrow$  SHA3(shdr || kss.pubkey)
24:    slen  $\leftarrow$  0
25:    Signkss.sigkey(phash, QSMP_DUPLEX_HASH_SIZE, resp.message)
26:    append kss.pubkey after the signature in resp.message
27:    cns.exflag  $\leftarrow$  connect_response
28:    return error_none
29:  else
30:    cns.exflag  $\leftarrow$  none
31:    return error_verify_failure
32:  end if
33: end procedure

```

Client: Exchange Request. For the Duplex exchange request, the client first validates the server connect response header in the surrounding driver. It parses the response message into the server signature and ephemeral public encapsulation key, recomputes the hash of the serialized response header and the public key, and verifies that this matches the signed hash. If the check fails the exchange is aborted. Otherwise the client uses the server ephemeral public key to encapsulate the first shared secret and obtain ciphertext *cpta*. The client stores the shared secret *seca* for later use, generates its own ephemeral asymmetric encryption key pair, computes a hash over its ephemeral public key, *cpta*, and the serialized exchange request header, and signs this hash with its signing key. The exchange request message consists of the client signature, *cpta* and the client ephemeral

public key and the exchange flag is updated accordingly.

Algorithm 8 Duplex.ClientExchangeRequest

```

1: procedure DUPLEXCLIENTEXCHANGEREQUEST(kcs, cns, resp, pkt)
2:   parse resp.message as (sigS, pkS)
3:   shdr ← HeaderSerialize(resp.header)
4:   phash ← SHA3(shdr || pkS)
5:   Verifykcs.rverkey(sigS, phash)
6:   if verification fails then
7:     cns.exflag ← none
8:     return error_verify_failure
9:   end if
10:  (seca, cpta) ← Encap(pkS)
11:  store seca in kcs for later key derivation
12:  (kcs.pubkey, kcs.prikey) ← EncapKeyGen()
13:  len ← KEX_DUPLEX_EXCHANGE_REQUEST_MESSAGE_SIZE
14:  pkt.header ← HeaderCreate(exchange_request, cns.txseq, len)
15:  shdr ← HeaderSerialize(pkt.header)
16:  kch ← SHA3(kcs.pubkey || cpta || shdr)
17:  Signkcs.sigkey(kch, QSMP_DUPLEX_HASH_SIZE, sigC)
18:  pkt.message ← sigC || cpta || kcs.pubkey
19:  cns.exflag ← exchange_request
20:  return error_none
21: end procedure

```

Server: Exchange Response. When the server receives a Duplex exchange request, it validates the header and timestamp in the driver, then parses the message into the client signature, ciphertext *cpta*, and client ephemeral public key. It verifies the client signature by recomputing the hash of the serialized header and message fields and comparing this to the signed hash. If verification fails the exchange is aborted. If it succeeds, the server decapsulates *cpta* using its private ephemeral key to obtain the first shared secret *seca*, then encapsulates a second shared secret *secb* to the client ephemeral public key and obtains ciphertext *cptb*. The server combines *seca*, *secb*, and the session cookie hash *sch_D* in cSHAKE to derive two symmetric keys and two nonces which are assigned to the receive and transmit RCS instances. Finally, the server computes a hash of *cptb* and the serialized exchange response header, signs this hash, and sends the signature and *cptb* as the exchange response message.

Algorithm 9 Duplex.ServerExchangeResponse

```

1: procedure DUPLEXSERVEREXCHANGERESPONSE( $kss, cns, req, resp$ )
2:   parse  $req.message$  as  $(sig_C, cpta, pk_C)$ 
3:    $shdr \leftarrow \text{HeaderSerialize}(req.header)$ 
4:    $kch \leftarrow \text{SHA3}(pk_C \parallel cpta \parallel shdr)$ 
5:    $\text{Verify}_{kss.rverkey}(sig_C, kch)$ 
6:   if verification fails then
7:      $cns.exflag \leftarrow \text{none}$ 
8:     return  $\text{error\_verify\_failure}$ 
9:   end if
10:   $seca \leftarrow \text{Decap}(cpta, kss.prikey)$ 
11:  if decapsulation fails then
12:    return  $\text{error\_decapsulation\_failure}$ 
13:  end if
14:   $(secb, cptb) \leftarrow \text{Encap}(pk_C)$ 
15:   $prnd \leftarrow \text{cSHAKE}(seca \parallel secb, \text{custom} = kss.schash, \text{outlen} = 3 \cdot$ 
     $\text{QSC\_KECCAK\_512\_RATE})$ 
16:  derive  $(k_1, n_1, k_2, n_2)$  from  $prnd$  using the byte layout in  $kex.c$ 
17:  copy part of the cSHAKE state into  $cns.rtcs$  as ratchet seed
18:   $\text{RCS.Init}(cns.rxcpr, k_1, n_1, \text{false})$ 
19:   $\text{RCS.Init}(cns.txcpr, k_2, n_2, \text{true})$ 
20:  zeroize  $seca, secb, prnd$ 
21:   $len \leftarrow \text{KEX\_DUPLEX\_EXCHANGE\_RESPONSE\_MESSAGE\_SIZE}$ 
22:   $resp.header \leftarrow \text{HeaderCreate}(\text{exchange\_response}, cns.txseq, len)$ 
23:   $shdr \leftarrow \text{HeaderSerialize}(resp.header)$ 
24:   $cpth \leftarrow \text{SHA3}(cptb \parallel shdr)$ 
25:   $\text{Sign}_{kss.sigkey}(cpth, \text{QSMP\_DUPLEX\_HASH\_SIZE}, sig_S)$ 
26:   $resp.message \leftarrow sig_S \parallel cptb$ 
27:   $cns.exflag \leftarrow \text{exchange\_response}$ 
28:  return  $\text{error\_none}$ 
29: end procedure

```

Client: Establish Request. The client establish request handler processes the server exchange response. It validates the header and timestamp, parses the message into the server signature and ciphertext $cptb$, and recomputes the hash over $cptb$ and the serialized header. If the signature verification or hash comparison fails, the exchange is aborted. When verification succeeds, the client decapsulates $cptb$ using its private ephemeral encryption key to obtain $secb$, and combines $seca$, $secb$, and the session cookie hash sch_D in cSHAKE to derive two symmetric keys and two nonces. These keys are assigned to the client transmit and receive RCS instances and a ratchet seed is extracted from the internal state. The client then encrypts the session cookie hash under the transmit cipher, using the serialized establish request header as associated data, and sends this ciphertext as the Duplex establish request message.

Algorithm 10 Duplex.ClientEstablishRequest

```

1: procedure DUPLEXCLIENTESTABLISHREQUEST( $kcs, cns, resp, pkt$ )
2:   parse  $resp.message$  as  $(sig_S, cptb)$ 
3:    $shdr \leftarrow \text{HeaderSerialize}(resp.header)$ 
4:    $cpth \leftarrow \text{SHA3}(cptb \parallel shdr)$ 
5:    $\text{Verify}_{kcs.rverkey}(sig_S, cpth)$ 
6:   if verification fails then
7:      $cns.exflag \leftarrow \text{none}$ 
8:     return  $error\_verify\_failure$ 
9:   end if
10:   $secb \leftarrow \text{Decap}(cptb, kcs.prikey)$ 
11:  if decapsulation fails then
12:    return  $error\_decapsulation\_failure$ 
13:  end if
14:   $prnd \leftarrow \text{cSHAKE}(seca \parallel secb, \text{custom} = kcs.schash, \text{outlen} = 3 \cdot$ 
     $\text{QSC\_KECCAK\_512\_RATE})$ 
15:  derive  $(k_1, n_1, k_2, n_2)$  from  $prnd$ 
16:  copy part of the cSHAKE state into  $cns.rtcs$  as ratchet seed
17:   $\text{RCS.Init}(cns.txcpr, k_1, n_1, \text{true})$ 
18:   $\text{RCS.Init}(cns.rxcpr, k_2, n_2, \text{false})$ 
19:  zeroize  $secb$  and  $prnd$ 
20:   $len \leftarrow \text{KEX\_DUPLEX\_ESTABLISH\_REQUEST\_MESSAGE\_SIZE}$ 
21:   $pkt.header \leftarrow \text{HeaderCreate}(\text{establish\_request}, cns.txseq, len)$ 
22:   $shdr \leftarrow \text{HeaderSerialize}(pkt.header)$ 
23:   $cm \leftarrow \text{RCS.Encrypt}(cns.txcpr, ad = shdr, pt = kcs.schash)$ 
24:   $pkt.message \leftarrow cm$ 
25:   $cns.exflag \leftarrow \text{establish\_request}$ 
26:  return  $error\_none$ 
27: end procedure

```

Server: Establish Response. The Duplex server establish response handler processes the client establish request. It validates the request header and timestamp and decrypts the message using the receive cipher while treating the serialized establish request header as associated data. The decrypted value is compared to the locally stored session cookie hash; if they differ, the exchange is aborted and the tunnel is torn down. If the session cookie hashes match, the server hashes the session cookie, encrypts this hash under the transmit cipher with the establish response header as associated data, and sends the resulting ciphertext as the establish response message. The operational state is set to session established.

Algorithm 11 Duplex.ServerEstablishResponse

```

1: procedure DUPLEXSERVERESTABLISHRESPONSE(kss, cns, req, resp)
2:   shdr  $\leftarrow$  HeaderSerialize(req.header)
3:   sch'  $\leftarrow$  RCS.Decrypt(cns.rxcpr, ad = shdr, ct = req.message)
4:   if sch'  $\neq$  kss.schash then
5:     cns.exflag  $\leftarrow$  none
6:     return error_decryption_failure
7:   end if
8:   hsch  $\leftarrow$  SHA3(kss.schash)
9:   len  $\leftarrow$  KEX_DUPLEX_ESTABLISH_RESPONSE_MESSAGE_SIZE
10:  resp.header  $\leftarrow$  HeaderCreate(establish_response, cns.txseq, len)
11:  shdrR  $\leftarrow$  HeaderSerialize(resp.header)
12:  cm  $\leftarrow$  RCS.Encrypt(cns.txcpr, ad = shdrR, pt = hsch)
13:  resp.message  $\leftarrow$  cm
14:  cns.exflag  $\leftarrow$  session_established
15:  return error_none
16: end procedure

```

Client: Establish Verify. The final Duplex step on the client side is to verify the establish response. The client first checks that the response header carries the establish response flag, that the sequence number and timestamp are valid, and that the message length matches the expected Duplex establish response size. It then uses the receive cipher to decrypt the message with the serialized establish response header as associated data and obtains the hash of the session cookie. The client recomputes the hash of its local session cookie and compares the result to the decrypted value in constant time. If the values match the client confirms that both parties have established the same encrypted channel, marks the session as established, and is ready to process application data. Any failure causes the tunnel to be torn down.

Algorithm 12 Duplex.ClientEstablishVerify

```

1: procedure DUPLEXCLIENTESTABLISHVERIFY(kcs, cns, resp)
2:   shdr  $\leftarrow$  HeaderSerialize(resp.header)
3:   hsch'  $\leftarrow$  RCS.Decrypt(cns.rxcpr, ad = shdr, ct = resp.message)
4:   hsch  $\leftarrow$  SHA3(kcs.schash)
5:   if hsch'  $\neq$  hsch then
6:     cns.exflag  $\leftarrow$  none
7:     return error_decryption_failure
8:   end if
9:   cns.exflag  $\leftarrow$  session_established
10:  return error_none
11: end procedure

```

2.6 Session Keys, Ratcheting, and Rekeying

In both modes the KDF is realized by a cSHAKE based construction that takes as keying input the concatenation of one or two KEM secrets and uses the session cookie as customization string. The output stream is parsed into symmetric keys and nonces for the RCS channels and an internal ratchet state *rtcs*. The ratchet state is not used directly as an encryption key but is instead stored as a seed for future key derivation.

QSMP supports two forms of rekeying. A symmetric ratchet derives fresh keys from the internal ratchet state and fresh randomness or derived material, updating *rtcs* and

replacing the RCS keys while preserving the same role assignment for transmit and receive channels. An asymmetric ratchet, available in Duplex mode, injects new asymmetric KEM secrets into the KDF along with the existing ratchet state, providing stronger recovery from long-term key compromise. In both cases the previous RCS keys are erased once the new keys are installed, and the ratchet state is updated in a one-way fashion so that past keys cannot be reconstructed.

2.6.1 Asymmetric Ratchet Integration in Duplex Mode

In Duplex mode the asymmetric ratchet is represented at the key exchange layer by the persistent remote verification key field `rverkey` in the client and server Duplex key exchange states. The key exchange code never generates or updates `rverkey` itself; instead it relies on an external `key_query` interface (on the server side) or higher level configuration (on the client side) to provide the current remote verification key. The presence of the compile time flag `QSMP_ASYMMETRIC_RATCHET` controls whether `rverkey` is cleared when a Duplex state is reset. When the flag is enabled, the remote verification key persists across key exchange resets and thus can be advanced by an external ratchet mechanism without being erased by QSMP's key exchange layer.

The same `rverkey` value is:

- used in the Duplex session cookie hash `schash` on both client and server, and
- used as the verification key for Duplex connect and exchange signatures.

The following pseudo-code extracts the asymmetric ratchet relevant parts of the Duplex key exchange implementation in `kex.c`.

Duplex client ratchet sensitive reset. The Duplex client reset function clears all local key exchange state. When the asymmetric ratchet is disabled, it also clears the stored remote verification key `rverkey`. When the asymmetric ratchet flag is enabled, `rverkey` is left untouched so that it can persist across sessions and be updated only by the external ratchet logic.

Algorithm 13 Duplex.ClientRatchetAwareReset

```

1: procedure DUPLEXCLIENTRATCHETAWARERESET(kcs)
2:   clear kcs.keyid over QSMP_KEYID_SIZE
3:   clear kcs.schash over QSMP_DUPLEX_SCHASH_SIZE
4:   clear kcs.prikey over QSMP_ASYMMETRIC_PRIVATE_KEY_SIZE
5:   clear kcs.pubkey over QSMP_ASYMMETRIC_PUBLIC_KEY_SIZE
6:   clear kcs.verkey over QSMP_ASYMMETRIC_VERIFY_KEY_SIZE
7:   clear kcs.sigkey over QSMP_ASYMMETRIC_SIGNING_KEY_SIZE
8:   clear kcs.ssec over QSMP_SECRET_SIZE
9:   if QSMP_ASYMMETRIC_RATCHET is not defined then
10:    clear kcs.rverkey over QSMP_ASYMMETRIC_VERIFY_KEY_SIZE
11:   end if
12:   kcs.expiration ← 0
13: end procedure

```

Duplex server ratchet sensitive reset. The Duplex server reset routine mirrors the client behavior. All local server key exchange state, including its own long-term verification key, is cleared. The remote verification key `rverkey` is only cleared when the asymmetric ratchet is disabled. With the ratchet flag enabled, `rverkey` persists so that a higher layer ratchet mechanism can retain and evolve the peer's verification key across sessions.

Algorithm 14 Duplex.ServerRatchetAwareReset

```

1: procedure DUPLEXSERVERRATCHETAWARERESET(kss)
2:   clear kss.keyid over QSMP_KEYID_SIZE
3:   clear kss.schash over QSMP_DUPLEX_SCHASH_SIZE
4:   clear kss.prikey over QSMP_ASYMMETRIC_PRIVATE_KEY_SIZE
5:   clear kss.pubkey over QSMP_ASYMMETRIC_PUBLIC_KEY_SIZE
6:   clear kss.sigkey over QSMP_ASYMMETRIC_SIGNING_KEY_SIZE
7:   if QSMP_ASYMMETRIC_RATCHET is not defined then
8:     clear kss.rverkey over QSMP_ASYMMETRIC_VERIFY_KEY_SIZE
9:   end if
10:  clear kss.verkey over QSMP_ASYMMETRIC_VERIFY_KEY_SIZE
11:  kss.expiration  $\leftarrow$  0
12: end procedure

```

Client side ratchet binding into the Duplex cookie. When preparing a Duplex connect request, the client binds the current remote verification key *rverkey* into the Duplex session cookie hash. This hash *schash* is later used as the customization input to cSHAKE for channel key derivation. As a result, changing *rverkey* (for example by advancing an asymmetric ratchet at a higher layer) will change the derived session keys even if other parameters remain fixed.

Algorithm 15 Duplex.ClientCookieHashWithRatchet

```

1: procedure DUPLEXCLIENTCOOKIEHASHWITHRATCHET(kcs)
2:   clear kcs.schash over QSMP_DUPLEX_SCHASH_SIZE
3:   initialize SHA3 state K
4:   update K with QSMP_CONFIG_STRING over QSMP_CONFIG_SIZE
5:   update K with kcs.keyid over QSMP_KEYID_SIZE
6:   update K with kcs.verkey over QSMP_ASYMMETRIC_VERIFY_KEY_SIZE
7:   update K with kcs.rverkey over QSMP_ASYMMETRIC_VERIFY_KEY_SIZE
8:   finalize K into kcs.schash over QSMP_DUPLEX_SCHASH_SIZE
9: end procedure

```

Server side ratchet binding and lookup. On the server side, the key exchange layer does not construct *rverkey* directly. Instead it calls the configured *key_query* interface with the incoming key identity. That interface is responsible for loading the current remote verification key for the client, which may have been advanced by an asymmetric ratchet. Once *rverkey* has been loaded, it is used both to verify the client signatures and to build the Duplex session cookie hash with the same ordering of fields as on the client.

Algorithm 16 Duplex.ServerLoadRatchetKeyAndCookieHash

```

1: procedure DUPLEXSERVERLOADRATCHETKEYANDCOOKIEHASH(kss, pkid)
2:   if kss.key_query(kss.rverkey, pkid) = false then
3:     return error_invalid_input
4:   end if
5:   t ← now()
6:   if t > kss.expiration then
7:     return error_key_expired
8:   end if
9:   clear kss.schash over QSMP_DUPLEX_SCHASH_SIZE
10:  initialize SHA3 state K
11:  update K with QSMP_CONFIG_STRING over QSMP_CONFIG_SIZE
12:  update K with kss.keyid over QSMP_KEYID_SIZE
13:  update K with kss.rverkey over QSMP_ASYMMETRIC_VERIFY_KEY_SIZE
14:  update K with kss.verkey over QSMP_ASYMMETRIC_VERIFY_KEY_SIZE
15:  finalize K into kss.schash
16:  return error_none
17: end procedure

```

Use of ratchet key in Duplex signature verification. In Duplex mode all client and server signatures in the connect and exchange stages are verified under the current remote verification key *rverkey*. When the asymmetric ratchet flag is enabled and *rverkey* is updated externally, the key exchange layer implicitly starts using the new ratcheted public key without any further changes. The following pseudo-code illustrates the verification pattern used in *kex.c*.

Algorithm 17 Duplex.VerifyWithRatchetKey

```

1: procedure DUPLEXVERIFYWITHRATCHETKEY(kxs, packet, khash)
2:   slen ← 0
3:   mlen ← QSMP_ASYMMETRIC_SIGNATURE_SIZE + QSMP_DUPLEX_HASH_SIZE
4:   ok ← SignatureVerify(khash, slen, packet.pmessage, mlen, kxs.rverkey)
5:   if ok = false then
6:     return error_authentication_failure
7:   end if
8:   return error_none
9: end procedure

```

2.6.2 Symmetric Ratchet State Generation

QSMP uses a simple symmetric ratchet that is initialized during the key exchange. In both Simplex and Duplex modes, the key exchange derives the initial transmit and receive keys from a cSHAKE instance, then permutes the internal Keccak state and copies part of that state into the connection state field *cns*→*rtcs*. This stored value does not equal any active channel key, and is intended to serve as the seed for future symmetric rekeying operations at the channel layer.

In the Simplex case the cSHAKE instance is keyed with the Simplex KEM secret and Simplex session cookie hash. In the Duplex case it is keyed with the pair of Duplex KEM secrets and the Duplex session cookie hash. The number of squeezed blocks and the symmetric key size constants follow *kex.c* exactly, and the ratchet seed always has size QSMP_DUPLEX_SYMMETRIC_KEY_SIZE.

Simplex symmetric ratchet seed. In Simplex mode, the symmetric ratchet seed is generated on both the client and the server immediately after the Simplex channel keys and nonces have been derived from the KEM secret and the Simplex session cookie hash. The implementation initializes a cSHAKE instance with the shared secret `ssec` and the Simplex cookie hash `schash`, squeezes two rate sized blocks into a local buffer `prnd`, permutes the Keccak state one more time so that the current channel keys are not directly stored, and then copies the first `QSMP_DUPLEX_SYMMETRIC_KEY_SIZE` bytes of the internal Keccak state into `cns->rtcs`. The buffer `prnd` is then split into the Simplex transmit and receive keys and nonces as described in the Simplex handshake pseudo-code.

Algorithm 18 Simplex.SymmetricRatchetSeed

```

1: procedure SIMPLEXYMMETRICRATCHETSEED(ssec, schash, cns)
2:   initialize Keccak state  $K$ 
3:   allocate prnd[ $2 \cdot \text{QSC\_KECCAK\_256\_RATE}$ ] as zero bytes
4:   // cSHAKE initialization:  $k = H(\text{ssec}, \text{schash})$ 
5:   cSHAKE.Init( $K$ , rate = qsc_keccak_rate_256, key = ssec, custom = schash)
6:   securely clear ssec
7:   // generate two rate blocks of key material
8:   cSHAKE.SqueezeBlocks( $K$ , rate = qsc_keccak_rate_256, out = prnd, blocks = 2)
9:   // permute so the stored state does not equal current keys
10:  KeccakPermute( $K$ )
11:  // copy next ratchet seed from internal state
12:  rtcs  $\leftarrow K.\text{state}[0 : \text{QSMP\_DUPLEX\_SYMMETRIC\_KEY\_SIZE}]$ 
13:  cns.rtcs  $\leftarrow$  rtcs
14:  // caller splits prnd into tx/rx keys and nonces
15:  derive Simplex tx and rx keys and nonces from prnd as in kex.c
16:  securely clear prnd
17: end procedure

```

In the reference implementation this logic appears in both the Simplex client exchange request handler and the Simplex server exchange response handler, with identical ordering of cSHAKE initialization, squeezing, permutation, and the `cns->rtcs` update.

Duplex symmetric ratchet seed. In Duplex mode the symmetric ratchet seed is generated on both sides after both KEM secrets `seca` and `secb` have been established and before the Duplex transmit and receive channels are initialized. The implementation initializes a cSHAKE instance with `seca`, `secb`, and the Duplex cookie hash `schash`, squeezes three rate sized blocks into `prnd`, permutes the Keccak state, and copies the first `QSMP_DUPLEX_SYMMETRIC_KEY_SIZE` bytes of the permuted state into `cns->rtcs`. The buffer `prnd` is then split into the Duplex transmit and receive keys and nonces.

Algorithm 19 Duplex.SymmetricRatchetSeed

```

1: procedure DUPLEXSYMMETRICRATCHETSEED(seca, secb, schash, cns)
2:   initialize Keccak state  $K$ 
3:   allocate prnd[ $3 \cdot \text{QSC\_KECCAK\_512\_RATE}$ ] as zero bytes
4:   // cSHAKE initialization:  $k = H(\text{seca}, \text{secb}, \text{schash})$ 
5:   cSHAKE.Init( $K$ , rate = qsc_keccak_rate_512, key1 = seca, custom = schash, key2 = secb)
6:   securely clear seca and secb
7:   // generate three rate blocks of key material
8:   cSHAKE.SqueezeBlocks( $K$ , rate = qsc_keccak_rate_512, out = prnd, blocks = 3)
9:   // permute so the stored state does not equal current keys
10:  KeccakPermute( $K$ )
11:  // copy next ratchet seed from internal state
12:  rtcs ←  $K$ .state[0 : QSMP_DUPLEX_SYMMETRIC_KEY_SIZE]
13:  cns.rtcs ← rtcs
14:  // caller splits prnd into tx/rx keys and nonces
15:  derive Duplex tx and rx keys and nonces from prnd as in kex.c
16:  securely clear prnd
17: end procedure

```

On the Duplex client side, this sequence appears in the handler that processes the server exchange response and prepares the establish request. On the Duplex server side, the same pattern appears in the handler that processes the client exchange request and prepares the exchange response. In both cases the ratchet seed stored in `cns->rtcs` is independent of the active channel keys, since the state is permuted after the key bytes have been extracted into `prnd`.

2.7 Channel Protection and Packet Processing

All application data is carried in packets that use the encrypted message flag in the header. For each outbound packet, the sender constructs the header with the appropriate flag, the current sequence number, the length of the plaintext, and the current UTC time. The header is serialized into a fixed length byte array and set as associated data for the RCS cipher. The sender then encrypts the plaintext and computes the authentication tag as part of the RCS transform, producing a ciphertext that occupies the message body. The transmit sequence counter is incremented after the packet is sent.

On reception, the peer deserializes the header and validates that the flag is an encrypted message, that the message length is within bounds, that the sequence number equals the current receive sequence counter, and that the timestamp lies inside the acceptable time window. It serializes the header to a byte array and sets it as associated data on the receive cipher. If decryption and tag verification succeed, the plaintext is delivered to the application and the receive sequence counter is incremented. Any failure in header validation or authentication results in an error and session teardown. Sequence numbers are monotonically increasing and are never reused within a session. Timestamps are interpreted relative to a configured deviation window; packets with times too far in the past or future are rejected even if authentication succeeds. This coupling of sequence numbers and timestamps to the AEAD associated data provides replay and reordering resistance under the assumption that the RCS channel meets standard AEAD security definitions.

Table 1: QSMP packet header fields and their security role.

Field	Description	Security purpose
flag	Encodes the packet type, for example Simplex or Duplex handshake step, data, keep alive, or error.	Binds the ciphertext to a specific protocol stage, which prevents a ciphertext that was valid in one stage from being replayed as a different kind of packet without failing AEAD verification.
sequence number	Monotonically increasing counter maintained per direction and incremented for every packet sent on that direction.	Detects replay and reordering within a session, since packets with reused or unexpected sequence numbers are rejected by the receiver.
payload length	Length of the encrypted payload in bytes as seen by the sender.	Protects framing of the encrypted stream. Any truncation or extension of the ciphertext changes the authenticated header and causes the AEAD tag check to fail.
timestamp	Sender supplied UTC time value associated with the packet.	Enables enforcement of a time window for packet validity and detection of stale or excessively delayed packets, under the clock skew assumptions in the model.

Table 1 makes explicit how each header field contributes to the channel security properties described in Section 2.7. The serialized header is always supplied as associated data to the RCS AEAD, so any successful modification of these fields without detection would imply a break of RCS integrity.

2.8 Error Handling, Time Validation, and Session Teardown

QSMP defines a set of error conditions that cover invalid input, key expiration, header validation failures, authentication failures, decapsulation failures, message time violations, sequence mismatches, and internal allocation failures. When a handshake function detects an error it constructs an error packet where the message body carries encrypted error code and diagnostic information, and it sends this packet with an error flag set in the header. After sending the error packet, both endpoints close the underlying transport connection and erase all connection state, including RCS keys, ratchet state, and any ephemeral secrets.

Time validation is enforced whenever a packet is received. The implementation compares the timestamp in the header to the current local time and rejects packets whose time difference exceeds a configured limit. In the handshake this prevents reuse of stale handshake messages. In the data phase it limits the window in which replayed packets can be accepted even if their sequence numbers match the expected value.

Keep alive behavior is modeled as an application level policy that periodically sends empty or minimal payload packets over an established session. These packets use the same encryption and authentication rules as normal application data and are subject to the same sequence and time validation. Failure to receive packets within a deployment specific timeout or repeated reception of invalid packets is treated as a signal that the session is no longer usable, and the implementation tears down the connection and clears all associated state.

3 Formal Protocol Specification

This section formalizes the behavior of QSMP as an abstract protocol executed between a client role C and a server role S . The specification is aligned with the engineering description but expressed in symbolic terms suitable for later definitions and proofs. All cryptographic primitives are treated as idealized interfaces, and all processing steps correspond to the logical behavior of the reference implementation.

3.1 Notation and Conventions

Let $\{0, 1\}^n$ denote the set of bitstrings of length n . Concatenation of bitstrings is written as $x \parallel y$. For a finite set X , $x \xleftarrow{\$} X$ denotes uniform sampling. For a probabilistic experiment Exp , $\Pr[\text{Exp} = 1]$ denotes its success probability.

Roles are C for client and S for server. Each party may maintain multiple sessions indexed by a local session identifier sid . A session is created whenever a party receives or generates a message that starts a handshake in either Simplex or Duplex mode. The set of all sessions at party P is denoted \mathcal{S}_P .

Each party maintains a local clock returning a value $\text{time}_P \in \mathbb{N}$. QSMP timestamps appear as integers in a fixed format and must satisfy a validity predicate

$$\text{ValidTime}(t, \text{time}_P) = 1$$

if and only if the deviation $|t - \text{time}_P|$ is below a protocol defined threshold.

A packet header is a tuple

$$H = (\text{flag}, \text{msglen}, \text{seq}, t),$$

and its serialized byte representation is written $\llbracket H \rrbracket$. The associated message body is written M . The entire packet is then the pair (H, M) .

The protocol references the following cryptographic interfaces:

- A KEM with algorithms (KGen, Enc, Dec).
- A signature scheme with algorithms (SigGen, Sign, Verify).
- A collision resistant hash function $H(\cdot)$.
- A cSHAKE based key derivation function $\text{KDF}(\cdot)$.
- An AEAD channel RCS with Enc and Dec, each taking associated data $\text{AD} = \llbracket H \rrbracket$.

3.2 Execution Model and Sessions

The protocol is executed in a multi-session environment with a single adversary controlling all network communication. The adversary may schedule activations of parties through a set of oracles defined later, and may deliver, modify, drop, or reorder packets arbitrarily. Each party $P \in \{C, S\}$ maintains local state for each active session:

$$\sigma_{P, sid} = (\text{mode}, \text{txseq}, \text{rxseq}, \text{cfg}, \text{kid}, \text{svk}, \text{ratchet}, k_{tx}, k_{rx}, n_{tx}, n_{rx}),$$

where:

- $\text{mode} \in \{\text{Simplex}, \text{Duplex}\}$,
- $\text{txseq}, \text{rxseq} \in \mathbb{N}$ are sequence counters,
- cfg is the configuration string bound into the session,
- kid is the peer key identity,

- svk is the peer verification key,
- ratchet is the current ratchet state,
- k_{tx}, k_{rx} and n_{tx}, n_{rx} are the transmit and receive keys and nonces for the AEAD channel.

Sessions begin in the *initiated* state and progress through the handshake stages described below. A session is considered *accepted* at a party once it has derived its session keys. A session is *established* when both parties reach acceptance.

3.3 Message Flows for Simplex Mode

The Simplex handshake consists of three ordered symbolic messages exchanged between client C and server S :

1. **Connect request.** The client sends:

$$C \rightarrow S : (\text{CR}, \text{kid}, \text{cfg}).$$

Upon sending, the client computes the Simplex session cookie

$$\text{sch}_S = H(\text{cfg} \parallel \text{kid} \parallel \text{svk}_S).$$

2. **Connect response.** The server generates an ephemeral KEM key pair (pk, sk) and a transcript hash

$$h = H(\llbracket H_{\text{resp}} \rrbracket \parallel \text{pk}),$$

and replies with:

$$S \rightarrow C : (\text{CRsp}, \text{pk}, \text{Sign}_S(h)).$$

The server computes the same cookie sch_S .

3. **Exchange request.** The client verifies the signature, encapsulates to obtain $(\text{cpt}, \text{sec}) = \text{Enc}(\text{pk})$, and computes:

$$(k_{tx}, n_{tx}, k_{rx}, n_{rx}) = \text{KDF}(\text{sec}, \text{sch}_S).$$

The client sends:

$$C \rightarrow S : (\text{EX}, \text{cpt}).$$

The server decapsulates $\text{sec} = \text{Dec}(\text{sk}, \text{cpt})$ and derives the same symmetric keys.

At this point the session is established.

3.4 Message Flows for Duplex Mode

The Duplex handshake consists of a connect stage, an exchange stage, and an establish stage. Both roles possess long-term signing keys and verify each other's ephemeral contributions. Let pvk_C and pvk_S denote the long-term verification keys of client and server. Define the Duplex session cookie:

$$\text{sch}_D = H(\text{cfg} \parallel \text{kid} \parallel \text{pvk}_C \parallel \text{pvk}_S).$$

1. **Client connect request.** The client samples an ephemeral KEM key pair $(\text{pk}_C, \text{sk}_C)$ and computes

$$h_C = H(\llbracket H_{\text{req}} \rrbracket \parallel \text{pk}_C).$$

It sends:

$$C \rightarrow S : (\text{CR}, \text{kid}, \text{cfg}, \text{pk}_C, \text{Sign}_C(h_C)).$$

2. Server connect response. The server samples its ephemeral pair $(\text{pk}_S, \text{sk}_S)$, verifies the client signature, computes

$$h_S = H(\llbracket H_{resp} \rrbracket \parallel \text{pk}_S),$$

and sends:

$$S \rightarrow C : (\text{CRsp}, \text{pk}_S, \text{Sign}_S(h_S)).$$

3. Exchange. Both parties perform two KEM operations so that each obtains a pair of secrets $(\text{sec}_1, \text{sec}_2)$:

$$\text{sec}_1 = \begin{cases} \text{Dec}(\text{sk}_C, \text{cpt}_1) & \text{client side,} \\ \text{Enc}(\text{pk}_C) & \text{server side,} \end{cases} \quad \text{sec}_2 = \begin{cases} \text{Enc}(\text{pk}_S) & \text{client side,} \\ \text{Dec}(\text{sk}_S, \text{cpt}_2) & \text{server side.} \end{cases}$$

The unified KDF computes:

$$(k_{tx}, n_{tx}, k_{rx}, n_{rx}) = \text{KDF}(\text{sec}_1, \text{sch}_D, \text{sec}_2).$$

4. Establish. The client encrypts the cookie:

$$C \rightarrow S : (\text{EST}, \text{Enc}_{k_{tx}}(\text{sch}_D)).$$

The server decrypts and checks equality. If valid, it returns:

$$S \rightarrow C : (\text{ESTR}, \text{Enc}_{k_{tx}}(H(\text{sch}_D))).$$

Upon matching, the client and server mark the session as established.

3.5 Partnering and Session Matching

Two sessions (P, sid) and (P', sid') are *partners* if:

1. They run in opposite roles $P \neq P'$,
2. They derive the same session cookie sch ,
3. They derive the same session keys $k_{tx}, k_{rx}, n_{tx}, n_{rx}$,
4. Their transcripts correspond to the same ordered handshake messages.

In Simplex, partnering is defined only for the client side because the server does not authenticate an identity for the client. In Duplex, partnering is symmetric.

3.6 Adversarial Interface and Oracles

The adversary controls all communication and interacts with the protocol through the following oracles:

- $\text{Send}(P, \text{sid}, m)$: delivers message m to session (P, sid) . The protocol may generate a response packet.
- $\text{RevealState}(P, \text{sid})$: returns the local state $\sigma_{P, \text{sid}}$ except for ephemeral secrets that have already been erased.
- $\text{RevealKey}(P, \text{sid})$: returns the session keys for (P, sid) if the session is accepted.
- $\text{CorruptLongTerm}(P)$: returns the long-term signing key for party P .

- **TamperHeader(H)**: allows the adversary to propose altered header values, constrained by validity of timestamps and sequence numbers.
- **Deliver(m)**: delivers an arbitrary packet constructed by the adversary to any party.

The adversary cannot violate the local time validity predicate or force a session to accept a packet with an invalid timestamp or sequence number. All AEAD decryption operations are carried out with the serialized header as associated data, and decryption failures terminate the session.

Table 2: Adversarial oracles in the QSMP security model.

Oracle	Arguments	Informal purpose
Send	Session identifier, message	Gives the adversary active control of the network. The adversary can inject, modify, and deliver messages to any local session and observe the resulting outputs.
Reveal	Accepted session	Returns the session key of an accepted session. Models post establishment key compromise and is used to define freshness in the key indistinguishability experiments.
Corrupt	Party identifier	Returns the long-term secret key material for that party and marks it as corrupted. Models full compromise of an identity and is used in the forward secrecy and authentication definitions.
Test	Fresh accepted session	Returns either the real session key or a random key of the same length, depending on a hidden bit. Defines the key indistinguishability advantage of the adversary.
Encrypt	Session, header, plaintext	Uses the established transmit key of the session to produce an RCS ciphertext and tag under the supplied header as associated data. Models chosen plaintext access to the channel.
Decrypt	Session, header, ciphertext, tag	Uses the established receive key of the session to decrypt the ciphertext under the supplied header as associated data. Returns either the plaintext or a distinguished failure symbol and models chosen ciphertext access to the channel.

Table 2 collects the adversarial interface defined in Section 3.6 into a single view. It summarizes which inputs each oracle accepts and which aspect of protocol or channel compromise it is intended to model in the subsequent security definitions and proofs.

4 Security Definitions

This section formalizes the security properties that QSMP aims to achieve. All definitions are stated within the multi-session execution model introduced earlier, with a single probabilistic polynomial time adversary controlling all network scheduling, message delivery, and corruption operations. Each security notion is expressed as an experiment whose advantage is the adversary's probability of causing a violation.

4.1 Simplex Authentication of the Server

In the SIMPLEX protocol, authentication is unidirectional and only the server possesses an asymmetric signature verification key known to the client. The session cookie binds the protocol configuration and server identity to the exchange. The session cookie is defined as:

$$\text{sch} \leftarrow H(\text{cfg} \parallel \text{kid} \parallel \text{pvk}_S)$$

where pvk_S denotes the server's public signature verification key.

Simplex mode provides one way authentication. The client authenticates the server, but the server does not authenticate the client and does not attempt to associate an identity with the initiator. This asymmetry must be reflected in the authentication experiment.

Let $\text{Exp}_{\text{AUTH-CLIENT}}^{\text{QSMP-SIMPLEX}}(A)$ be the experiment where the adversary interacts with the client and server oracles. The experiment outputs 1 only if:

1. The client session (C, sid) reaches acceptance with a derived transcript, cookie, and session keys.
2. No honest server session produces a compatible transcript. That is, there exists no (S, sid') whose session cookie and transcript match those of (C, sid) .
3. The adversary has not corrupted the server's long-term signing key prior to the acceptance of (C, sid) .

The adversary's advantage in breaking Simplex client side authentication is defined as:

$$\text{Adv}_{\text{AUTH-CLIENT}}^{\text{QSMP-SIMPLEX}}(A) = \Pr[\text{Exp}_{\text{AUTH-CLIENT}}^{\text{QSMP-SIMPLEX}}(A) = 1].$$

There is intentionally no corresponding AUTH-SERVER definition for Simplex, since the server does not establish any authenticated client identity. Any attempt by the server to attribute a unique partner identity to the client in Simplex is outside the intended security model.

4.2 Duplex Mutual Authentication

In Duplex mode both parties authenticate each other's long-term verification keys. Let a session (P, sid) be *accepted* if it reaches the established state with derived keys and cookie sch_D .

The experiment $\text{Exp}_{\text{AUTH}}^{\text{QSMP-DUPLEX}}(A)$ outputs 1 if:

1. An accepted session (P, sid) exists at some party P .
2. There is no accepted partner session (P', sid') at the peer such that:

$$\text{sch}_D^{P, \text{sid}} = \text{sch}_D^{P', \text{sid}'} \quad \text{and} \quad k_{tx}, k_{rx}, n_{tx}, n_{rx} \text{ match up to the role permutation.}$$

3. The adversary has not corrupted either party's long-term signing key before both sessions finish their respective connect stages.

The adversary's mutual authentication advantage is:

$$\text{Adv}_{\text{AUTH}}^{\text{QSMP-DUPLEX}}(A) = \Pr[\text{Exp}_{\text{AUTH}}^{\text{QSMP-DUPLEX}}(A) = 1].$$

4.3 Key Indistinguishability

Key indistinguishability (KI) captures the confidentiality of the derived session keys with respect to an adversary who may schedule sessions, observe all transcripts, and corrupt long-term keys after the handshake. The Duplex key derivation combines two independent IND CCA secure Kyber secrets, producing a KDF input of length 512-bits. Under cSHAKE_512 extraction, the resulting RCS-512 keys inherit a security level corresponding to the joint entropy of both shared secrets.

The KI experiment $\text{Exp}_{\text{KI}}^{\text{QSMP}}(A)$ proceeds as follows:

1. The adversary activates parties and may cause them to complete handshakes in Simplex or Duplex mode.
2. At most once, the adversary issues a challenge query $\text{Test}(P, \text{sid})$ to an accepted session that has not had its session keys revealed.
3. The experiment samples a random bit $b \in \{0, 1\}$. If $b = 0$, the experiment returns the real session keys (k_{tx}, k_{rx}) ; if $b = 1$, it returns uniformly random keys of equal length.
4. The adversary continues interacting and finally outputs a guess b' .

The advantage is:

$$\text{Adv}_{\text{KI}}^{\text{QSMP}}(A) = |\Pr[b' = b] - \frac{1}{2}|.$$

The experiment implicitly captures both Simplex and Duplex modes, because the source of the keying material (one or two KEM secrets) and the applicable long-term key compromise constraints differ between modes.

4.4 Forward Secrecy in Simplex and Duplex

Forward secrecy (FS) is defined relative to long-term key compromise events that occur *after* a session has completed. Since the Simplex client has no long-term key, FS in Simplex is meaningful only with respect to compromise of the server's long-term signing key. In Duplex, both parties possess long-term signing keys whose compromise affects future authentication but should not reveal past session keys.

Let $\text{Exp}_{\text{FS}}^{\text{QSMP}}(A)$ be the experiment:

1. The adversary causes a session (P, sid) to accept and then issues $\text{CorruptLongTerm}(P)$ against the corresponding role(s) after acceptance.
2. The adversary issues a $\text{Test}(P, \text{sid})$ query on that session.
3. The experiment responds using the same random bit technique as in the KI experiment.

The FS advantage is:

$$\text{Adv}_{\text{FS}}^{\text{QSMP}}(A) = |\Pr[b' = b] - \frac{1}{2}|.$$

In Simplex mode this captures secrecy of the client derived session key under post compromise leakage of the server's long-term signing key. In Duplex mode this captures secrecy under compromise of either party's long-term signing key, provided the compromise occurs after the connect stage.

4.5 Channel Confidentiality and Integrity

Channel security is defined using the AEAD security of the RCS encrypted channel. Every encrypted packet has the form (H, C) with associated data $\text{AD} = \llbracket H \rrbracket$, where the header contains seq and t which enforce ordering and time based freshness.

Confidentiality experiment. In $\text{Exp}_{\text{CHAN-CONF}}^{\text{QSMP}}(A)$, the adversary may:

- Query encryption of chosen plaintexts under chosen headers, with the restriction that it cannot cause tag verification failures or reveal session keys.
- Issue a single **TestChan** query for an unopened session, receiving either the real ciphertext or a random ciphertext of equal length.

The advantage is defined as usual by the adversary's ability to distinguish the two cases.

Integrity and replay resistance. In $\text{Exp}_{\text{CHAN-INT}}^{\text{QSMP}}(A)$, the adversary attempts to produce a forged packet (H^*, C^*) such that:

1. The receiver's time validation predicate accepts H^* .
2. The sequence number in H^* matches the receiver's expected sequence.
3. The RCS decryption and tag verification succeed.
4. The adversary did not previously obtain an encryption of the same header body pair.

Replays and reorderings necessarily violate the integrity condition because the sequence number is included in the associated data. Any successful forgery corresponds to an INT-CTXT forgery against the underlying AEAD scheme.

4.6 Ratcheting and Post Compromise Guarantees

For Duplex mode, QSMP supports asymmetric and symmetric ratchets. The ratchet state is a one way evolving value ratchet_i derived from the KDF output and any injected KEM secrets.

Define the post compromise experiment $\text{Exp}_{\text{PCS}}^{\text{QSMP}}(A)$:

1. The adversary compromises the long-term signing key of a party and observes ciphertexts under current session keys.
2. The adversary triggers a ratchet update by causing the party to execute a rekeying operation.
3. The adversary issues a challenge query on the keys derived *after* the ratchet update.

The PCS advantage is:

$$\text{Adv}_{\text{PCS}}^{\text{QSMP}}(A) = \left| \Pr[b' = b] - \frac{1}{2} \right|.$$

This definition models the ability of the protocol to recover confidentiality after compromise, assuming fresh asymmetric or symmetric entropy is injected into the KDF during ratcheting. Past keys remain unrecoverable because the ratchet state evolves in a one way fashion.

5 Assumptions on Cryptographic Primitives

QSMP relies on four classes of cryptographic primitives: a post-quantum key encapsulation mechanism, a post-quantum digital signature scheme, a hash and extendable output family, and a permutation based authenticated encryption channel. This section states the assumptions made about each primitive in the security analysis. These assumptions are standard and correspond to the security definitions under which the underlying constructions have been formally analyzed.

5.1 KEM Security

QSMP uses a post quantum key encapsulation mechanism ($\text{KGen}, \text{Enc}, \text{Dec}$) to derive one or more shared secrets during the handshake. The analysis assumes that the KEM satisfies indistinguishability under adaptive chosen ciphertext attack (IND CCA). Formally, for any probabilistic polynomial time adversary A , the KEM advantage is

$$\text{Adv}_{\text{IND-CCA}}^{\text{KEM}}(A) = |\Pr[\text{Exp}_{\text{IND-CCA}}^{\text{KEM}}(A) = 1] - \frac{1}{2}|,$$

and this advantage is assumed to be negligible. The reductions for both Simplex and Duplex key indistinguishability rely on this assumption, since the KEM shared secrets are the primary source of entropy fed into the key derivation function.

5.2 Signature Scheme Security

QSMP uses a post quantum digital signature scheme ($\text{SigGen}, \text{Sign}, \text{Verify}$) to authenticate ephemeral key material and bind handshake transcripts to long-term identities. The analysis assumes that the signature scheme provides existential unforgeability under chosen message attack (EUF CMA). For any adversary A , the advantage in producing a forgery is

$$\text{Adv}_{\text{EUF-CMA}}^{\text{SIG}}(A) = \Pr[\text{Exp}_{\text{EUF-CMA}}^{\text{SIG}}(A) = 1],$$

and this must be negligible. In Simplex mode this guarantees that an adversary cannot impersonate the server identity. In Duplex mode it guarantees that neither party can be impersonated during the connect stage and that both signatures binding the ephemeral public keys are unforgeable.

5.3 Hash and KDF Assumptions

QSMP uses the SHA3 hash function and the cSHAKE extendable output construction for transcript binding, session cookie computation, and key derivation. The security analysis makes the following assumptions.

Collision resistance. The hash function $H(\cdot)$ used for transcript hashing and session cookie computation is assumed to be collision resistant. That is, for any adversary A ,

$$\text{Adv}_{\text{CR}}^H(A) = \Pr[A \text{ finds } x \neq x' \text{ with } H(x) = H(x')]$$

is negligible. This ensures that the transcript hash and session cookies uniquely bind the handshake fields and long-term verification keys.

Random oracle or indifferentiability style behavior. The analysis requires that the session cookie value sch be indistinguishable from a random value unless the adversary produces a collision in the inputs. This is satisfied if SHA3 is treated either as a random oracle or as a sponge construction secure under the indifferentiability framework.

cSHAKE as a randomness extractor and pseudo-random function. In both Simplex and Duplex modes, the KDF is realized by cSHAKE with the KEM secret(s) as keying input and the session cookie as customization string. The analysis assumes that cSHAKE extracts pseudo-random bits from any input of high min entropy and that its output is computationally indistinguishable from uniform. Formally, for any adversary A ,

$$\text{Adv}_{\text{PRF}}^{\text{KDF}}(A) = |\Pr[A(\text{KDF}(\cdot)) = 1] - \Pr[A(U) = 1]|$$

is negligible, where U is a uniformly random function sampled from the appropriate domain. This assumption is standard for Keccak based extendable output functions and is required for the key indistinguishability and forward secrecy proofs.

5.4 RCS Channel Security

QSMP uses an RCS based authenticated encryption channel to protect all application data and the encrypted portions of the Duplex establish stage. The RCS construction is modeled as an authenticated encryption with associated data (AEAD) scheme. The analysis assumes both confidentiality and integrity under chosen ciphertext attack.

Confidentiality. For any adversary A , the AEAD confidentiality advantage is

$$\text{Adv}_{\text{IND-CPA}}^{\text{RCS}}(A) = |\Pr[\text{Exp}_{\text{IND-CPA}}^{\text{RCS}}(A) = 1] - \frac{1}{2}|,$$

and this is assumed negligible. This guarantees that encrypted packets reveal no information about their plaintexts, even under adaptive queries.

Integrity. RCS must also satisfy ciphertext integrity. For any adversary A ,

$$\text{Adv}_{\text{INT-CTXT}}^{\text{RCS}}(A) = \Pr[\text{Exp}_{\text{INT-CTXT}}^{\text{RCS}}(A) = 1]$$

is assumed negligible. Because sequence numbers, timestamps, and message lengths are included in the associated data for all encrypted packets, a successful forgery includes replay and reordering attempts, which reduce directly to a violation of AEAD integrity.

Together, these assumptions form the cryptographic foundation on which the QSMP security analysis is constructed. All reductions in later sections invoke only these primitive level properties.

6 Simplex Security Proofs

In this section we analyze the security of QSMP in Simplex mode. The client authenticates the server, derives a session key from a single KEM secret and a binding hash of public configuration data, and initializes a unidirectional encrypted channel. The server does not authenticate a client identity, so the Simplex authentication guarantee is explicitly one sided.

6.1 Client Authentication in Simplex

Recall the Simplex client authentication experiment $\text{Exp}_{\text{AUTH-CLIENT}}^{\text{QSMP-SIMPLEX}}(A)$ defined in the security definitions. The adversary succeeds if it causes an honest client session to accept without a corresponding honest server partner session, under the constraint that the server's long-term signing key is not corrupted before that acceptance.

Theorem 1 (Simplex client authentication). *Let A be any probabilistic polynomial time adversary attacking Simplex client authentication. Then there exists an adversary B against the underlying signature scheme such that*

$$\text{Adv}_{\text{AUTH-CLIENT}}^{\text{QSMP-SIMPLEX}}(A) \leq \text{Adv}_{\text{EUF-CMA}}^{\text{SIG}}(B) + \text{negl}(\lambda),$$

where λ is the security parameter and negl denotes a negligible function.

Proof sketch. We describe a reduction from any successful Simplex client authentication attack to a forgery against the server's signature scheme.

Assume that A succeeds in $\text{Exp}_{\text{AUTH-CLIENT}}^{\text{QSMP-SIMPLEX}}(A)$ with non-negligible probability. Consider the first client session (C, sid) that accepts without a matching server partner session. By definition of acceptance, this session must have:

- received a connect response containing an ephemeral KEM public key pk and a signature σ ,
- verified σ as valid on the transcript hash $h = H(\llbracket H_{resp} \rrbracket \parallel \text{pk})$,
- proceeded to the exchange stage using pk .

Since there is no matching honest server session, this signature was not produced by the honest server's signing algorithm on this transcript. Hence, the pair (h, σ) constitutes a valid existential forgery for the signature scheme, provided that the reduction can simulate all other protocol operations consistently with the view of A .

Adversary B interacts with the EUF CMA challenger for the signature scheme and simulates the QSMP environment for A . Whenever the protocol requires a signature under the server's key, B queries the signing oracle. When A produces an accepting Simplex client session without a partner server session, B outputs the corresponding (h, σ) as its forgery. The only subtlety is that header fields and ephemeral keys enter the transcript hash. The simulation ensures that these values are honestly generated or relayed, so the forgery is nontrivial.

The reduction overhead is polynomial in the running time of A , and the probability that B outputs a valid forgery is at least the probability that A wins the authentication experiment minus a negligible term that accounts for abort events in the simulation. This yields the stated bound. \square

The theorem shows that Simplex provides client side authentication of the server under the EUF CMA security of the server's signature scheme. The server side property is intentionally weaker, since the protocol does not attempt to authenticate a client identity in Simplex.

6.2 Key Indistinguishability in Simplex

We next show that under the IND CCA security of the KEM and the pseudo-randomness of the cSHAKE based KDF, the Simplex session keys are indistinguishable from random in the presence of an active adversary who controls the network and may schedule many concurrent handshakes.

Theorem 2 (Simplex key indistinguishability). *Let A be any probabilistic polynomial time adversary in the Simplex key indistinguishability experiment. Then there exist adversaries B_1 and B_2 such that*

$$\text{Adv}_{\text{KI}}^{\text{QSMP-SIMPLEX}}(A) \leq \text{Adv}_{\text{IND-CCA}}^{\text{KEM}}(B_1) + \text{Adv}_{\text{PRF}}^{\text{KDF}}(B_2) + \text{negl}(\lambda).$$

Hybrid proof sketch. We consider a sequence of hybrid games transitioning from the real Simplex execution to an ideal execution in which the test session keys are uniform.

Game G_0 . This is the real key indistinguishability experiment. The adversary interacts with honest parties running Simplex, may adaptively corrupt long-term keys subject to the usual constraints, and issues a single **Test** query on an accepted session. The test session returns either the real keys or random keys according to a hidden bit b . The adversary's advantage in distinguishing the two cases is $\text{Adv}_{\text{KI}}^{\text{QSMP-SIMPLEX}}(A)$.

Game G_1 . In G_1 we modify how the KEM secret sec is obtained for the test session. Instead of using the real encapsulation and decapsulation algorithms, we embed an IND CCA KEM challenge. When the test session is created, the reduction obtains a challenge ciphertext and one of two possible secrets from the KEM challenger, and uses these in place of (cpt, sec) . All other KEM operations are simulated honestly.

If A can distinguish G_0 from G_1 with nonnegligible probability, then we build B_1 that distinguishes real KEM secrets from random in the IND CCA experiment by using A 's decision as its own. Hence

$$|\Pr[G_0 = 1] - \Pr[G_1 = 1]| \leq \text{Adv}_{\text{IND-CCA}}^{\text{KEM}}(B_1).$$

Game G_2 . In G_2 we replace the output of the KDF on the challenge KEM secret and binding hash sch_S by uniform random keys of the same length. For all non test sessions the KDF is evaluated honestly.

If the KDF behaves as a pseudo-random function on high entropy inputs, then any difference in the adversary's view between G_1 and G_2 can be used to construct an adversary B_2 against the KDF. Therefore

$$|\Pr[G_1 = 1] - \Pr[G_2 = 1]| \leq \text{Adv}_{\text{PRF}}^{\text{KDF}}(B_2).$$

Game G_3 . In the final game we give the adversary direct access to uniformly random keys as the response to the **Test** query for the challenge session. Since in G_2 the keys used in the protocol are already uniform and independent of the KEM secret from the adversary's point of view, the distribution of transcripts and ciphertexts in G_2 and G_3 is identical. Thus

$$\Pr[G_2 = 1] = \Pr[G_3 = 1].$$

In G_3 the adversary's advantage is zero by construction, since the test session keys are independent of the bit b . Combining the hybrid steps and applying the triangle inequality yields the stated bound. \square

6.3 Forward Secrecy for Simplex

We now consider forward secrecy in the sense defined previously. In Simplex mode the client has no long-term key, so forward secrecy is meaningful only relative to compromise of the server's long-term signing key after a session completes. The goal is to show that past Simplex session keys remain indistinguishable from random even if the adversary later learns the server signing key.

Theorem 3 (Simplex forward secrecy). *Let A be any probabilistic polynomial time adversary in the forward secrecy experiment for Simplex. Suppose that A may corrupt the server's long-term signing key at any time after the completion of the target session. Then there exist adversaries B_1 and B_2 such that*

$$\text{Adv}_{\text{FS}}^{\text{QSMP-SIMPLEX}}(A) \leq \text{Adv}_{\text{IND-CCA}}^{\text{KEM}}(B_1) + \text{Adv}_{\text{PRF}}^{\text{KDF}}(B_2) + \text{negl}(\lambda).$$

Proof sketch. The proof follows the structure of the key indistinguishability argument, with the additional complication that the adversary eventually learns the server's long-term signing key.

The crucial observation is that in Simplex mode the derived session key depends only on the KEM secret and the binding hash of public values $\text{sch}_S = H(\text{cfg} \parallel \text{kid} \parallel \text{svk}_S)$. Once the handshake completes and the session key is derived, the server's signing key no longer influences any future computations. The KEM secret and the KDF output are not recomputed upon later key compromise.

We construct hybrids similar to those in the previous theorem, embedding the KEM challenge in the target session and replacing the KDF output by uniform keys. The corruption of the server signing key after the handshake completion does not change the distribution of the KEM challenge or the KDF output. It only allows the adversary to forge signatures on new transcripts, which does not retroactively affect the already derived session key.

By the same reasoning as in the key indistinguishability proof, any advantage in distinguishing the real session key from random in the forward secrecy experiment can be translated into an advantage against the KEM or the KDF. The post handshake corruption of the signing key does not provide additional information about the KEM secret beyond what is captured by the standard IND CCA model. This yields the stated bound. \square

6.4 Discussion of Limitations in Simplex

The Simplex analysis above clarifies both the guarantees and the intentional limitations of this mode.

First, authentication is strictly one sided. The client is guaranteed that any accepting Simplex session corresponds to a run with a server that holds the correct long-term signing key, except with probability bounded by the EUF CMA advantage of the signature scheme. The server does not obtain any guarantee about the identity or uniqueness of the client. Multiple clients can share the same server verification key, and the server cannot distinguish honest clients from active adversaries based solely on the Simplex handshake.

Second, forward secrecy is scoped to compromise of the server's long-term signing key and does not cover compromise of the KEM or the KDF. If the KEM fails to provide IND CCA security or if the KDF fails to behave as a pseudo-random extractor, an adversary might recover past session keys through structured attacks on the underlying primitives. These possibilities are ruled out only under the stated assumptions.

Third, the Simplex mode does not provide any protection against misuse of client side state or compromise of application level credentials. The cryptographic analysis focuses on the confidentiality and authenticity of the transport keys and the associated encrypted channel, not on higher layer authentication or authorization decisions.

Finally, the Simplex mode does not model or guarantee any notion of post compromise recovery in the sense of advanced ratcheting schemes. Once the server's signing key is compromised, future Simplex sessions that rely on that key are vulnerable to impersonation. Simplex therefore should be used in settings where the server signing key is well protected or where higher layer mechanisms can detect and respond to compromise.

7 Duplex Security Proofs

In this section we analyze QSMP in Duplex mode. Both parties possess long-term signing keys and verification keys, and both contribute ephemeral KEM key pairs. The session keys are derived from two KEM secrets and a binding hash of the configuration string and both long-term verification keys. Duplex therefore provides mutual authentication and stronger resistance to certain compromise patterns than Simplex, subject to the same primitive level assumptions.

7.1 Mutual Authentication in Duplex

Recall the experiment $\text{Exp}_{\text{AUTH}}^{\text{QSMP-DUPLEX}}(A)$ for Duplex mutual authentication. The adversary succeeds if it causes an accepted session at one party without a matching partner session at the peer, while not corrupting the relevant long-term keys before both sides complete the connect stage.

Theorem 4 (Duplex mutual authentication). *Let A be any probabilistic polynomial time adversary in the Duplex mutual authentication experiment. Then there exist adversaries B_C and B_S against the client and server signature schemes respectively such that*

$$\text{Adv}_{\text{AUTH}}^{\text{QSMP-DUPLEX}}(A) \leq \text{Adv}_{\text{EUF-CMA}}^{\text{SIG}}(B_C) + \text{Adv}_{\text{EUF-CMA}}^{\text{SIG}}(B_S) + \text{negl}(\lambda).$$

Proof sketch. The Duplex connect stage involves two signed messages. The client sends a connect request containing an ephemeral public key and a signature over the transcript hash h_C , and the server sends a connect response containing its own ephemeral public key and a signature over h_S . Both hashes bind the header fields, the configuration string, the key identity, and the ephemeral key material.

Assume that A produces an accepted session at one party that has no matching partner session at the other party. Consider the first such event in the execution.

If the unmatched session is at the client, then there must exist a connect response whose signature verifies under the server's verification key but which does not correspond to any run of the server signing algorithm on the transcript observed by the client. This provides an existential forgery against the server signature scheme. A reduction B_S simulates all other protocol behavior for A using its signing oracle, and outputs the first such forged transcript hash and signature pair.

If the unmatched session is at the server, the same reasoning applies to the client signature scheme. An adversary B_C embeds the client signing key provided by the EUF CMA challenger into the protocol simulation and uses any valid server session that accepts without a matching client session to extract a forgery against that scheme.

The probability that A wins the mutual authentication experiment is therefore bounded by the sum of the EUF CMA advantages of B_C and B_S , plus negligible terms for abort events in the simulation. This yields the claimed inequality. \square

7.2 Key Indistinguishability in Duplex

We now turn to key indistinguishability in Duplex mode. The main difference from Simplex is that the session keys are derived from two KEM secrets and a shared binding hash. This provides additional entropy and robustness, but the proof approach is similar, using a series of hybrids that reduce to IND CCA security of the KEM and pseudo-randomness of the KDF.

Theorem 5 (Duplex key indistinguishability). *Let A be any probabilistic polynomial time adversary in the Duplex key indistinguishability experiment. Then there exist adversaries B_1 , B_2 against the KEM and B_3 against the KDF such that*

$$\text{Adv}_{\text{KI}}^{\text{QSMP-DUPLEX}}(A) \leq \text{Adv}_{\text{IND-CCA}}^{\text{KEM}}(B_1) + \text{Adv}_{\text{IND-CCA}}^{\text{KEM}}(B_2) + \text{Adv}_{\text{PRF}}^{\text{KDF}}(B_3) + \text{negl}(\lambda).$$

Hybrid proof sketch. We define a sequence of games.

Game G_0 . This is the real Duplex key indistinguishability experiment. The adversary interacts with honest parties, may cause many Duplex sessions to complete, and selects one accepted session for the test query.

Game G_1 . In G_1 we embed an IND CCA challenge for the first KEM secret sec_1 of the test session. Instead of computing $(\text{cpt}_1, \text{sec}_1)$ using the real KEM algorithms, the reduction obtains an IND CCA challenge ciphertext and one of two possible secrets from the KEM challenger. All other KEM operations are carried out honestly. If A can distinguish G_0 from G_1 , we construct an adversary B_1 that wins the KEM IND CCA game.

Game G_2 . In G_2 we similarly embed an IND CCA challenge for the second KEM secret sec_2 of the test session. The reduction uses the KEM challenger output in place of the real encapsulation or decapsulation results for this secret. All other operations are unchanged. Any distinguishability between G_1 and G_2 yields an adversary B_2 against the KEM.

Game G_3 . In G_3 we keep the challenge KEM secrets as in G_2 but replace the KDF output for the test session by uniformly random keys of the same length. Specifically, instead of computing

$$(k_{tx}, n_{tx}, k_{rx}, n_{rx}) = \text{KDF}(\text{sec}_1, \text{sch}_D, \text{sec}_2),$$

we sample a tuple of keys and nonces from the uniform distribution. For all non test sessions the KDF is evaluated honestly. Any distinguisher between G_2 and G_3 implies an adversary B_3 against the pseudo-randomness of the KDF.

Game G_4 . In the final game, the keys returned in response to the test query are replaced by freshly sampled uniform keys that do not participate in any encryption operation. The transcript distribution and all ciphertexts observed by the adversary remain unchanged from G_3 , because the KDF outputs for the test session were already uniform and independent. Therefore $\Pr[G_3 = 1] = \Pr[G_4 = 1]$ and the adversary's advantage in G_4 is zero. Combining the differences across games by the triangle inequality and applying the bounds provided by the KEM and KDF assumptions yields the stated result. \square

7.3 Forward Secrecy and Ratcheting in Duplex

Duplex mode supports both forward secrecy with respect to long-term key compromise and post compromise recovery through ratcheting. After the initial handshake, new KEM secrets or symmetric inputs can be injected into the KDF along with the ratchet state in order to derive new session keys while erasing old ones.

Theorem 6 (Duplex forward secrecy and ratcheting). *Let A be any probabilistic polynomial time adversary in the forward secrecy and post compromise security experiments for Duplex.*

Suppose that:

- *the KEM is IND CCA secure,*
- *the KDF behaves as a pseudo-random function and extracts entropy from its inputs,*
- *ratchet updates incorporate either fresh KEM entropy or high entropy symmetric inputs.*

Then there exist adversaries B_1, B_2, B_3 such that

$$\begin{aligned} \text{Adv}_{\text{FS}}^{\text{QSMP-DUPLEX}}(A) + \text{Adv}_{\text{PCS}}^{\text{QSMP-DUPLEX}}(A) &\leq \text{Adv}_{\text{IND-CCA}}^{\text{KEM}}(B_1) + \text{Adv}_{\text{PRF}}^{\text{KDF}}(B_2) \\ &\quad + \text{Adv}_{\text{CR}}^H(B_3) + \text{negl}(\lambda). \end{aligned}$$

Proof sketch. Forward secrecy for the initial Duplex session follows the same pattern as in Simplex and is covered by Theorem 5, since compromise of long-term signing keys after acceptance does not reveal the KEM secrets or retroactively change the KDF output.

For post compromise security, we consider a scenario where the adversary learns both long-term signing keys and even the current session keys at some time t , then a ratchet update is performed at a later time $t' > t$. The ratchet state ratchet_i is a one way function of previous KDF outputs and any previously injected secrets. At the update step, a fresh KEM secret or high entropy symmetric input r is combined with ratchet_i and the session cookie sch_D inside the KDF to produce new keys and a new ratchet state ratchet_{i+1} .

Under the IND CCA assumption on the KEM, the new secret r is indistinguishable from random even in the presence of past compromise, as long as the new KEM operation is carried out with uncompromised ephemeral keys. Under the pseudo-randomness and extractor properties of the KDF, the output keys after the ratchet update are

indistinguishable from random even given the previous keys and ratchet state. Collision resistance of the hash function that defines the cookie ensures that the binding to long-term verification keys remains sound.

Therefore any adversary that can distinguish post ratchet keys from random, or that can use past compromises to break confidentiality of post ratchet traffic, can be turned into an adversary against the KEM, the KDF, or the hash function. This yields the bound stated in the theorem. \square

7.4 Replay and Reordering Resistance

Finally we formalize the replay and reordering resistance properties of the Duplex channel. These properties arise from the use of sequence numbers and timestamps as associated data for the RCS AEAD, together with the sequence number and time window checks enforced by the implementation.

Theorem 7 (Replay and reordering resistance). *Assume that RCS provides ciphertext integrity as an AEAD scheme and that the receiver enforces monotonic sequence numbers and valid time windows. Then any adversary that causes an honest party to accept a replayed or reordered application packet in Duplex mode can be turned into an adversary that forges an RCS ciphertext. In particular,*

$$\text{Adv}_{\text{CHAN-INT}}^{\text{QSMP-DUPLEX}}(A) \leq \text{Adv}_{\text{INT-CTXT}}^{\text{RCS}}(B) + \text{negl}(\lambda).$$

Proof sketch. Consider an adversary A that succeeds in delivering a replayed or reordered packet that is accepted by the receiver as valid. Let (H^*, C^*) be the first such packet. Since the receiver enforces a strictly increasing sequence counter for each session and uses the serialized header $\llbracket H^* \rrbracket$ as associated data, acceptance implies that:

- the sequence number in H^* equals the current expected receive sequence,
- the timestamp in H^* satisfies the time validity predicate,
- RCS decryption under the session key with associated data $\llbracket H^* \rrbracket$ produces a valid plaintext and tag.

If (H^*, C^*) was previously output by the sender's encryption algorithm, then acceptance does not represent a successful attack because the sequence number would already have been advanced and the packet either would be rejected as stale or violate the monotonicity check. Thus the first accepted replay or reorder cannot coincide with a previously generated packet and must be a new ciphertext and header pair from the AEAD point of view.

We can therefore define an adversary B against RCS that uses A to obtain a valid ciphertext and header pair that is accepted by the decryption oracle without having been produced by the encryption oracle. From the AEAD perspective this is an INT CTXT forgery. The reduction simulates all honest encryptions for A , forwards decryption attempts to the RCS challenger, and when A succeeds in convincing the receiver to accept a new packet, B outputs that packet as its forgery.

This shows that any nonnegligible advantage for A in breaking replay or reordering resistance would contradict the assumed INT CTXT security of RCS, up to negligible simulation error. \square

These theorems together establish that Duplex mode provides mutual authentication, session key indistinguishability, forward secrecy with post compromise recovery through ratcheting, and strong channel integrity, under the standard assumptions stated earlier.

8 Channel Security and AEAD Binding

The security of the QSMP data channel relies on the correct use of the RCS authenticated encryption scheme with packet headers bound as associated data. This section formalizes that use, clarifies how replay and reordering attacks are captured as AEAD forgeries, and discusses what aspects of denial of service and liveness are outside the cryptographic model.

8.1 AEAD Model for QSMP

Let $\text{AEAD} = (\text{Enc}, \text{Dec})$ be an authenticated encryption scheme with associated data. For each established session in QSMP, the parties share:

- a transmit key and nonce (k_{tx}, n_{tx}) ,
- a receive key and nonce (k_{rx}, n_{rx}) ,
- a pair of sequence counters $(\text{txseq}, \text{rxseq})$.

Each application packet is represented as a pair (H, M) , where

$$H = (\text{flag}, \text{msglen}, \text{seq}, t)$$

is the header and M is the plaintext payload. The serialized header

$$\text{AD} = \llbracket H \rrbracket$$

is used as associated data for AEAD. The sender computes

$$C = \text{Enc}_{k_{tx}}(n_{tx}, M, \text{AD})$$

and transmits the packet (H, C) . The receiver, upon accepting H as syntactically valid and satisfying local checks on seq and t , performs

$$M' = \text{Dec}_{k_{rx}}(n_{rx}, C, \text{AD}),$$

and accepts the plaintext only if decryption returns a non error symbol.

The abstract model assumes that:

1. The same pair (k_{tx}, k_{rx}) is never reused across distinct sessions.
2. For a fixed session and direction, nonces and sequence numbers are never reused.
3. Every acceptance event corresponds to a single successful AEAD decryption with the correct associated data.

Under these conditions, the channel security analysis reduces to the standard confidentiality and integrity properties of the underlying AEAD scheme.

8.2 Replay and Reordering as Forgery Events

We formalize how replay and reordering attacks on QSMP map to the ciphertext integrity game $\text{Exp}_{\text{PINT-CTXT}}^{\text{AEAD}}$ for RCS. An adversary is considered successful if it causes an honest party to accept a packet whose ciphertext and associated data pair (C^*, AD^*) was not previously output by any honest encryption query for that party and direction.

In QSMP, two types of attacks are of interest:

- *Pure replays*, where an adversary resends an old packet (H, C) to the receiver.

- *Reorderings or header tampering*, where the adversary changes seq or t while reusing the same ciphertext C , possibly with a modified header H^* .

Let (H, C) be a packet originally produced by $\text{Enc}_{k_{tx}}$ with associated data $\text{AD} = \llbracket H \rrbracket$. When the adversary sends (H, C) again without modification, the receiver will reject it because the sequence number in H no longer matches the expected rxseq . Even if the timestamp remains within the valid window, the sequence check fails and the packet is discarded before AEAD decryption. Such replays therefore do not impact AEAD security and are handled entirely by local state checks.

Now consider an adversary that attempts to reorder packets or change header fields. The adversary must produce a pair (H^*, C^*) such that:

1. $\text{AD}^* = \llbracket H^* \rrbracket$ passes the receiver's syntactic checks and satisfies $\text{ValidTime}(t^*, \text{time}_P) = 1$.
2. The sequence number seq^* equals the receiver's current rxseq .
3. Decryption under the session key and nonce with associated data AD^* succeeds.
4. The pair (C^*, AD^*) was not previously produced by an honest encryption call in this direction.

Any successful replay or reordering attack on QSMP that results in acceptance must therefore correspond to a successful INT CTXT forgery against the AEAD scheme with respect to the pair (C^*, AD^*) . This is because the receiver's sequence and time checks rule out acceptance of any previously seen encryption output. Thus an adversary who can cause an honest party to accept a replayed or reordered packet distinct from all prior encryptions yields an adversary in the AEAD integrity game with essentially the same advantage.

8.3 Denial of Service and Liveness Considerations

The formal model and reductions capture confidentiality and integrity of the QSMP channel, including resistance to replay and reordering attacks that aim to inject forged ciphertexts. However, several aspects of denial of service (DoS) and liveness lie outside the scope of the cryptographic analysis.

First, the adversary is allowed to drop, delay, or reorder packets arbitrarily. The model does not attempt to guarantee that honest parties will eventually complete handshakes or deliver application messages. It only guarantees that if a packet is accepted, then it satisfies the integrity and freshness properties implied by AEAD security and the local sequence and time checks.

Second, the use of timestamps and sequence numbers introduces explicit policies for session timeouts and packet freshness. These policies are modeled via the ValidTime predicate and the monotonic sequence counters, but the choice of thresholds and retry strategies is a deployment decision. Parameters that are too strict may cause honest sessions to fail under network delay, while parameters that are too loose may increase the window for benign replays. Balancing these trade offs is an operational matter rather than a cryptographic one.

Third, error handling and logging on authentication failures can be abused by an adversary to trigger resource exhaustion or connection churn. The analysis assumes that sessions which encounter repeated failures are torn down and that implementations enforce global limits on concurrent sessions and queued connections. These measures mitigate DoS but are not modeled in the security games.

In summary, the formal treatment of AEAD binding in QSMP ensures that accepted packets are authentic and fresh, but it does not guarantee availability. Liveness and

robustness against resource level DoS attacks remain system design concerns that must be addressed through transport level mechanisms, rate limiting, and deployment specific policies.

9 Cryptanalytic Evaluation

This section examines QSMP from a cryptanalytic perspective beyond the idealized reductions. The focus is on practical attack surfaces, the effect of long-term key compromise under different patterns, and how QSMP compares at a high level with related secure messaging and key exchange protocols.

9.1 Attack Surfaces and Adversarial Capabilities

The formal model gives the adversary full control over the network, access to key corruption oracles, and the ability to interact with many concurrent sessions. In practice, additional attack strategies must be considered.

Transcript manipulation and downgrade attempts. QSMP does not negotiate the configuration string `cfg` inside the handshake. Instead, the configuration is fixed by deployment and included in the session cookie computation and transcript hashes. This choice reduces downgrade risks that arise when multiple suites are negotiated interactively. An adversary who attempts to modify `cfg` in transit must either forge signatures on altered transcripts or produce collisions in the hash computations that define `sch`. Both strategies are bounded by the EUF CMA and collision resistance assumptions.

Key identity confusion. The key identity `kid` is a short fixed length identifier mapped to long-term verification keys. An adversary who can trick a client into associating the wrong verification key with a given `kid` could redirect connections to an unintended server. This risk is orthogonal to the protocol design and depends on the correctness of the external key distribution and binding mechanisms, such as certificates or configuration databases. Once the mapping from `kid` to `svk` is fixed, the binding of `kid` inside the session cookie and transcript hashes prevents silent substitution on the wire.

Side channel leakage. The analysis treats KEM, signature, and RCS operations as ideal black boxes with no side channel leakage. In practice, timing, cache, or power side channels could reveal information about long-term or ephemeral secrets. Mitigation requires constant time implementations, masking, and other engineering level defenses. The security model does not account for such leakage, and attacks that exploit side channels fall outside the formal guarantees.

Randomness quality. QSMP depends on high quality randomness for KEM key generation, encapsulation randomness where applicable, and signature operations. Weak random number generators can lead to predictable ephemeral keys, repeated KEM secrets, or biased signatures. These failures would undermine the assumptions used in the reductions and could enable key recovery attacks. The protocol assumes that the underlying system provides sufficient entropy and that random number generation is correctly implemented and seeded.

Implementation bugs and parsing errors. The protocol uses a compact binary header format with length fields and timestamps. Incorrect parsing, integer overflow, or failure to validate lengths and reserved values can lead to memory safety vulnerabilities or

logic bypasses that are not captured by the symbolic model. The cryptographic analysis assumes correct implementation of the specified checks, including strict enforcement of message sizes and rejection of unsupported flags.

9.2 long-term Key Compromise and Ratchet Behavior

QSMP separates long-term signing keys from session keys that are derived from KEM secrets and cookies. This separation limits the effect of various compromise patterns.

Server signing key compromise. In Simplex mode, compromise of the server signing key before a handshake allows full impersonation to clients, but does not directly reveal past KEM secrets or session keys. In Duplex mode, compromise of one party’s signing key before the connect stage enables impersonation of that party in future runs but does not retroactively affect past sessions. The formal forward secrecy results show that past session keys remain secure as long as the KEM and KDF assumptions hold.

Ephemeral KEM key compromise. If an attacker obtains an ephemeral private KEM key sk for a specific session, it can recover the corresponding secret for that handshake. This exposes that session’s keys, but because the KEM key pairs are generated per session and erased after key derivation, the damage is localized. In Duplex mode, compromise of a single ephemeral key reveals only one of the two KEM secrets, and the combined KDF input still includes entropy from the other secret.

Session key compromise and ratchet recovery. If a Duplex session is configured with an asymmetric or symmetric ratchet, and an attacker compromises the current session keys, the ratchet mechanism can provide recovery. Once a new ratchet step injects fresh KEM or symmetric entropy into the KDF, subsequent keys become independent of the previously compromised values. The ability to recover depends on correct implementation of key erasure and on the assumption that at least one fresh high entropy input is available. If the attacker can persistently compromise every new ephemeral key, then the ratchet cannot restore confidentiality.

Cookie and transcript binding under compromise. The session cookies sch_S and sch_D bind configuration strings and long-term verification keys to the KEM secrets. Compromise of long-term verification keys does not allow an attacker to retroactively alter the cookie for a past session without finding collisions in the hash function. This property is important for replay defenses in Duplex mode, where the establish stage confirms that both parties hold the same cookie value before accepting the session as fully authenticated.

9.3 Comparison with Related Protocols

QSMP occupies a design space similar to other post quantum aware secure channel protocols, while making different trade offs in handshake structure and channel binding.

Relation to SSH and TLS. Traditional SSH and TLS deployments rely on classical key exchange and signature algorithms and often support a large number of negotiated cipher suites. QSMP adopts a more constrained approach. It fixes a post quantum KEM and signature scheme per configuration and avoids interactive cipher suite negotiation. This simplifies downgrade analysis, since there is no path for an attacker to force a weaker suite within the protocol itself. The use of a compact binary header with explicit sequence numbers and timestamps is conceptually similar to record oriented protocols but is more tightly coupled to the AEAD associated data.

Relation to Noise and modern secure messaging frameworks. Noise based protocols and modern secure messaging systems such as Signal emphasize a clear separation between handshake patterns, key derivation, and ratcheting. QSMP shares this modular structure. It uses explicit handshake stages in Simplex and Duplex mode, a Keccak based KDF that incorporates multiple secrets and binding data, and a ratchet state that supports rekeying and post compromise recovery. Unlike generic frameworks that allow many pattern instantiations, QSMP fixes a small number of handshake flows and hard codes the interpretation of flags and header fields, which simplifies analysis but reduces flexibility.

Post quantum focus and cryptographic assumptions. QSMP is designed from the outset for post quantum security. Its core assumptions are the IND CCA security of a post quantum KEM, the EUF CMA security of a post quantum signature scheme, and well studied properties of SHA3 and cSHAKE. The RCS channel relies on Keccak style permutation based encryption rather than block ciphers. This contrasts with protocols that layer post quantum KEMs on top of classical designs or that retain classical signatures for authentication. By binding both long-term verification keys and configuration strings into the cookie and KDF inputs, QSMP attempts to give a clear and explicit mapping from primitive level assumptions to end to end guarantees.

Overall, the cryptanalytic evaluation indicates that QSMP does not introduce unusual structural weaknesses relative to established secure channel designs, provided that implementations satisfy the constant time and randomness requirements and that key distribution mechanisms correctly bind key identities to verification keys. Remaining risks arise primarily from implementation quality, side channels, and system level misconfiguration, rather than from the cryptographic design itself.

10 Implementation Conformance and Side Channel Considerations

This section examines the relationship between the formal model and the reference implementation of QSMP, the side channel assumptions required by the analysis, and operational considerations relating to randomness, time synchronization, and error handling. The goal is to show that the implementation follows the abstract specification closely enough for the formal results to apply, and to identify the engineering properties that the formal model treats as idealizations.

10.1 Mapping Between Model and Reference Implementation

The reference implementation follows the symbolic protocol specification with a direct correspondence between code level functions and formal handshake stages.

Handshake structure. The Simplex and Duplex modes each implement a sequence of connect, exchange, and (for Duplex) establish stages. Each stage corresponds to the symbolic message flows defined earlier. The implementation uses explicit flag values to represent message types and constructs packet bodies in the exact order required by the formal specification.

Session cookies and transcript binding. The session cookies sch_S and sch_D are computed as hashes of configuration strings, key identities, and verification keys. The code computes these values using SHA3 in the same order and with the same input framing as in the symbolic model. Transcript hashes for signed messages include the serialized header and the sender's ephemeral public key, consistent with the formal definition of h_C and h_S .

Key derivation and ratchet state. The KDF is implemented through cSHAKE with the KEM secret or secrets as keying input and the session cookie as customization string. The output buffer is partitioned into RCS keys, nonces, and a ratchet state extracted from the post permutation Keccak state. This matches the formal specification of $\text{KDF}(\text{sec}, \text{sch})$ in Simplex and of $\text{KDF}(\text{sec}_1, \text{sch}_D, \text{sec}_2)$ in Duplex.

AEAD usage and associated data. In the implementation, the header is serialized into a contiguous byte array and provided to RCS as associated data for both encryption and decryption. Decryption fails if tag verification fails or if the internal nonce and sequence number state does not match expectations. This behavior corresponds exactly to the formal AEAD model described earlier.

State transition and cleanup. The code erases ephemeral KEM private keys, shared secrets, and derived symmetric keys once they are no longer needed, aligning with the formal assumption that ephemeral values are erased immediately after use. Session state transitions follow the order specified in the model, ensuring that no packet is accepted out of order or with stale transcript data.

10.2 Constant Time Requirements

The formal analysis assumes that QSMP’s cryptographic operations do not leak information through timing or other side channels. To satisfy this assumption, the implementation must enforce constant time or time independent behavior in several places.

KEM decapsulation. The KEM decapsulation operation must run in constant time with respect to the input ciphertext. In particular, decapsulation failures must be handled without revealing whether the failure was caused by an invalid ciphertext or by an invalid public key. This prevents chosen ciphertext timing attacks that could recover the ephemeral secret or allow oracle based key recovery.

Signature verification. Verification operations should avoid control flow branches on secret dependent values. The attacker should not be able to distinguish verification failures arising from malformed packets versus failures from forged signatures based on timing.

AEAD decryption. The RCS tag comparison must be constant time, and the implementation must not leak whether decryption failed early due to header checks or late due to authentication failure. Ideally, decryption attempts proceed using a constant time comparison after the header has been validated syntactically and temporally.

Equality checks. All equality tests involving sch_D , its hashed confirmation value $H(\text{sch}_D)$, or any other sensitive field must be performed with constant time comparison functions to avoid leaking partial equality or length dependent information.

10.3 Random Number Generation and Time Synchronization

Several operations in QSMP rely on high quality entropy or synchronized time.

Randomness requirements. Secure randomness is required for:

- generation of ephemeral KEM key pairs,
- encapsulation randomness where required by the KEM,

- signing randomness where required by the signature scheme,
- generation of fresh entropy for ratchet updates.

If any of these sources produce low entropy values or repeat outputs, session keys may become predictable or attackers may be able to identify relationships across sessions. The formal model assumes perfect randomness, so real deployments must use cryptographically secure random number generators backed by system entropy sources.

Time synchronization. QSMP uses timestamps in packet headers, and the receiver validates that each packet lies within a bounded time window. Loose synchronization between peers is required. If clocks drift too far apart, legitimate packets may fail the `ValidTime` predicate, causing liveness issues. If the window is too large, adversaries gain a marginally larger window to replay old packets that still fall within the time bounds but will nevertheless be rejected based on sequence numbers. The time bound therefore must be calibrated for the intended deployment environment.

Sequence numbers and strict monotonicity. Both the transmit and receive sequence numbers must be strictly monotonic during the lifetime of a session. If an implementation incorrectly increments or resets sequence values, an attacker might bypass replay detection or force undefined behavior. The formal model presumes strict monotonicity and relies on it for channel integrity.

10.4 Error Handling, Logging, and Teardown

Error handling behavior affects both security and robustness. The formal model treats decryption failures and handshake verification failures as terminal events that erase state. The implementation conforms to this behavior.

Error generation and propagation. On detecting an error in header validation, time validation, sequence checking, signature verification, or AEAD decryption, the implementation constructs an error packet indicating the failure cause and sends it to the peer if the session state permits. Immediately afterward, it clears all cryptographic state and tears down the connection. This prevents use of keys after validation failures and limits the scope of adversarial influence.

Logging. The implementation includes logging hooks for failures. While logging provides valuable diagnostic information, care must be taken that log messages do not leak sensitive data or timing patterns visible to external observers. Logs should not contain plaintext fragments, key identifiers beyond what is necessary for debugging, or detailed error codes that could reveal parsing decisions.

Teardown behavior. After any unrecoverable error, the implementation resets session state, clears RCS keys, ratchet state, ephemeral keys, and local buffers. This aligns with the formal assumption that state is erased upon error detection. Teardown behavior prevents adversaries from exploiting partially initialized or stale state in subsequent protocol operations.

In summary, the reference implementation tracks the formal model closely in its cryptographic structure, message processing, and key derivation logic. While the formal analysis abstracts away many engineering details, the implementation must satisfy strict constant time, randomness, and state erasure requirements for the formal guarantees to hold in practice.

11 Concrete Security Estimates

This section provides concrete security interpretations for the QSMP construction. The goal is not to produce exact security numbers, but to show how the reductions and assumptions translate into practical confidence levels when instantiated with standard post quantum primitives.

Table 3 summarizes the quantitative security levels of the primitives used in QSMP and identifies the dominant computational assumptions for each. All values correspond to the parameter sets defined in Section 11.1.

11.1 Parameter Choices and Security Levels

QSMP is parameterized by a configuration string `cfg` that selects fixed cryptographic primitives. A typical configuration pairs:

- a post quantum KEM such as Kyber or a code based KEM at a target level comparable to NIST Category 3 or Category 5,
- a post quantum signature scheme such as Dilithium or SPHINCS that meets EUF CMA at the same target level,
- SHA3 and cSHAKE with capacity and rate parameters that match 256-bit or 512-bit strength,
- the RCS authenticated channel built on the Keccak permutation, configured to provide either 256-bit or 512-bit AEAD security.

When these primitives are instantiated with Category 3 or Category 5 parameters, their concrete resistance to classical and quantum adversaries is on the order of 2^{160} operations or greater for Category 3, and on the order of 2^{200} operations or greater for Category 5. Because QSMP composes these primitives in a hybrid structure, the effective security level for the session keys is dominated by the weakest primitive. For Duplex, the combination of two KEM secrets can raise the entropy of the KDF input but does not raise the security level beyond the bound given by the lowest strength KEM input.

Table 3: Security levels of QSMP primitives and dominant assumptions.

Primitive	Parameter set	Security level	Dominant assumption
KEM	Kyber (configured)	≈ 256 bits	IND CCA hardness of MLWE
Signature	Dilithium (configured)	≈ 256 bits	EUF CMA hardness of MLWE
Hash (Simplex)	SHA3_256	256-bit preimage	Sponge indistinguishability
Hash (Duplex)	SHA3_512	512-bit preimage	Sponge indistinguishability
KDF (Simplex)	cSHAKE_256	256-bit output	Pseudo-randomness from SHA3 permutation
KDF (Duplex)	cSHAKE_512	512-bit output	Pseudo-randomness from SHA3 permutation
AEAD	RCS (Simplex)	256-bit integrity	Strength of tag, associated data binding
AEAD	RCS (Duplex)	512-bit integrity	Strength of tag, associated data binding
Session cookie (Simplex)	SHA3_256	256-bit hash	Collision resistance
Session cookie (Duplex)	SHA3_512	512-bit hash	Collision resistance

11.2 Quantitative Bounds from the Reductions

The reductions provide explicit bounds on the adversarial advantage in each security game. For illustration, suppose that:

- the underlying KEM achieves IND CCA advantage at most ϵ_{KEM} ,
- the signature scheme achieves EUF CMA advantage at most ϵ_{SIG} ,
- the cSHAKE based KDF behaves as a pseudo-random function with advantage at most ϵ_{KDF} ,
- the AEAD channel has confidentiality and integrity advantages at most $\epsilon_{\text{RCS,conf}}$ and $\epsilon_{\text{RCS,int}}$.

Then, ignoring negligible simulation terms, the reductions give the following upper bounds:

Simplex client authentication.

$$\text{Adv}_{\text{AUTH-CLIENT}}^{\text{QSMP-SIMPLEX}} \leq \epsilon_{\text{SIG}}.$$

Simplex key indistinguishability.

$$\text{Adv}_{\text{KI}}^{\text{QSMP-SIMPLEX}} \leq \epsilon_{\text{KEM}} + \epsilon_{\text{KDF}}.$$

Simplex forward secrecy.

$$\text{Adv}_{\text{FS}}^{\text{QSMP-SIMPLEX}} \leq \epsilon_{\text{KEM}} + \epsilon_{\text{KDF}}.$$

Duplex mutual authentication.

$$\text{Adv}_{\text{AUTH}}^{\text{QSMP-DUPLEX}} \leq 2\epsilon_{\text{SIG}}.$$

Duplex key indistinguishability.

$$\text{Adv}_{\text{KI}}^{\text{QSMP-DUPLEX}} \leq 2\epsilon_{\text{KEM}} + \epsilon_{\text{KDF}}.$$

Duplex forward secrecy and post compromise recovery.

$$\text{Adv}_{\text{FS}}^{\text{QSMP-DUPLEX}} + \text{Adv}_{\text{PCS}}^{\text{QSMP-DUPLEX}} \leq \epsilon_{\text{KEM}} + \epsilon_{\text{KDF}} + \epsilon_{\text{CR}},$$

where ϵ_{CR} is the collision bound for SHA3.

Channel confidentiality and integrity.

$$\text{Adv}^{\text{QSMP-CHAN-CONF}} \leq \epsilon_{\text{RCS,conf}}, \quad \text{Adv}^{\text{QSMP-CHAN-INT}} \leq \epsilon_{\text{RCS,int}}.$$

These bounds remain valid across many concurrent sessions, provided that each session uses fresh KEM secrets, fresh KDF calls, and non overlapping key material. If the adversary performs q sessions or queries, standard hybrid arguments give linear scaling, for example

$$\text{Adv}_{\text{KI}}^{\text{QSMP-SIMPLEX}}(A_q) \leq q\epsilon_{\text{KEM}} + q\epsilon_{\text{KDF}}.$$

This scaling reflects the fact that each session independently uses the KEM and KDF in a manner that does not share secrets with other sessions.

11.3 Discussion of Security Margins

Practical security margins depend on the exact primitives chosen.

KEM and signature margins. Modern post quantum KEMs and signature schemes have undergone extensive cryptanalytic evaluation. Their concrete security levels reflect both classical and quantum attack costs. When using Category 3 or Category 5 parameters, the margin against the best known attacks is significant, with no known shortcuts that substantially reduce security beyond the NIST estimates. QSMP inherits these margins directly, since its session key entropy flows from the KEM secrets.

Hash and KDF margins. The SHA3 family and cSHAKE have strong indifferentiability and collision resistance bounds. For 256-bit output configurations, collision searches require on the order of 2^{128} operations. This provides ample margin for both transcript binding and cookie computation. The KDF’s pseudo-randomness margin is similarly strong, because the full Keccak permutation is used with appropriate rate and capacity settings.

AEAD channel margins. The RCS channel relies on Keccak and a domain separated duplex construction. Its integrity and confidentiality rely on the security of the underlying permutation. Keccak has a large security margin, and no practical attacks threaten the capacity used by RCS in the QSMP configuration. As long as keys and nonces are not reused and associated data inputs are correct, the AEAD channel inherits these margins.

Multiplicative effects of composition. In composed protocols it is common for reductions to lose security bits due to multiple uses of primitives. QSMP avoids heavy multiplicative losses by using only one or two KEM secrets and one KDF invocation per session, and by avoiding symmetric key reuse. As a result, the overall security parameter is close to that of the weakest primitive, rather than suffering significant reduction from composition.

Operational margins. Time windows for packet acceptance and bounds on sequence numbers do not reduce cryptographic security, but they do influence robustness. Adversarial manipulation of timing may cause disruptions but does not significantly change confidentiality or integrity guarantees. Implementations should treat time windows conservatively to avoid unnecessary rejection of legitimate packets.

In summary, QSMP retains the concrete security levels of its underlying post quantum primitives, with no significant reduction introduced by the protocol structure. When instantiated with conservative parameter sets, the protocol achieves security levels suitable for long-term confidentiality against both classical and quantum attackers.

12 Conclusion

12.1 Summary of Results

This paper presented a formal and cryptanalytic analysis of the Quantum Secure Messaging Protocol, covering both its Simplex and Duplex operating modes. The study introduced a precise engineering level specification derived directly from the reference implementation, then developed a symbolic protocol model suitable for rigorous cryptographic reasoning. Within this model, we defined authentication, key indistinguishability, forward secrecy, channel integrity, and post compromise recovery, and we proved that QSMP satisfies these properties under standard assumptions.

For Simplex mode, we established unilateral authentication of the server to the client based on the existential unforgeability of the server’s signature scheme. We proved that session keys achieve indistinguishability under attacks permitted by the multi session model, and that forward secrecy holds for past sessions when the server’s long-term signing key is compromised after the handshake. For Duplex mode, we proved mutual authentication based on the unforgeability of both parties’ signatures, demonstrated that session keys remain indistinguishable from random under adaptive adversaries, and showed that the ratchet design supports recovery from certain compromise patterns when fresh entropy is injected.

In both modes, we showed that data channel confidentiality and integrity reduce to the AEAD properties of the RCS construction when the packet header is included as associated data. The sequence and timestamp validation enforced by the implementation ensure that replay and reordering attacks translate to INT CTXT forgery attempts. The cryptanalytic evaluation confirmed that QSMP’s security derives cleanly from its underlying KEM, signature scheme, hash, and AEAD components, with no unusual structural vulnerabilities introduced by the protocol design.

12.2 Limitations and Caveats

The analysis relies on several idealizations that must be recognized when interpreting the results. The multi session model assumes perfect randomness, correct enforcement of monotonic sequence numbers, and reliable time synchronization within bounded drift. Failures in system entropy, clock management, or sequence number handling fall outside the formal guarantees and may weaken security in practice.

Side channels are not modeled. The proofs assume that KEM, signature, and AEAD operations are implemented in constant time and do not leak secret dependent information. Without such protections, practical attacks may violate confidentiality or authentication even when the underlying primitives remain cryptographically sound.

The analysis also does not cover the correctness of external key distribution mechanisms. The binding between key identities and verification keys is assumed to be correct and authenticated by out of band means. Attacks on certificate validation, key provisioning, or key revocation fall outside the protocol and thus beyond the scope of the formal model.

Denial of service, connection churn, and resource exhaustion attacks are likewise not modeled. The results guarantee that accepted packets satisfy authenticity and freshness, but they do not guarantee liveness in the presence of an active adversary. Operational safeguards such as rate limiting, connection quotas, and robust error handling must complement the cryptographic protections.

12.3 Directions for Future Work

Several directions merit further investigation.

Extended ratcheting. QSMP includes optional support for simple symmetric and asymmetric ratcheting, but the design could be enriched with a more advanced ratchet structure that provides stronger post compromise security or continuous key evolution. Exploring formal models for multi stage ratchets and integrating them into QSMP’s handshake framework could increase resilience under persistent compromise.

Formal verification of the implementation. The protocol’s correctness depends on rigorous enforcement of length checks, state transitions, and constant time operations. Formal verification of the reference implementation, for example using symbolic execution or model checking, would provide additional assurance that the implementation conforms to the formal specification.

Broader comparative analysis. QSMP’s design philosophy differs from protocols such as SSH, TLS, or the Noise framework. A deeper comparative study could illuminate trade offs in handshake structure, state management, and post quantum assumptions. Such analysis could also explore hybrid designs where QSMP components are combined with classical mechanisms or layered inside larger transport protocols.

Integration with deployment level mechanisms. The behavior of QSMP is influenced by the quality of time synchronization, key distribution, and randomness sources. Future work could address how best to integrate QSMP into system architectures that provide these services, ensuring that the protocol’s assumptions hold under practical conditions.

In summary, QSMP provides a compact and well structured design for post quantum secure messaging. This paper shows that, when implemented correctly and supported by robust system level defenses, the protocol achieves strong authentication, confidentiality, and integrity guarantees grounded in standard assumptions on its cryptographic components.

References

1. National Institute of Standards and Technology (NIST). *CRYSTALS-Kyber, Public-Key Encryption and Key-Establishment Algorithm*. NIST PQC Standard, 2024. Available at: <https://doi.org/10.6028/NIST.FIPS.203>.
2. National Institute of Standards and Technology (NIST). *CRYSTALS-Dilithium, Digital Signature Algorithm*. NIST PQC Standard, 2024. Available at: <https://doi.org/10.6028/NIST.FIPS.204>.
3. National Institute of Standards and Technology (NIST). *Falcon: Lattice-Based Digital Signature Algorithm (Round 4 Candidate)*. NIST PQC Project, 2023. Available at: <https://csrc.nist.gov/projects/post-quantum-cryptography/round-4-submissions>.
4. National Institute of Standards and Technology (NIST). *FIPS 202: SHA-3 Standard, Permutation-Based Hash and Extendable Output Functions*. U. S. Department of Commerce, 2015. Available at: <https://doi.org/10.6028/NIST.FIPS.202>.
5. Bertoni, G., Daemen, J., Peeters, M., and Van Assche, G. *The Keccak Reference*. Submission to the NIST SHA-3 Competition, 2011. Available at: <https://keccak.team/files/Keccak-reference-3.0.pdf>.
6. Krawczyk, H. *Cryptographic Extraction and Key Derivation: The HKDF Scheme*. In CRYPTO 2010. Available at: https://link.springer.com/chapter/10.1007/978-3-642-14623-7_34.
7. Bellare, M., and Namprempre, C. *Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm*. In ASIACRYPT 2000. Available at: https://link.springer.com/chapter/10.1007/3-540-44448-3_29.
8. Rogaway, P. *Nonce-Based Symmetric Encryption*. In FSE 2004. Available at: https://link.springer.com/chapter/10.1007/978-3-540-25937-4_10.
9. Rogaway, P., and Shrimpton, T. *A Provable-Security Treatment of the Key-Wrap Problem*. In EUROCRYPT 2006. Available at: https://link.springer.com/chapter/10.1007/11761679_3.
10. Bellare, M., and Rogaway, P. *Entity Authentication and Key Distribution*. In CRYPTO 1993. Available at: <https://www.iacr.org/archive/crypto93/>.
11. Canetti, R., and Krawczyk, H. *Analysis of Key Exchange Protocols and Their Use for Building Secure Channels*. In EUROCRYPT 2001. Available at: https://link.springer.com/chapter/10.1007/3-540-44987-6_18.
12. Abadi, M., and Rogaway, P. *Reconciling Two Views of Cryptography*. Journal of Cryptology, 2002. Available at: <https://link.springer.com/article/10.1007/s00145-002-0042-4>.
13. Bernstein, D. J., Chuengsatiansup, C., Lange, T., and van Vredendaal, C. *NTRU Prime: Round 3 Specification*. NIST PQC Project, 2020. Available at: <https://ntruprime.cr.yp.to/nist/ntruprime-20201007.pdf>.
14. Alkim, E., Ducas, L., Pöppelmann, T., and Schwabe, P. *NewHope: Algorithm Specifications and Supporting Documentation*. NIST PQC Project, 2019. Available at: <https://newhopecrypto.org/resources/newhope-specification-2019.pdf>.
15. Underhill, J. G. *Quantum Secure Messaging Protocol (QSMP) Specification*. Quantum Resistant Cryptographic Solutions Corporation, 2025. Available at: https://www.qrcs corp.ca/documents/qsmp_specification.pdf.
16. Underhill, J. G. *Quantum Secure Messaging Protocol Implementation Analysis*. Quantum Resistant Cryptographic Solutions Corporation, 2025. Available at: https://www.qrcs corp.ca/documents/qsmp_analysis.pdf.

17. Underhill, J. G. *QSMP Reference Implementation (C Source Code)*. Quantum Resistant Cryptographic Solutions Corporation, 2025. Available at: <https://github.com/QRCS-CORP>.