Post Quantum Shell - PQS 1.0

Revision 1.0, October 21, 2024

John G. Underhill - contact@qrcscorp.ca

This document is an engineering level description of the Post Quantum Shell protocol.
This document describes the network protocol PQS, a post-quantum secure shell.

Contents **Page**

Foreword

This document is the first revision of the specification of PQS, further revisions may become necessary during the pursuit of a standard model, and revision numbers shall be incremented with changes to the specification. The reader is asked to consider only the most recent revision of this draft, as the authoritative implementation of the PQS specification.

The author of this specification is John G. Underhill, and can be reached at john.underhill@protonmail.com

PQS, the algorithm constituting the PQS messaging protocol is patent pending, and is owned by John G. Underhill and Quantum Resistant Cryptographic Solutions Corporation. The code described herein is copyrighted, and owned by John G. Underhill and Quantum Resistant Cryptographic Solutions Corporation.

1. Introduction

The **Post Quantum Shell (PQS)** is a high security, post-quantum communication protocol designed to replace traditional Secure Shell (SSH) for remote server access and communication. With the advent of quantum computing, classical encryption algorithms like RSA, ECDH, and DSA are at risk of being broken, making quantum-resistant cryptographic techniques essential. PQS leverages post-quantum cryptographic primitives, including Kyber (lattice-based key encapsulation), and digital signatures from Dilithium, which are designed to resist both classical and quantum computing threats.

PQS is particularly well suited for industries and applications where long-term confidentiality is critical, such as financial technology (fintech), government communication, healthcare, and critical infrastructure. By offering enhanced security against quantum attacks, PQS ensures that sensitive communications remain secure for years to come.

2. Protocol Description

The PQS exchange is a one-way trust, client-server key-exchange model in which the client trusts the server, and a single shared secret is securely shared between them. Designed for efficiency, the Simplex exchange is fast and lightweight, while providing 256-bit post-quantum security, ensuring protection against future quantum-based threats.

This protocol is versatile and can be used in a wide range of applications, such as client registration on networks, secure cloud storage, hub-and-spoke model communications, commodity trading, and electronic currency exchange—essentially, any scenario where an encrypted tunnel using strong, quantum-safe cryptography is required.

The server in this model is built as a multi-threaded communications platform capable of generating a uniquely keyed encrypted tunnel for each connected client. With a lightweight state footprint of less than 4 kilobytes per client, a single server instance has the capability to handle potentially hundreds of thousands of simultaneous connections. The cipher encapsulation keys utilized during each key exchange are ephemeral and unique, ensuring that every key exchange remains secure and independent from previous key exchanges.

The server distributes a public signature verification key to its clients. This key is used to authenticate the server's public cipher encapsulation key during the key exchange process. The server's public verification key can be shared with clients through various secure methods, including during a registration event, pre-embedding in client software, or via other secure distribution channels.

3.Terms and Definitions

3.1 Cryptographic Primitives

3.1.1 Kyber

The Kyber asymmetric cipher and NIST Post Quantum Competition winner.

3.1.2 McEliece

The McEliece asymmetric cipher and NIST Round 3 Post Quantum Competition candidate.

3.1.3 Dilithium

The Dilithium asymmetric signature scheme and NIST Post Quantum Competition winner.

3.1.5 SPHINCS+

The SPHINCS+ asymmetric signature scheme and NIST Post Quantum Competition winner.

3.1.6 RCS

The wide-block Rijndael hybrid authenticated symmetric stream cipher.

3.1.7 SHA-3

The SHA3 hash function NIST standard, as defined in the NIST standards document FIPS-202; SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions.

3.1.8 SHAKE

The NIST standard Extended Output Function (XOF) defined in the SHA-3 standard publication FIPS-202; SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions.

3.1.9 KMAC

The SHA3 derived Message Authentication Code generator (MAC) function defined in NIST special publication SP800-185: SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash and ParallelHash.

3.2 Network References

3.2.1 Bandwidth

The maximum rate of data transfer across a given path, measured in bits per second (bps).

3.2.2 Byte

Eight bits of data, represented as an unsigned integer ranged 0-255.

3.2.3 Certificate

A digital certificate, a structure that contains a signature verification key, expiration time, and serial number and other identifying information. A certificate is used to verify the authenticity of a message signed with an asymmetric signature scheme.

3.2.4 Domain

A virtual grouping of devices under the same authoritative control that shares resources between members. Domains are not constrained to an IP subnet or physical location but are a virtual group of devices, with server resources typically under the control of a network administrator, and clients accessing those resources from different networks or locations.

3.2.5 Duplex

The ability of a communication system to transmit and receive data; half-duplex allows one direction at a time, while full-duplex allows simultaneous two-way communication.

3.2.6 Gateway: A network point that acts as an entrance to another network, often connecting a local network to the internet.

3.2.7 IP Address

A unique numerical label assigned to each device connected to a network that uses the Internet Protocol for communication.

3.2.8 IPv4 (Internet Protocol version 4): The fourth version of the Internet Protocol, using 32-bit addresses to identify devices on a network.

3.2.9 IPv6 (Internet Protocol version 6): The most recent version of the Internet Protocol, using 128-bit addresses to overcome IPv4 address exhaustion.

3.2.10 LAN (Local Area Network)
A network that connects computers within a limited area such as a residence, school, or office building.

3.2.11 Latency
The time it takes for a data packet to move from source to destination, affecting the speed and performance of a network.

3.2.12 Network Topology
The arrangement of different elements (links, nodes) of a computer network, including physical and logical aspects.

3.2.13 Packet
A unit of data transmitted over a network, containing both control information and user data.

3.2.14 Protocol
A set of rules governing the exchange or transmission of data between devices.

3.2.15 TCP/IP (Transmission Control Protocol/Internet Protocol)
A suite of communication protocols used to interconnect network devices on the internet.

3.2.16 Throughput: The actual rate at which data is successfully transferred over a communication channel.

3.2.17 UDP (User Datagram Protocol)
A communication protocol that offers a limited amount of service when messages are exchanged between computers in a network that uses the Internet Protocol.

3.2.18 VLAN (Virtual Local Area Network)
A logical grouping of network devices that appear to be on the same LAN regardless of their physical location.

3.2.19 VPN (Virtual Private Network)
Creates a secure network connection over a public network such as the internet.

3.3 Normative References

The following documents serve as references for cryptographic components used by QSTP:

3.3.1 FIPS 202: SHA-3 Standard: Permutation-Based Hash and Extendable Output Functions: This standard specifies the SHA-3 family of hash functions, including SHAKE extendable-output functions. https://doi.org/10.6028/NIST.FIPS.202

3.3.2 FIPS 203: Module-Lattice-Based Key Encapsulation Mechanism (ML-KEM): This standard specifies ML-KEM, a key encapsulation mechanism designed to be secure against quantum computer attacks. https://doi.org/10.6028/NIST.FIPS.203

3.3.3 FIPS 204: Module-Lattice-Based Digital Signature Standard (ML-DSA): This standard specifies ML-DSA, a set of algorithms for generating and verifying digital signatures, believed to be secure even against adversaries with quantum computing capabilities. https://doi.org/10.6028/NIST.FIPS.204

3.3.4 NIST SP 800-185: SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash, and ParallelHash: This publication specifies four SHA-3-derived functions: cSHAKE, KMAC, TupleHash, and ParallelHash. https://doi.org/10.6028/NIST.SP.800-185

3.3.5 NIST SP 800-90A Rev. 1: Recommendation for Random Number Generation Using Deterministic Random Bit Generators: This publication provides recommendations for the generation of random numbers using deterministic random bit generators. https://doi.org/10.6028/NIST.SP.800-90Ar1

3.3.6 NIST SP 800-108: Recommendation for Key Derivation Using Pseudorandom Functions: This publication offers recommendations for key derivation using pseudorandom functions. https://doi.org/10.6028/NIST.SP.800-108

3.3.7 FIPS 197: The Advanced Encryption Standard (AES): This standard specifies the Advanced Encryption Standard (AES), a symmetric block cipher used widely across the globe. https://doi.org/10.6028/NIST.FIPS.197

4: Cryptographic Primitives

PQS relies on a set of cryptographic primitives designed to provide resilience against both classical and quantum-based attacks. The following sections detail the specific cryptographic algorithms and mechanisms that form the foundation of PQS's encryption, key exchange, and authentication processes.

4.1 Asymmetric Cryptographic Primitives

PQS employs post-quantum secure asymmetric algorithms to ensure the integrity and confidentiality of key exchanges, as well as to facilitate digital signatures. The primary asymmetric primitives used are:

- **Kyber**: An IND-CCA secure lattice-based key encapsulation mechanism that provides secure and efficient key exchange resistant to quantum attacks. Kyber is valued for its balance between computational speed and cryptographic strength, making it suitable for scenarios requiring rapid key generation and exchange.
- **McEliece**: A code-based cryptosystem that remains one of the most established and trusted post-quantum algorithms. It leverages the difficulty of decoding general linear codes, offering a high level of security even against advanced quantum decryption techniques.
- **Dilithium**: A lattice-based digital signature scheme based on that of the underlying MLWE and MSIS problems, that offers fast signing and verification while maintaining strong security guarantees against quantum attacks.
- **Sphincs+**: A stateless hash-based signature scheme, which provides long-term security without reliance on specific problem structures, making it robust against future advancements in cryptographic research.

These asymmetric primitives are selected for their proven resilience against quantum cryptanalysis, ensuring that PQS's key exchange and signature operations remain secure in the face of evolving computational threats.

4.2 Symmetric Cryptographic Primitives

PQS's symmetric encryption employs the **Rijndael Cryptographic Stream (RCS)**, a stream cipher adapted from the Rijndael (AES) symmetric cipher to meet post-quantum security needs. Key features of the RCS cipher include:

- **Wide-Block Cipher Design**: RCS extends the original AES design with a focus on increasing the block size (from 128 to 256 bits) and number of transformation rounds (from 14 to 21 for a 256-bit key, and 30 rounds for a 512-bit key), thereby enhancing its resistance to differential and linear cryptanalysis.
- **Enhanced Key Schedule**: The key schedule in RCS is cryptographically strong using Keccak (cSHAKE), ensuring that derived keys are resistant to known attacks, including algebraic-based and differential attacks. RCS replaces Rijndael’s cryptographically-weak key schedule, with a strong post-quantum secure key expansion function.

- **Authenticated Encryption with Associated Data (AEAD)**: RCS integrates with KMAC (Keccak-based Message Authentication Code) to provide both encryption and message authentication in a single operation. This approach ensures that data integrity is maintained alongside confidentiality.

The RCS stream cipher's design is optimized for high-performance environments, making it suitable for low-latency applications that require secure and efficient data encryption. It leverages AES-NI instructions embedded in modern CPUs.

4.3 Hash Functions and Key Derivation

Hash functions and key derivation functions (KDFs) are essential to PQS's ability to transform raw cryptographic data into secure keys and hashes. The following primitives are used:

- **SHA-3**: SHA-3 serves as PQS's primary hash function, providing secure, collision-resistant hashing capabilities.
- **SHAKE**: PQS employs the Keccak SHAKE XOF function for deriving symmetric keys from shared secrets. This ensures that each session key is uniquely generated and unpredictable, enhancing the protocol's security against key reuse attacks.
- **KMAC**: The SHA-3 keyed hashing function (MAC), part of the SHA-3 family of post-quantum resistant hashing functions.

These cryptographic primitives ensure that PQS's key management processes remain secure, even in scenarios involving high-risk adversaries and quantum-capable threats.

5. Protocol Components and State Structures

5.1 Protocol String

The protocol string in PQS is composed of four key components, each representing a specific cryptographic element used in the secure communication process:

1. **Asymmetric Signature Scheme**: Specifies the signature scheme along with its security strength (e.g., s1, s3, s5) from low to high. Example: dilithium-s3 correlates to the NIST *level 3* security designation (192 bits of post-quantum security).
2. **Asymmetric Encapsulation Cipher**: Defines the asymmetric encryption algorithm and its security strength. Example: mceliece-s5.
3. **Hash Function Family**: The designated hash function used within the protocol, which is set as SHA3.
4. **Symmetric Cipher**: The symmetric cipher used for data encryption, set as the authenticated stream cipher RCS.

The protocol string plays a crucial role during the initial negotiation phase to ensure that both the client and server agree on a common set of cryptographic parameters. If the client and server do not support the same protocol settings, a secure connection cannot be established.

Signature Scheme	Asymmetric Cipher	HASH Function	Symmetric Cipher
Dilithium	Kyber	SHA3	RCS
Dilithium	McEliece	SHA3	RCS
Sphincs+	McEliece	SHA3	RCS

Table 5.1: The Protocol string choices in revision PQS 1.3a.

5.2 Client Key Structure

The client key is a publicly exportable structure that contains the signature verification key and associated metadata. It includes parameters such as the key expiration time, protocol string, public signature verification key, and key identity array.

Parameter	Data Type	Bit Length	Function
Expiration	Uint64	64	Validity check
Configuration	Uint8 array	320	Protocol check
Key ID	Uint8 array	128	Identification
Verification Key	Uint8 array	Variable	Authentication

Table 5.2: The client key structure.

- **Expiration**: A 64-bit unsigned integer indicating the number of seconds since the epoch (01/01/1900) until the UTC time when the key remains valid. If the key has expired, the client must request a new public key from the server.

- **Configuration**: Contains the protocol string that defines the cryptographic parameters. If the protocol string on both hosts does not match, the connection is aborted.
- **Key ID**: A unique identifier for the public verification key, facilitating quick reference on the server.
- **Verification Key**: The public asymmetric signature verification key used for authenticating asymmetric encapsulation keys and data during the key exchange.

The client key can be distributed openly or could be encapsulated using X.509 certificates to create a chain of trust, enhancing its security in diverse environments.

5.3 Server Key Structure

The server key is a private (secret) key retained by the server. It contains all elements of the client key plus an additional parameter, the asymmetric signing key.

Data Name	Data Type	Bit Length	Function
Expiration	Uint64	64	Validity check
Configuration	Uint8 array	320	Protocol check
Key ID	Uint8 array	128	Identification
Verification Key	Uint8 array	Variable	Authentication
Signing Key	Uint8 array	Variable	Signing

Table 5.3: The server key structure.

The inclusion of the signing key in the server key structure allows the server to sign messages during the key exchange, ensuring that data exchanges are authenticated and trusted.

5.4 Keep Alive State

PQS uses an internal keep-alive mechanism to maintain active connections. The server periodically sends a keep-alive packet to the client, which the client must acknowledge within the defined interval.

Parameter	Data Type	Bit Length	Function
Expiration Time	Uint64	64	Validity check
Packet Sequence	Uint64	64	Protocol check
Received Status	Bool	8	Status

Table 5.4: The keep alive state.

If the server does not receive a response within the timeout period, it logs a keep-alive error and terminates the connection to prevent stale sessions.

5.5 Connection State

The internal connection state structure stores the critical information needed for PQS operations, including cipher states, sequence counters, and the ratchet key.

Data Name	Data Type	Bit Length	Function
Target	Socket struct	664	Validity check
Cipher Send State	Structure	Variable	Symmetric Encryption
Cipher Receive State	Structure	Variable	Symmetric Decryption
Receive Sequence	Uint64	64	Packet Verification
Send Sequence	Uint64	64	Packet Verification
Connection Instance	Uint32	32	Identification
KEX Flag	Uint8	8	KEX State Flag
Ratchet Key	Uint8 array	512	Symmetric Ratchet
PkHash	Uint8 array	256	Authentication
Session Token	Uint8 array	256	Authentication
ExFlag	Uint8	8	Protocol Check

Table 5.5: The connection state structure.

This data structure ensures secure handling of connection parameters, packet sequencing, and cryptographic states during active communication sessions.

5.8 Client KEX State

The Simplex protocol's client and server state structures focus on one-way authentication, storing essential key exchange data:

Data Name	Data Type	Bit Length	Function
Key ID	Uint8 array	128	Key Identification
Session Token	Uint8 array	512	Verification
Remote Verification Key	Uint8 array	Variable	Asymmetric Authentication
Signature Key	Uint8 array	Variable	Asymmetric Authentication
Shared Secret	Uint8 array	256	Symmetric Key
Verification Key	Uint8 array	Variable	Asymmetric Authentication
Expiration	Uint64	64	Verification

Table 5.7: The Simplex client KEX state structure.

5.9 Server KEX State

The Simplex server state structure stores the asymmetric cipher and signature keys used during the key exchange execution.

Data Name	Data Type	Bit Length	Function

Key ID	Uint8 array	128	Key Identification
Session Token	Uint8 array	512	Verification
Private Cipher Key	Uint8 array	Variable	Asymmetric Encryption
Public Cipher Key	Uint8 array	Variable	Asymmetric Encryption
Signature Key	Uint8 array	Variable	Asymmetric Authentication
Shared Secret	Uint8 array	256	Symmetric Key
Verification Key	Uint8 array	Variable	Asymmetric Authentication
Expiration	Uint64	64	Verification

Table 5.8: The Simplex server KEX state structure.

5.10 PQS Packet Header

The PQS packet header is 21 bytes in length, and contains:

1. The **Packet Flag**, the type of message contained in the packet; this can be any one of the key-exchange stage flags, a message flag, or an error flag.
2. The **Packet Sequence**, this indicates the sequence number of the packet in the exchange.
3. The **Message Size**, this is the size in bytes of the message payload.
4. The **UTC time**, the time the packet was created, used in an anti-replay attack mechanism.

The message is a variable sized array, up to PQS_MESSAGE_MAX in size.

Packet Flag **1 byte**	**Packet Sequence** **8 bytes**	**Message Size** **4 bytes**	**UTC Time** **8 bytes**
Message **Variable Size**			

Figure 5.7: The PQS packet structure.

This packet structure is used for both the key exchange protocol, and the communications stream.

5.11 Flag Types

The following is a list of packet flag types used by PQS:

Flag Name	Numerical Value	Flag Purpose
None	0x00	No flag was specified, the default value.
Connect Request	0x01	The key-exchange client connection request flag.

Connect Response	0x02	The key-exchange server connection response flag.
Connection Terminated	0x03	The connection is to be terminated.
Encrypted Message	0x04	The message has been encrypted by the communications stream.
Exchange Request	0x07	The key-exchange client exchange request flag.
Exchange Response	0x08	The key-exchange server exchange response flag.
Establish Request	0x09	The key- exchange client establish request flag.
Establish Response	0x0A	The key- exchange server establish response flag.
Keep Alive Request	0x0B	The packet contains a keep alive request.
Keep Alive Response	0x0C	The packet contains a keep alive response.
Remote Connected	0x0D	The remote host has terminated the connection.
Remote Terminated	0x0E	The remote host has terminated the connection.
Session Established	0x0F	The session is in the established state.
Establish Verify	0x10	The session is in the verify state.
Unrecognized Protocol	0x11	The protocol string is not recognized
Asymmetric Ratchet Request	0x12	The packet contains an asymmetric ratchet request.
Asymmetric Ratchet Response	0x13	The packet contains an asymmetric ratchet response.
Symmetric Ratchet Request	0x14	The packet contains a symmetric ratchet request.
Error Condition	0xFF	The connection experienced an error.

Table 5.8: Packet header flag types.

5.12 Error Types

The following is a list of error messages used by PQS:

Error Name	Numerical Value	Description
None	0x00	No error condition was detected.
Authentication Failure	0x01	The symmetric cipher had an authentication failure.
Bad Keep Alive	0x02	The keep alive check failed.
Channel Down	0x03	The communications channel has failed.

Connection Failure	0x04	The device could not make a connection to the remote host.
Connect Failure	0x05	The transmission failed at the KEX connection phase.
Decapsulation Failure	0x06	The asymmetric cipher failed to decapsulate the shared secret.
Establish Failure	0x07	The transmission failed at the KEX establish phase.
Exstart Failure	0x08	The transmission failed at the KEX exstart phase.
Exchange Failure	0x09	The transmission failed at the KEX exchange phase.
Hash Invalid	0x0A	The public-key hash is invalid.
Invalid Input	0x0B	The expected input was invalid.
Invalid Request	0x0C	The packet flag was unexpected.
Keep Alive Expired	0x0D	The keep alive has expired with no response.
Key Expired	0x0E	The PQS public key has expired.
Key Unrecognized	0x0F	The key identity is unrecognized.
Packet Un-Sequenced	0x10	The packet was received out of sequence.
Random Failure	0x11	The random generator has failed.
Receive Failure	0x12	The receiver failed at the network layer.
Transmit Failure	0x13	The transmitter failed at the network layer.
Verify Failure	0x14	The expected data could not be verified.
Unknown Protocol	0x15	The protocol string was not recognized.
Listener Failure	0x16	The listener function failed to initialize.
Accept Failure	0x17	The socket accept function returned an error.
Hosts Exceeded	0x18	The server has run out of socket connections.
Allocation Failure	0x19	The server has run out of memory.
Decryption Failure	0x1A	The decryption authentication has failed.
Ratchet Failure	0x1C	The ratchet operation has failed.

Table 5.9: Error type messages.

5.10 Function Definitions in pqs.h

Function	Description
pqs_packet_to_stream	Serializes a network packet into a byte stream for transmission.

pqs_stream_to_packet	Deserializes a byte array back into a packet structure.
pqs_public_key_decode	Decodes a public key string into a client verification key structure.
pqs_public_key_encode	Encodes a client key structure into a public key string.
pqs_public_key_hash	Hashes a public key structure and returns the result in a byte array.
pqs_signature_key_deserialize	Decodes a secret signature key from a byte array into a server key structure.
pqs_signature_key_serialize	Encodes a server key structure into a serialized byte array.

5.10.1 Function: pqs_packet_to_stream

Purpose: Serializes a network packet into a byte stream for transmission.

Description: Converts a structured packet, containing fields such as message, sequence number, and timestamp, into a byte stream ready for network transmission. The function ensures proper byte ordering, padding, and integrity checks before sending the data over the network.

5.10.2 Function: pqs_stream_to_packet

Purpose: Deserializes a byte array back into a packet structure.

Description: Converts a byte array received from the network back into a packet structure. It verifies the packet format, validates sequence numbers, timestamps, and other fields to ensure the message is legitimate and intact.

5.10.3 Function: pqs_public_key_decode

Purpose: Decodes a public key string into a client verification key structure.

Description: Converts a serialized public key string (typically in base64 or hexadecimal form) into a structured key format that can be used for cryptographic operations.

5.10.4 Function: pqs_public_key_encode

Purpose: Encodes a client key structure into a public key string.

Description: Converts a structured public key into a string format, often used for transmission or storage in human-readable forms like base64.

5.10.5 Function: pqs_public_key_hash

Purpose: Hashes a public key structure and returns the result in a byte array.

Description: Applies a secure hash function to the public key and returns a fixed-size hash value that can be used for verification or integrity checks.

5.10.6 Function: pqs_signature_key_serialize

Purpose: Encodes a server key structure into a serialized byte array.

Description: Converts a signature key (either private or public) into a byte array for storage or transmission.

5.10.7 Function: pqs_signature_key_deserialize

Purpose: Decodes a secret signature key from a byte array into a server key structure.

Description: Converts a serialized byte array containing a signature key back into its structured format to be used in cryptographic operations such as signing or verification.

6 Protocol Operational Overview

6.1 Connection Request

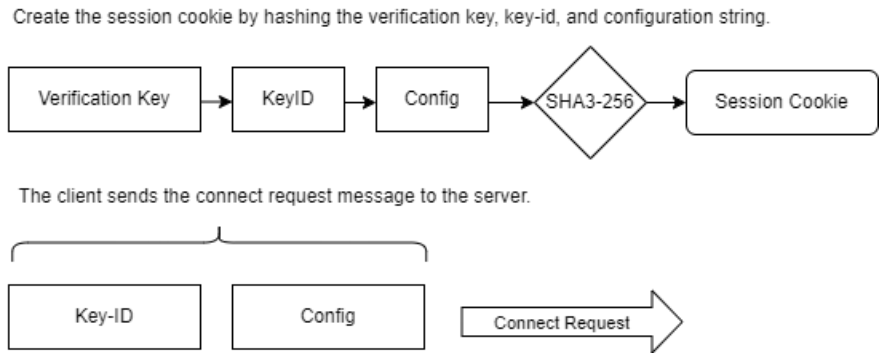

Figure 6.1: PQS Simplex connection request.

1) The client inspects the packet header for the correct flag, sequence number, expected message size, and that the valid-time has not expired.
2) The client begins the key exchange operation by sending a connect request packet to the server. This packet contains the server's key identification array and the protocol configuration string.
3) The client hashes the configuration string, the key identification array, and its signature verification key. This combined hash is stored in the session cookie state value (*sch*) and is used as a unique session identifier. This approach ensures that the session's cryptographic parameters are referenced and that the session state is uniquely identifiable.
4) The client adds the key-id and the configuration string, and sends the connection request to the server.

6.2 Connection Response

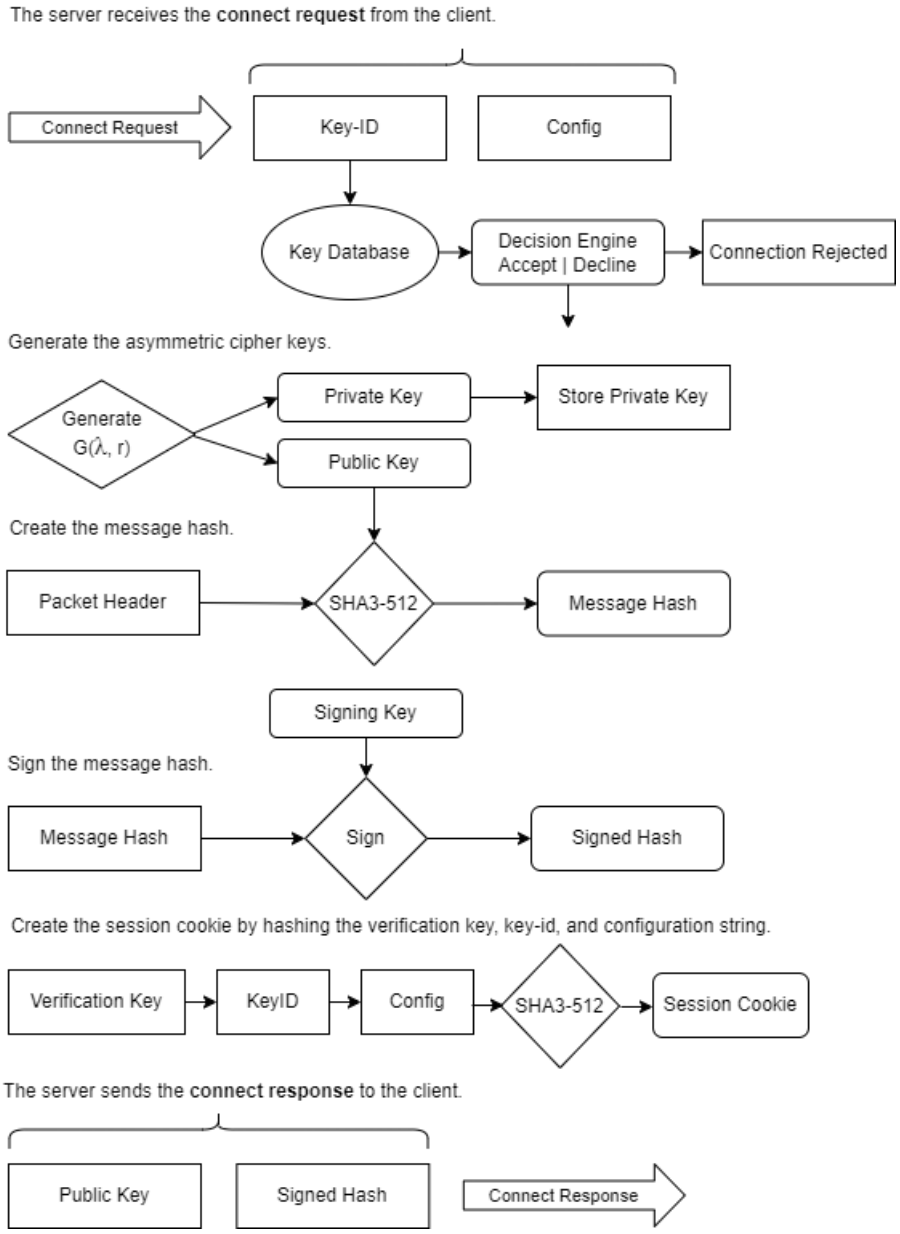

Figure 6.2: PQS server connection response.

1) The server inspects the packet header for the correct flag, sequence number, expected message size, and that the valid-time has not expired.
2) The server checks its database for a key that matches the key identification array provided in the request. If the verification key is not found, the server sends an *unknown key* error message to the client, aborts the key exchange, logs the event, and tears down the session.
3) The server compares the protocol configuration string sent by the client with its own stored protocol string to ensure compatibility.

4) The server verifies the expiration time of the key. If all these fields are validated successfully, the server loads the key into its active state.
5) The server hashes the configuration string, the key identification array, and its signature verification key, and stores this combined hash in its session cookie state value (sch).
6) The server generates a new public/private asymmetric cipher key pair. It hashes the public encapsulation key and the serialized connection response packet header, and signs this hash with its private signing key.
7) The server adds the public asymmetric encapsulation key and the signed hash of the public key to the connect response message and sends it to the client to continue the key exchange process.

6.3 Exchange Request

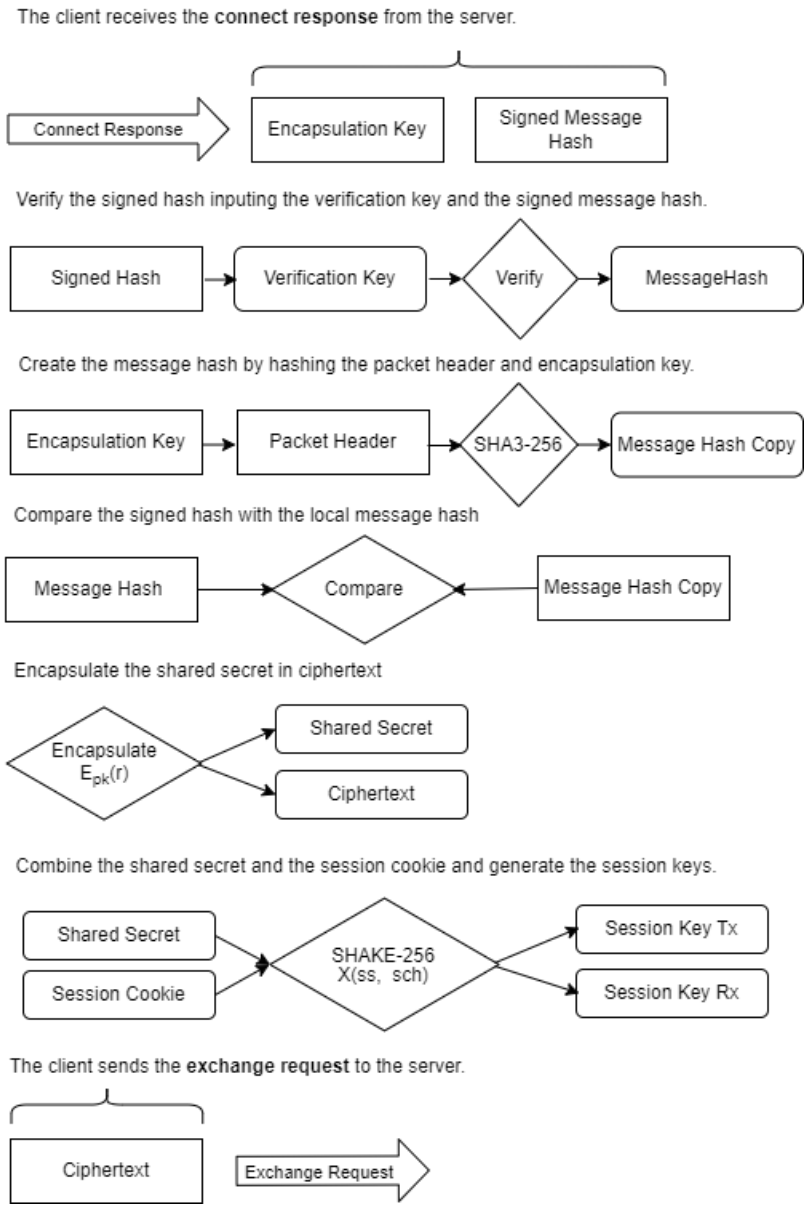

Figure 7.3: PQS client exchange request.

1) The client inspects the packet header for the correct flag, sequence number, expected message size, and that the valid-time has not expired.
2) The client uses the server's signature verification key to verify the signature on the hash of the asymmetric encapsulation key and serialized packet header. If the signature verification fails, the client sends an *authentication failure* message and terminates the connection.
3) If the signature is successfully verified, the client hashes the asymmetric cipher key and serialized header, and compares this hash to the signed hash in the server's response message.

If the hash check fails, the client sends a *hash invalid* error message and closes the connection.

4) The client uses the asymmetric cipher key to encapsulate a *shared secret*, creating the ciphertext.
5) The shared secret is combined with the session cookie to key the KDF, which generates the symmetric cipher keys and nonces used to key the transmit and receive cipher instances.
6) The cipher *rx* and *tx* symmetric instances are initialized and ready to transmit and receive data.
7) The asymmetric ciphertext is then included in the exchange request packet, which the client sends to the server.

6.4 Exchange Response

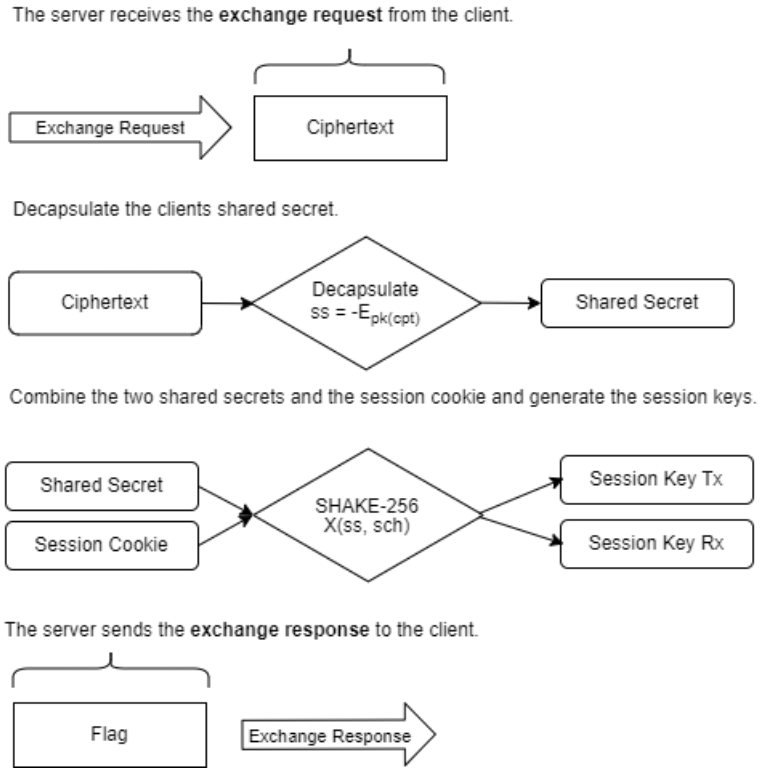

Figure 7.4: PQS server exchange response.

1) The server inspects the packet header for the correct flag, sequence number, expected message size, and that the valid-time has not expired.
2) The server uses its stored asymmetric cipher private key to decapsulate the shared secret from the ciphertext.
3) The decapsulated shared secret is combined with the session cookie to derive the two symmetric session keys and nonces.
4) These derived session keys are used to initialize the symmetric cipher instances, activating both the transmit and receive channels of the encrypted tunnel.

6.5 Establish Verify

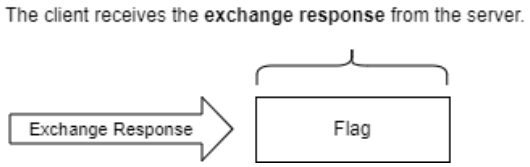

Figure 7.5: PQS client establish request.

1) The client inspects the packet header for the correct flag, sequence number, expected message size, and that the valid-time has not expired.
2) The client verifies that the encrypted tunnel is now active and fully operational. If the packet contains an error flag, indicating that an issue occurred during the tunnel setup, the client immediately initiates a connection teardown.
3) The client **should** then handle the error according to its predefined procedures, ensuring the user or application is informed of the failure.

7. Mathematical Description

Mathematical Symbols

← ↔ →	-Assignment and direction symbols
:=, !=, ?=	-Equality operators; assign, not equals, evaluate
C	-The client host, initiates the exchange
S	-The server host, listens for a connection
G(λ, r)	-The asymmetric cipher key generation with parameter set and random source
$-E_{sk}$	-The asymmetric decapsulation function and secret key
E_{pk}	-The asymmetric encapsulation function and public key
S_{sk}	-Sign data with the secret signature key
V_{pk}	-Verify a signature the public verification key
cfg	-The protocol configuration string
cpr_{rx}	-A receive channels symmetric cipher instance
cpr_{tx}	-A transmit channels symmetric cipher instance
cpt	-The symmetric ciphers cipher-text
cpta	-The asymmetric ciphers cipher-text
$-E_k$	-The symmetric decryption function and key
E_k	-The symmetric encryption function and key
H	-The hash function (SHA3)
k, *mk*	-A symmetric cipher or MAC key
KDF	-The key expansion function (SHAKE)
kid	-The public keys unique identity array
M_{mk}	-The MAC function and key (KMAC)
pk, *sk*	-Asymmetric public and secret keys
pvk	-Public signature verification key
sch	-A hash of the configuration string and and asymmetric verification-keys
sec	-The shared secret derived from asymmetric encapsulation and decapsulation
spkh	-The signed hash of the asymmetric public encapsulation-key

Simplex Key Exchange Sequence

Preamble

The Simplex key exchange sequence begins with the client verifying the validity of the server's public signature verification key. The client checks the expiration date of this key, and if it is found to be invalid or expired, the client initiates a re-authentication session with the server. During this session, a new key is distributed over an encrypted channel, and the client verifies the new key's certificate using the designated authentication authority or scheme implemented by the server and client software.

7.1 Connect Request

The client initiates the connection process by sending a *connection request* to the server that includes its configuration string and asymmetric public signature key identity.

Key Identity

The *key identity (kid)* is a multi-part, 16-byte array that acts as a public asymmetric verification key and device identification string. It is used to match the target server to its corresponding cryptographic key, ensuring that the correct key is used during the exchange.

Configuration String

The *configuration string (cfg*) specifies the cryptographic protocol set being used in the key exchange process. For the exchange to proceed successfully, the configuration strings used by both the client and server must match, indicating that they are using the same cryptographic parameters.

Session Cookie

To securely manage the state of the key exchange, the client generates a *session cookie* by hashing a combination of the configuration string, the key identity, and the server public asymmetric signature verification key:

$sch \leftarrow \mathrm{H}(cfg \,||\, kid \,||\, pvk)$

Where:

- *cfg* is the configuration string.
- *kid* is the key identity.
- *pvk* is the server's public signature verification key.

This session cookie (*sch*) serves as a unique identifier for the session, helping to ensure that the cryptographic parameters are consistently referenced throughout the exchange.

The client then sends the key identity string (*kid*) and the configuration string (*cfg*) to the server to initiate the connection:

$C\{ kid, cfg \} \rightarrow S$

7.2 Connect Response

The server processes the client's connection request and responds with either an error message or a connect response packet. If any error occurs during the key exchange, the server generates an error packet and sends it to the remote host, which triggers a teardown of the session and network connection on both sides.

Key Verification and Protocol Check

The server begins by verifying that it has the appropriate asymmetric signature verification key that corresponds to the client's request, using the key-identity array (*kid*).

It then checks that its protocol configuration matches the one specified by the client. To securely manage the state of the exchange, the server creates a session cookie by hashing the configuration string, the key identity, and the public signature verification key:

$sch \leftarrow H(cfg \parallel kid \parallel pvk)$

Where:

- *cfg* is the configuration string.
- *kid* is the key identity.
- *pvk* is the server's public signature verification key.

This session cookie (*sch*) serves as a unique identifier for the session, helping maintain the integrity of the key exchange.

Asymmetric Key Generation and Signing

The server generates a new asymmetric encryption key pair and securely stores the private key. It hashes the public encapsulation key and the serialized connect response packet header, and signs this hash using its private asymmetric signature key. The signature provides a cryptographic guarantee that the public asymmetric cipher key has not been tampered with during transmission.

Key generation and signing steps:

Generate the public (*pk*) and private (*sk*) asymmetric encryption keys.

$pk, sk \leftarrow G(\lambda, r)$

Create a hash of the public key and serialized *connect response* packet header (*sh*).

$pkh \leftarrow H(pk \parallel sh)$

Sign the hashed public key using the server's private signature key.

$spkh \leftarrow S_{sk}(pkh)$

The public signature verification key itself can be signed using a 'chain of trust' model, such as X.509, to ensure further authentication through a signature verification extension to the protocol.

Server Response

The server sends a connect response message back to the client, containing the signed hash of the public asymmetric encapsulation key (*spkh*) and a copy of the public key itself:

$S\{ spkh, pk \} \rightarrow C$

7.3 Exchange Request

The client processes the server's connect response and initiates the next steps of the key exchange by verifying the received data, encapsulating a shared secret, and preparing the session keys.

Signature Verification and Hash Check

The client begins by verifying the signature of the hash using the server's public verification key. It then generates its own hash of the server's public key and compares it to the hash contained in the server's message. If the hashes match, the client proceeds to encapsulate the shared secret. If the hashes do not match, the key exchange is aborted.

The client uses the server's public verification key to check the hash of the public key. If the verification is successful, the process continues; otherwise, the key exchange fails.

$V_{pk}(H(pk)) \leftarrow (\text{true } ?= pk : 0)$

The public encapsulation key and connect response packet header are hashed, and the hash is compared with signed hash received from the server. Once the packet header and public key are verified, the client uses the server's public key to encapsulate a shared secret.

The client generates a ciphertext (*cpt*) and encapsulates the shared secret (*sec*) using the server's public key.

$cpt, sec \leftarrow E_{pk}(sec)$

The client combines the shared secret and the session cookie to derive the session keys and two unique nonces for the communication channels.

The Key Derivation Function (KDF) generates two session keys (*k1*, *k2*) and two nonces (*n1*, *n2*) using the shared secret (*sec*) and the session cookie (*sch*).

$k1, k2, n1, n2 \leftarrow \text{KDF}(sec \parallel sch)$

Cipher Initialization

The receive and transmit channel ciphers are then initialized using the derived keys and nonces.

Initializes the receive channel cipher with key *k2* and nonce *n2*.

$\text{cpr}_{\text{rx}}(k2, n2)$

Initializes the transmit channel cipher with key *k1* and nonce *n1*.

$\text{cpr}_{\text{tx}}(k1, n1)$

Client Transmission

The client sends the ciphertext to the server as part of the exchange request.

The client transmits the encapsulated shared secret to the server.

$C\{ cpt \} \rightarrow S$

7.4 Exchange Response

The server processes the client's exchange request by decapsulating the shared secret, deriving the session keys, and confirming the secure communication channel.

Shared Secret Decapsulation

The server decapsulates the shared secret from the ciphertext received from the client.

The server uses its private asymmetric key to decapsulate the shared secret (*sec*) from the received ciphertext (*cpt*).

$sec \leftarrow -\text{E}_{\text{sk}}(cpt)$

Session Key Derivation

The server combines the decapsulated shared secret and the session cookie hash to derive two session keys and two unique nonces for the communication channels.

The Key Derivation Function (KDF) generates two symmetric session keys (*k1*, *k2*) and two nonces (*n1*, *n2*) using the shared secret (*sec*) and the session cookie (*sch*).

$k1, k2, n1, n2 \leftarrow \text{KDF}(sec \parallel sch)$

Cipher Initialization

The server initializes the symmetric ciphers for the receive and transmit channels.

Initializes the receive channel cipher with key *k1* and nonce *n1*.

$\text{cpr}_{\text{rx}}(k1, n1)$

Initializes the transmit channel cipher with key *k2* and nonce *n2*.

$\text{cpr}_{\text{tx}}(k2, n2)$

Server Response

The server sets the packet flag to "exchange response," indicating that the encrypted channels have been successfully established. It then sends this notification back to the client to confirm the secure communication channel.

The server sends an exchange response flag to the client, confirming that the secure tunnel is established.

$\text{S}\{f\} \rightarrow \text{C}$

The server updates its operational state to *session established*, indicating that it is now ready to securely process data over the encrypted channels.

7.5 Establish Verify

In the final step of the key exchange sequence, the client verifies the status of the encrypted tunnel based on the server's exchange response.

Client Verification

The client inspects the flag of the exchange response packet received from the server. If the flag indicates an error state, the client immediately tears down the tunnel to prevent any further data transmission. This ensures that no data is sent over an insecure or compromised connection.

If the flag does not indicate an error state, the client confirms that the tunnel is successfully established and in an operational state.

Operational State

Once the verification is complete and the tunnel is confirmed, the client updates its internal state to *session established*, indicating that the secure communication channels are fully operational. The client is now ready to process data over the encrypted tunnel.

7.6 Transmission

During the transmission phase, either the client or server sends messages over the established encrypted tunnel using the RCS stream cipher's MAC, AEAD (Authenticated Encryption with Associated Data), and encryption functions. This process ensures the integrity and confidentiality of the transmitted data.

Message Serialization and Encryption

The transmitting host (client or server) starts by serializing the packet header, which includes critical details such as the message size, timestamp, protocol flag, and sequence number. This serialized header is then added to the symmetric cipher's associated data parameter, which adds metadata authentication to the encryption process.

The message encryption process is as follows:

1. **Encrypt the Message**: The plaintext message is encrypted using the symmetric encryption function of the RCS stream cipher. The symmetric encryption function (*Ek*) is applied to the plaintext message (*m*) to produce the ciphertext (*cpt*).
   $cpt \leftarrow E_k(m)$

2. **Update the MAC State**: The serialized packet header is added to the MAC (Message Authentication Code) state through the additional-data parameter of the RCS cipher. The MAC function (M_{mk}) is updated with the serialized packet header (*sh*) and the ciphertext (*cpt*) to produce the MAC code (*mc*).
   $mc \leftarrow M_{mk}(sh, cpt)$

3. **Append the MAC Code**: The MAC code is appended to the end of the ciphertext, ensuring that any tampering with the data during transmission will be detected.

Packet Decryption and Verification

Upon receiving the packet, the recipient host deserializes the packet header and adds it to the MAC state along with the received ciphertext. The MAC computation is then finalized and compared with the MAC code that was appended to the ciphertext. The packet timestamp is

compared to the *UTC* time, if the time is outside of a tolerance threshold, the packet is rejected and the session is torn down.

1. **Generate the MAC Code:** Add the serialized packet header to the cipher AEAD. Add the ciphertext and generate the MAC code.
   $mc` \leftarrow M_{mk}(sh, cpt)$
   Compare the MAC tag copy with the MAC tag appended to the ciphertext.
   $mc` ?= mc$
   If the MAC check fails, indicating potential data tampering or corruption, the decryption function returns an empty message array and an error status. The application **shall** handle this error accordingly.

2. **Decrypt the Ciphertext:** If the MAC code matches, the ciphertext is considered authenticated, and the message is decrypted.
   The ciphertext (*cpt*) is decrypted back into the plaintext message (*m*) if the MAC verification succeeds.
   $m \leftarrow -E_k(cpt)$

This process ensures that the transmitted data remains confidential and tamper-evident, providing both encryption and authentication to protect the integrity of the communication. Any errors during decryption signal an immediate response to prevent the further exchange of potentially compromised data.

8. Security Analysis

PQS is designed to withstand both classical and quantum attacks, offering future-proof cryptographic security. This section analyzes the protocol's defense mechanisms against common cryptographic threats, focusing on man-in-the-middle attacks, replay attacks, side-channel attacks, and quantum-specific threats. Additionally, we perform a cryptanalysis of the key exchange construction and compare it to other common key exchange protocols.

8.1 Post-Quantum Cryptographic Primitives

PQS relies on post-quantum cryptographic primitives that are resistant to quantum computers, which can break classical cryptography through algorithms like Shor's algorithm and Grover's algorithm. The following primitives are used in PQS:

- **Kyber**: A lattice-based key encapsulation mechanism (KEM) that offers both efficiency and post-quantum security. Kyber ensures that shared session keys are secure against quantum adversaries.
- **Dilithium**: A lattice-based digital signature algorithm used for authentication. Dilithium ensures that digital signatures cannot be forged even with quantum capabilities.

These primitives replace classical algorithms like RSA and ECDH, which are vulnerable to quantum attacks.

8.2 Resistance to Classical Attacks

8.2.1 Man-in-the-Middle (MiTM) Attacks

PQS prevents man-in-the-middle attacks through the use of authenticated key exchange. The key exchange involves signing the server's public key using a post-quantum digital signature (Dilithium) which the client verifies using the server's certificate.

- **Mathematical Defense**: $C_s = \text{Sign}(S_r, P_s)$, $\text{Verify}(P_r, P_s, C_s)$

  This guarantees that only the legitimate server's public key is used for key exchange.

8.2.2 Replay Attacks

Replay attacks are mitigated using a **message valid-time check** (UTC_t). Each packet includes a timestamp, and the server only processes messages that fall within a predefined time window. This ensures that even if an attacker intercepts and replays a packet, it will be rejected if it falls outside the valid time window.

- **Mathematical Defense**: Accept if $| \text{UTC}_{\text{received}} - \text{UTC}_{\text{current}} | < \text{Threshold}$

8.2.3 Forward Secrecy

PQS provides **forward secrecy** through the use of **ephemeral key pairs**. During the key exchange, both the client and server generate new public-private key pairs for each session. Once

the session is complete, the ephemeral keys are discarded, ensuring that even if a long-term private key is compromised, past communications remain secure.

- **Mathematical Defense**: The ephemeral session key k is derived for each session:

$$k = \text{KEX}(S_s, P_l) = \text{KEX}(S_l, P_s)$$

- Since S_s and S_l are discarded after use, future key compromises do not affect past session security.

8.2.4 Message Integrity and Authentication

PQS uses **KMAC (Keccak-based Message Authentication Code)** for message integrity and authentication. After encrypting a message, a MAC is generated over the ciphertext to ensure that the message has not been tampered with during transmission. Both the encryption key and the MAC key are derived from the shared session key.

- **Mathematical Defense**:

$$T = \text{MAC}_{kauth}(M), \text{ Verify } T' = T$$

If the MAC check fails, the message is rejected.

8.3 Resistance to Quantum Attacks

Quantum computers pose a significant threat to classical cryptography by being able to solve problems like factoring large integers or computing discrete logarithms in polynomial time. PQS uses post-quantum cryptographic primitives that are resistant to quantum attacks.

8.3.1 Kyber (Key Encapsulation Mechanisms)

Kyber is used for key encapsulation in PQS, and is the NIST Post Quantum competition winner and standardized for quantum resistant asymmetric ciphers. Kyber is based on the learning with errors (LWE) problem involving findng the shortest path through a lattice. Lattice problems are believed to be resistant to quantum algorithms such as Shor's algorithm.

8.3.2 Dilithium

Dilithium is based on the hardness of finding short vectors in lattices, and is the NIST Post Quantum competition winner and standardized for quantum resistant digital signature schemes.

8.3.3 McEliece

McEliece is a code-based cryptosystem that remains one of the most established and trusted post-quantum algorithms. It leverages the difficulty of decoding general linear codes, offering a high level of security even against advanced quantum decryption techniques.

8.3.4 SPHINCS+

Sphincs+ is a stateless hash-based signature scheme, which provides long-term security without reliance on specific problem structures, making it robust against future advancements in cryptographic research.

8.4 Cryptanalysis of Key Exchange

The PQS key exchange uses a hybrid post-quantum approach, combining classical techniques (like hashing and MACs) with post-quantum encryption and digital signatures. This combination ensures that the key exchange is secure even in the presence of quantum adversaries.

8.4.1 Key Exchange Process

The key exchange in PQS relies on two main operations:

- Key encapsulation using Kyber or McEliece.
- Digital signatures using Dilithium or SPHINCS+.

The server's public key is signed using a post-quantum signature scheme and verified by the client. The client then generates a shared secret using the server's public cipher key and its own private key, ensuring confidentiality and authenticity.

8.4.2 Defense Against Known Attacks

- **Quantum Attacks**: The use of post-quantum algorithms ensures that PQS is secure against both classical and quantum adversaries.
- **Man-in-the-Middle Attacks**: PQS's key exchange mechanism authenticates both the client and server, preventing attackers from intercepting or altering the exchange.
- **Replay Attacks**: The **message valid-time check** prevents an attacker from reusing packets to replay previous communications.
- **Side-Channel Attacks**: The use of constant-time implementations and secure memory management helps mitigate side-channel attacks, where attackers try to exploit information leaks from cryptographic operations.

8.5 Summary of Security Benefits

- **Post-Quantum Security**: PQS's use of McEliece, Kyber, Dilithium, and SPHINCS+ ensures that it is resistant to quantum attacks.
- **Forward Secrecy**: Each session uses ephemeral key pairs, ensuring that past communications remain secure even if long-term keys are compromised.
- **Replay Attack Prevention**: The **message valid-time check** ensures that previously transmitted packets cannot be replayed by an adversary.

- **Man-in-the-Middle Attack Prevention**: PQS authenticates both the server and client, preventing any third party from injecting or altering messages.
- **Resilience Against Side-Channel Attacks**: PQS is designed to resist side-channel attacks through constant-time implementations and secure memory management.

9. Cryptanalysis of the PQS 1.0 Simplex Protocol

9.1 Assumptions, Adversary Model, and Goals

We extend the informal discussion in § 8 by adopting the **Canetti–Krawczyk (CK)** authenticated-key-exchange model specialized to the one-way-trust setting that PQS embraces. The active, adaptive adversary $\mathfrak{A}$

- has full control of the network (eavesdrop, drop, delay, replay, reorder, inject);
- can obtain chosen-ciphertext queries to the IND-CCA KEM (Kyber or McEliece) and chosen-message queries to the EUF-CMA signature (Dilithium or SPHINCS+);
- may compromise long-term server secrets *after* or *before* a run;
- owns a large-scale quantum computer **after protocol termination**.

Target properties:

Label	Informal statement
Server-Auth	Client accepts ⇔ it verified a Dilithium / SPHINCS+ signature under the server's certified key.
Key-Secrecy	The session key *k* is indistinguishable from random for $\mathfrak{A}$ unless IND-CCA (KEM) or EUF-CMA (SIG) is broken.
Forward Secrecy (FS)	Compromise of any static signing key after the session leaks no past ciphertext.
Post-Compromise Security (PCS)	Optional symmetric & asymmetric ratchets (§ 5.11 flags 0x12–0x14) recover secrecy once a fresh ratchet completes.
Replay / Downgrade	Sequence-number ‖ UTC field inside every MACed header rejects stale or mixed-version packets. PQS Specification

9.2 Handshake-Level Cryptanalysis

Phase (flag)	Critical check	Result & argument
Connect Req (0x01)	*kid* + *cfg* → hash sch = H(cfg‖kid‖pvk)	Binds protocol string to session; mismatch aborts → **no silent downgrade**.
Connect Resp (0x02)	Server signs pkh = H(pk‖hdr) → spkh	Forgery ⇒ EUF-CMA break; hash couples *pk* to header (inc. timestamp) → **MitM blocked**.

Exchange Req (0x07)	Client verifies spkh, encapsulates sec	If verify fails → abort; cpt = Enc_pk(sec) is IND-CCA → **k remains secret**.
Exchange Resp (0x08)	Server decapsulates, derives (k_1,k_2)	Equality of *k* on both sides is implicit; explicit confirmation occurs in **Establish Verify** (0x10).
Establish Verify (0x10)	Flag indicates success / error	Any cryptographic mismatch triggers error flag; data phase never starts with diverging keys.

Key-secrecy sketch k = KDF(sec‖sch), where *sec* is IND-CCA secret and sch is public. 𝔄 can replace *sch* only by forging the signature in Connect Resp; otherwise the output of **SHAKE** behaves as a PRF, yielding a uniform key.

9.3 Data-Phase Confidentiality & Integrity

The stream cipher **RCS** is keyed with k_tx , k_rx; each packet:

1. Serializes a 21-byte header (flag‖seq‖utc‖len).
2. Runs c = RCS_AEAD(k, header, m) producing ciphertext ‖ KMAC-tag.
3. Receiver re-computes the tag; mismatch aborts.

Assuming RCS is a PRF-secure wide-block cipher and KMAC is UF-CMA, the channel attains **IND-CPA + INT-CTXT**. Because transmit and receive use *independent* keys, even a tag-forgery oracle on one direction yields no information about the opposite direction.

9.4 Resistance to Standard Attacks

Attack vector	Mitigation in PQS	Residual risk
Man-in-the-Middle	Full-packet signature of server *pk*; all subsequent packets MAC'd with fresh key.	None, unless signature or KEM broken.
Replay	seq (64-bit) + utc (±Δ window) inside MAC'd header.	Requires clock sync; widen Δ only under NTP.
Downgrade	cfg hashed into sch; any mismatch aborts.	Client must pin cfg string it demands.
Key-Compromise Impersonation	Client has no long-term secret; attacker with server's signing key can impersonate server *until key rotation*.	Mitigated by short-lived certs.

Side channels	Reference implementation constant-time; no table lookups in Kyber/Dilithium fast path.	RCS uses AES-NI – must fall back to masked S-boxes on non-AES CPUs.
Quantum adversary	Kyber/McEliece IND-CCA; Dilithium/SPHINCS+ EUF-CMA; SHA-3 family resists Grover (≥ 256-bit security margin).	Quantum brute-force halves RCS strength: prefer 512-bit profile for long-term secrets.

9.5 Performance Snapshot

Measured on Ryzen 7 5800U (gcc-14):

Step	Kyber-768/Dilithium-3	McEliece-4608/SPHINCS-256s
Keypair + sign (server)	0.32 ms	4.9 ms
Encaps + verify (client)	0.24 ms	5.1 ms
Decaps (server)	0.15 ms	1.7 ms
Cipher setup (SHAKE+RCS)	0.04 ms	0.04 ms
Total 3-RTT latency	**≈ 0.9 ms LAN**	**≈ 12 ms LAN**

Bandwidth overhead: 21-byte header + 32-byte (Kyber) or 64-byte (McEliece) tag per packet; handshake < 2 KiB.

9.6 Comparison with Classical SSH and OpenSSH-Hybrid

Dimension	PQS 1.0	OpenSSH-9.7 (RSA/ECDH)	OpenSSH-ML-KEM (draft)
Post-quantum	**Native** (Kyber/McEliece, Dilithium)	✗	Partial (KEM only)
Authentication	One-way (server)	Mutual (optional cert)	Mutual
RTT to first byte	3	4	4
FS / PCS	✓ / ✓ (sym & asym ratchets)	✓ / ✗	✓ / ✗
Replay defense	seq ‖ utc in MAC	seq only	seq only

MitM surface	Signed *pk* + cfg-hash; no fallback algs	Signature on hostkey only; KEX list suffers downgrade unless disabled	Same as OpenSSH classic
Code footprint (LOC, client)	~12 k	~180 k	~190 k
Throughput (AES-NI)	≈ 9 Gb/s @512-bit RCS	≈ 11 Gb/s AES-CTR	≈ 10 Gb/s AES-CTR

PQS trades the **mutual** authentication of SSH for a lighter, one-way trust—well-suited to cloud or IoT deployments where servers are centralized and clients are numerous. Compared with the OpenSSH hybrid draft, PQS offers PCS, stronger replay defenses, and a significantly smaller trusted code base.

9.7 Conclusion

Under standard hardness assumptions (IND-CCA Kyber/McEliece, EUF-CMA Dilithium/SPHINCS+, PRF-secure SHAKE/KMAC, and an unbroken RCS), **PQS 1.0 Simplex** delivers authenticated server-only tunnels with forward secrecy, post-compromise security, and strong replay/downgrade resilience, even against quantum-capable adversaries while incurring less latency and code complexity than SSH-style protocols.

10. Application Scenarios

As quantum computing technology progresses, many existing cryptographic protocols, particularly those based on classical algorithms like **RSA**, **ECDH**, and **DSA**, will no longer be secure. **PQS** addresses this challenge by offering quantum-resistant cryptographic solutions that ensure long-term security, making it a prime candidate to replace **SSH** and other classical protocols in critical industries. Below are several key areas where PQS could be applied.

10.1 Financial Technology (Fintech)

In the financial technology industry, secure communications are essential for a variety of tasks, including secure online transactions, financial data transfers, and remote access to trading systems. SSH is currently widely used to secure communications between remote systems in financial institutions. However, with the looming threat of quantum computers, the existing cryptographic methods (such as RSA and ECDH) will soon become vulnerable.

PQS can provide a quantum-secure alternative to SSH in the following ways:

- **Quantum-Secure Transaction Processing**: Payment gateways and financial servers rely on remote secure shell access to manage systems. PQS ensures that even if quantum computing becomes a reality, the transaction channels remain secure.
- **Data Protection in Stock Trading**: Remote trading systems can use PQS to prevent unauthorized access or tampering with trading data.
- **Long-Term Data Confidentiality**: Fintech companies that store large amounts of sensitive data, such as transaction records and customer information, will benefit from PQS's forward secrecy and resistance to quantum attacks. This ensures that data remains secure well into the future, even after quantum computers become operational.

10.2 Government and Military Communication

Government and military communications require the highest level of security, especially for classified data and remote access to critical infrastructure. **SSH** is commonly used in secure environments for server management, file transfers, and system monitoring. However, its reliance on classical cryptographic primitives makes it vulnerable to future quantum threats.

PQS offers several advantages for secure government and military applications:

- **Secure Remote Access to Critical Infrastructure**: PQS can replace SSH for remote management of servers and critical systems that control infrastructure such as electricity grids, water supply systems, and communication networks.
- **Classified Communications**: PQS ensures the confidentiality and integrity of sensitive information, protecting against both classical and quantum attacks. This is particularly crucial for secure messaging and file transfers involving classified government and military data.

- **Post-Quantum Secure Diplomatic Communication**: Diplomatic channels used by embassies and consulates for secure communication with the home country can be upgraded with PQS to prevent future breaches of sensitive information.

By implementing PQS, governments can future-proof their communication infrastructure, ensuring long-term security even as quantum computing develops.

10.3 Healthcare Systems

The healthcare industry is increasingly reliant on remote access solutions for managing medical devices, electronic health records (EHRs), and telemedicine platforms. SSH is widely used to manage and secure these systems. However, the sensitive nature of healthcare data, combined with the long-term requirement to store patient records securely, makes the adoption of quantum-resistant cryptography essential.

PQS can improve the security of healthcare systems in the following ways:

- **Secure Access to Medical Devices**: Remote management of critical medical devices, such as MRI machines, ventilators, and infusion pumps, can be secured using PQS to prevent unauthorized access or tampering.
- **Telemedicine Platforms**: Doctors can use PQS to securely communicate with patients over encrypted channels, ensuring that sensitive medical information is protected.
- **Secure Health Record Storage**: PQS ensures that patient data, including health records and diagnostic results, remain encrypted and safe from quantum-enabled breaches, preserving patient privacy in the long term.

The **US HIPAA** law requires the protection of sensitive patient information. PQS helps organizations comply with these regulations by providing advanced encryption that remains secure against quantum attacks, thus mitigating future risks.

10.4 Critical Infrastructure

Critical infrastructure systems that control national power grids, transportation networks, water supply systems, and industrial control systems require secure remote management solutions. SSH is often used for remote access to these systems, but as the quantum threat becomes more prominent, it is crucial to transition to post-quantum solutions like PQS.

- **Quantum-Secure Power Grid Management**: Power grids, which require constant remote monitoring and control, can benefit from PQS's secure key exchange and authentication to protect against both classical and quantum threats.
- **Secure Transportation Networks**: Autonomous and remotely managed transportation systems, including train control systems and smart highways, can use PQS to ensure that malicious actors cannot compromise their communications.
- **Industrial Control Systems (ICS)**: Remote access to ICS, such as SCADA systems used in manufacturing, can be secured with PQS to protect against unauthorized control or sabotage.

By integrating PQS, critical infrastructure can be protected against the vulnerabilities posed by future quantum threats, ensuring national security and system reliability.

10.5 Cloud Service Providers

Cloud computing environments rely heavily on remote shell protocols like SSH to manage data centers, virtual machines, and customer environments. As quantum computers become more capable, securing the communication between cloud providers and their customers will be essential to maintaining trust in cloud-based services.

- **Secure Cloud Data Centers**: PQS can replace SSH in data centers where administrators manage thousands of virtual machines and sensitive data. This will ensure that critical administrative tasks, such as system updates and security patching, are quantum-safe.
- **Post-Quantum Secure Cloud Storage**: Customers that rely on cloud storage for confidential business or personal data can use PQS to securely access and manage their data.
- **Virtual Private Cloud (VPC) Management**: PQS can be used to manage remote access to VPCs, ensuring that customer environments remain secure even in a quantum computing world.

By adopting PQS, cloud service providers can offer quantum-safe communication channels to their customers, ensuring continued trust in their services.

10.6 PQS as a Replacement for SSH

PQS is designed as a **post-quantum alternative to SSH**, offering quantum-resistant cryptography without sacrificing the functionality that SSH provides for secure remote access and communication. Some key comparisons include:

- **Key Exchange Security**: SSH relies on ECDH and RSA, which are vulnerable to quantum attacks. PQS uses Kyber and Dilithium, offering quantum resistance while maintaining secure key exchange and authentication.
- **Message Authentication**: PQS uses KMAC for message integrity, offering quantum-resistant authentication in place of SSH's HMAC-based approach, which will be vulnerable in a quantum world.
- **Scalability and Performance**: PQS, with its optimized post-quantum algorithms, can provide similar or better performance compared to SSH when used in environments that require high-throughput, secure communication, such as cloud environments or data centers.

PQS is an ideal replacement for SSH, providing **future-proof security**, ensuring that systems remain safe from both classical and quantum threats.

11. Conclusion

The **Post Quantum Shell (PQS)** represents a significant advancement in secure communication protocols, providing robust protection against both classical and quantum adversaries. As quantum computing capabilities evolve, many currently used cryptographic systems, including SSH, will become vulnerable. PQS addresses this critical challenge by incorporating post-quantum cryptographic primitives that are designed to remain secure even when faced with powerful quantum attacks.

11.1 Summary of Key Findings

Throughout the analysis, several key advantages of PQS over traditional protocols like SSH have emerged:

11.1.1 Quantum-Resistant Cryptography

The primary strength of PQS lies in its foundation on post-quantum cryptographic primitives. By using Kyber, Dilithium, RCS and Keccak, PQS ensures that its cryptographic operations such as key exchange, encryption, and digital signatures are secure against both classical and quantum adversaries. These algorithms rely on mathematical problems (such as lattice-based, hash-based, and code-based cryptography) that are believed to be hard for quantum computers to solve.

- **Kyber** provides strong protection for key exchange.
- **McEliece** proven code-based key encapsulation alternative.
- **Dilithium** ensures that signatures are quantum-safe and secure.
- **SPHINCS+** strong hash based signature scheme alternative.
- **Keccak (SHA-3)** and **KMAC** provide secure hashing and message authentication.
- **RCS** provides a powerful post-quantum symmetric cipher with AEAD message authentication.

This quantum-resistance makes PQS a highly secure protocol that will remain effective even as quantum computers become a reality.

11.1.2 Enhanced Key Exchange Security

Compared to SSH, which relies on vulnerable ECDH and RSA for key exchange, PQS uses Kyber or McEliece, which is are resistant to quantum attacks. The key exchange process in PQS is also authenticated using Dilithium signatures, ensuring that the public keys cannot be forged or compromised.

The **message valid-time check** (previously referred to as UTC_t) and other security mechanisms within PQS ensure protection against replay attacks and man-in-the-middle attacks, providing enhanced integrity and authenticity for every communication session.

11.1.3 Future-Proof Security

PQS is built with future security challenges in mind. Its use of quantum-resistant cryptography ensures that it will continue to provide secure communication channels long after quantum computers become practical. This future-proofing makes PQS an ideal replacement for SSH, particularly in industries where long-term confidentiality and integrity are critical, such as financial technology (fintech), government, military, healthcare, and cloud service providers.

As countries like the United States and the the nations of the European Union begin to mandate the use of stronger encryption standards (e.g., 256-bit keys), PQS offers a viable and efficient alternative to classical protocols, which would otherwise struggle with the increased computational load of larger key sizes.

11.1.4 Performance and Scalability

Despite its use of quantum-resistant algorithms, PQS has been designed to remain efficient and scalable. The post-quantum algorithms used in PQS such as Kyber, and RCS are optimized for performance, ensuring that PQS can handle large volumes of concurrent connections in environments like data centers and cloud platforms without introducing significant latency or overhead.

This makes PQS an ideal solution for industries that require high throughput and scalability, such as cloud computing and fintech, where millions of transactions or remote connections need to be securely managed.

11.2 Potential for Wide Adoption

As quantum computing moves from theory to practice, many industries will be forced to transition away from classical cryptographic protocols like SSH and TLS to ensure the security of their communications and data. PQS is well-positioned to replace these older protocols, offering:

- **Quantum-Secure Remote Access**: Industries that rely on remote access to manage infrastructure, such as cloud service providers, government agencies, and critical infrastructure operators, can deploy PQS as a quantum-resistant alternative to SSH.
- **Secure Transactions in Fintech**: Financial institutions can adopt PQS to ensure that online banking, payment systems, and transaction data remain secure against future quantum threats.
- **Long-Term Confidentiality in Healthcare**: Healthcare organizations can use PQS to protect patient data, ensuring compliance with privacy laws like HIPAA and maintaining the confidentiality of sensitive medical records well into the future.

By adopting PQS, organizations in these sectors can future-proof their operations against quantum-based attacks while maintaining the same functionality and performance offered by current classical protocols.