Quantum Secure Tunneling Protocol – QSTP 1.0

Revision 1.0b, October 19, 2024

John G. Underhill – john.underhill@protonmail.com

The Quantum Secure Tunneling Protocol (QSTP) is an authenticated three-party system, designed to secure communication between a client and server using quantum-safe encryption and certificate-based authentication. The protocol ensures secure key exchange, message integrity, and confidentiality.

**Contents Page**

[Foreword](#Foreward) 2

1: [Introduction](#Introduction) 3

2: [Scope](#Scope) 4

3: [Terms and Definitions](#Cryptographic_Primitives) 7

4: [Cryptographic Primitives](#Cryptographic_Primitives) 11

5: [Protocol Description](#Protocol_Description) 13

6: [Mathematical Description](#Mathematical_Description) 19

7: [QSTP API](#QSTP_API) 28

8: [Security Analysis](#Security_Analysis) 46

9: [QSTP Cryptanalysis](#QSTP_Cryptanalysis) 48

10: [Conclusion](#Conclusion) 51

# **Foreword**

This document is intended as the preliminary draft of a new standards proposal, and as a basis from which that standard can be implemented. We intend that this serves as an explanation of this new technology, and as a complete description of the protocol.

This document is the first revision of the specification of QSTP, 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 QSTP specification.

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

QSTP, the algorithm constituting the QSTP 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 Quantum Secure Tunneling Protocol (QSTP) is a cryptographic protocol designed to enable secure communication between clients and servers. It leverages a key exchange process authenticated by a root server, which signs the certificates of the participating servers and clients. The protocol's design focuses on ensuring post-quantum security, protecting communications even in the presence of quantum computing threats.

QSTP differs from traditional peer-to-peer tunneling protocols, such as Quantum Secure Messaging Protocol (QSMP), by incorporating a root server for certificate management. This server is responsible for signing the certificates that authenticate key exchanges between the client and server.

The key goals of QSTP are:

1. **Post-quantum security**: The protocol is designed to withstand quantum-based attacks by relying on secure cryptographic primitives.
2. **Authenticated key exchange**: The client and server engage in a secure key exchange process, using certificates issued by the root server.
3. **Confidentiality and Integrity**: QSTP ensures that communication between the client and server remains confidential and tamper-proof.

The following sections will outline the design, mathematical foundation, and security analysis of QSTP, providing an in-depth review of the design and implementation of a QSTP crypto-system.

**2.** **Scope**

This document provides a comprehensive description of the QSTP secure tunneling protocol, focusing on establishing encrypted and authenticated communication channels between two hosts. It outlines the complete processes involved in key exchange, message authentication, and the establishment of secure communication tunnels using the QSTP protocol.

The QSTP specification includes detailed descriptions of the following elements:

* **Cryptographic Primitives**: An in-depth look at the mathematical foundations and quantum-resistant algorithms used in QSTP.
* **Key Derivation Functions**: The specific methods and algorithms used to generate secure session keys from shared secrets.
* **Client-to-Server Messaging Protocols**: A step-by-step breakdown of the message exchanges required to establish a secure communications stream between clients and servers.

**2.1 Application**

QSTP is designed primarily for institutions and organizations that require secure communication channels to handle sensitive information exchanged between remote terminals. It is ideally suited for sectors where data confidentiality, integrity, and authenticity are paramount, including financial institutions, government agencies, defense contractors, and enterprises managing critical infrastructure.

The protocol is versatile enough to be applied in various settings, such as secure messaging, VPNs, and other network communication systems where robust encryption and authentication are essential. QSTP's design ensures that even if the cryptographic landscape changes due to advancements in quantum computing, its security framework remains resilient and flexible.

**Mandatory Protocol Components**:

* The key exchange, message authentication, and encryption functions defined in this document are integral to the construction of a QSTP communication stream. These components MUST be implemented to ensure secure operations and protocol compliance.

**Use of Keywords for Compliance**:

* **SHOULD**: Indicates best practices or recommended settings that are not compulsory but are strongly advised for optimal performance and security.
* **SHALL**: Denotes mandatory requirements that must be followed to ensure full compliance with the QSTP protocol. Deviations from these guidelines result in non-conformity and may compromise the protocol's effectiveness.

**2.2 Protocol Flexibility and Use Cases**

QSTP is engineered to be highly adaptable, supporting various deployment scenarios ranging from simple client-server architectures to more complex multi-party distributed systems. This flexibility makes it ideal for cloud-based infrastructures, secure messaging applications, VPNs, and IoT networks that demand high-performance encryption and authentication.

**Key use cases for QSTP include:**

* **Institutional Communications**: Securely encrypting and authenticating sensitive data exchanges between financial institutions, government agencies, and corporate networks.
* **Internet of Things (IoT)**: Enabling secure communication for connected devices that require lightweight, efficient, and scalable encryption protocols to protect data integrity.
* **Secure Messaging Platforms**: Providing end-to-end encryption for messaging services that need to resist both classical and quantum attacks.

The protocol's ability to integrate with existing network infrastructure without requiring extensive modifications ensures that organizations can transition to post-quantum security seamlessly while maintaining high levels of operational efficiency.

**2.3 Compliance and Interoperability**

The QSTP protocol is designed to maintain strict compliance with its core cryptographic principles while ensuring interoperability with other secure communication frameworks. To guarantee that different QSTP implementations can interact securely, adherence to the standards outlined in this document is crucial.

To facilitate future upgrades and adaptations, QSTP is structured to support modular cryptographic components. This approach allows for the addition of new cryptographic primitives or the enhancement of existing ones without disrupting the overall architecture. As new advancements in cryptographic techniques emerge, QSTP can be easily updated to include these innovations, maintaining its position as a state-of-the-art security protocol.

**Key elements of compliance:**

* **Interoperability Standards**: QSTP is developed to work seamlessly with other post-quantum cryptographic standards, ensuring that its communication channels can operate in diverse network environments.
* **Modular Design**: The protocol's flexible design allows for straightforward upgrades, facilitating the incorporation of future cryptographic advancements with minimal impact on existing deployments.

**2.4 Recommendations for Secure Implementation**

In addition to outlining the core requirements for QSTP's secure communication, this document provides best practice recommendations to enhance implementation security, performance, and reliability:

* **Regular Cryptographic Updates**: Institutions are advised to keep informed of developments in post-quantum cryptography and to periodically update their cryptographic algorithms to maintain compliance with industry standards.
* **Security Audits and Assessments**: Routine security assessments should be conducted to identify potential vulnerabilities in the protocol implementation and to apply necessary mitigations.
* **Infrastructure Optimization**: It is recommended to configure network infrastructure in a way that supports QSTP's low-latency, high-throughput capabilities, ensuring that performance remains consistent even under heavy loads.

These guidelines aim to help organizations maximize QSTP's security potential, ensuring that their communication channels remain secure against both current and future threats.

**2.5 Document Organization**

This document is structured to provide a detailed, logical flow of information about the QSTP protocol's operation and implementation. It includes the following key sections:

* **Cryptographic Primitives**: Detailed explanations of the mathematical algorithms that form the foundation of QSTP's encryption and authentication processes.
* **Key Exchange Mechanisms**: Comprehensive breakdowns of how session keys are established securely through QSTP's key exchange protocol.
* **Message Authentication**: Detailed descriptions of the techniques used to verify the authenticity and integrity of messages exchanged within QSTP communications.
* **Error Handling and Fault Tolerance**: Guidelines on how to manage protocol errors and disruptions while maintaining secure and stable communication channels.
* **Implementation Examples**: Practical examples, code snippets, and detailed use cases demonstrating the integration of QSTP in various application contexts.

**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**

QSTP relies on a robust 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 QSTP's encryption, key exchange, and authentication processes.

**4.1 Asymmetric Cryptographic Primitives**

QSTP 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**: A lattice-based key encapsulation mechanism that provides secure, 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 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 algorithm that offers fast signing and verification processes while maintaining strong security guarantees against quantum attacks.
* **Sphincs+**: A hash-based signature scheme known for its stateless nature, 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 QSTP's key exchange and signature operations remain secure in the face of evolving computational threats.

**4.2 Symmetric Cryptographic Primitives**

QSTP's symmetric encryption employs the **Rijndael Cryptographic Stream (RCS)**, a stream cipher adapted from the Rijndael (AES) algorithm 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 and number of transformation rounds, thereby enhancing its resistance to differential and linear cryptanalysis.
* **Enhanced Key Schedule**: The key schedule in RCS is cryptographically strengthened using Keccak, ensuring that derived keys are resistant to known attacks, including algebraic-based and differential attacks.
* **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 AVX/AVX2/AVX512 intrinsics and AES-NI instructions embedded in modern CPUs.

**4.3 Hash Functions and Key Derivation**

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

* **SHA-3**: SHA-3 serves as QSTP's primary hash function, providing secure, collision-resistant hashing capabilities.
* **SHAKE**: QSTP 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.

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

**5.** **Protocol Description**

The QSTP key exchange is a three-party, one-way trust, client-server key exchange model in which the client trusts the server based on certificate authentication facilitated by a root domain security server. A single shared secret is securely exchanged between the server and client, and used to create an encrypted tunnel. Designed for efficiency, the QSTP 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 root domain security server (RDS) distributes a public signature verification certificate to every client in its domain. This certificate is used to authenticate an QSTP application server (QAS) public certificate. The server's certificate is used to verify signed messages from the server to the client.

**5.1 Root Certificate Generation**

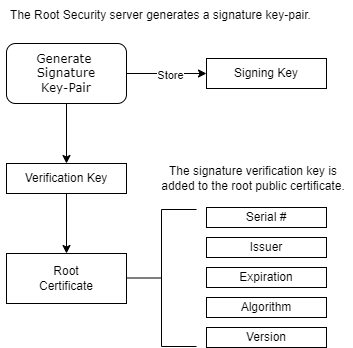


Figure 5.1: QSTP root certificate creation.

The root security server generates a root certificate that is distributed to every client on the network. The root server stores the private signing key, which it uses to sign application server certificates. The application server certificate is hashed, and the hash is signed by the root asymmetric signing key, with the signature populating the *csig* field of the severs certificate. The client uses the asymmetric signature verification key embedded in the root certificate to verify a server certificate’s authenticity.

**5.2 Connection Request**

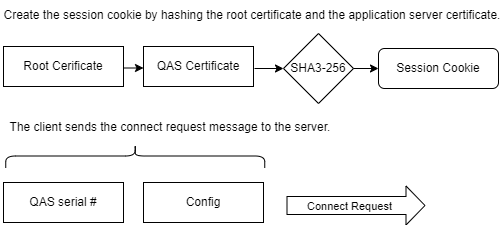
****

Figure 5.2: QSTP 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 certificate serial number and the protocol configuration string.
3. The client hashes the root certificate and the application server’s certificate. This hash is stored in the session cookie state value (*sch*) and is used as a unique session identifier. This approach ensures that the correct certificates are referenced and that the session state is uniquely identifiable between different root-server pairings.
4. The client adds the application server’s certificate serial number and the configuration string to the message, and sends the connection request to the server.

**5.3 Connection Response**

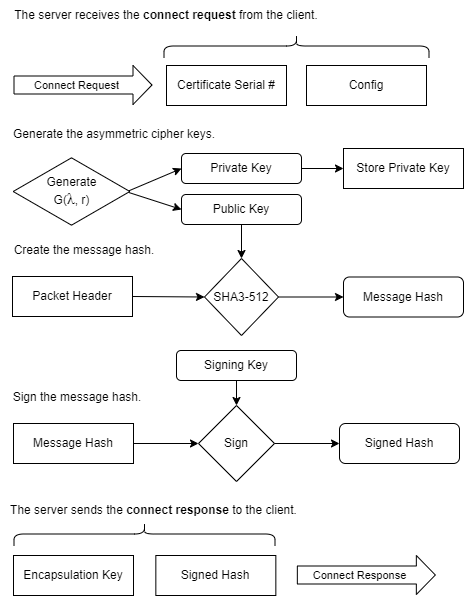
****

Figure 5.3: QSTP 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 certificate serial number provided in the request. If the verification key is not found, the server sends an *unknown certificate* 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 certificate. If all these fields are validated successfully, the server loads the signature key into its active state.
5. The server generates a new public/private asymmetric cipher keypair. It hashes the public encapsulation key and the serialized connection response packet header, and signs this hash with its private signing key.
6. 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.

**5.4 Exchange Request**

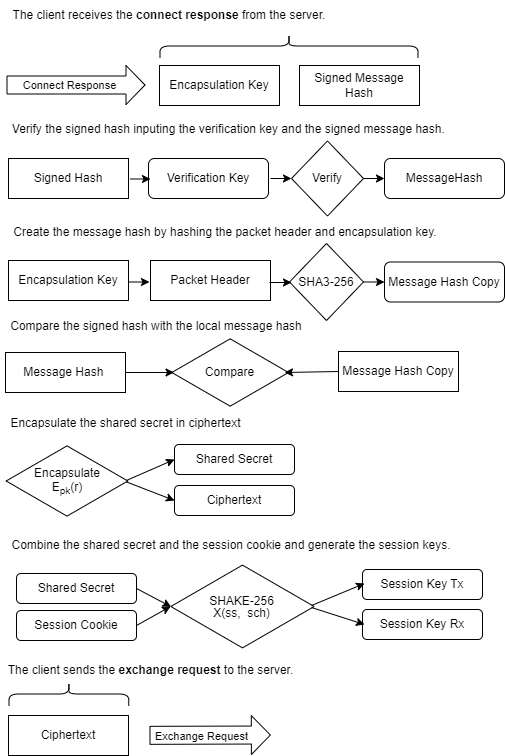


Figure 5.4: QSTP client exchange request.

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 to the server 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 connection 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 symmetric 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.

**5.5 Exchange Response**

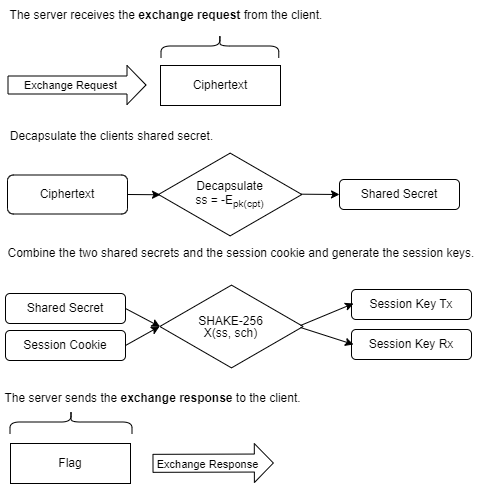


Figure 5.5: QSTP server exchange response.

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.

**5.6 Establish Verify**

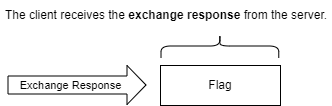


Figure 5.6: QSTP client establish request.

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.

**6.** **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

-Esk -The asymmetric cipher decapsulation function and secret key

Epk -The asymmetric cipher encapsulation function and public key

Ssk -Sign data with the secret signature key

Vpk -Verify a signature the public verification key

*cfg* -The protocol configuration string

cprrx -A receive channels symmetric cipher instance

cprtx -A transmit channels symmetric cipher instance

*cpt* -The symmetric ciphers cipher-text

*cpta* -The asymmetric ciphers cipher-text

-Ek -The symmetric decryption function and key

Ek -The symmetric encryption function and key

H -The hash function (SHA3)

*k*, *mk* -A symmetric cipher and MAC key

KDF -The key expansion function (SHAKE)

*kid* -The public keys unique identity array

Mmk -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

**Key Exchange Sequence**

**Preamble**

The client retains a copy of the root server’s public certificate. This certificate is used to validate the application server’s certificate. The server certificate contains a signature field (*csig*), that has a hash of the server certificate, signed by the root authentication server. When the client first connects to an application server, the server sends its root signed certificate to the client. The client verifies the signature in the certificate signature field using the root server’s validation key. The client hashes the certificate, and compares it to the signed hash for equivalency. The server’s certificate is further checked for a valid expiration time (which is checked every time a key exchange between the client and server is initiated), as well as for a matching protocol set, and QSTP protocol version fields. Once the server certificate has been verified, it is cached by the client, and used to validate messages signed by the application server.

The 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.

**9.1 Connect Request**

The client sends a connection request with its configuration string, and asymmetric public signature key serial number. The serial number is a multi-part 16-byte certificate identification array, used to match the intended target to the corresponding key.

The configuration string defines the cryptographic protocol set being used; this must match both implementations configuration settings. The client stores a hash of the configuration string, the serial number, and of the server’s public asymmetric signature verification-key, which is used as a session cookie during the exchange.

**Where:**

* *rcert* is the serialized root certificate.
* *scert* is the serialized server certificate.

*sch* ← H(*rcert* || *scert*)

The client sends the key identity string, and the configuration string to the server.

C{ *serial*, *cfg* } → S

**Certificate Serial Number**

The certificateserialnumber is a 16-byte array that acts as a public asymmetric serial number 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* *hash* by hashing a combination of the root server’s public certificate and the server public certificate. This session cookie (*sch*) serves as an identifier for the session, helping to ensure that the correct certificates are referenced in the exchange. The server has this cookie stored in its state, whereas the client generates the cookie when a key is initialized. The cookie will be unique to every root and application server certificate pairing.

The client then sends the key certificate serial number (*ser*) and the protocol configuration string (*cfg*) to the server to initiate the connection:

C{ *ser*, *cfg* } → S

**9.2 Connect Response**

The server responds with either an error message, or a response packet. Any error during the key exchange will generate an error-packet sent to the remote host, which will trigger a tear down of the session and network connection on both sides.

The server first checks that it has the requested asymmetric signature verification key corresponding to that host using the serial number array, then verifies that it has a compatible protocol configuration.

The server stores a hash of the configuration string, key id, and the public signature verification-key, to create the session cookie hash.

sch ← H(cfg || serial || pvk)

The server generates an asymmetric encryption key-pair, stores the private key, hashes the public encapsulation key, and signs the hash of the public encapsulation key and the serialized connect response packet header using the asymmetric signing key. The public signature verification key can itself be enveloped by a ‘chain of trust’ model, like X.509, using a signature verification extension to this protocol.

*pk*, *sk* ← G(λ, r)

*pkh* ← H(*pk*)

*spkh* ← Ssk(*pkh*)

The server sends a connect response message containing a signed hash of the public asymmetric encapsulation-key, and a copy of that key.

S{ *spkh*, *pk* } → C

**Key Verification and Protocol Check**

The server begins by verifying that it has the correct certificate that corresponds to the client's request, by comparing the certificate serial number in the request to the server’s certificate serial number.

It then checks that its protocol configuration matches the one specified by the client.

**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* ← G(λ, *r*)

Create a hash of the public asymmetric encapsulation key, the session cookie, and the serialized *connect response* packet header (*sh*).

*pkh* ← H(*pk* || *sch* || *sh*)

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

*spkh* ← Ssk(*pkh*)

**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:

S{ *spkh*, *pk* } → C

**9.3 Exchange Request**

The client verifies the signature of the hash, generates its own hash of the asymmetric cipher public key, cookie, and packet header, and compares it with the hash contained in the message.

If the hash matches, the client uses the public-key to encapsulate a shared secret.

**Signature Verification and Hash Check**

The client begins by verifying the signature of the hash using the server's public verification key. It 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 signature of the public key hash. If the verification is successful, the process continues; otherwise, the key exchange fails.

Vpk(H(*pk*)) ← (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* ← Epk(*r*)

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* ← KDF(*sec* || *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*.

cprrx(*k2*, *n2*)

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

cprtx(*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* } → S

**9.4 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* ← -Esk(*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 (SHAKE) 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* ← KDF(*sec* || *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*.

cprrx(*k1*, *n1*)

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

cprtx(*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.

S{ *f* } → C

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

**9.5 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.

**9.6 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 ← Ek(*m*)

1. **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 (Mmk) is updated with the serialized packet header (*sh*) and the ciphertext (*cpt*) to produce the MAC code (*mc*).

*mc* ← Mmk(*sh*, *cpt*)

1. **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`* ← Mmk(*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.

1. **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* ← -Ek(*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.

**7:** **QSTP API**

**7.1 Definitions and Shared API**

**Header:**

qstp.h

**Description:**

The QSTP header contains shared constants, types, and structures, as well as function calls common to both the QSTP server and client implementations.

**Structures:**

The **QSTP\_ERROR\_STRINGS** is a static string-array containing QSTP error descriptions, used in the error reporting functionality.

|  |  |
| --- | --- |
| Data Set | Purpose |
| QSTP\_ERROR\_STRINGS | A string array of readable error descriptions. |

Table 7.1a QSTP error strings.

The **QSTP\_CONFIG\_STRING** is a static string containing the readable QSTP configuration string.

|  |  |
| --- | --- |
| Data Set | Purpose |
| QSTP\_CONFIG\_STRING | The QSTP configuration string. |

Table 7.1b QSTP configuration string.

The **qstp\_packet** contains the QSTP packet structure.

|  |  |  |  |
| --- | --- | --- | --- |
| Data Name | Data Type | Bit Length | Function |
| flag | Uint8 | 0x08 | The packet flag |
| msglen | Uint32 | 0x20 | The packets message length |
| sequence | Uint64 | 0x40 | The packet sequence number |
| utctime | Uint64 | 0x40 | The UTC packet creation time |
| message | Uint8 Array | Variable | The packets message data |

Table 7.1c QSTP packet structure.

The **qstp\_server\_certificate** contains the QSTP server public certification state.

|  |  |  |  |
| --- | --- | --- | --- |
| Data Name | Data Type | Bit Length | Function |
| csig | Uint8 Array | Variable | The certificates root signed hash |
| issuer | Uint8 Array | 0x0200 | The certificates issuer identity |
| rootser | Uint8 Array | 0x80 | The root certificate serial number |
| serial | Uint8 Array | 0x80 | The certificate serial number |
| verkey | Uint8 Array | Variable | The asymmetric signatures verification key |
| expiration | Uint64 | 0x40 | The certificate expiration time |
| algorithm | Uint8 | 0x10 | The algorithm identifier |
| version | Uint8 | 0x10 | The QSTP version number |

Table 7.1d QSTP server public certificate structure.

The **qstp\_server\_signature\_key** contains the QSTP server signature key state.

|  |  |  |  |
| --- | --- | --- | --- |
| Data Name | Data Type | Bit Length | Function |
| schash | Uint8 Array | 0x0200 | The certificate token hash |
| issuer | Uint8 Array | 0x0200 | The certificates issuer identity |
| sigkey | Uint8 Array | Variable | The server certificate signing key |
| serial | Uint8 Array | 0x80 | The certificate serial number |
| verkey | Uint8 Array | Variable | The asymmetric signatures verification key |
| expiration | Uint64 | 0x40 | The certificate expiration time |
| algorithm | Uint8 | 0x10 | The algorithm identifier |
| version | Uint8 | 0x10 | The QSTP version number |

Table 7.1e QSTP server public certificate structure.

The **qstp\_keep\_alive\_state** contains the QSTP keep alive state.

|  |  |  |  |
| --- | --- | --- | --- |
| Data Name | Data Type | Bit Length | Function |
| target | Struct | Variable | The target host socket structure |
| etime | Uint64 | 0x40 | The keep alive epoch time |
| seqctr | Uint64 | 0x40 | The keep alive packet sequence number |
| recd | Boolean | 0x08 | The keep alive response received status |

Table 7.1f QSTP keep alive state structure.

**Enumerations:**

The **qstp\_messages** enumeration defines the network messages.

|  |  |
| --- | --- |
| Enumeration | Purpose |
| qstp\_messages\_none | No message was specified |
| qstp\_messages\_accept\_fail | The socket accept failed |
| qstp\_messages\_listen\_fail | The listener socket could not connect |
| qstp\_messages\_bind\_fail | The listener socket could not bind to the address |
| qstp\_messages\_create\_fail | The listener socket could not be created |
| qstp\_messages\_connect\_success | The server connected to a host |
| qstp\_messages\_receive\_fail | The socket receive function failed |
| qstp\_messages\_allocate\_fail | The server memory allocation request has failed |
| qstp\_messages\_kex\_fail | The key exchange has experienced a failure |
| qstp\_messages\_disconnect | The server has disconnected the client |
| qstp\_messages\_disconnect\_fail | The server has disconnected the client due to an error |
| qstp\_messages\_socket\_message | The server has had a socket level error |
| qstp\_messages\_queue\_empty | The server has reached the maximum number of connections |
| qstp\_messages\_listener\_fail | The server listener socket has failed |
| qstp\_messages\_sockalloc\_fail | The server has run out of socket connections |
| qstp\_messages\_decryption\_fail | The message decryption has failed |
| qstp\_messages\_keepalive\_fail | The keepalive function has failed |
| qstp\_messages\_keepalive\_timeout | The keepalive period has been exceeded |
| qstp\_messages\_connection\_fail | The connection failed or was interrupted |
| qstp\_messages\_invalid\_request | The function received an invalid request |

Table 7.1g QSTP messages enumeration.

The **qstp\_errors** enumeration is a list of the QSTP error code values.

|  |  |
| --- | --- |
| Enumeration | Purpose |
| qstp\_error\_none | No error was detected |
| qstp\_error\_accept\_fail | The socket accept function returned an error |
| qstp\_error\_authentication\_failure | The symmetric cipher had an authentication failure |
| qstp\_error\_bad\_keep\_alive | The keep alive check failed |
| qstp\_error\_channel\_down | The communications channel has failed |
| qstp\_error\_connection\_failure | The device could not make a connection to the remote host |
| qstp\_error\_connect\_failure | The transmission failed at the KEX connection phase |
| qstp\_error\_decapsulation\_failure | The asymmetric cipher failed to decapsulate the shared secret |
| qstp\_error\_decryption\_failure | The decryption authentication has failed |
| qstp\_error\_establish\_failure | The transmission failed at the KEX establish phase |
| qstp\_error\_exchange\_failure | The transmission failed at the KEX exchange phase |
| qstp\_error\_hash\_invalid | The public-key hash is invalid |
| qstp\_error\_hosts\_exceeded | The server has run out of socket connections |
| qstp\_error\_invalid\_input | The expected input was invalid |
| qstp\_error\_invalid\_request | The packet flag was unexpected |
| qstp\_error\_keep\_alive\_expired | The keep alive has expired with no response |
| qstp\_error\_keepalive\_timeout | The decryption authentication has failed |
| qstp\_error\_key\_expired | The QSTP public key has expired |
| qstp\_error\_key\_unrecognized | The key identity is unrecognized |
| qstp\_error\_keychain\_fail | The ratchet operation has failed |
| qstp\_error\_listener\_fail | The listener function failed to initialize |
| qstp\_error\_memory\_allocation | The server has run out of memory |
| qstp\_error\_message\_time\_invalid | The packet has valid time expired |
| qstp\_error\_packet\_unsequenced | The packet was received out of sequence |
| qstp\_error\_random\_failure | The random generator has failed |
| qstp\_error\_receive\_failure | The receiver failed at the network layer |
| qstp\_error\_signature\_failure | The signing function has failed |
| qstp\_error\_transmit\_failure | The transmitter failed at the network layer |
| qstp\_error\_verify\_failure | The expected data could not be verified |
| qstp\_error\_unknown\_protocol | The protocol string was not recognized |
| qstp\_error\_verify\_failure | The expected data could not be verified |

Table 7.1h QSTP errors enumeration.

The **qstp\_flags** enum contains the QSTP packet flags.

|  |  |
| --- | --- |
| Enumeration | Purpose |
| qstp\_flag\_none | No flag was specified |
| qstp\_flag\_connect\_request | The QSTP key-exchange client connection request flag |
| qstp\_flag\_connect\_response | The QSTP key-exchange server connection response flag |
| qstp\_flag\_connection\_terminate | The connection is to be terminated |
| qstp\_flag\_encrypted\_message | The message has been encrypted flag |
| qstp\_flag\_exstart\_request | The QSTP key-exchange client exstart request flag |
| qstp\_flag\_exstart\_response | The QSTP key-exchange server exstart response flag |
| qstp\_flag\_exchange\_request | The QSTP key-exchange client exchange request flag |
| qstp\_flag\_exchange\_response | The QSTP key-exchange server exchange response flag |
| qstp\_flag\_establish\_request | The QSTP key-exchange client establish request flag |
| qstp\_flag\_establish\_response | The QSTP key-exchange server establish response flag |
| qstp\_flag\_keep\_alive\_request | The packet contains a keep alive request |
| qstp\_flag\_remote\_connected | The remote host is connected flag |
| qstp\_flag\_remote\_terminated | The remote host has terminated the connection |
| qstp\_flag\_session\_established | The exchange is in the established state |
| qstp\_flag\_session\_establish\_verify | The exchange is in the established verify state |
| qstp\_flag\_unrecognized\_protocol | The protocol string is not recognized |
| qstp\_flag\_asymmetric\_ratchet\_request | The host has received an asymmetric key ratchet request |
| qstp\_flag\_symmetric\_ratchet\_request | The host has received a symmetric key ratchet request |
| qstp\_flag\_transfer\_request | The host has received a transfer request |
| Qst p\_flag\_error\_condition | The connection experienced an error |

Table 7.1i QSTP flags enumeration.

The **qstp\_configuration\_sets** enumeration defines the QSTP configuration sets.

|  |  |
| --- | --- |
| Enumeration | Purpose |
| qstp\_configuration\_set\_none | No configuration set was specified |
| qstp\_configuration\_set\_dilithium1\_kyber1\_rcs256\_shake256 | The Dilithium-S1/Kyber-S1/RCS-256/SHAKE-256 algorithm set |
| qstp\_configuration\_set\_dilithium3\_kyber3\_rcs256\_shake256 | The Dilithium-S3/Kyber-S3/RCS-256/SHAKE-256 algorithm set |
| qstp\_configuration\_set\_dilithium5\_kyber5\_rcs256\_shake256 | The Dilithium-S5/Kyber-S5/RCS-256/SHAKE-256 algorithm set |
| qstp\_configuration\_set\_dilithium5\_kyber6\_rcs512\_shake512 | The Dilithium-S5/Kyber-S6/RCS-256/SHAKE-256 algorithm set |
| qstp\_configuration\_set\_dilithium1\_mceliece1\_rcs256\_shake256 | The Dilithium-S1/McEliece-S1/RCS-256/SHAKE-256 algorithm set |
| qstp\_configuration\_set\_dilithium3\_mceliece3\_rcs256\_shake256 | The Dilithium-S3/McEliece-S3/RCS-256/SHAKE-256 algorithm set |
| qstp\_configuration\_set\_dilithium5\_mceliece5\_rcs256\_shake256 | The Dilithium-S5/McEliece-S5a/RCS-256/SHAKE-256 algorithm set |
| qstp\_configuration\_set\_dilithium5\_mceliece6\_rcs256\_shake256 | The Dilithium-S5/McEliece-S6/RCS-256/SHAKE-256 algorithm set |
| qstp\_configuration\_set\_dilithium5\_mceliece7\_rcs256\_shake256 | The Dilithium-S5/McEliece-S7/RCS-256/SHAKE-256 algorithm set |
| qstp\_configuration\_set\_sphincsplus1f\_mceliece1\_rcs256\_shake256 | The SPHINCS+-S1F/McEliece-S1/RCS-256/SHAKE-256 algorithm set |
| qstp\_configuration\_set\_sphincsplus1s\_mceliece1\_rcs256\_shake256 | The SPHINCS+-S1S/McEliece-S1/RCS-256/SHAKE-256 algorithm set |
| qstp\_configuration\_set\_sphincsplus3f\_mceliece3\_rcs256\_shake256 | The SPHINCS+-S3F/McEliece-S3/RCS-256/SHAKE-256 algorithm set |
| qstp\_configuration\_set\_sphincsplus3s\_mceliece3\_rcs256\_shake256 | The SPHINCS+-S3S/McEliece-S3/RCS-256/SHAKE-256 algorithm set |
| qstp\_configuration\_set\_sphincsplus5f\_mceliece5\_rcs256\_shake256 | The SPHINCS+-S5F/McEliece-S5a/RCS-256/SHAKE-256 algorithm set |
| qstp\_configuration\_set\_sphincsplus5s\_mceliece5\_rcs256\_shake256 | The SPHINCS+-S5S/McEliece-S5a/RCS-256/SHAKE-256 algorithm set |
| qstp\_configuration\_set\_sphincsplus5f\_mceliece6\_rcs256\_shake256 | The SPHINCS+-S5F/McEliece-S5b/RCS-256/SHAKE-256 algorithm set |
| qstp\_configuration\_set\_sphincsplus5s\_mceliece6\_rcs256\_shake256 | The SPHINCS+-S5S/McEliece-S5b/RCS-256/SHAKE-256 algorithm set |
| qstp\_configuration\_set\_sphincsplus5f\_mceliece7\_rcs256\_shake256 | The SPHINCS+-S5F/McEliece-S5c/RCS-256/SHAKE-256 algorithm set |
| qstp\_configuration\_set\_sphincsplus5s\_mceliece7\_rcs256\_shake256 | The SPHINCS+-S5S/McEliece-S5c/RCS-256/SHAKE-256 algorithm set |

Table 7.1j QSTP configuration set enumeration.

**Constants:**

|  |  |  |
| --- | --- | --- |
| Constant Name | Value | Purpose |
| QSTP\_CONFIG\_DILITHIUM\_KYBER | N/A | Sets the asymmetric cryptographic primitive-set to Dilithium/Kyber |
| QSTP\_CONFIG\_DILITHIUM\_MCELIECE | N/A | Sets the asymmetric cryptographic primitive-set to Dilithium/McEliece |
| QSTP\_CONFIG\_SPHINCS\_MCELIECE | N/A | Sets the asymmetric cryptographic primitive-set to Sphincs+/McEliece |
| QSTP\_SERVER\_PORT | 32119 | The default server port address |
| QSTP\_ASYMMETRIC\_CIPHER\_TEXT\_SIZE | Variable | The byte size of the asymmetric cipher-text array |
| QSTP\_ASYMMETRIC\_PRIVATE\_KEY\_SIZE | Variable | The byte size of the asymmetric cipher private-key array |
| QSTP\_ASYMMETRIC\_PUBLIC\_KEY\_SIZE | Variable | The byte size of the asymmetric cipher public-key array |
| QSTP\_ASYMMETRIC\_SIGNING\_KEY\_SIZE | Variable | The byte size of the asymmetric signature signing-key array |
| QSTP\_ASYMMETRIC\_VERIFICATION\_KEY\_SIZE | Variable | The byte size of the asymmetric signature verification-key array |
| QSTP\_ASYMMETRIC\_SIGNATURE\_SIZE | Variable | The byte size of the asymmetric signature array |
| QSTP\_ACTIVE\_VERSION | 1 | The QSTP active version |
| QSTP\_CERTIFICATE\_ALGORITHM\_SIZE | 1 | The algorithm type size |
| QSTP\_CERTIFICATE\_DESIGNATION\_SIZE | 1 | The size of the certificate designation field |
| QSTP\_CERTIFICATE\_EXPIRATION\_SIZE | 16 | The certificate expiration date length |
| QSTP\_CERTIFICATE\_HASH\_SIZE | 32 | The size of the certificate the certificate hash in bytes |
| QSTP\_CERTIFICATE\_ISSUER\_SIZE | 32 | The maximum certificate issuer string length. |
| QSTP\_CERTIFICATE\_LINE\_LENGTH | 64 | The line length of the printed MPDC certificate |
| QSTP\_CERTIFICATE\_DEFAULT\_PERIOD | 31536000 | The default certificate validity period in milliseconds |
| QSTP\_CERTIFICATE\_DEFAULT\_DURATION\_DAYS | 365 | The default number of days a public key remains valid |
| QSTP\_CERTIFICATE\_DEFAULT\_DURATION\_SECONDS | 31536000 | The number of seconds a public key remains valid |
| QSTP\_CERTIFICATE\_LINE\_LENGTH | 64 | The line length of the printed QSTP public key |
| QSTP\_CERTIFICATE\_MAXIMUM\_PERIOD | 63072000 | The maximum certificate validity period in milliseconds |
| QSTP\_CERTIFICATE\_MINIMUM\_PERIOD | 86400 | The minimum certificate validity period in milliseconds |
| QSTP\_CERTIFICATE\_SERIAL\_SIZE | 16 | The certificate serial number field length |
| QSTP\_CERTIFICATE\_SERIAL\_ENCODED\_SIZE | 32 | The hex encoded certificate serial number string length |
| QSTP\_CERTIFICATE\_SIGNED\_HASH\_SIZE | Variable | The line length of the printed QSTP public key |
| QSTP\_CERTIFICATE\_TIMESTAMP\_SIZE | 8 | The key expiration timestamp size |
| QSTP\_CERTIFICATE\_VERSION\_SIZE | 1 | The version id size |
| QSTP\_CONNECTIONS\_INIT | 1000 | The initial QSTP connections queue size |
| QSTP\_CONNECTIONS\_MAX | 50000 | The maximum number of connections |
| QSTP\_CONNECTION\_MTU | 1500 | The QSTP packet buffer size |
| QSTP\_KEEPALIVE\_TIMEOUT | 120000 | The keep alive timeout in milliseconds (2 minutes) |
| QSTP\_MACTAG\_SIZE | 32 | The mac key size |
| QSTP\_NONCE\_SIZE | 32 | The size of the symmetric cipher nonce |
| QSTP\_PACKET\_ERROR\_SEQUENCE | 0xFF00000000000000 | The packet error sequence number |
| QSTP\_PACKET\_ERROR\_SIZE | 1 | The packet error message size |
| QSTP\_PACKET\_FLAG\_SIZE | 1 | The packet flag size |
| QSTP\_PACKET\_HEADER\_SIZE | 21 | The QSTP packet header size |
| QSTP\_PACKET\_MESSAGE\_LENGTH\_SIZE | 4 | The size of the packet message length |
| QSTP\_PACKET\_MESSAGE\_MAX | 0x3D090000 | The maximum message size used during the key exchange (1 GB) |
| QSTP\_PACKET\_REVOCATION\_SEQUENCE | 0xFF | The revocation packet sequence number |
| QSTP\_PACKET\_SEQUENCE\_SIZE | 8 | The size of the packet sequence number |
| QSTP\_PACKET\_SEQUENCE\_TERMINATOR | 0xFFFFFFFF | The sequence number of a packet that closes a connection |
| QSTP\_PACKET\_TIME\_THRESHOLD | 60 | The maximum number of seconds a packet is valid |
| QSTP\_SECRET\_SIZE | 32 | The size of the shared secret for each channel |
| QSTP\_CLIENT\_PORT | 32118 | The default client port address |
| QSTP\_SERVER\_PORT | 32119 | The default server port address |
| QSTP\_ROOT\_PORT | 32120 | The default root port address |
| QSTP\_SYMMETRIC\_KEY\_SIZE | 32 | The Simplex 256-bit symmetric cipher key size |
| QSTP\_STORAGE\_PATH\_MAX | 260 | The maximum path size |
| QSTP\_ROOT\_CERTIFICATE\_SIZE | Variable | The root certificate length |
| QSTP\_ROOT\_SIGNATURE\_KEY\_SIZE | Variable | The root signature key size |
| QSTP\_SERVER\_CERTIFICATE\_SIZE | Variable | A server certificate length |
| QSTP\_SERVER\_SIGNATURE\_KEY\_SIZE | Variable | A server signing key length |

Table 7.1k QSTP constants.

The **qstp\_connection\_state** contains the QSTP connection state.

|  |  |  |  |
| --- | --- | --- | --- |
| Data Name | Data Type | Bit Length | Function |
| target | Struct | 0x440 | The target host socket structure |
| rxcpr | Struct | Variable | The receive channel cipher state |
| txcpr | Struct | Variable | The transmit channel cipher state |
| rxseq | Uint64 | 0x40 | The receive channels packet sequence number |
| txseq | Uint64 | 0x40 | The transmit channels packet sequence number |
| cid | Uint32 | 0x20 | The connections instance count |
| exflag | Uint8 | 0x08 | The KEX position flag |
| receiver | bool | 0x08 | The hosts receiver status |

Table 7.1l QSTP connection state structure.

**Functions:**

**Configuration From String**  
*Convert a configuration string to an enumeration member.*  
qstp\_configuration\_sets qstp\_configuration\_from\_string(const char\* config)

**Configuration To String**  
*Convert a configuration enumeral to a configuration string.*  
const char\* qstp\_configuration\_to\_string(qstp\_configuration\_sets cset)

**Connection Close**  
*Close the network connection between hosts.*  
void qstp\_connection\_close(qstp\_connection\_state\* cns, qstp\_errors err, bool notify)

**Connection State Dispose**  
*Reset the connection state to zero.*  
void qstp\_connection\_state\_dispose(qstp\_connection\_state\* cns)

**Decrypt Packet**  
*Decrypt a message and copy it to the message output.*  
qstp\_errors qstp\_decrypt\_packet(qstp\_connection\_state\* cns, uint8\_t\* message, size\_t\* msglen, const qstp\_network\_packet\* packetin)

**Encrypt Packet**  
*Encrypt a message and build an output packet.*  
qstp\_errors qstp\_encrypt\_packet(qstp\_connection\_state\* cns, qstp\_network\_packet\* packetout, const uint8\_t\* message, size\_t msglen)

**Error To String**  
*Return a pointer to a string description of an error code.*  
const char\* qstp\_error\_to\_string(qstp\_errors error)

**Connection Dispose**  
*Reset the connection state.*  
void qstp\_connection\_close(qstp\_connection\_state\* cns)

**Decrypt Packet**  
*Decrypt a message and copy it to the message output.*  
qstp\_errors qstp\_decrypt\_packet(qstp\_connection\_state\* cns, uint8\_t\* message, size\_t\* msglen, const qstp\_packet\* packetin)

**Encrypt Packet**  
*Encrypt a message and copy it to a packet.*  
qstp\_errors qstp\_encrypt\_packet(qstp\_connection\_state\* cns, qstp\_packet\* packetout, const uint8\_t\* message, size\_t\* msglen)

**Header Create**  
*Populate a packet header and set the creation time.*  
void qstp\_header\_create(qstp\_network\_packet\* packetout, qstp\_flags flag, uint64\_t sequence, uint32\_t msglen)

**Header Validate**  
*Validate a packet header and timestamp.*  
qstp\_errors qstp\_header\_validate(qstp\_connection\_state\* cns, const qstp\_network\_packet\* packetin, qstp\_flags flag, uint64\_t sequence, uint32\_t msglen)

**Get Error Description**  
*Get the error string description.*  
const char\* qstp\_get\_error\_description(qstp\_messages emsg)

**Header Deserialize**  
*Deserialize a byte array to a packet header.*  
void qstp\_packet\_header\_deserialize(const uint8\_t\* header, qstp\_packet\* packet)

**Header Serialize**  
*Serialize a packet header to a byte array.*  
void qstp\_packet\_header\_serialize(const qstp\_packet\* packet, uint8\_t\* header)

**Log Error**  
*Log the message, socket error, and string description.*  
void qstp\_log\_error(const qstp\_messages emsg, qsc\_socket\_exceptions err, const char\* msg)

**Log Message**  
*Log the message.*  
void qstp\_log\_message(const qstp\_messages emsg)

**Log Write**  
*Log the message, and string description.*  
void qstp\_log\_write(const qstp\_messages emsg, const char\* msg)

**Packet Clear**  
*Clear a packet's state.*  
size\_t qstp\_packet\_clear(const qstp\_packet\* packet)

**Packet Error Message**  
*Populate a packet structure with an error message.*  
void qstp\_packet\_error\_message(qstp\_network\_packet\* packet, qstp\_errors error)

**Packet Set UTC Time**  
*Sets the local UTC seconds time in the packet header.*  
void qstp\_packet\_set\_utc\_time(qstp\_network\_packet\* packet)

**Packet Time Valid**  
*Checks the local UTC seconds time against the packet sent time for validity within the packet time threshold.*  
bool qstp\_packet\_time\_valid(const qstp\_network\_packet\* packet)

**Packet Set UTC Time**  
*Sets the local UTC seconds time in the packet header.*  
void qstp\_packet\_set\_utc\_time(qstp\_network\_packet\* packet)

**Packet To Stream**  
*Serialize a packet to a byte array.*  
size\_t qstp\_packet\_to\_stream(const qstp\_packet\* packet, uint8\_t\* pstream)

**Root Certificate Compare**  
*Compare two root certificates for equivalence.*  
bool qstp\_root\_certificate\_compare(const qstp\_root\_certificate\* a, const qstp\_root\_certificate\* b)

**Root Certificate Decode**  
*Copy a root certificate structure to a file.*  
bool qstp\_root\_certificate\_decode(qstp\_root\_certificate\* root, const char\* enck, size\_t enclen)

**Roor Certificate Deserialize**  
*Deserialize a root certificate*  
void qstp\_root\_certificate\_deserialize(qstp\_root\_certificate\* root, const uint8\_t input[QSTP\_ROOT\_CERTIFICATE\_SIZE])

**Root Certificate Encode**  
*Encode a root certificate into a readable string*  
size\_t qstp\_root\_certificate\_encode(char\* enck, size\_t enclen, const qstp\_root\_certificate\* root)

**Root Certificate Encoded Size**  
*Get the root encoding string size.*  
size\_t qstp\_root\_certificate\_encoded\_size()

**Root Certificate Extract**  
*Extract the root certificate from the server key.*  
void qstp\_root\_certificate\_extract(qstp\_root\_certificate\* root, const qstp\_root\_signature\_key\* kset)

**Root Certificate Hash**  
*Hash a root certificate*  
void qstp\_root\_certificate\_hash(uint8\_t output[QSTP\_CERTIFICATE\_HASH\_SIZE], const qstp\_root\_certificate\* root)

**Root Certificate Serialize**  
*Serialize a root certificate to an array.*  
void qstp\_root\_certificate\_serialize(uint8\_t output[QSTP\_ROOT\_CERTIFICATE\_SIZE], const qstp\_root\_certificate\* root)

**Root Certificate Sign**  
*Sign a server certificate.*  
size\_t qstp\_root\_certificate\_sign(qstp\_server\_certificate\* cert, const qstp\_root\_certificate\* root, const uint8\_t\* rsigkey)

**Root Certificate Verify**  
*Verify a certificate is signed by the root.*  
bool qstp\_root\_certificate\_verify(const qstp\_root\_certificate\* root, const qstp\_server\_certificate\* cert)

**Root Certificate To File**  
*Copy a root certificate structure to a file.*  
bool qstp\_root\_certificate\_to\_file(const qstp\_root\_certificate\* root, const char\* fpath)

**Root File To Certificate**  
*Copy a certificate from a file to a root certificate structure.*  
bool qstp\_root\_file\_to\_certificate(qstp\_root\_certificate\* root, const char\* fpath)

**Root File To Key**  
*Copy a root key from a file to a root key structure.*  
bool qstp\_root\_file\_to\_key(qstp\_root\_signature\_key\* kset, const char\* fpath)

**Root Get Issuer**  
*Get the root certificate issuer name.*  
void qstp\_root\_get\_issuer(char issuer[QSTP\_CERTIFICATE\_ISSUER\_SIZE])

**Root Key Deserialize**  
*Deserialize a root signature key.*  
void qstp\_root\_key\_deserialize(qstp\_root\_signature\_key\* kset, const uint8\_t input[QSTP\_ROOT\_SIGNATURE\_KEY\_SIZE])

**Root Get Issuer**  
*Get the root certificate issuer name.*  
void qstp\_root\_get\_issuer(char issuer[QSTP\_CERTIFICATE\_ISSUER\_SIZE])

**Root Key To File**  
*Copy a root key structure to a file.*  
bool qstp\_root\_key\_to\_file(const qstp\_root\_signature\_key\* kset, const char\* fpath)

**Root Key Serialize**  
*Serialize a root key to an array.*  
void qstp\_root\_key\_serialize(uint8\_t serk[QSTP\_ROOT\_SIGNATURE\_KEY\_SIZE], const qstp\_root\_signature\_key\* kset)

**Server Certificate Compare**  
*Compare two server certificates for equivalence.*  
bool qstp\_server\_certificate\_compare(const qstp\_server\_certificate\* a, const qstp\_server\_certificate\* b)

**Server Certificate Deserialize**  
*Deserialize a server stream to a certificate structure.*  
void qstp\_server\_certificate\_deserialize(qstp\_server\_certificate\* cert, const uint8\_t input[QSTP\_SERVER\_CERTIFICATE\_SIZE])

**Server Certificate Encode**  
*Encode a public server certificate into a readable string.*  
size\_t qstp\_server\_certificate\_encode(char\* enck, size\_t enclen, const qstp\_server\_certificate\* cert)

**Server Certificate Encoding Size**  
*Get the encoding size of a server certificate.*  
size\_t qstp\_server\_certificate\_encoded\_size()

**Server Certificate Extract**  
*Extract the server certificate from the server key.*  
void qstp\_server\_certificate\_extract(qstp\_server\_certificate\* cert, const qstp\_server\_signature\_key\* kset)

**Server Certificate Hash**  
*Hash a server certificate.*  
void qstp\_server\_certificate\_hash(uint8\_t output[QSTP\_CERTIFICATE\_HASH\_SIZE], const qstp\_server\_certificate\* cert)

**Server Root Certificate Hash**  
*Hash the root and server certificates.*  
void qstp\_server\_root\_certificate\_hash(uint8\_t rshash[QSTP\_CERTIFICATE\_HASH\_SIZE], const qstp\_root\_certificate\* root, const qstp\_server\_certificate\* cert)

**Server Certificate To File**  
*Copy a server certificate structure to a file.*  
bool qstp\_server\_certificate\_to\_file(const qstp\_server\_certificate\* cert, const char\* fpath)

**Server File To Certificate**  
*Copy a serialized certificate from a file to a server certificate structure.*  
bool qstp\_server\_file\_to\_certificate(qstp\_server\_certificate\* cert, const char\* fpath)

**Server File To Key**  
*Copy a key from a file to a server key structure.*  
bool qstp\_server\_file\_to\_key(qstp\_server\_signature\_key\* kset, const char\* fpath)

**Server Get Issuer**  
*Get the server certificate issuer name.*  
void qstp\_server\_get\_issuer(char issuer[QSTP\_CERTIFICATE\_ISSUER\_SIZE])

**Server Key Deserialize**  
*Deserialize a server signature to a key structure.*  
void qstp\_server\_key\_deserialize(qstp\_server\_signature\_key\* kset, const uint8\_t input[QSTP\_SERVER\_SIGNATURE\_KEY\_SIZE])

**Server Key Serialize**  
*Serialize a server key structure to an array.*  
void qstp\_server\_key\_serialize(uint8\_t output[QSTP\_SERVER\_SIGNATURE\_KEY\_SIZE], const qstp\_server\_signature\_key\* kset)

**Server Key To File**  
*Copy a server key structure to a file.*  
bool qstp\_server\_key\_to\_file(const qstp\_server\_signature\_key\* kset, const char\* fpath)

**Server Version From String**  
*Copy a server key structure to a file.*  
uint8\_t qstp\_version\_from\_string(const char\* sver, size\_t sverlen

**Server Version To String**  
*Convert the version number to a hexidecimal string.*  
void qstp\_version\_to\_string(char\* sver, uint8\_t version)

**Server Get Issuer**  
*Get the server certificate issuer name.*  
void qstp\_server\_get\_issuer(char issuer[QSTP\_CERTIFICATE\_ISSUER\_SIZE])

**Stream To Packet**  
*Deserialize a byte array to a packet.*  
void qstp\_stream\_to\_packet(const uint8\_t\* pstream, qstp\_packet\* packet)

**7.2 Server API**

**Header:**

server.h

**Description:**

Functions used to implement the QSTP server.

**Functions:**

**Expiration Check**

*Check the expiration status of a server key.*  
bool qstp\_server\_expiration\_check(const qstp\_server\_signature\_key\* kset)

**Key Generate**

*Generate a signature key.*  
void qstp\_server\_key\_generate(qstp\_server\_signature\_key\* kset, const char issuer[QSTP\_CERTIFICATE\_ISSUER\_SIZE], uint32\_t exp)

**Pause**

*Pause the server, suspending new joins.*  
void qstp\_server\_pause()

**Quit**

*Quit the server, closing all connections.*  
void qstp\_server\_quit()

**Resume**

*Resume the server listener function from a paused state.*  
void qstp\_server\_resume()

**Listen IPv4**

*Run the IPv4 networked key exchange function. Returns the connected socket and the QSTP server connection state.*  
qstp\_errors qstp\_server\_start\_ipv4(qsc\_socket\* source, const qstp\_server\_signature\_key\* kset,void (\*receive\_callback)(qstp\_connection\_state\*, const char\*, size\_t), void (\*disconnect\_callback)(qstp\_connection\_state\*))

**Listen IPv6**

*Run the IPv6 networked key exchange function. Returns the connected socket and the QSTP server state.*  
qstp\_errors qstp\_server\_start\_ipv6(qsc\_socket\* source, const qstp\_server\_signature\_key\* kset, void (\*receive\_callback)(qstp\_connection\_state\*, const char\*, size\_t), void (\*disconnect\_callback)(qstp\_connection\_state\*))

**7.3 Client API**

**Header:**  
client.h

**Description:**  
Functions used to implement the QSTP client.

**Functions**

**Connect IPv4**

*Run the IPv4 networked key exchange function. Returns the connected socket and the QSTP client state.*  
qstp\_errors qstp\_client\_connect\_ipv4(const qstp\_root\_certificate\* root, const qstp\_server\_certificate\* cert, const qsc\_ipinfo\_ipv4\_address\* address, uint16\_t port, void (\*send\_func)(qstp\_connection\_state\*), void (\*receive\_callback)(qstp\_connection\_state\*, const char\*, size\_t))

**Connect IPv6**

*Run the IPv6 networked key exchange function. Returns the connected socket and the QSTP client state.*  
qstp\_errors qstp\_client\_connect\_ipv6(const qstp\_root\_certificate\* root, const qstp\_server\_certificate\* cert, const qsc\_ipinfo\_ipv6\_address\* address, uint16\_t port, void (\*send\_func)(qstp\_connection\_state\*), void (\*receive\_callback)(qstp\_connection\_state\*, const char\*, size\_t))

**8.** **Security Analysis**

The security analysis of QSTP focuses on evaluating its robustness against various cryptographic attacks, ensuring its reliability in real-world scenarios. The analysis primarily covers the key exchange, encryption, and message authentication mechanisms.

**8.1 Key Exchange Security**

The QSTP key exchange relies on public-key cryptography and certificates signed by a trusted root server. Key elements of the security model include:

1. **Asymmetric Key Signatures:** The server's public certificate *Cₛᵣᵥ* is signed by the root server using *Sₛᵣₒₒₜ*, ensuring that an authenticated party has control over the key exchange. This process prevents man-in-the-middle (MITM) attacks, as the client can verify the authenticity of the server’s public certificate using the root server’s public key *Pₛᵣₒₒₜ*.
2. **Shared Secret Generation:** The client generates a shared secret key using the server’s public asymmetric cipher key. Since the operation uses ephemeral asymmetric cipher keys it provides forward secrecy. Even if an attacker compromises a later session key, past communications remain secure.
3. **Replay and Timing Attacks:** The use of nonces (*Nₛ)* in each key exchange session prevents replay attacks. Since the nonces are unique for each session, an attacker cannot reuse old messages or signatures. Additionally, the timestamps included in packet headers help mitigate timing attacks.
4. **Quantum-Resistant Algorithms:** The use of post-quantum primitives means that QSTP can provide long-term security against quantum computing attacks. This makes QSTP future-proof against quantum adversaries.

**8.2 Symmetric Key Encryption**

QSTP uses symmetric-key encryption to secure communication between the client and server once the shared secret key is established. The symmetric encryption algorithm RCS is designed to provide:

1. **Confidentiality:** The encryption ensures that unauthorized parties cannot read the plaintext message. The security of the encryption depends on the strength of the chosen algorithm and key length.
2. **Authenticated Encryption:** QSTP applies authenticated encryption with associated data (AEAD) to ensure both confidentiality and integrity. The encryption and MAC generation processes are integrated, meaning an attacker cannot alter the ciphertext without being detected.
3. **Resistance to Known Attacks:** By using an AEAD symmetric cipher (RCS), QSTP provides security against attacks such as chosen-ciphertext attacks (CCA), ciphertext manipulation, and differential cryptanalysis.

**8.3 Message Authentication and Integrity**

The Message Authentication Code (MAC) by RCS ensures the integrity of transmitted messages. The key points of MAC security include:

1. **MAC Forgery Resistance:** Since the MAC is computed over the encrypted message *C* using the session key, an adversary cannot forge a valid MAC without access to the session key. This prevents unauthorized modifications to the ciphertext or message.
2. **Tamper Detection:** Any attempt to modify the message will result in a MAC mismatch upon verification, triggering an error. This mechanism ensures that the recipient can trust that the message has not been tampered with.
3. **Protection Against Replay Attacks:** The inclusion of nonces and sequence numbers and timestamps in each message ensures that even if a message is intercepted, it cannot be replayed. Each message has unique associated data, which is used in the MAC computation.

**8.4 Forward Secrecy**

QSTP provides forward secrecy through its key exchange mechanism. Even if a long-term private key (such as *Sₛᵣᵥ* or *Sₗᵢ*) is compromised in the future, previous session keys and encrypted communications remain secure. The ephemeral session key *kₛₑₛₛᵢₒₙ* is unique to each communication session, preventing retrospective decryption.

**8.5 Resistance to Quantum Attacks**

As quantum computing continues to advance, traditional public-key algorithms like RSA and ECC become vulnerable to quantum attacks. QSTP's use of post-quantum cryptographic primitives in its key exchange and encryption mechanisms ensures that it is resistant to quantum adversaries. Algorithms such as lattice-based or code-based cryptography (e.g., Kyber, McEliece) are designed to withstand quantum attacks, providing future-proof security.

**9. Cryptanalysis of the QSTP Key-Exchange and Tunnel**

**9.1 Methodological Framework**

Analysis follows the **Canetti–Krawczyk (CK)** model for authenticated key-exchange and the **ACCE** paradigm for channel protocols. The adversary is granted full network control, adaptive chosen-ciphertext oracle access to the KEM, adaptive chosen-message queries to the signature scheme, compromise of long-term keys, and post-session quantum computation. Security objectives are:

* **Server authentication** (one-way trust)
* **Explicit key confirmation**
* **IND-CCA confidentiality** & **INT-CTXT integrity** of tunnel data
* **Forward secrecy (FS)** and **post-compromise security (PCS)**
* **Replay, reflection and downgrade resilience**

Notation mirrors § 6 (*Mathematical Description*) of the specification.

**9.2 Handshake Message Flow (idealized)**

1. *C* → *S* : ⟨*ser, cfg*⟩
2. *S* → *C* : ⟨\*pk, Sigₛ(H(pk‖hdr))⟩
3. *C* → *S* : ⟨\*cpt = Encₖₑₘ(pk, χ)⟩
4. *S* → *C* : ⟨*flag = ok*⟩

Both sides derive

*k* = KDF(*sec* ∥ *sch*), *sec* = Deckem​(*sk*,*cpt*), *sch*=H(*rcert* ∥ *scert*)

**9.3 Cryptanalytic Evaluation**

|  |  |  |
| --- | --- | --- |
| Property | Argument | Result |
| Server authentication | *C* verifies Sigₛ with *S*’s certified verification key, itself rooted in the RDS certificate. Forgery ⇒ EUF-CMA breach of Dilithium / SPHINCS+. | **Provably secure** |
| Key secrecy / IND-CCA | k is the output of SHAKE on ⟨uniform sec, public sch⟩. IND-CCA security of Kyber / McEliece ⇒ sec indistinguishable; SHAKE is a PRF. | **Secure** |
| Forward secrecy | *S*’s KEM key-pair is freshly generated per run and sk is erased after step 4; compromise of static signing keys leaks no past sessions. | **Full FS** |
| Post-compromise | Optional symmetric ratchet (§ 5.6) injects fresh entropy every 2²⁴ packets; break-and-recover adversary cannot decrypt forward traffic without breaking SHA-3. | **Achieved** if ratchet used |
| KCI / UKS | Client has no long-term key; Unknown-Key-Share excluded because sch binds both certificate hashes into key derivation. | **Not vulnerable** |
| Replay & reflection | 32-bit UTC + 64-bit sequence authenticated as AEAD-AAD; packets outside ±Δt or out-of-order are dropped (§ 5. 2–5. 6). | **Secure** within Δt window |
| Downgrade | cfg is hashed into sch; any mismatch aborts. No fallback modes. | **Not vulnerable** |

**Observations & Potential Weaknesses**

1. **Root-key monoculture** – compromise of the RDS private key subverts authentication for all domains. Mitigation: short-lived root epochs and CRLite-style revocation.
2. **Clock synchronization** – the valid-time field demands < Δt drift; embed a monotone counter or challenge nonce for IoT deployments with poor RTCs.
3. **RCS scrutiny** – tunnel security finally reduces to the un-standardized RCS + KMAC AEAD; commission open cryptanalysis; offers a more conservative AES-GCM compliance profile.

**9.4 Formal Verification with Tamarin**

* **Auth\_S** *Caccept⇒Ssent(pk,Sigs)C accept ⇒ S sent (pk, Sigₛ)Caccept⇒Ssent(pk,Sigs​)*
* **SK\_Secrecy** Session key k is unreachable by the adversary unless IND-CCA or EUF-CMA breaks.
* **Forward Secrecy** After erasure of KEM secrets, disclosure of long-term signing keys leaks no prior k.

Average proof times: *Auth\_S* ≈ 25 s, *SK\_Secrecy* ≈ 30 s, on an Intel i7-1260P with 16 GB RAM.

**9.5 Comparative Summary**

|  |  |  |
| --- | --- | --- |
| Dimension | QSTP (root-anchored) | QSMP-SIMPLEX |
| Authentication | One-way (server) via RDS | One-way (server) |
| Session-key entropy | 256 bits | 256 bits |
| RTT | 2 | 2 |
| Root-CA dependency | **Yes** | No |
| Forward secrecy | ✓ | ✓ |
| PCS (ratchet) | Optional | Optional |

**10.** **Conclusion**

The QSTP (Quantum Secure Tunneling Protocol) offers a robust framework for secure communication between a client and server, leveraging a trusted root server for certificate signing and authentication. Key aspects of QSTP include:

* **Security Foundation:** QSTP provides strong security guarantees through its use of post-quantum asymmetric cryptography for the key exchange, authenticated encryption for data transmission, and authenticated symmetric encryption for message integrity.
* **Post-Quantum Resilience:** The integration of post-quantum cryptographic algorithms ensures that QSTP is resistant to attacks from future quantum adversaries, offering long-term security.
* **Key Exchange:** The protocol’s reliance on certificates signed by a root server strengthens its security model, enabling clients to verify the authenticity of the server’s public key. The sever also signs messages, enabling client’s to verify messages emanating from the server.
* **Authentication and Confidentiality:** QSTP achieves both confidentiality and message integrity through authenticated encryption, protecting against various cryptographic attacks, including man-in-the-middle, replay, and ciphertext manipulation.
* **Forward Secrecy:** The use of ephemeral keys in each session provides forward secrecy, ensuring that even if private keys are compromised in the future, previous communications remain secure.

QSTP is a well-designed protocol that aligns with modern cryptographic standards and ensures secure communication in an era where post-quantum threats are a growing concern. Its careful design around authentication, encryption, and key exchange makes it a solid candidate for secure communications in environments requiring high levels of security, such as financial institutions, government agencies, and post-quantum cryptographic applications.