Dual Key Tunneling Protocol - DKTP

Revision 1.0a, July 30, 2025

John G. Underhill – contact@qrcscorp.ca

This document is an engineering level description of the Dual Key Tunneling Protocol (DKTP), an authenticated and encrypted network tunneling protocol. This document describes the network protocol DKTP; the key exchange, authentication scheme, and encrypted tunneling functions that comprise the DKTP protocol.

Contents **Page**

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 DKTP, 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 DKTP specification.

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

1. Introduction

The Dual-Key Tunneling Protocol (DKTP) was conceived as a deliberate exploration of cryptographic boundaries, inspired by a fundamental question: how close can we come to Shannon's notion of perfect secrecy using standardized, implementable cryptographic primitives? While Claude Shannon proved that true perfect secrecy is only achievable with a one-time pad and a truly random key, rendering it largely impractical for dynamic communication; this protocol represents a serious attempt to asymptotically approach that ideal. By combining the strengths of both asymmetric and symmetric cryptography in a state-synchronized, bidirectional tunnel, DKTP aspires to meet the highest attainable standard of information-theoretic security, within the constraints of real-world systems and modern computational assumptions.

This effort is more than academic; as the global security community braces for a post-quantum future, there is a pressing need for encrypted tunnel designs that are not only resilient to emerging cryptanalytic capabilities, but also architecturally forward-looking. Traditional systems like TLS and SSH, while mature and in wide-use, still depend on certificate-based trust infrastructures and classical assumptions such as elliptic curve or finite-field hardness, which are vulnerable to quantum attacks. DKTP rejects these legacy dependencies and instead defines a tunnel rooted in quantum-resistant key encapsulation, mutual contribution of secret entropy, signed ephemeral key material, and fully authenticated symmetric stream channels.

At its heart, DKTP relies on a dual-key mechanism where both communicating parties independently generate ephemeral asymmetric key-pairs and contribute shared secrets through key encapsulation mechanisms. These secrets are then fused with long-term pre-shared symmetric keys, which are continuously updated and authenticated throughout the tunnel lifecycle. The resulting tunnel keys are uniquely derived for each session and direction, transmit and receive, ensuring that even partial compromise or asymmetric impersonation yields no meaningful insight into the ciphertext streams. Each channel operates with its own instance of RCS, a Rijndael-based AEAD stream cipher with high entropy diffusion, integrated nonce handling, and built-in timing resistance. The use of RCS further strengthens the tunnel against a range of side-channel attacks and injects additional post-quantum safety into the symmetric layer.

Rather than depend on certificate authorities or revocation infrastructure, DKTP uses device-level signature keys mapped to a cryptographic identity array (kid), ensuring each peer is authenticated through an explicit verification chain. This decentralized trust model is well suited to peer-to-peer networks, sovereign systems, and embedded deployments where no live PKI infrastructure exists or is desirable. Session cookies, hashed from protocol configuration, key identity, and the exchanged public keys, anchor the entire session in a cryptographically bound context that resists downgrade, replay, and cross-session attacks. The protocol's attention to sequence validation, UTC timestamp checking, and header integrity provides a layered defense model that reinforces every stage of the exchange.

DKTP is capable of full 512-bit security; in the enhanced mode, keys, tokens, and primitives are scaled up to 512-bit. This includes the symmetric cryptography, which uses SHA3-512, SHAKE-

512, KMAC-512 and RCS-512. DKTP was designed for maximum crypto agility, making 512-bit encrypted tunneling a reality for the most high-value communications systems.

By design, DKTP achieves properties rare in conventional protocols: mutual forward secrecy, bidirectional key derivation, state advancement through key injection, and complete tunnel initialization before any user data is accepted. These attributes are critical for environments where tunnel compromise cannot be tolerated, such as financial services, government systems, or classified communication infrastructure. Moreover, the clean separation between handshake and transmission layers, along with authenticated stream cipher integration, enables compact implementations suitable for embedded systems and high-performance applications alike.

In summary, DKTP is not merely a hybrid tunnel; it is a philosophical rethinking of encrypted session design in a post-quantum world. It prioritizes entropy, state renewal, and structural asymmetry to produce an architecture that is not only cryptographically robust, but also operationally forward-looking. By pushing the envelope of what can be achieved with standardized, well-vetted primitives, DKTP offers a credible path toward a future-proof secure tunnel protocol, one that closely echoes the timeless aspiration of Shannon's perfect secrecy, within the technical boundaries of modern cryptography.

1.1 Purpose

The Dual-Key Tunneling Protocol (DKTP) was created to address the need for a cryptographically strong, future-proof encrypted tunnel that provides mutual authentication, post-quantum resistance, and full-duplex authenticated communication. Its design aligns with emerging cryptographic standards and modern threat models, with specific emphasis on resisting quantum-era adversaries and eliminating structural dependencies on traditional PKI.

The purpose of DKTP is as follows:

- **To provide a quantum-secure tunnel** that does not rely on legacy cryptographic assumptions such as RSA or elliptic curve cryptography, but instead uses post-quantum asymmetric primitives and robust symmetric designs.
- **To enable mutual authentication** through signed ephemeral keys and session-bound identity verification without dependence on certificate authorities or revocation infrastructure.
- **To create independently derived transmit and receive keys** for each direction of communication, ensuring asymmetric secrecy and preventing loopback or reflection attacks.
- **To maintain synchronized pre-shared keys** between peers, with automated forward secrecy through key injection and state updating at every session establishment.
- **To bind the cryptographic handshake to both time and configuration**, using hashed session cookies and timestamp validation to detect downgrade, replay, or misconfiguration attempts.
- **To support scalable deployments** in zero-trust, federated, or embedded systems where centralized key distribution or third-party trust anchors are impractical or undesirable.

- **To enforce integrity and confidentiality** using RCS, an authenticated stream cipher with wide-block diffusion, AEAD support, and high resistance to fault or side-channel attacks.

DKTP is suitable for securing high-risk communication environments such as encrypted infrastructure control channels, financial transaction systems, embedded secure device networks, and sovereign communications systems. It is designed not only to secure the data in motion, but to do so in a way that anticipates and defends against threats not yet realized. By combining strong, standards-based cryptographic primitives with a structurally asymmetric, state-renewing architecture, DKTP stands as a forward-looking alternative to classical tunnels, built for a world in which cryptography must remain resilient far beyond today's capabilities.

2. Scope

This document provides a comprehensive description of the DKTP 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, key ratcheting, and the establishment of secure communication tunnels using the DKTP protocol.

The DKTP specification includes detailed descriptions of the following elements:

- **Cryptographic Primitives**: An in-depth look at the mathematical foundations and quantum-resistant algorithms used in DKTP.

  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

DKTP 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. DKTP'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 DKTP 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 DKTP protocol. Deviations from these guidelines result in non-conformity and may compromise the protocol's effectiveness.

2.2 Protocol Flexibility and Use Cases

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

To facilitate future upgrades and adaptations, DKTP 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, DKTP 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**: DKTP 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 DKTP'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 DKTP's low-latency, high-throughput capabilities, ensuring that performance remains consistent even under heavy loads.

These guidelines aim to help organizations maximize DKTP'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 DKTP protocol's operation and implementation. It includes the following key sections:

- **Cryptographic Primitives**: Detailed explanations of the mathematical algorithms that form the foundation of DKTP's encryption and authentication processes.
- **Key Exchange Mechanisms**: Comprehensive breakdowns of how session keys are established securely through DKTP's key exchange protocol.
- **Message Authentication**: Detailed descriptions of the techniques used to verify the authenticity and integrity of messages exchanged within DKTP 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 DKTP 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.

3.2.18 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 DKTP:

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

DKTP 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 DKTP's encryption, key exchange, and authentication processes.

4.1 Asymmetric Cryptographic Primitives

DKTP 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. The QSC library contains an optional (non-standard) K5 parameter set option for FIPS ML-KEM, capable of approximately 320 bits of post-quantum security
- **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 DKTP's key exchange and signature operations remain secure in the face of evolving computational threats.

4.2 Symmetric Cryptographic Primitives

DKTP'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 DKTP's ability to transform raw cryptographic data into secure keys and hashes. The following primitives are used:

- **SHA-3**: SHA-3 serves as DKTP's primary hash function, providing secure, collision-resistant hashing capabilities.
- **SHAKE**: DKTP 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 DKTP's key management processes remain secure, even in scenarios involving high-risk adversaries and quantum-capable threats.

5. Protocol Components

5.1 Protocol String

The protocol string in DKTP 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 stage 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 DKTP 1.3a.

5.2 Remote Peer Key Structure

The remote host peering key is a private (secret) structure that contains the pre-shared key, the signature verification key, and associated metadata. It includes parameters such as the key expiration time, protocol string, signature verification key, and key identity array. This key is given to a remote host, and used to connect to the local host.

Parameter	Data Type	Bit Length	Function
Channel Key	Uint8 array	256	Keying Material
Configuration	Uint8 array	320	Protocol check
Expiration	Uint64	64	Validity check
Key ID	Uint8 array	128	Identification

Verification Key	Uint8 array	Variable	Authentication

Table 5.2: The client key structure.

- **Channel Key** The pre-shared secret (*pss*) used as seeding material during the encrypted tunnel establishment.
- **Configuration**: Contains the protocol string that defines the cryptographic parameters. If the protocol string on both hosts does not match, the connection is aborted.
- **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 peering key from the host.
- **Key ID**: A unique identifier for the signature verification key, facilitating quick reference on the server.
- **Verification Key**: The asymmetric signature verification key used for authenticating asymmetric encapsulation keys and data during the key exchange.

The host peering keys are considered secret, and must be exchanged over a secure connection (e.g. QKD, QSTP, or IPSec), or installed when the client is initialized.

5.3 Local Peering Key Structure

Parameter	Data Type	Bit Length	Function
Channel Key	Uint8 array	256	Keying Material
Configuration	Uint8 array	320	Protocol check
Expiration	Uint64	64	Validity check
Local Key ID	Uint8 array	128	Identification
Remote Key ID	Uint8 array	128	Identification
Signing Key	Uint8 array	Variable	Authentication
Verification Key	Uint8 array	Variable	Authentication

Table 5.3: The client key structure.

The local secret peer key structure is the peer key state generated and stored locally. It is identical to the peering state, but contains two additional fields; the remote peering identity string, and the asymmetric signing key. This key is not distributed, but is the peer key with the signature and remote identity field, required during the session establishment. Local and remote peer keys are single-use pairings, shared between two hosts. Once the peer keys have been exchanged, they are linked; the local peer key records the remote peer key identity. This value is checked each time a key exchange is run and an encrypted tunnel is created. The value stored in the local peer key for the remote host, is verified during the connect stage of the key exchange in the peer-linking function.

- **Channel Key** The pre-shared secret (*pss*) used as seeding material during the encrypted tunnel establishment.
- **Configuration**: Contains the protocol string that defines the cryptographic parameters. If the protocol string on both hosts does not match, the connection is aborted.
- **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 peering key from the host.
- **Local Key ID**: A unique identifier for the peering key assigned by the creator of the key, facilitating quick reference on the server.
- **Remote Key ID**: The key identity of the corresponding peering key; each peering key is paired with a remote peering key.
- **Signing Key**: The signature scheme signing key used to sign messages sent to the remote peer.
- **Verification Key**: The asymmetric signature verification key used for authenticating asymmetric encapsulation keys and data during the key exchange.

5.4 Keep Alive State

DKTP SIMPLEX 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 data required by DKTP operations, including cipher states, sequence counters, and the ratchet key.

Data Name	Data Type	Bit Length	Function
Cipher Receive State	Structure	Variable	Symmetric Encryption
Cipher Transmit State	Structure	Variable	Symmetric Decryption
Connection Instance	Uint32	32	Identification
Receive Sequence	Uint64	64	Packet Verification

Target	Socket struct	664	Validity check
Transmit Sequence	Uint64	64	Packet Verification

Table 5.5: The connection state structure.

- **Cipher Receive State:** the symmetric cipher's state connected to the receive channel.
- **Cipher Transmit State:** the symmetric cipher's state connected to the transmit channel.
- **Connection Instance:** the connections unique identification string.
- **ExFlag:** the protocol state flag, indicates the current state of the tunnel.
- **Receive Sequence:** the packet sequence counter for the receive channel interface.
- **Target:** the remote device TCP/IP socket connection structure.
- **Transmit Sequence:** the packet sequence counter for the transmit channel interface.

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

5.6 Client Kex State

The client key exchange (KEX) state holds information about asymmetric and symmetric keys during the key exchange process.

Data Name	Data Type	Bit Length	Function
Channel Key	Uint8 array	256	Symmetric Key
Cipher Decapsulation Key	Uint8 array	Variable	Asymmetric Encryption
Cipher Encapsulation Key	Uint8 array	Variable	Asymmetric Encryption
Expiration	Uint64	64	Verification
Key ID	Uint8 array	128	Key Identification
Remote Channel Key	Uint8 array	256	Symmetric Key
Remote Verification Key	Uint8 array	Variable	Asymmetric Authentication
Session Cookie Token	Uint8 array	512	Verification
Pre-Shared Secret Local	Uint8 array	256	Symmetric Key
Pre-Shared Secret Remote	Uint8 array	256	Symmetric Key
Remote Verification Key	Uint8 array	Variable	Asymmetric Authentication
Signature Key	Uint8 array	Variable	Asymmetric Authentication
Verification Key	Uint8 array	Variable	Asymmetric Authentication

Table 5.6: The client KEX state structure.

- **Channel Key:** the local host's pre-shared secret.
- **Cipher Decapsulation Key:** the cipher decapsulation key used in the key exchange.
- **Cipher Encapsulation Key:** the cipher encapsulation key used in the key exchange.
- **Expiration:** the expiration time of the key in seconds from the epoch.

- **Key ID:** the key identity string, used to uniquely identify this key.
- **Remote Channel Key:** the remote host's pre-shared secret.
- **Remote Verification Key:** the signature verification key for the remote host.
- **Session Token:** the session cookie token, used for authentication and key derivation.
- **Shared Secret:** the shared secret derived from the local encapsulation.
- **Signature Key:** the asymmetric signature key used to sign an outgoing message.
- **Verification Key:** the local asymmetric signature verification key.

This state ensures that all required keys and tokens are securely managed throughout the key exchange process. This state is securely erased once the key exchange has completed.

5.7 Server Kex State

The server KEX state structure mirrors the client state, with additional functionality for handling server-specific key queries.

Data Name	Data Type	Bit Length	Function
Channel Key	Uint8 array	256	Symmetric Key
Cipher Decapsulation Key	Uint8 array	Variable	Asymmetric Encryption
Cipher Encapsulation Key	Uint8 array	Variable	Asymmetric Encryption
Expiration	Uint64	64	Verification
Key ID	Uint8 array	128	Key Identification
Key Query Callback	Uint64	64	Function Pointer
Remote Channel Key	Uint8 array	256	Symmetric Key
Remote Verification Key	Uint8 array	Variable	Asymmetric Authentication
Session Token	Uint8 array	256/512	Verification
Shared Secret	Uint8 array	256	Symmetric Key
Signature Key	Uint8 array	Variable	Asymmetric Authentication
Verification Key	Uint8 array	Variable	Asymmetric Authentication

Table 5.7: The server KEX state structure.

- **Channel Key:** the local host's pre-shared secret.
- **Cipher Decapsulation Key:** the cipher decapsulation key used in the key exchange.
- **Cipher Encapsulation Key:** the cipher encapsulation key used in the key exchange.
- **Expiration:** the expiration time of the key in seconds from the epoch.
- **Key ID:** the key identity string, used to uniquely identify this key.
- **Key Query Callback:** the callback function pointer for the ratchet mechanism.
- **Remote Channel Key:** the remote host's pre-shared secret.
- **Remote Verification Key:** the remote host's signature verification key.
- **Session Token:** the session cookie token, used for authentication and key derivation.
- **Shared Secret:** the shared secret derived from the local encapsulation.

- **Signature Key:** the asymmetric signature key used to sign an outgoing message.
- **Verification Key:** the local asymmetric signature verification key.

This state is securely erased once the key exchange has completed.

5.8 Packet Header

The DKTP 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 DKTP_MESSAGE_MAX in size.

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

Figure 5.8: The DKTP packet structure.

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

5.9 Flag Types

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

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 5.9: Packet header flag types.

5.10 Error Types

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

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 stage.

Decapsulation Failure	0x06	The asymmetric cipher failed to decapsulate the shared secret.
Establish Failure	0x07	The transmission failed at the KEX establish stage.
Exstart Failure	0x08	The transmission failed at the KEX exstart stage.
Exchange Failure	0x09	The transmission failed at the KEX exchange stage.
Hash Invalid	0x0A	The hash authentication 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 DKTP peering 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 5.10: Error type messages.

6. Operational Overview

During the device initialization stage, clients generate an asymmetric signature key-pair. This key-pair consists of a private key, which the client uses to sign messages in the key exchange, and a signature verification key, which is shared with other approved hosts and used to verify a message signature (though each peering key is bonded to a single remote host with unique pre-shared secrets, verification keys are host-based and common to peering keys from that host). The peering key contains the asymmetric signature verification key, a pre-shared secret channel-key, a key identity array, the protocol configuration string, and the key expiration date. The sign/verify signature keys generated by the clients, function as the primary authentication for the key exchange in a bi-directional authentication scheme.
The peering keys can be distributed between hosts through some other cryptographic means that possesses the equivalent post-quantum 256-bit security (e.g. QKD, QSTP, IPSec), or installed locally when the device is initialized.

DKTP is capable of full 512-bit authenticated and encrypted tunnels; in enhanced mode (by enabling the DKTP_ENHANCED_SECURITY macro), SHA3, SHAKE, KMAC, and RCS are all running in 512-bit security mode, and keys and tokens are upgraded to 512-bits in length. This mode is reserved for the highest security setting, and the greatest assurance of crypto agility when protecting the most high-value message streams.

Participating hosts are assigned roles during the connection stage as either a server (TCP listener), which accepts network connection requests, or a client, which initiates the connection request, but a device can be both, initiating or accepting a connection.

The client begins the connection process by sending a *connection request* packet, and if the server recognizes the key-id contained in the request message as valid, the key exchange sequence is initiated. The identity array is used to look up the corresponding local and remote peer keys and load them into the key exchange state. If the correct matching keys are not found, the exchange is aborted locally or denied by the remote host.

During this exchange, the asymmetric cipher keys and ciphertext are signed, verified, and mutually exchanged between the client and server. This process results in the generation of a pair of asymmetrically derived shared secrets, which are used along with the pre-shared secrets in the peering key, to key a cryptographic derivation function (KDF). The KDF derives the symmetric keys and nonces used to initialize symmetric cipher instances for both the transmit and receive channel interfaces.

Each transmit and receive channel is keyed independently, using different source asymmetrically-derived and pre-shared keys. These keys create cryptographically independent circuits; breaking one channel, does not break the other channel. Breaking either the asymmetric cryptography or the pre-shared key, does not break the protocol. Two disparate encryption

systems (asymmetric and pre-shared symmetric key), must be broken simultaneously, on two independently keyed circuits, to compromise the entire communications stream.

If an error occurs during the key exchange the affected sender or listener immediately sends an error message to the other host, disconnects, and terminates the session.
Error handling includes checks for message synchronization, timing, expected message size during the key exchange, authentication failures, packet expiration, and any internal errors triggered by cryptographic or network operations integral to the key exchange and communication flow.

6.1 Connection Request

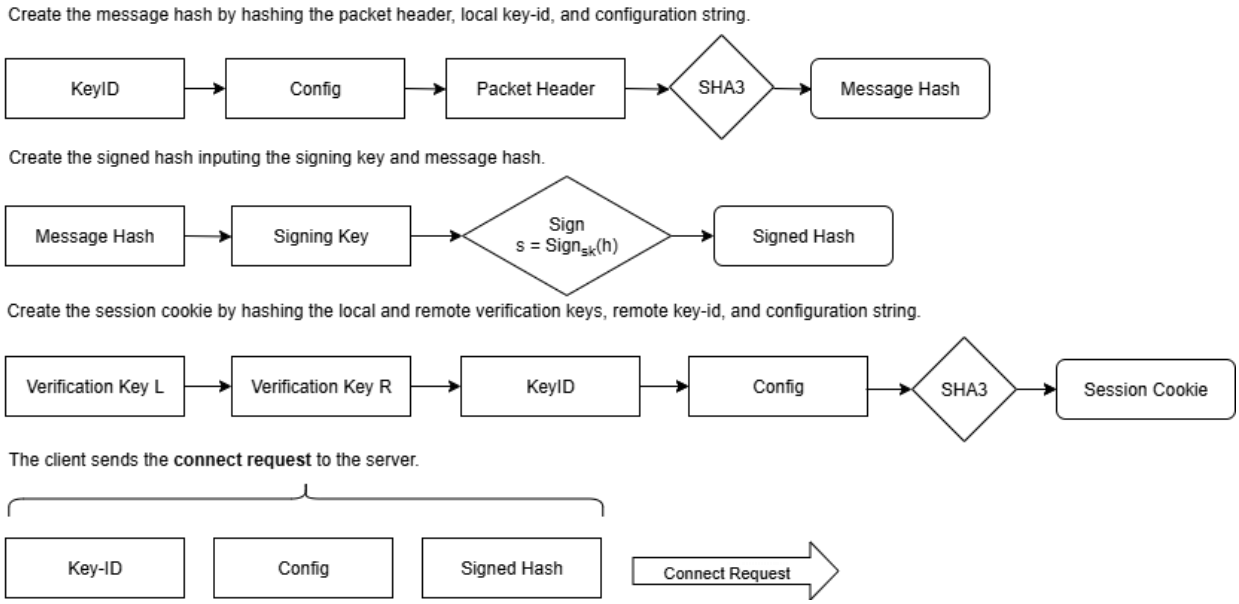

Figure 6.1: DKTP connection request.

1) The client initiates the key exchange operation by sending a connection request packet to the server. This packet includes the client's key identification array, the protocol configuration string, and a signed hash of the message including the packet header sequence number and timestamp values.
2) The packet header fields are verified by the server; message size, sequence number, flag, and the timestamp, all of which are added to the message hash. The message hash is signed, and this guarantees protection from replay attacks.
3) The client generates a hash of the protocol string, along with both the client's and server's asymmetric signature verification keys, and the remote key ID, and stores this information in the session cookie (*sch*) state value for later use in the key exchange. This ensures that the correct verification keys and cryptographic parameters are referenced throughout the key exchange process.

6.2 Connection Response

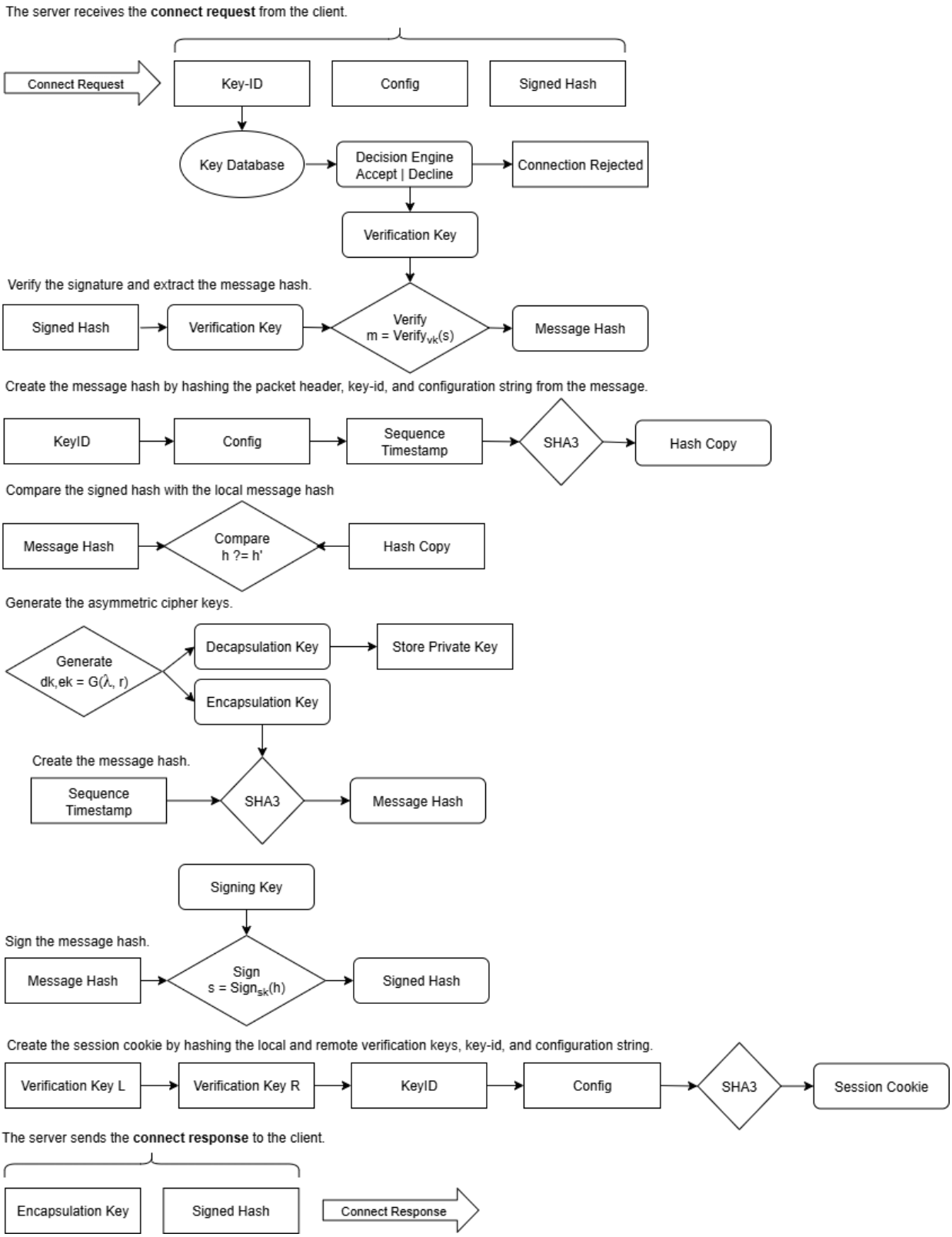

Figure 6.2: DKTP server connection response.

1) The server inspects the packet header for the correct flag, sequence number, expected message size, and that the packet valid-time has not expired.
2) The server checks its database for a key matching the key identification array sent by the client in the connect request message. The server looks through its list of peering keys, identifying the *remote key id*, and loads that peering key into state during the key exchange. If the key is not known to the server, the server sends a *key unrecognized* error message to the client.
3) The server compares the configuration string contained in the message against its own protocol string for a match. If the protocol configuration strings do not match, the server will send an *unknown protocol* error to the client and close the connection.
4) The server verifies the key's expiration time, if the client's key has expired, the server will abort the connection response and send a *key expired* error message.
5) The server checks the signature of the client's message hash using the client's signature verification key stored in the peering key. If the signature check fails, the server will send a *verify failure* error to the client and close the connection.
6) If the signature is authenticated, the server hashes the local key-id, the config string, and the *connect request* packet header sequence number and timestamp values, and compares this to the hash embedded in the signature for equivalence. If the hashes are equal, the server loads the client key into state. If the hash comparison fails the server sends a *hash invalid* error to the client and closes the connection.
7) The server generates an encapsulation/decapsulation asymmetric cipher key-pair (*dk, ek*).
8) The server hashes the encapsulation key and the *connect response* packet header sequence number and timestamp values, and signs the hash with its asymmetric signing key. The client has a copy of the asymmetric signature verification key, that will be used to verify this signature.
9) The server securely stores the asymmetric cipher decapsulation key temporarily in its state.
10) The server hashes the key-id array, the configuration string, and the local and remote copies of the signature verification keys, and stores the hash in its session cookie state value (*sch*), for future use as a session cookie.
11) The server adds the public asymmetric encapsulation key, and the public key's signed hash, to the *connect response* message, and sends it to the client.

6.3 Exchange Request

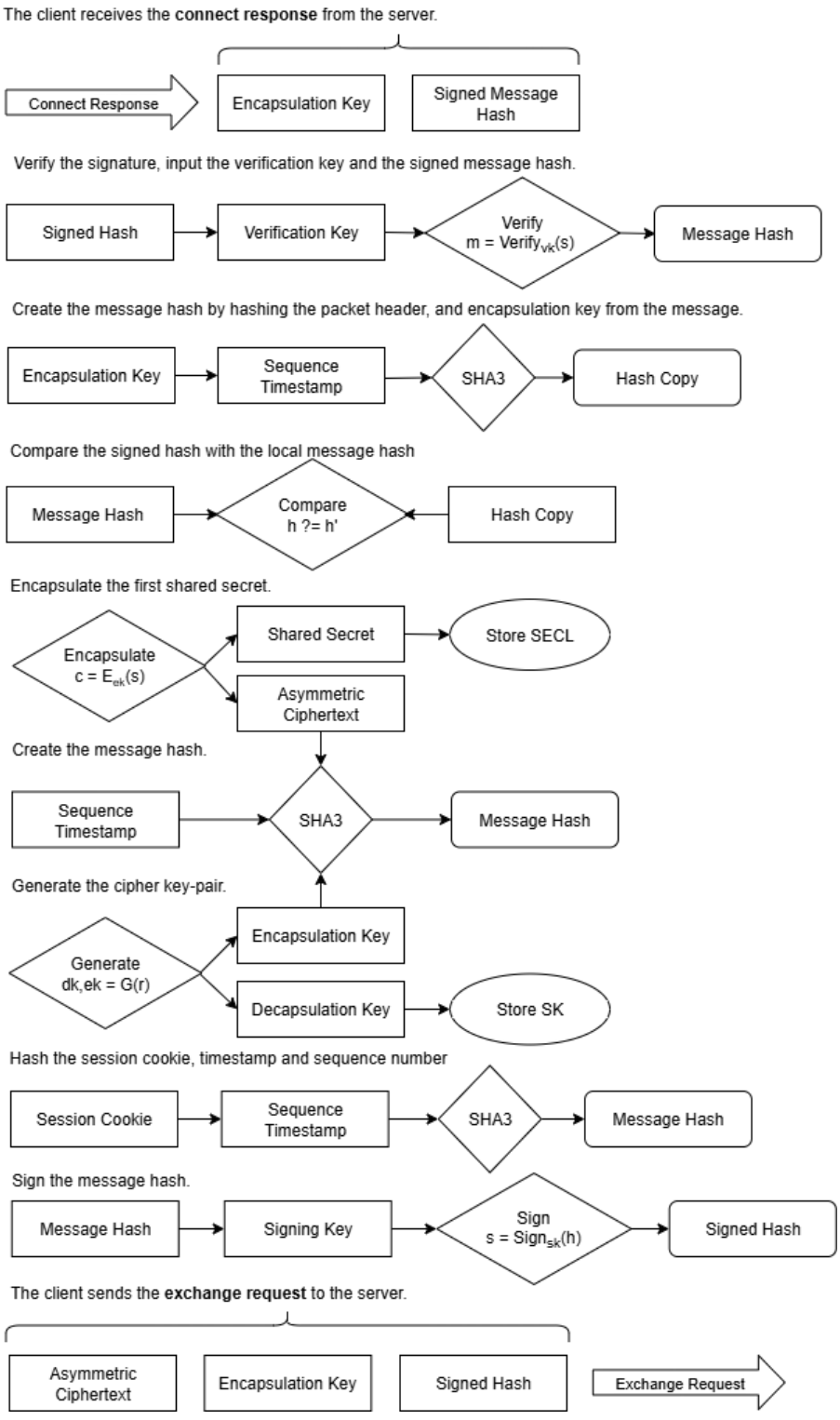

Figure 6.3: DKTP client exchange request.

1) The client inspects the *connect response* packet header for the correct flag, sequence number, expected message size, and that the UTC timestamp has not expired.

2) The client uses the server's signature verification key to validate the signature on the message hash.
3) If the signature verification is successful, the client hashes the asymmetric encapsulation key and *connection response* packet header sequence and timestamp values, and compares this hash to the signed hash received from the server. If the signature verification fails, the client sends an *authentication failure* message to the server and terminates the connection. Similarly, if the hash comparison fails, the client sends a *hash invalid* error message and closes the connection.
4) Once the signature and hash have been successfully authenticated, the client uses the asymmetric cipher key to encapsulate a shared secret, generating a ciphertext that will be sent to the server. This ciphertext will be used by the server to decapsulate the shared secret (*secl*), which the client securely stores for later use in deriving the tunnel channel key.
5) The client generates a new asymmetric cipher key-pair (*dk*, *ek*), and stores the decapsulation key.
6) The client hashes the encapsulation key, the ciphertext, and the *exchange request* packet header sequence number and timestamp, signing the hash using its asymmetric signing key.
7) The client adds the asymmetric ciphertext, the encapsulation key, and the signed hash to the exchange request packet, which is sent to the server to continue the key exchange process.

6.4 Exchange Response

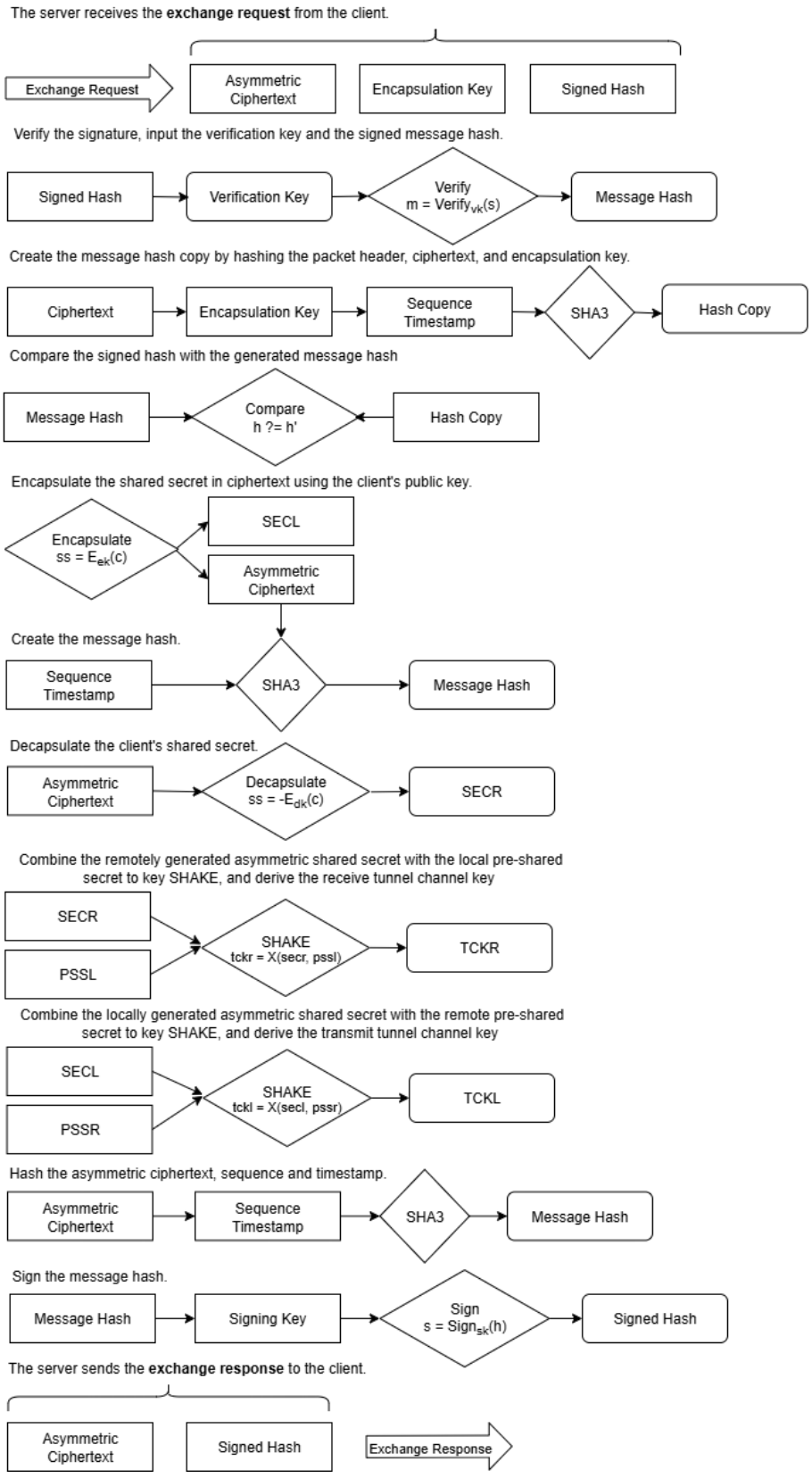

Figure 6.4: DKTP server exchange response.

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 verifies the signature of the hash of the message, using the client's signature verification key.
3) The server hashes the encapsulation key, cipher-text, and *exchange request* header sequence and timestamp values, and verifies the hash for equivalence to the one embedded in the signed hash.
4) The server uses the stored asymmetric cipher decapsulation key to decapsulate the first shared secret (*secr*).
5) The server uses the public key sent by the client to generate a new shared secret and encapsulate it in ciphertext (*secl*).
6) The remote asymmetrically derived shared secret *secr* and the local pre-shared secret *pssl* are used to key the KDF, which derives the receive channel tunnel key (*tckr*).
7) The local asymmetrically derived shared secret *secl* and the remote pre-shared secret *pssr* are used to key the KDF, which derives the transmit channel tunnel key (*tckl*).
8) The cipher-text and exchange response header sequence number and timestamp values are hashed, the hash is signed by the server's private asymmetric signature key, and these are sent back to the client in an *exchange response* packet.
9) The symmetric cipher instances are keyed with the tunnel channel keys, raising both the transmit and receive channels of the encrypted tunnel.

6.5 Establish Request

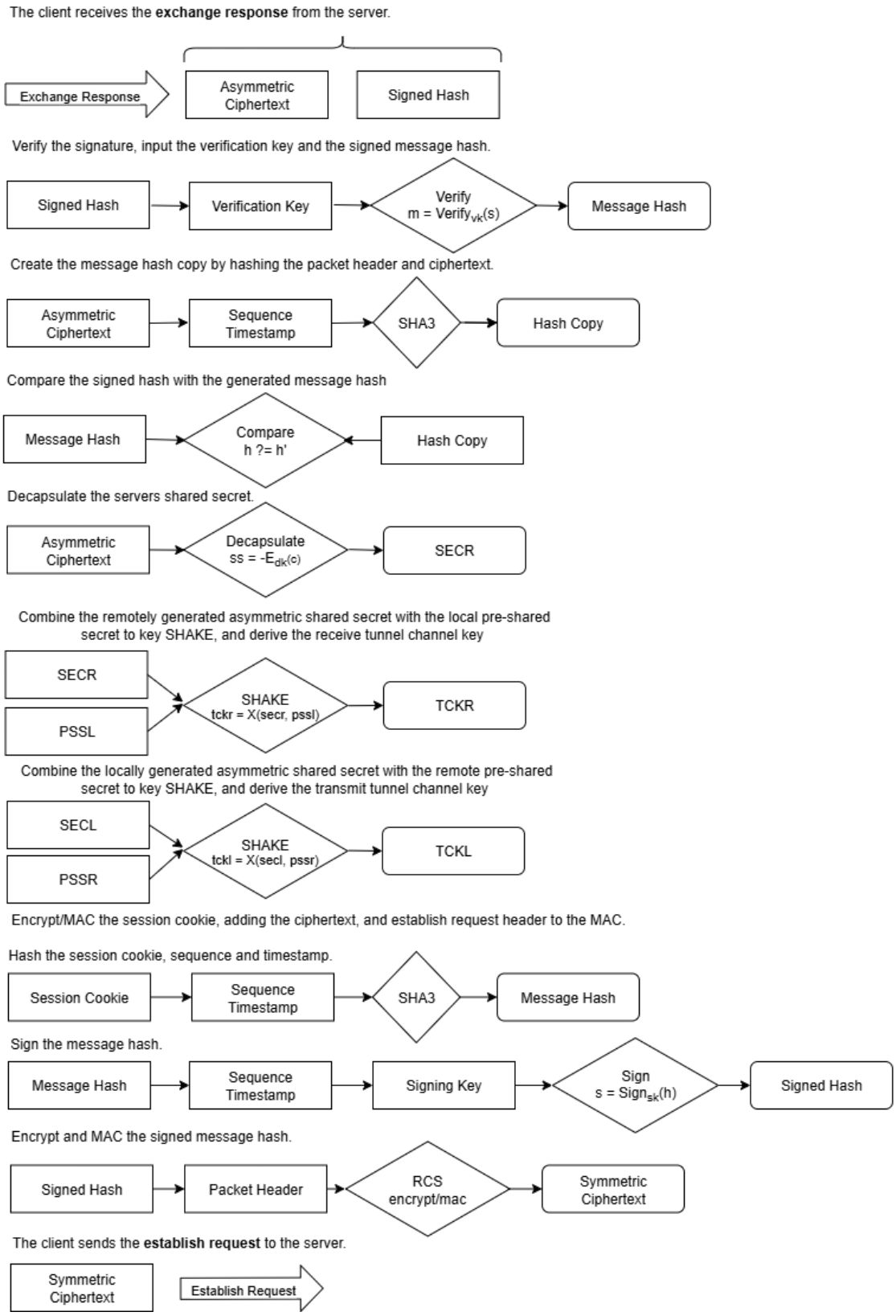

Figure 6.5: DKTP client establish request.

1) The client inspects the packet header for the correct flag, sequence number, expected message size, and that the valid-time timestamp has not expired.
2) The client extracts the asymmetric ciphertext and the signed hash of the ciphertext. It uses the server's signature verification key to verify the signature on the hash, ensuring its authenticity.
3) The client hashes the ciphertext and the serialized *exchange response* header sequence number and timestamp values, and compares the generated hash with the hash embedded in the signature. If the hashes match, the client confirms the integrity of the data.
4) The client decapsulates the shared secret from the ciphertext (*secr*).
5) The remote asymmetrically derived shared secret *secr* and the local pre-shared secret *pssl* are used to key the KDF, which derives the receive channel tunnel key (*tckr*).
6) The local asymmetrically derived shared secret *secl* and the remote pre-shared secret *pssr* are used to key the KDF, which derives the transmit channel tunnel key (*tckl*).
7) The symmetric cipher instances are keyed with the tunnel channel keys, raising both the transmit and receive channels of the encrypted tunnel.
8) The client hashes the session cookie and the *establish request* packet timestamp and sequence number, and encrypts the hash with the *transmit* instance of the symmetric cipher, and adds the serialized *establish request* header to the additional data of the AEAD stream cipher (RCS).

6.6 Establish Response

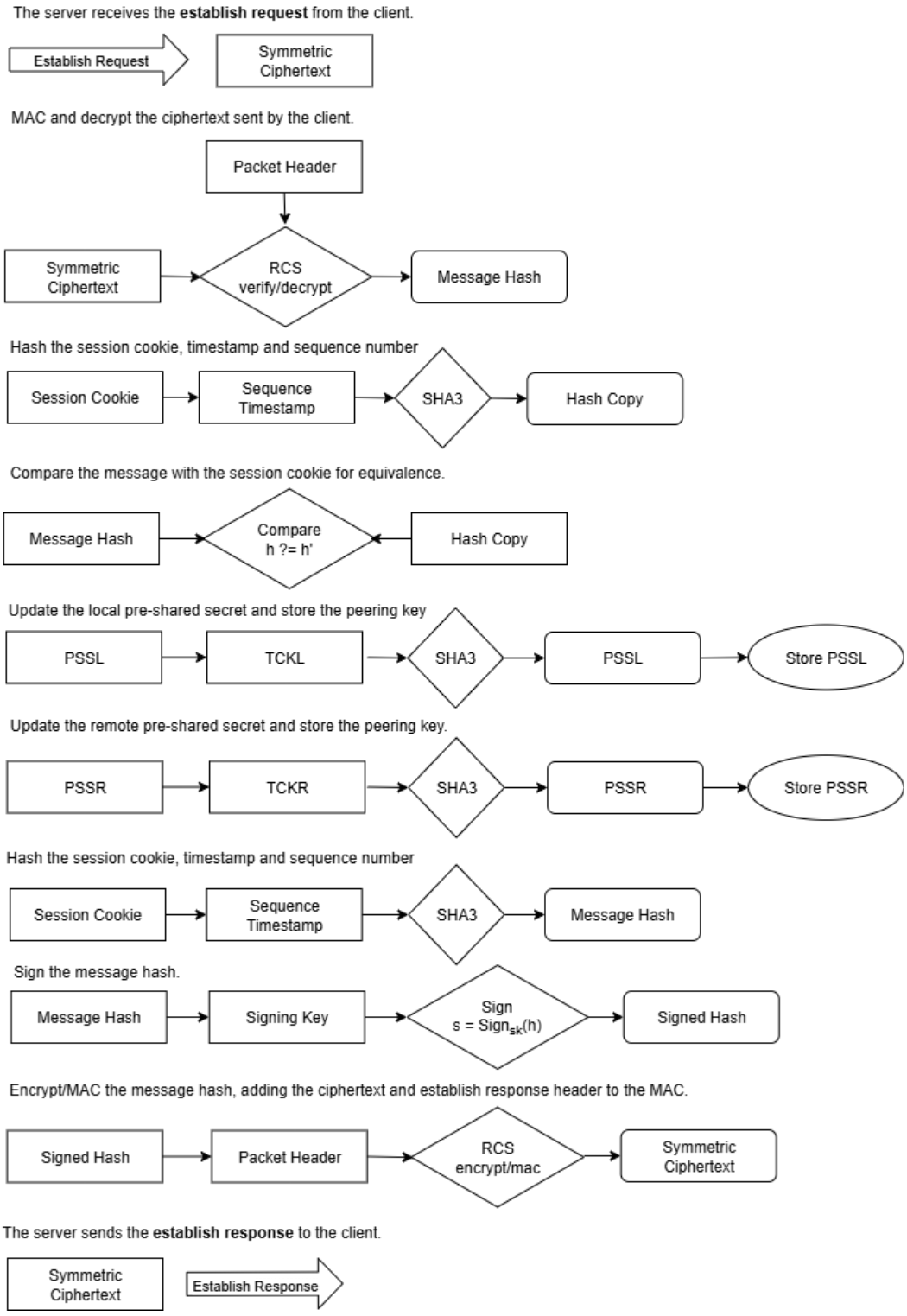

Figure 6.6: DKTP server establish response.

1) The server inspects the packet header for the correct flag, sequence number, expected message size, and that the valid-time timestamp has not expired.

2) The server decrypts the ciphertext using the *receive* cipher instance, adding the serialized *establish request* packet header as additional data.
3) The message is compared to the hash of the session cookie and packet sequence number and valid-time timestamp for equivalence. If the decryption succeeds and the message equals the session cookie hash, the tunnel has been verified.
4) The server hashes the local pre-shared secret (*pssl*) with the local tunnel key (*tckl*), to update the local pre-shared secret. The server saves the local peering key to permanent storage.
5) The server hashes the remote pre-shared secret (*pssr*) with the remote tunnel key (*tckr*), to update the remote pre-shared secret. The server saves the remote peering key to permanent storage.
6) The server hashes the session cookie and the *establish response* packet timestamp and sequence number, and encrypts the hash with the *transmit* instance of the symmetric cipher, and adds the serialized packet header to the additional data of the AEAD stream cipher (RCS).
7) The tunnel interface is changed to the active state on the server, peering keys and kex state are unloaded and cleared, and the message is sent to the client.

6.7 Establish Verify

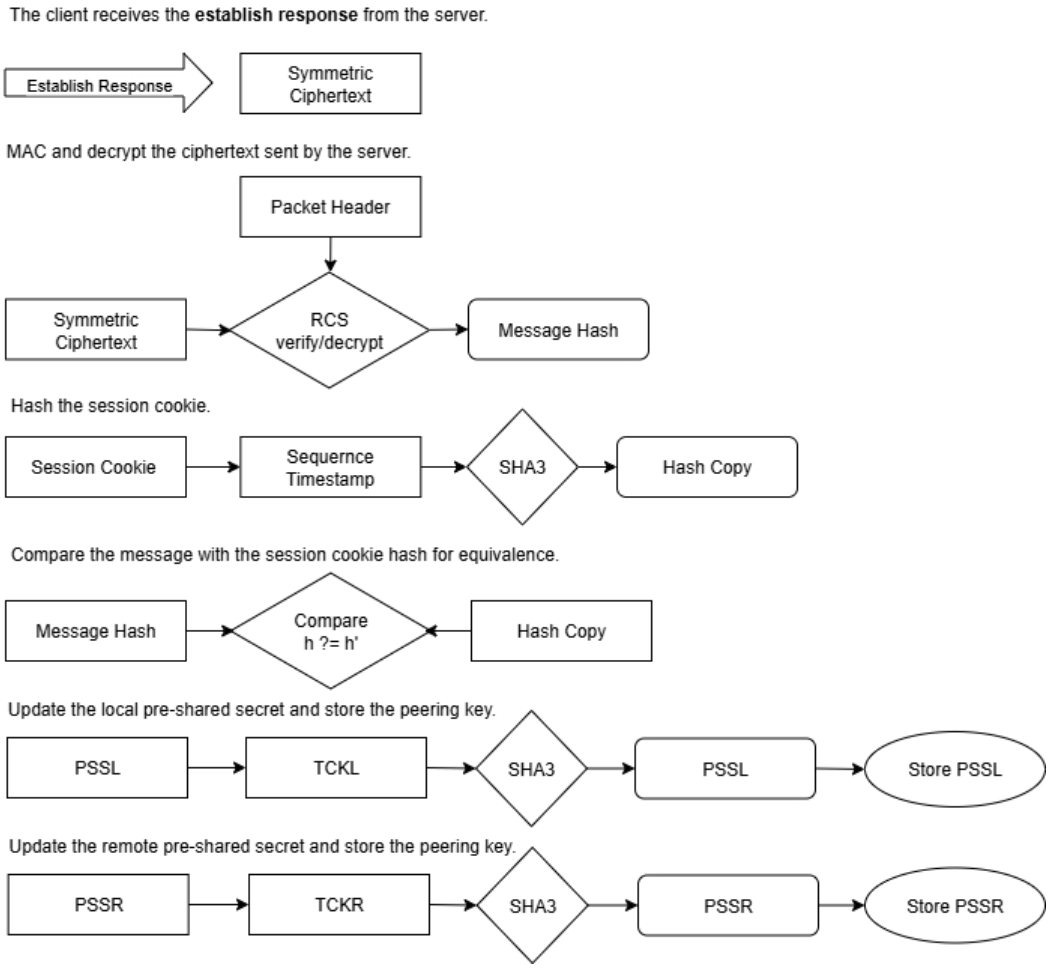

Figure 6.7: DKTP client establish verify.

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 decrypts the ciphertext using the *receive* instance of the cipher, adding the serialized *establish request* packet header to the cipher MAC.
3) The client hashes the session cookie and packet valid-time timestamp and sequence number, and compares it to the decrypted message for equivalence. If the hashes match, the session has been verified.
4) The client hashes the local pre-shared secret (*pssl*) with the local tunnel key (*tckl*), to update the local pre-shared secret. The client saves the local peering key to permanent storage.
5) The client hashes the remote pre-shared secret (*pssr*) with the remote tunnel key (*tckr*), to update the remote pre-shared secret. The client saves the remote peering key to permanent storage.
6) The key exchange has completed, peering keys and kex state are unloaded and cleared, the tunnel state is changed to active, and the encrypted tunnel interfaces are now ready to process data.

6.8 Asymmetric Ratchet

The [optional] asymmetric ratchet mechanism in DKTP injects new entropy into the tunnel construction by periodically re-keying the symmetric ciphers that provide the tunnel's encryption and decryption functionality. This ratchet function can be called by either host; client or server, at any time after the key exchange is completed and the tunnel is active. This process involves combining a hash of a pre-shared secret (*pssl* and *pssr*) with new keying material (*rtokr* and *rtokl*), obtained through a bidirectional asymmetric key exchange. The derived keys are used to re-key the transmit and receive symmetric cipher instances on both the client and the server. Pre-shared keys are updated after the exchange is confirmed, by hashing them with the corresponding ratchet token.

6.8.1 Asymmetric Ratchet Connection Request

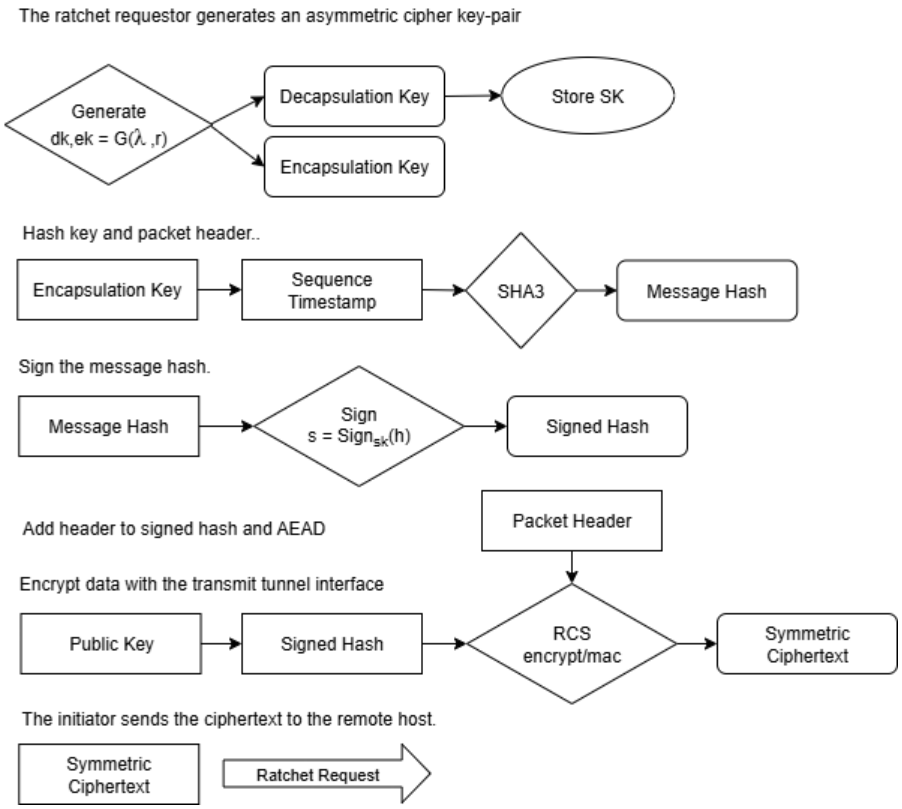

Figure 6.8a Asymmetric ratchet request.

1. Upon invocation of the asymmetric ratchet function, the client (the initiator in the exchange) generates a new asymmetric cipher key-pair.
2. The encapsulation key, the packet header sequence number and timestamp values are hashed.
3. The message hash is signed using the client's asymmetric signing key to ensure its authenticity.
4. The client MACs and encrypts the encapsulation key and signed hash using the transmit channel tunnel interface (RCS), and transmits the ciphertext to the remote host (the server) over the tunnel.

6.8.2 Asymmetric Ratchet Connection Response

Figure 6.8b Asymmetric ratchet connection response.

1. The server verifies and decrypts the ciphertext using the receive channel cipher RCS.
2. The server verifies the message signature using the client's signature verification key.
3. If the signature is valid, the host hashes the encapsulation key and the *ratchet connect request* packet header sequence number and timestamp values, and compares it to the hash embedded in the message signature.

4. If the hashes are equivalent, the server uses the encapsulation key to generate a new shared secret; the *ratchet token*, and corresponding ciphertext. The ratchet token (*rtok*) and the *ratchet key*, are used to key the KDF and derive a new *transmit* channel tunnel key (*tck*), which is used to re-key the symmetric cipher corresponding to the transmit channel of the tunnel.
5. The server hashes the the ciphertext, and the *ratchet exchange response* packet header sequence number and timestamp values.
6. The message hash is signed using the server's asymmetric signing key.
7. The server MACs and encrypts the ciphertext, and the signed hash using the transmit channel symmetric cipher instance. The server sends the ciphertext to the client.

6.8.3 Asymmetric Ratchet Establish Verify

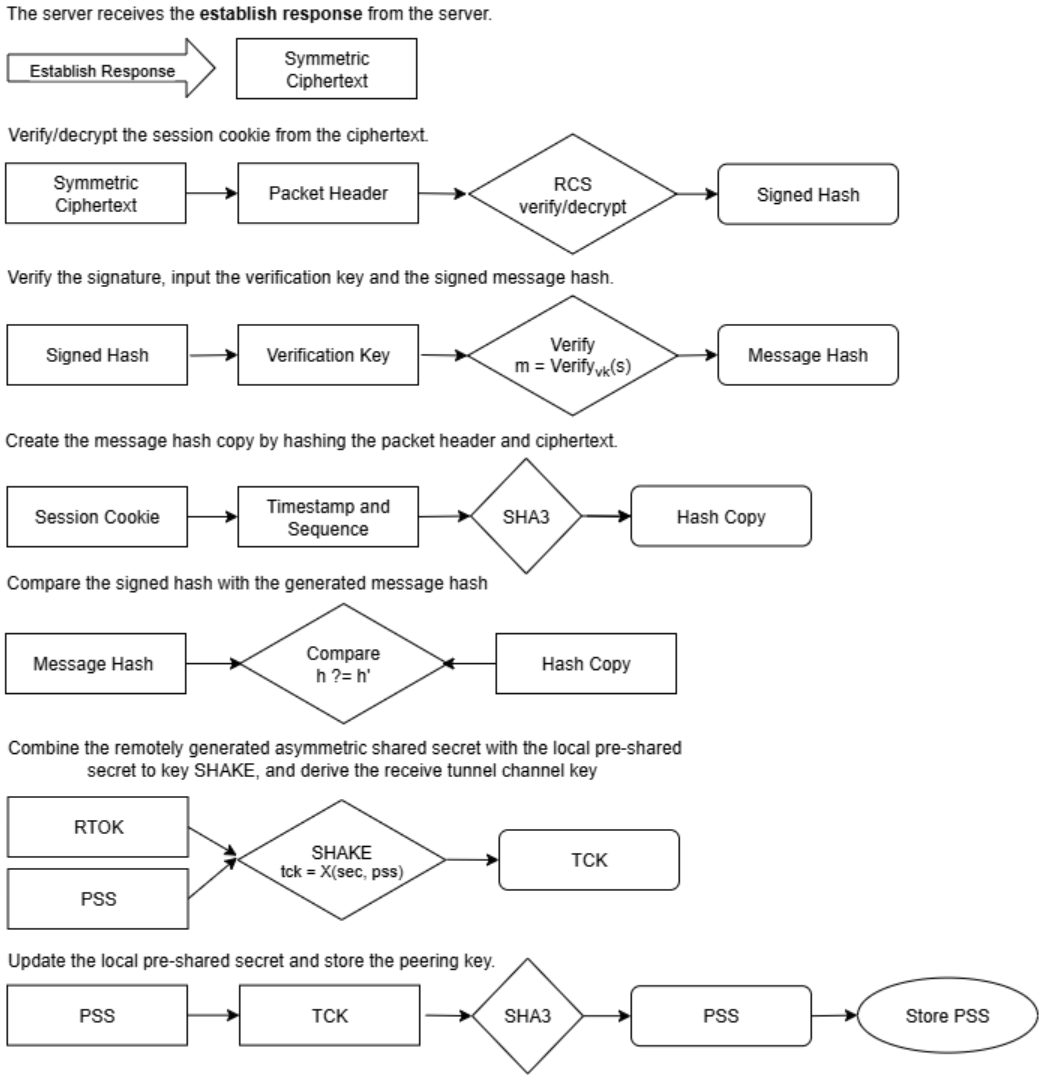

Figure 6.8c Asymmetric ratchet establish verify

1. The client decrypts the *connect response* message using the receive channel symmetric cipher instance.
2. The signed hash signature is verified using the server's asymmetric signature verification key.
3. If the signature verification succeeds, the client hashes the session cookie with the *ratchet connection response* packet timestamp and sequence number. This hash is compared to the hash embedded in the signature for equality.
4. The client keys a KDF with the remote token (*rtok*) and local pre-shared secret (*pss*) and generates the remote tunnel key (*tck*) and nonce. The receive channel symmetric cipher is re-keyed.

5. The client keys a KDF with the local token (*rtok*), and remote pre-shared secret (*pss*) and generates the local tunnel key (*tck*) and nonce. The transmit channel symmetric cipher is re-keyed.
6. The client updates the pre-shared keys by hashing the remote token(*rtok*) and remote pre-shared secret (*pss*), and hashing the local token (*rtok*) and pre-shared secret (*pss*).

7.0 Formal Description

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
$-E_{dk}$	-The asymmetric decapsulation function and secret key
E_{ek}	-The asymmetric encapsulation function and key
E_k, $-E_k$	-The symmetric encryption and decryption functions and key
G(λ, r)	-The asymmetric cipher key generation with parameter set and random source
H	-The hash function (SHA3)
KDF	-The key expansion function (SHAKE)
S_{sk}	-Sign data with the secret signature key
V_{vk}	-Verify a signature using the signature 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
kid	-The peering keys unique identity array
dk, *ek*	-Asymmetric cipher decapsulation and encapsulation keys
pssl, *pssr*	-The local and remote pre-shared symmetric keys
secl, *secr*	-The shared secret derived from asymmetric encapsulation and decapsulation
shpk	-The signed hash of the asymmetric cipher encapsulation-key
sk, vk	-The asymmetric signature signing and verification keys
st	-The serialized packet sequence number and timestamp.
tckl, *tckr*	-The tunnel channel keys for the transmit/receive symmetric cipher instances

Key Exchange Sequence

Preamble:

The key exchange is designed to facilitate secure communication in a peer-to-peer architecture. Hosts exchange peering keys with each other, which uniquely bind them together as authorized communicators. Every host peering requires a unique set of peering keys; these bonded keys are used only for communications between the two peer hosts. Peering keys are exchanged through a secondary process; fetched from a directory server over an encrypted channel, installed during host initialization, or distributed using an encrypted tunnel of equal cryptographic security (256-bit post quantum, e.g. QKD, QSTP, QSMP, IPSec).

In the architecture, since one node must initiate the connection while the other must accept it, the initiator is designated as the *client*, and the recipient of the request is referred to as the *server* within the key exchange context.

7.1 Connect Request

The client initiates the connection by sending a *connection request* to the server, which includes its configuration string and key identity string.

The key identity (kid) is a multi-part, 16-byte array that serves as both a device and key identification array. This identifier is used to link the intended target with its corresponding cryptographic key, ensuring that the correct signature verification key is used during the secure exchange.

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

To maintain the integrity of the key exchange, the client generates a session cookie (*sch*) by hashing a combination of the configuration string, the key identity, and the asymmetric signature verification keys from both the client and the server:

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

Where:

- *cfg* is the configuration string.
- *kid* is the local peering key identity string.
- *pvka* is the client's signature verification key.
- *pvkb* is the server's signature verification key.

This session cookie hash (*sch*) serves as a identifier for the session pairing, ensuring secure reference to the cryptographic parameters during the establish stage of the key exchange.

The client serializes the *connection request* packet header sequence number and timestamp values (*st*), are added to a hash along with the key id and configuration string. The client signs the hash with its asymmetric signing key, and adds this to the packet message along with the *kid* and *cfg* arrays.

$shm \leftarrow S_{sk}(H(kid \parallel cfg \parallel st))$

The client then transmits the *connection request* to the server to begin the key exchange operation:

$C\{ kid \parallel cfg \parallel shm \} \rightarrow S$

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 message packet and sends it to the remote host, triggering a teardown of both the key exchange and the network connection on both sides.

Key Verification and Configuration Check

The server checks the *connect request* packet header for validity, including the sequence number, message size, protocol flag, and valid-time timestamp. This check is done at each step of the exchange, verifying inbound packets for correctness of the expected flag, message size, creation time, and sequence number. The UTC timestamp is tested for a valid-time threshold; if the local time is different from the packet creation time by more than the threshold (default is 60 seconds) the packet is rejected, and the exchange is torn down. This mechanism protects the exchange from replay attacks and packet header tampering. Serialized packet headers are either added to the hash of a message and signed, or added to the *additional data* of the authenticated stream cipher (RCS) to guarantee authenticity.

The server verifies that it has the requested peering key that matches the client's host by searching its cached peering keys *remote kid* fields against the client *kid* contained in the message. It then checks that its protocol configuration is equivalent with that of the client.

The server verifies the message signature, then hashes the message, which is compared to the hash signed by the client for equivalence.

Where:

- *shm* is the signed message hash received from the client.
- *hm* is the hashed message signed by the client.
- *hm'* is the message hashed by the server.
- *st* are the sequence number and timestamp values.
- *m* is the packet message : *kid* || *cfg* || *sh*

$V_{vk}(shm) \leftarrow (\text{true} ?= hm : 0)$

$hm' \leftarrow H(kid \,||\, cfg \,||\, st)$

$hm' ?= hm : m : 0$

The server creates a session cookie by hashing the configuration string, the key identity, and both the asymmetric signature verification keys:

Where:

- *cfg* is the configuration string.
- *kid* is the remote peering key identity string.
- *pvka* is the client's signature verification key.
- *pvkb* is the server's signature verification key.

$sch \leftarrow H(cfg \,||\, kid \,||\, pvkb \,||\, pvka)$

This hash acts as a unique session identifier for the established stage of the key exchange.

Asymmetric Key Generation and Signing

The server generates a new asymmetric cipher key-pair and securely stores the private key. It then hashes the encapsulation key and the serialized outbound packet header, and signs this hash using its private asymmetric signature key.

Key generation and signing steps are as follows:

Generate the encapsulation (*ek*) and decapsulation (*dk*) asymmetric encryption keys. Store the decapsulation key.

$dk, ek \leftarrow G(\lambda, r)$

Create a hash of the encapsulation key and serialized *connection response* packet header sequence number and timestamp values (*st*).

$hek \leftarrow H(ek \,||\, st)$

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

$shek \leftarrow S_{sk}(hek)$

The server then sends the *connection response* message to the client, which contains the signed hash of the asymmetric encapsulation key (*shek*) and a copy of the encapsulation key:

$S\{ shek \,||\, ek \} \rightarrow C$

7.3 Exchange Request

The client processes the *connect response* message from the server and proceeds with the next steps in the key exchange. This stage involves verifying the server's encapsulation key, encapsulating a shared secret, and authenticating the message.

Signature Verification and Hash Check

The client checks the *connect response* packet header for validity; the flag, expected message size, the valid-time timestamp, and the sequence number.

The client verifies the server's signature on the hashed encapsulation key and serialized packet header. It then generates its own hash of the received encapsulation key and serialized header and compares it to the one included in the server's message. If the hashes match, the client proceeds with the key exchange. If the hashes do not match, the key exchange is aborted, and the session is terminated.

The client verifies the hash of the encapsulation key using the server's signature verification key. If the hash is valid, the process continues; otherwise, the exchange fails.

$V_{vk}(H(ek \parallel st))$?= (true ?= ek : 0)

Once the encapsulation key is verified, the client uses it to encapsulate a shared secret. The client generates a ciphertext (*cpta*) and encapsulates the shared secret (*secl*) using the encapsulation key.

$cpta = E_{ek}(secl)$

The client stores the shared secret (*secl*), which will be combined with the remote pre-shared secret (*pssr*) to create the (local) transmit tunnel channel key (*tckl*).

Asymmetric Key Generation and Signing

The client generates its own asymmetric encryption key-pair and securely stores the private key. It then creates a hash of its encapsulation key, the ciphertext and the serialized outbound packet header, and signs this hash using its asymmetric signing key.

Key generation and signing steps:

Generate the client's encapsulation (*ek*) and decapsulation (*dk*) asymmetric encryption keys.

$dk, ek \leftarrow G(\lambda, r)$

Hash the client's encapsulation key, the ciphertext, and the serialized exchange request packet sequence number and timestamp values (*st*).

$hkc \leftarrow H(ek \parallel cpta \parallel st)$

Sign the hashed value using the client's asymmetric signing key.

$shkc \leftarrow S_{sk}(hkc)$

The client sends an exchange request message back to the server. This message contains the signed hash of its asymmetric encapsulation key and ciphertext, the ciphertext itself, and a copy of the encapsulation key:

$C\{ cpta \parallel ek \parallel shkc \} \rightarrow S$

7.4 Exchange Response

The server processes the *exchange request* from the client, verifying the integrity of the message. The server decapsulates the ciphertext deriving the remote shared secret (*secr*). The remote shared secret and the local pre-shared secret (*pssl*) are hashed to create the remote receive tunnel key (*tckr*).

Signature Verification and Hash Check

The server checks the *exchange request* packet header, the flag, expected message size, the valid-time timestamp and sequence number.

The server verifies the signature of the hash included in the client's message. It then generates its own hash of the client's encapsulation key and the ciphertext, comparing this hash with the one embedded in the message signature. If the hashes match, the server continues with the key exchange; otherwise, the process is terminated, and the key exchange is aborted.

The server uses the client's signature verification key (*vk*) to verify the hash of the encapsulation key, ciphertext and exchange request packet header sequence number and timestamp values (*st*). If the verification is successful, the process continues; otherwise, the server halts the exchange.

$V_{vk}(H(ek \parallel cpta \parallel st)) \leftarrow (\text{true ?= ek} \parallel \text{cpta} : 0)$

Shared Secret Decapsulation

The server decapsulates the shared secret received from the client (*secr*). The server uses its asymmetric cipher key to decapsulate the remote shared secret (*secr*) from the ciphertext (*cpta*) provided by the client.

$secr \leftarrow -E_{dk}(cpta)$:

This shared secret is securely stored for use in generating the session keys.

Generation of Second Shared Secret

The server generates a new ciphertext and a second shared secret (*secl*) using the client's encapsulation key. The server generates the second ciphertext (*cptb*) and shared secret using the client's encapsulation key.

$cptb \leftarrow E_{ek}(secl)$

Session Key Derivation

The server combines the two shared secrets; the local shared secret and the remote pre-shared secret (*secl* and *pssr*) to derive a symmetric tunnel channel key (*tckl*) and unique nonce (*nl*) for the transmit channel.

$tckl, nl \leftarrow \text{KDF}(secl, pssr)$

The server combines the two shared secrets; the remote shared secret and the local pre-shared secret (*secr* and *pssl*) to derive a symmetric tunnel channel key (*tckr*) and corresponding nonce (*nr*) for the receive channel.

$tckr, nr \leftarrow \text{KDF}(secr, pssl)$

Cipher Initialization

The symmetric cipher instances for the receive and transmit channels are then initialized with the derived tunnel keys and nonces. Each channel; transmit and receive is keyed with an independently derived set of keys, making both transmit and receive channels independently keyed circuits.

Initializes the receive channel cipher with tunnel channel key *tckr* and the corresponding nonce *nr*.

$cpr_{rx}(tckr, nr)$

Initializes the transmit channel cipher with tunnel channel key *tckl* and the corresponding nonce *nl*.

$cpr_{tx}(tckl, nl)$

Hash and Signature of Ciphertext

To complete the exchange response, the server hashes the newly generated ciphertext and packet header sequence number and timestamp values (*st*), and signs the hash to ensure its integrity and authenticity before sending it back to the client.

$hcpt \leftarrow \text{H}(cptb \parallel st)$

$shcp \leftarrow S_{sk}(hcpt)$

The server sends the cipher-text, and the signed hash of the ciphertext to the client.

S{ *cptb*, *shcp* } → C

7.5 Establish Request

In the final stage of the key exchange process, the client completes the establishment of the encrypted communication channel by validating the message received from the server, decapsulating the shared secret, and generating the symmetric tunnel channel keys.

Signature Verification and Hash Check

The client validates the *exchange response* packet header, the flag, expected message size, the valid-time timestamp and sequence number.

The client verifies the server's signature of the hash of the ciphertext and exchange response packet header sequence number and timestamp values (*st*). It generates its own hash of the ciphertext and compares it with the one provided by the server. If the hashes match, the client proceeds to decapsulate the shared secret; otherwise, the key exchange is aborted.

The client verifies the hash of the server's ciphertext (*cptb*) using the server's signature verification key. If the verification is successful, the client continues; otherwise, it terminates the exchange.

$V_{vk}(H(cptb \parallel st)) \leftarrow (\text{true} \mathrel{?=} cptb : 0)$:

Shared Secret Decapsulation

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

The client uses its asymmetric decapsulation key to decapsulate the second shared secret (*secr*) from the server's ciphertext (*cptb*).

$secr \leftarrow -E_{ek}(cptb)$

Session Key Derivation

The client combines the two shared secrets; the remote shared secret and the local pre-shared secret (*secr* and *pssl*) to derive the remote symmetric tunnel channel key (*tckr*) and corresponding nonce (*nr*) for the receive channel.

tckr, nr = KDF(*secr, pssl*)

The client combines the two shared secrets; the local shared secret and the remote pre-shared secret (*secl* and *pssr*) to derive a symmetric tunnel channel key (*tckl*) and the corresponding nonce (*nl*) for the transmit channel.

tckl, nl = KDF(*secl, pssr*)

Cipher Initialization

The client initializes the symmetric ciphers for both communication channels.

Initializes the receive channel cipher with tunnel channel key *tckr* and nonce *nr*.

cpr_{rx}(*tckr*, *nr*)

Initializes the transmit tunnel channel cipher with key *tckl* and nonce *nl*.

cpr_{tx}(*tckl*, *nl*)

Establish Request Message

Once the symmetric channels are successfully initialized, the client sends a hash of the session cookie (*sch*) and the packet sequence number and timestamp (*st*) through the encrypted tunnel to the server, signaling that both encrypted channels of the tunnel are now active and that the tunnel is in its operational state.

hsch ← H(*sch* || *st*)

The establish request packet header is serialized and added to the *additional data* of the transmit instance of the authenticated cipher (RCS). The session cookie is encrypted and sent to the server.

cpt ← E_k(*hsch*, *sh*)

In the event of an error during this process, the client sends an error message to the server, which causes the key exchange to abort and the connection to be terminated on both ends.

The client sends the establish request to the server, indicating the successful establishment of the encrypted tunnel.

C{ *cpt* } → S

7.6 Establish Response

This step verifies the tunnel has been established by both hosts, and updates the pre-shared keys and the peering key structure containers.

Server Response Verification

The server checks the *establish request* packet header, the flag, expected message size, the valid-time timestamp and sequence number.

The server adds the serialized *establish request* packet header to the *additional data* of the receive instance of the authenticated stream cipher, and decrypts the session cookie.

$hsch \leftarrow -E_k(cpt, sh)$

The decrypted session cookie is compared to a hash of the local session cookie and establish request packet header's session number and timestamp fields (*st*) for equivalence.

$hsch' \leftarrow H(sch \parallel st)$

Compare the two hashes for equivalence. If the hash check fails, the server returns a *hash invalid* error message to the client and tears down the connection.

$hsch' \text{ ?= } hsch$ (true : 0):

The server updates its pre-shared keys, both for the local and remote host peer key structures. The server updates the exiting pre-shared local key by hashing in the local tunnel channel key, and updates the remote key by hashing in the remote tunnel key. The keys are saved to permanent storage, preserving the peering key updates.

$pssl \leftarrow H(pssl \parallel tckl)$

$pssr \leftarrow H(pssr \parallel tckr)$

The server re-hashes the session cookie, adding the sequence number and valid-time timestamp from the *establish response* packet to the hash.

$hsch \leftarrow H(sch \parallel st)$

The server adds the serialized *establish response* packet header to the additional data of the transmit cipher instance, and encrypts the hashed session cookie.

$cpt \leftarrow E_k(hsch, sh)$

Once the server sends the establish response, it sets its internal state to "session established," signaling that the encrypted tunnel is fully operational and ready to process data transmissions.

$S\{ cpt \} \rightarrow C$

7.7 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 response.

Client Verification

The client checks the *establish response* packet header for correct values; the flag, expected message size, the valid-time timestamp and sequence number.

If the flag does not indicate an establish response, the client identifies that the tunnel is in an error state as specified by the message. In the cases of an error, the client initiates a teardown of the tunnel on both sides to ensure that no data is transmitted over an insecure connection.

Operational State

The client adds the serialized establish response packet header to the additional data of the receive instance of the authenticated stream cipher. The client decrypts the session cookie, hashes its own session cookie along with the sequence number and valid-time timestamp fields from the establish response packet header, and compares the two hashes for equivalence.

$hsch \leftarrow -E_k(cpt, sh)$

The client hashes the session cookie with the establish response packets sequence number and timestamp, and compares the result with the hash sent by the server. If the hash check fails, the server returns a *hash invalid* error message to the client and tears down the connection.

$hsch' \leftarrow H(sch \parallel st)$

$hsch' \text{ ?= } hsch$ (true : 0):

If the two hashes are equal the encrypted tunnel is in the up state, and ready to transmit and receive data, and the session has been verified established.

The client updates its pre-shared keys, both for the local and remote host peering key structures. It updates the exiting pre-shared local key by hashing in the local tunnel channel key, and updates the remote key by hashing in the remote tunnel key. The keys are saved to permanent storage, preserving the peering key updates.

$pssl \leftarrow H(pssl \parallel tckl)$

$pssr \leftarrow H(pssr \parallel tckr)$

7.8 Transmission

This section describes operations that happen *internally* in the RCS symmetric AEAD stream cipher; and are used to clarify cryptographic operations executed during encryption and decryption of data using the cipher.

During message transmission, either the client or server initiates the process of securely sending data over the encrypted tunnel. This involves encrypting the message, updating the message authentication code (MAC), and preparing the packet for secure delivery.

Message Serialization and Encryption

The transmitting host, whether it is the client or server, first serializes the packet header, which includes 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 to ensure that it is securely integrated into the encryption process.

The host proceeds to encrypt the message using the RCS (Rijndael Cryptographic Stream) stream cipher's Authenticated Encryption with Associated Data (AEAD) functions. The encryption process generates a ciphertext, which is then passed through the MAC function to produce a verification code.

The plaintext message (*m*) is encrypted using the symmetric encryption function (E_k) to generate the ciphertext (*cpt*).

$cpt \leftarrow \mathrm{E_k}(m)$

The MAC code (*mc*) is calculated by updating the MAC function with the serialized packet header (*sh*) and the ciphertext (*cpt*).

$mc \leftarrow \mathrm{M_{mk}}(sh, cpt)$

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 host deserializes the packet header and adds it to the MAC state, along with the received ciphertext. The host then finalizes the MAC computation and compares the output code with the MAC code appended to the ciphertext. If the codes match, the ciphertext is authenticated and can be safely decrypted.

If the MAC verification succeeds, the ciphertext (*cpt*) is decrypted back into the plaintext message (*m*).

$m \leftarrow -\mathrm{E_k}(cpt)$

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.

If the MAC check fails, the decryption function returns an empty message array and an error signal, indicating that the message was either corrupted or tampered with.

This process guarantees the integrity and confidentiality of the transmitted data, allowing the application to handle any errors in a controlled manner.

8. Security Analysis

The Dual-Key Tunneling Protocol (DKTP) is a forward-secure, post-quantum resistant cryptographic tunneling protocol that integrates independently keyed asymmetric and symmetric operations, bidirectional shared secret contribution, and authenticated encrypted communication through the RCS stream cipher.
Its construction is both layered and recursive, meaning that cryptographic operations reinforce one another at each stage; handshake, key exchange, session derivation, and transmission, rather than relying on a singular root-of-trust. This section provides a complete cryptographic analysis of the protocol, including its handshake construction, symmetric and asymmetric key derivation logic, authenticated encryption, and ratcheting behavior.
The analysis demonstrates the protocol's resilience to classical and quantum attacks, while also identifying design strengths and trade-offs across the cryptographic stack.

8.1 Asymmetric Primitives and Key Exchange

The key exchange mechanism in DKTP employs a mutual contribution model where both parties independently generate ephemeral asymmetric key pairs and perform encapsulation or decapsulation operations. In a typical session, the server first generates a public encapsulation key and signs a hash of this key with its long-term signature key, transmitting both the key and its signature to the client. The client verifies the signature using the server's signature verification key and encapsulates a shared secret using the provided encapsulation key. The client then generates its own ephemeral key-pair and signs a hash of its encapsulation key and generated ciphertext, returning the ciphertext, key, and signature to the server. The server decapsulates the ciphertext to recover the client-generated shared secret and uses the client's signature verification key to validate the exchange.
This mutual key contribution model has two major benefits over classical designs such as TLS or SSH. First, it offers strong resistance to asymmetric downgrade attacks and partial key compromise, as both parties must generate and contribute to the tunnel key material independently. Second, because ephemeral key-pairs are used and destroyed upon session finalization, the resulting shared secrets (*secl* and *secr*) provide forward secrecy even in the event of a future compromise of the long-term signing keys.
Cryptographic strength here derives from the security of the encapsulation mechanism (e.g., Kyber or McEliece) and the unforgeability of the signature scheme (e.g., Dilithium or SPHINCS+). The encapsulated secrets are never exposed, and signature binding ensures that even an active man-in-the-middle cannot substitute keys without detection. Both shared secrets are independently derived and combined with symmetric pre-shared keys (*pssl*, *pssr*), ensuring dual-entropy in both directions of the tunnel.

A cryptographic adversary targeting DKTP would need to:

- Forge one of the ephemeral public keys to inject malicious ciphertext (prevented by signature validation),
- Compromise a party's ephemeral private key before key derivation completes (requires immediate, pre-image attack),
- Or guess both the pre-shared key and the asymmetric shared secret to recover a derived tunnel key (highly infeasible under current cryptographic assumptions).

The structure of the handshake ensures that replayed or malformed packets will be rejected due to signature validation failures, header mismatches, and UTC timestamp thresholding. Every cryptographic input; public keys, ciphertexts, packet headers, and configuration strings, is authenticated via hash functions before being signed, hashed, or encapsulated. The use of non-interactive KEMs eliminates the need for long-lived Diffie-Hellman key exchanges and removes reliance on round-trip state.

8.2 Symmetric Key Derivation and Pre-shared Key Updating

A defining feature of DKTP is its dual-channel key derivation structure. After the handshake, each party derives a transmit and receive tunnel channel key (*tckl* and *tckr*) using the combination of a shared secret (from KEM) and a pre-shared key (from prior state). This is done via a post-quantum KDF (e.g., SHAKE) with nonce mixing. Because the derivation is direction-specific (transmit and receive keys are derived separately), the cipher state of each channel is cryptographically isolated.

The pre-shared symmetric keys (*pssl*, *pssr*) are updated at the end of each handshake by hashing the current pre-shared key with the newly derived tunnel key. This approach ensures that the key state evolves forward, providing forward secrecy in the symmetric domain as well. The pre-shared key can be stored, rotated, or erased depending on the trust relationship between hosts, making DKTP adaptable to both ephemeral peering and long-lived device associations.

Unlike many protocols where the pre-shared key is used only to bootstrap the connection (e.g., PSK in TLS 1.3), DKTP treats the pre-shared key as a living component of the entropy pool. The repeated injection of tunnel-derived entropy into the pre-shared key ensures that if one tunnel key is ever compromised, future tunnel keys will be statistically unrelated and independent.

This update structure is not susceptible to rollback because the derived state is only accepted if both ends compute and confirm the new session key material. Additionally, the cryptographic binding to the session cookie (*sch*) ensures that no keying material can be inserted from an unrelated session or reused outside of its cryptographic context.

8.3 Authenticated Encryption and RCS Cipher Design

The encrypted data stream in DKTP is protected using RCS (Rijndael Cipher Stream), a wide-block Rijndael-based AEAD stream cipher with deep resistance to forgery, timing, and fault injection, and capable of up to 512-bit secure symmetric encryption and authentication. Each transmit and receive channel is initialized with a unique key and nonce derived from the

combined asymmetric and symmetric entropy pools, ensuring that even repeated communication between the same two parties never reuses keying material.
RCS provides built-in nonce handling, and its authenticated encryption supports associated data (AEAD), which is used in DKTP to bind packet headers into the cipher state. This means the packet flag, size, timestamp, and sequence number of every packet are authenticated along with the message body. The symmetric MAC function (KMAC) is also used to reinforce authenticity, ensuring that the ciphertext and metadata are bound to the correct cipher instance and packet context.
Each channel uses 256-bit or 512-bit derived keys (*enhanced mode*), ensuring an extremely high margin against brute force or related-key attacks. The MAC tags appended to each message are cryptographically bound to the session state, and any modification to the ciphertext, header, or associated data results in a MAC failure and tunnel teardown. RCS's Rijndael wide-block (256-bit state) core is subject to multiple rounds (256-bit: 22, 512-bit: 30), making it far more resistant to structural cryptanalysis than AES (128-bit block) based stream ciphers. The cipher also supports hardware acceleration and can be implemented in constant time, providing robust performance in embedded or high-security environments.
This combination of AEAD encryption with dual-key derivation, direction-specific state, and nonce uniqueness ensures message-level confidentiality and integrity, even under adversarial conditions. No ciphertext can be replayed, injected, or altered without detection, and any deviation from the expected protocol schedule causes the connection to abort.

8.4 Asymmetric Ratcheting Behavior

DKTP supports asymmetric ratcheting implicitly through its tunnel establishment and key update design. Each handshake generates fresh asymmetric key pairs, and each session injects newly derived entropy into the pre-shared symmetric key state. This means that each tunnel is cryptographically disconnected from the previous one, satisfying forward secrecy in both key dimensions.

Symmetric ratcheting is achieved via hash updates to *local* and *remote* pre-shared secrets (*pssl* and *pssr*) with tunnel key injection.

This operation:

- Prevents key reuse.
- Creates a non-linear key evolution path.
- Resists compromise chaining (i.e., knowledge of $pssr_n$ provides no advantage in predicting $pssr_{n+1}$).

Although not a formal double ratchet like Signal’s, the design achieves many of the same properties in a tunneling context, particularly:

- Perfect forward secrecy with minimal state.

- Asynchronous update safety (since the key state is not dependent on interactive ratchets).
- Stateless session restoration (if symmetric keys are preserved securely).

An optional re-key mechanism could be defined for long-lived tunnels, allowing asymmetric re-keying mid-session without disrupting the channel. However, the current design prioritizes strong key evolution per session rather than continuous ratcheting during transmission, which is appropriate for high-assurance tunnel security.

8.5 Attack Surface and Threat Resistance

DKTP's layered security properties allow it to defend against a broad range of attack classes:

- **Man-in-the-middle attacks** are mitigated through signed ephemeral keys and mutual signature verification.
- **Replay attacks** are blocked through strict UTC timestamp validation and sequence checking.
- **Ciphertext malleability** is prevented by AEAD binding and MAC code enforcement.
- **Key compromise** of long-term signing keys does not expose ephemeral sessions, due to forward secrecy of asymmetric exchange.
- **Quantum adversaries** are neutralized through the exclusive use of post-quantum KEMs and hash-based primitives.
- **Pre-shared key disclosure** does not expose past or future sessions due to state updates and non-invertibility of the key derivation process.

Any adversary attempting to compromise the tunnel must defeat both an asymmetric key exchange (e.g., Kyber, McEliece) and a MAC-authenticated AEAD layer, while also predicting the pre-shared state, the session cookie, and the tunnel key derivation structure. The compounded difficulty of this multi-dimensional attack space renders DKTP highly resistant to cryptographic breaks under all known cryptographic models.

8.6 Summation

DKTP's cryptographic structure is carefully designed to resist current and future threats, balancing protocol modularity with strong default properties such as mutual authentication, forward secrecy, bidirectional key separation, and authenticated stream encryption.
Its incorporation of post-quantum primitives, symmetric state advancement, and AEAD-protected communication makes it robust against both passive and active attacks, with no reliance on legacy assumptions or centralized trust anchors. The protocol achieves an excellent balance between cryptographic rigor and implementation practicality, offering a viable candidate for future-proof secure tunnels across infrastructure, embedded, and sovereign systems.

9. Use Cases

The Dual-Key Authenticated Tunneling Protocol (DKTP) was designed to operate under a rigorous set of post quantum, asymmetric-and-symmetric combined cryptographic requirements. Its architecture offers a number of unique capabilities that make it ideally suited for secure communications in both current and future threat landscapes. This section outlines the principal applications of DKTP in real-world environments, considering its cryptographic strengths, independence from certificate authorities, automatic forward and post-compromise security mechanisms, and adaptability to hybrid infrastructures.

High-Assurance Remote Access and Administrative Control

Modern infrastructure systems, including those in energy, transportation, healthcare, and defense, depend heavily on remote administration tools to maintain operational continuity. In these environments, DKTP provides a secure and verifiable communications channel between privileged operators and critical control nodes. The dual-key model, combining ephemeral asymmetric encapsulation with state-updated symmetric pre-shared keys; ensures that session keys are both forward-secure and post-compromise resistant.

Traditional VPNs and SSH-based tunnels often rely on asymmetric primitives whose long-term safety remains in question under quantum attack assumptions. By contrast, DKTP makes no trust assumptions on the longevity of any single cryptographic primitive. Even if the post-quantum KEM used is weakened in the future, the inclusion of a symmetric pre-shared key (PSK) layered into the tunnel key derivation ensures that the overall system degrades gracefully and never catastrophically. In high-assurance control environments where compromise of one channel can lead to cascading failures, this multi-layered security is essential.

Moreover, because each DKTP peer verifies its counterparty's signature using known public keys, without needing third-party validation or live certificate status checks; the protocol is ideally suited for deployments in segmented or sovereign infrastructure. It allows control nodes to authenticate administrators even in isolated or offline conditions, preserving availability and integrity under constrained conditions.

Field-Deployed Equipment and Tactical Communications

In field-deployable communications systems such as mobile command stations, forward-deployed satellites, emergency response networks, or temporary mesh infrastructure, DKTP provides a lightweight but robust tunneling framework. Traditional TLS-based VPNs often depend on uninterrupted certificate validation, revocation checks, or trusted time sources. DKTP requires none of these. Its model of locally maintained peering keys enables equipment to perform full identity validation and session key generation autonomously, even in fully disconnected states.

Each DKTP handshake provides independent transmit and receive key derivations using two encapsulated KEM exchanges and two symmetric keys (one local, one remote). This allows for asymmetric security domains—where, for example, outbound commands may be more sensitive

than inbound telemetry, and also aligns with asymmetric trust models, such as hierarchical command structures or multi-party mission coordination.
Further, DKTP's ratchet mechanism ensures that the pre-shared keys evolve automatically with each successful handshake. This means that even in hostile environments where capture of devices is a threat, the window of exposure is sharply bounded to the last completed session. If a peer is compromised, the symmetric ratchet ensures that future sessions do not reuse any key material derived from compromised state.

Financial Networks and Critical Transaction Channels

Financial institutions and central bank systems face two critical cryptographic demands: confidentiality over decades and resistance to quantum adversaries with access to intercepted traffic archives. DKTP was built to satisfy both. By combining a session-specific encapsulation handshake with long-lived stateful symmetric keys, the protocol offers the strongest guarantees of forward secrecy, and -critically- post-compromise security, which is a rarity in asymmetric-only key exchange models.
For settlement systems, ATM tunnels, secure messaging between clearinghouses, or secure interbank APIs, DKTP offers a lightweight yet tamper-evident alternative to TLS with built-in key rotation and bilateral trust. The signatures on ephemeral keys allow all handshakes to be audit-logged and attributed to specific entities, while the symmetric key derivation ensures deterministic authentication even in bandwidth-constrained settings.
Moreover, because DKTP uses a wide-block authenticated cipher (RCS) with KMAC for tag generation, it resists a broader class of integrity and timing attacks than narrow-block ciphers such as AES-GCM. This makes it suitable for use in environments where strong message ordering, replay prevention, and content authentication are essential, such as core banking systems or international fund transfer protocols.

Government and Diplomatic Communications

Governments and international diplomatic entities often require encrypted communications that are not only resilient but also independently verifiable, decentralized, and compatible with air-gapped or high-isolation deployments. DKTP's avoidance of a centralized CA or PKI infrastructure makes it particularly appealing in these domains. Mutual authentication is established using long-lived peering keys that are distributed securely at provisioning time. No online certificate validation is necessary at session time.
When embedded into secure phone systems, embassy-to-consulate tunnels, or internal secure document transport systems, DKTP's ratcheted key exchange ensures that intercepted messages cannot be decrypted at a later time, even if long-term key material is eventually exposed. In post-incident forensics, the clear logging of signature-verifiable handshakes enables attribution and non-repudiation; critical properties in state-level diplomatic or intelligence channels.

Summary Table - Application Fit of DKTP

APPLICATION DOMAIN	BENEFITS OF DKTP

SECURE ADMIN ACCESS	Mutual auth, key ratcheting, no CA
TACTICAL COMMS	Offline handshake, independent Rx/Tx keys
BANKING TUNNELS	PQ-resistance, forward & post-compromise security
GOVERNMENT LINKS	Sovereign trust model, audit-capable
INDUSTRIAL IOT	Lightweight symmetric support, scalable PSKs
DISASTER MESH	Self-authenticating, no live dependency
SECURE BOOTSTRAPPING	Device to device provisioning with full mutual auth
HYBRID VPNS	Works with or without CA support, X509 extension ready

Emerging Domains and Strategic Potential

With the imminent commoditization of quantum computing capabilities, many conventional VPNs and TLS tunnels will become structurally unfit for long-term confidentiality. DKTP's layered design, comprising two separate key classes, asymmetric and symmetric, directly addresses this. As a deployable technology, it enables organizations to design networks where session key compromise is not catastrophic and auditability, tamper-resistance, and cryptographic agility are preserved throughout the lifecycle.

DKTP also opens a pathway for hybrid deployments: organizations can gradually transition from classical key exchange protocols to DKTP by first enabling PSK-based fallback, then moving to dual-key mode as devices are updated. This flexibility ensures that DKTP is not merely an academic construct, but a future-proof communications infrastructure suitable for live operational rollout today.

Finally, the inclusion of an optional asymmetric ratchet mode, provides a defense-in-depth model whereby even if an attacker compromises one half of a session (e.g., the ephemeral private key), they gain no leverage over the opposite direction or any future session. This is the essence of post-compromise resilience, a property vital in adversarial network environments such as military, financial, or critical infrastructure control.

10. Conclusion

The Dual-Key Authenticated Tunneling Protocol (DKTP) represents a rigorous rethinking of cryptographic tunnel design for the post-quantum era. By integrating ephemeral asymmetric key exchanges with persistent, ratcheted symmetric key material, DKTP creates a layered cryptographic model that provides confidentiality, integrity, mutual authentication, and long-term resilience against both classical and quantum threats. It deliberately avoids reliance on fragile infrastructure such as certificate authorities or centralized trust hierarchies, instead offering a peer-driven trust framework suitable for sovereign, embedded, or high-assurance deployments.

The protocol's construction is characterized by cryptographic asymmetry and entropy duality. Each endpoint contributes to the final session state independently, both through KEM-based shared secrets and through directional pre-shared symmetric keys. This dual entropy contribution results in independently derived transmit and receive tunnel keys, ensuring that even if one direction of communication is compromised, the other remains protected. Through the use of post-quantum hash functions, KMAC authentication, and RCS stream encryption, DKTP guarantees message authenticity, replay resistance, and cryptographic unlinkability across all stages of the session.

In contrast to conventional tunnels, DKTP does not treat handshake and transmission as loosely coupled layers. Instead, it binds them tightly through cryptographic commitments, authenticated headers, and sequence-validated session identifiers. Each session begins with a signed assertion of configuration and identity, and concludes only when both parties confirm tunnel readiness through encrypted, authenticated acknowledgments. This structure prevents mid-session injection, downgrade, or time-based forgery, offering a robust defense model not present in simpler key agreement protocols.

The protocol’s ability to support both stateful symmetric key ratcheting and ephemeral asymmetric key regeneration positions it uniquely among secure tunneling systems. Forward secrecy is preserved across all sessions, and post-compromise security is achieved through progressive key updates. Each pre-shared key is evolved using output from the current session, ensuring that a compromise at any moment does not retroactively expose past communications, nor does it grant access to future sessions. This approach satisfies stringent cryptographic goals in adversarial threat models, including those involving state-level actors or long-term archival decryption attacks.

From a deployment perspective, DKTP is both adaptable and scalable. It can operate in fully decentralized networks, mesh architectures, peer-to-peer environments, and embedded control systems. Its trust model allows for autonomous key lifecycle management without external validation infrastructure, and its modular structure enables integration with existing post-quantum cryptographic libraries and secure enclave technologies. Moreover, its handshake structure, although longer than classical 2-RTT systems, provides an unparalleled degree of assurance in trust bootstrapping and key integrity.

In summary, DKTP offers a cryptographic and architectural response to the inadequacies of traditional tunnel designs in a world increasingly shaped by quantum capability and cryptanalytic acceleration. It is not merely compatible with existing security standards, it improves upon them. It defends against cryptographic obsolescence by employing primitives built to withstand future threat models, while at the same time enforcing operational principles rooted in sound design: entropy isolation, directionally distinct keys, tamper-evident handshake sequences, and built-in resilience against both temporal and persistent compromise. As such, DKTP stands as a secure, future-proof foundation for confidential and authenticated communication in the post-quantum world.