The Design and Formal Analysis of the Distributed Key Tunnel Protocol

John G. Underhill

Quantum Resistant Cryptographic Solutions Corporation
contact@qrcscorp.ca

November 24, 2025

Abstract

The Distributed Key Tunnel Protocol is a post quantum secure channel mechanism that combines asymmetric key exchange, digital signatures, pre-shared keys, state hashing, and a state evolving ratchet to provide authenticated confidentiality and integrity for long lived connections. The protocol uses a flexible construction that supports multiple post quantum key encapsulation and signature schemes, including lattice based families such as Kyber and Dilithium and optional code based or hash based alternatives such as Classic McEliece and SPHINCS+. Directional pre-shared keys may be configured to strengthen the tunnel establishment process and to provide hybrid symmetric protection that remains effective even if public key primitives are weakened in the future.

This paper provides a complete formal cryptanalysis of the protocol. It begins with an engineering level specification of all handshake messages, key schedule components, transcript hashing rules, and transport phase operations, including the derivation of directional root keys, RCS tunnel keys, and sequence based nonces. The description is based on a detailed examination of the DKTP specification and the reference implementation and is written to enable faithful verification of all state transitions and cryptographic inputs.

The paper then presents a formal model of DKTP that includes long term keys, configurable pre-shared keys, ephemeral key pairs, ratchet states, and the complete transcript driven key schedule. A game based adversarial model is defined with support for message manipulation, compromise of various key classes, quantum oracle queries to Keccak based func-

tions, and key indistinguishability tests. Security goals include authentication, handshake confidentiality, ciphertext integrity, replay resistance, forward secrecy, post compromise security, and ratchet security.
For each goal we provide a reduction to the underlying post quantum assumptions, including IND-CCA security of the selected key encapsulation mechanism, existential unforgeability of the selected signature scheme, pseudo-randomness of the domain separated Keccak based key derivation functions, and confidentiality and integrity of the RCS authenticated encryption system. The proofs include a transcript integrity lemma that incorporates pre-shared key inputs and formalizes binding of all handshake messages into a single authenticated transcript.
The formal analysis is complemented by an implementation conformance section that verifies that the state machine, message formats, KDF inputs, ratchet behavior, and failure handling in the reference implementation correspond to the formal model. A concrete security section provides security level estimates for typical parameter selections and quantifies the contribution of pre-shared keys. The results show that DKTP achieves strong post quantum security under a wide range of assumptions and provides a robust secure channel mechanism suitable for long term protection against both classical and quantum adversaries.

Contents

1 Introduction

The Distributed Key Tunnel Protocol is a post-quantum secure channel protocol designed to establish long lived, stateful, authenticated, and confidential tunnels between two peers. DKTP combines post quantum public key exchange, post quantum signatures, pre-shared symmetric keys, a cookie based session binding mechanism, and a state evolving tunnel key ratchet. The design emphasizes long term robustness, flexibility in the choice of cryptographic primitives, resistance to quantum capable adversaries, and precise control over session state transitions.

The protocol described in this paper is identical to the mechanism implemented in the DKTP reference code and documented in the DKTP technical specification. All handshake fields, key derivation inputs, associated data structures, replay protections, and ratcheting behaviors are drawn directly from those documents. The purpose of this formal analysis is to provide a rigorous, mathematically grounded foundation for the protocol as deployed, not for a simplified or idealized variant.

1.1 Background and Motivation

The transition from classical to post quantum cryptography presents several challenges for secure channel protocols. Classical authenticated key exchange protocols often rely on mathematical assumptions that are no longer viable under quantum adversaries. Post quantum alternatives exist, but many have different performance characteristics, different security reductions, and different requirements for key management.

DKTP addresses these issues by providing a modular construction that supports multiple post quantum key encapsulation mechanisms and digital signatures. The protocol does not assume a single fixed suite of primitives. Instead, it is compatible with a range of algorithms, including lattice based systems such as Kyber and Dilithium, code based systems such as Classic McEliece, and hash based systems such as SPHINCS+. Implementations select parameter sets at compile time through configuration and do not alter the structure of the protocol.

A second motivation is resilience in the presence of long term compromise. Many systems rely solely on public key material during the handshake. DKTP supplements this with directional pre-shared symmetric keys. These keys provide additional entropy, improve resistance to downgrade attacks, strengthen session establishment when long term static keys are exposed, and contribute to post compromise recovery after the session transitions into its transport mode.

Finally, DKTP is designed for systems that require stable stateful tunnels rather than single pass exchanges. The protocol integrates a tunnel level key derivation mechanism, per direction sequence numbers, authenticated timestamps, and a ratchet that updates symmetric keys after each handshake. This supports long-lived channel operation with periodic renewal of cryptographic state.

1.2 Design Objectives for DKTP

DKTP was created according to several engineering and cryptographic objectives.

- **Post-quantum security.** The protocol must rely exclusively on cryptographic primitives that remain secure in the presence of quantum adversaries. The KEM must satisfy IND-CCA security and the signature system must provide existential unforgeability under chosen message attack.
- **Support for algorithm agility.** The protocol must remain functional when instantiated with any supported post quantum KEM or signature scheme. This includes configurable options for Kyber, Dilithium, Classic McEliece, and SPHINCS+, selectable by the implementation without changing the packet structure.
- **Use of directional pre-shared keys.** Each direction of communication has an independent pre-shared key. The handshake must incorporate these keys to strengthen initial authentication and key derivation, and the protocol must update these keys during ratcheting to provide post compromise recovery.
- **Cookie based session binding.** The handshake does not use a transcript hash. Instead, the initiator and responder compute a session cookie from the session configuration and verification keys. This cookie is transmitted in early messages and authenticated in later messages, ensuring that both parties agree on the same session parameters and preventing tampering or reflection attacks.
- **Symmetric tunnel keys with directional separation.** The tunnel uses separate symmetric keys for sending and receiving. These keys are derived from the shared secrets produced by the KEM, combined with the directional pre-shared keys and the authenticated session cookie.
- **Replay and ordering guarantees.** All encrypted transport packets include a fixed format header containing a flag byte, a strictly increasing sequence number, a payload length, and a timestamp. This header is au-

thenticated during decryption, providing replay protection and ordering guarantees.

- **Secure implementation and state handling.** The protocol must be implementable in constant time. All ephemeral secrets, intermediate keying material, and ratchet states must be securely erased after use. Sequence numbers and nonce values must be monotonic, unique, and correctly authenticated.

These objectives shape the design and analysis of the protocol. They also guide the security model developed in this paper.

1.3 Contributions of this Paper

This paper provides the first complete, specification accurate, formal cryptanalysis of DKTP. Its contributions are:

- A detailed engineering description of the DKTP handshake and transport phases based strictly on the technical specification, including session cookies, directional pre-shared keys, KEM operations, signature validation, authenticated encryption, and state transitions.
- A formal model that captures all relevant components of the protocol, including long term asymmetric keys, directional pre-shared keys, configuration identifiers, ephemeral KEM keys, authenticated session cookies, shared secrets, tunnel keys, and ratchet values.
- Formal definitions of all security goals relevant to secure channel protocols, including authentication, confidentiality, integrity, replay resistance, forward secrecy, post compromise security, and ratchet security.
- Rigorous reduction based proofs showing how each security goal follows from the assumptions on the underlying primitives. These reductions include explicit treatment of pre-shared keys and the session cookie, and accurately reflect the behavior of the actual protocol.
- A cryptanalytic evaluation of the protocol that describes its attack surfaces, explains why each attack is mitigated, and identifies the security dependencies introduced by the KEM, the signature scheme, the AEAD mode, and the pre-shared key mechanism.
- A concrete security analysis that quantifies the strength of the protocol under different algorithm selections and different pre-shared key lengths.

- An implementation conformance section that verifies the alignment between the formal model and the DKTP reference implementation, including packet formats, KDF inputs, calls to cryptographic libraries, and erasure behavior.

1.4 Relationship to the Specification and Implementation

This formal analysis was developed using the DKTP technical specification, the reference C implementation, and the implementation analysis documents as authoritative sources. Every step of the handshake, every byte of every message, every input to the key derivation functions, and every update of the pre-shared keys is taken directly from those documents. The descriptions and proofs in this paper are therefore grounded in the protocol that is implemented and deployed.

By combining a detailed engineering description, a precise mathematical model, and complete security reductions, this document ensures that the DKTP protocol can be analyzed, implemented, and evaluated with confidence. The alignment among specification, implementation, and formal analysis provides a foundation for long term use of DKTP in environments that require high security against classical and quantum adversaries.

2 Engineering Description of DKTP

This section provides a complete and specification accurate engineering level description of the Distributed Key Tunnel Protocol. The purpose is to give a clear and detailed account of every component of the protocol before the formal model is introduced. All information in this section is drawn directly from the DKTP technical specification and the associated reference implementation.

DKTP establishes a secure channel between an initiator and a responder. The structure consists of a four pass post-quantum handshake followed by a stateful transport phase. The handshake authenticates the peers, agrees on configuration, derives directional shared secrets, incorporates pre-shared keys, and produces tunnel keys and initial nonces. The transport phase uses authenticated encryption with strict sequence numbers, timestamps, and flags to provide confidentiality, integrity, and replay protection.

2.1 Protocol Roles and State Machines

DKTP operates between two peers who take on the roles of initiator I and responder R. Each side maintains an explicit protocol state machine that transitions through the following phases.

- **Idle.** No session is active. The peer waits for a Connect-Request or initiates one.
- **Handshake.** The peer processes one of the handshake messages (Connect-Request, Connect-Response, Exchange, or Verification) and updates its handshake state accordingly.
- **Tunnel Established.** The peer has derived tunnel keys, initial nonces, and updated pre-shared keys. Transport packets may now be sent and received.
- **Ratchet Ready.** When a new handshake is initiated over an existing connection, the peer performs a ratchet operation to refresh directional PSKs and tunnel keys.

Each peer stores directional state variables, including sequence numbers, nonces, current tunnel keys, updated PSKs, and timers used for timestamp validation. The state machine is symmetric except for initiation order.

2.2 Long Term Keys and Configuration Parameters

Each peer has three classes of long lived cryptographic material.

Long term verification key pair. Each peer holds a static post quantum signature key pair (vk, sk) used to sign and verify handshake messages. The algorithm is configurable and may be Dilithium, SPHINCS+, or another supported post quantum signature system.

Directional pre-shared keys. DKTP maintains two independent symmetric pre-shared keys.

$$\mathsf{psk}_{I\to R}, \mathsf{psk}_{R\to I}.$$

These keys strengthen authentication, add symmetric protection to the handshake, and contribute to forward secrecy and post compromise recovery. Each PSK is updated at the end of the handshake using tunnel key material.

Configuration parameters. These include:

- the KEM algorithm and parameter set,
- the signature algorithm and parameter set,
- the allowed cipher suites,
- policy parameters such as timestamp window and maximum sequence advance.

These parameters are included in the session cookie to bind the session to the same configuration on both sides.

2.3 Core Primitives and Algorithm Agility

DKTP is designed to be agnostic to the underlying post-quantum primitives. Implementations select algorithms at compile time through the configuration file. DKTP supports:

- **Key Encapsulation Mechanisms:** Kyber parameter sets, Classic McEliece parameter sets.
- **Signature Schemes:** Dilithium parameter sets, SPHINCS+ variants.
- **Authenticated Encryption:** RCS, a wide-block Rijndael based authenticated encryption construction that uses cSHAKE for key schedule and KMAC style authentication.
- **Hash Function and KDF:** Keccak based functions with explicit domain separation, used to derive tunnel keys, nonces, and updated PSKs.

The protocol structure does not change when the cryptographic algorithms are changed.

2.4 Handshake Phases in Engineering Terms

From an engineering perspective, the DKTP handshake is realized as a fixed sequence of six protocol messages. These messages authenticate peer identities, perform a bidirectional KEM exchange, derive directional tunnel keys, and then explicitly confirm establishment before the tunnel is treated as operational.

M_1 : Connect-Request

M_2 : Connect-Response

M_3 : Exchange-Request

M_4 : Exchange-Response

M_5 : Establish-Request

M_6 : Establish-Response

Connect-Request (M1). The initiator transmits its identity and configuration material and authenticates the message using its long-term signature key. The message binds the packet header fields (sequence number and timestamp) to the authenticated payload.

Connect-Response (M2). The responder validates the initiator signature, generates an ephemeral KEM keypair, returns the encapsulation key, and signs the responder message hash. The responder also derives and stores a session cookie state value for later establishment confirmation.

Exchange-Request (M3). The initiator validates the responder signature, performs KEM encapsulation to the responder encapsulation key, generates its own ephemeral KEM keypair, and sends the encapsulated ciphertext and the initiator encapsulation key, authenticated under its long-term signature key.

Exchange-Response (M4). The responder validates the initiator signature, decapsulates the initiator ciphertext to obtain the initiator-to-responder shared secret, encapsulates to the initiator encapsulation key to obtain the responder-to-initiator shared secret, and derives the directional tunnel keys and nonces via the KDF.

Establish-Request (M5). Once both directional cipher instances are initialized, the initiator hashes the session cookie state together with the establish-request header fields and encrypts the resulting cookie-hash under the transmit channel cipher. This establishes explicit confirmation that the initiator derived the same tunnel state.

Establish-Response (M6). The responder decrypts and verifies the cookie-hash, updates the pre-shared keys, and returns an encrypted cookie-hash derived from the establish-response header fields. After the initiator validates this response, the tunnel is considered established and operational.
The tunnel is not treated as established solely by completion of the KEM exchange. Operational activation occurs only after the establish request and establish response messages are verified.

2.5 Key Schedule and Directional Separation

This subsection describes the DKTP tunnel key derivation from an engineering perspective, matching the specification. DKTP derives independent transmit and receive channel keys using the two KEM shared secrets produced during the exchange phase together with the pre-shared secrets provisioned in the peering key structures.

Directional secrets. After the exchange-response message, each peer holds two KEM-derived shared secrets:

- secl: the *local* shared secret derived from the peer's encapsulation operation to the remote ephemeral key (used for the local transmit channel).
- secr: the *remote* shared secret derived from decapsulation of the peer's received ciphertext (used for the local receive channel).

Pre-shared secrets. Each peer maintains two pre-shared secrets as part of the peering key material:

- pssl: the local host pre-shared secret.
- pssr: the remote host pre-shared secret.

Tunnel keys and nonces. Directional tunnel keys and initial nonces are derived using the DKTP KDF as specified:

$$(tckl,\ nl) \leftarrow \mathsf{KDF}_{\mathrm{DKTP}}(\mathsf{secl},\ \mathsf{pssr}), \qquad (tckr,\ nr) \leftarrow \mathsf{KDF}_{\mathrm{DKTP}}(\mathsf{secr},\ \mathsf{pssl}).$$

Here $tckl$ and nl initialize the local transmit cipher instance, while $tckr$ and nr initialize the local receive cipher instance. This assignment yields strict directional separation.

Activation. Cipher instances are initialized after key derivation, but the tunnel is treated as established only after the establish-request and establish-response messages are successfully processed.

PSK update. After establishment is confirmed, pre-shared secrets are updated as specified:

$$\mathsf{pssl} \leftarrow \mathsf{Hash}(\mathsf{pssl} \parallel tckl), \qquad \mathsf{pssr} \leftarrow \mathsf{Hash}(\mathsf{pssr} \parallel tckr).$$

These updates are persisted alongside the peering key structures.

2.6 Transport Phase, Ratcheting, and Tunnel Operation

Once the handshake finishes, both peers enter the tunnel phase. All transport packets are encrypted and authenticated using RCS with the current tunnel key and nonce.
Each packet header contains:

- a flag byte,
- an eight byte sequence number,
- a four byte payload length,
- an eight byte timestamp.

This header is authenticated as AEAD associated data.
The ratchet is triggered when a new handshake is initiated during an established session. The resulting handshake produces new directional root keys, which update:

$$\mathsf{psk}_{I\to R}, \mathsf{psk}_{R\to I}, tckl, tckr.$$

This provides post compromise recovery.

2.7 Replay Protection and Ordering Requirements

Replay protection and strict ordering are provided by:

- authenticated sequence numbers,
- monotonically increasing counters in each direction,
- timestamp freshness checks,
- AEAD authentication of the entire header.

The receiver rejects:

- duplicate sequence numbers,
- out of order packets,
- timestamp violations,
- packets failing integrity verification.

2.8 Failure Handling, Zeroization, and Implementation Notes

The DKTP implementation performs secure erasure of:

- ephemeral KEM private keys,
- derived shared secrets,
- tunnel keys and nonces,
- pre-shared keys immediately after update,
- intermediate buffers used during encryption and decryption.

All cryptographic routines use constant time operations where applicable. The random number generator is required to be cryptographically secure and to provide entropy for ephemeral keys and nonces.
Errors at any phase of the handshake cause immediate session termination and state destruction.

3 Formal Preliminaries

This section introduces the notation, cryptographic primitives, and adversarial capabilities that form the basis of the security model for DKTP. The formalism presented here aligns with the engineering description of the protocol and matches the state variables and operations found in the DKTP specification and reference implementation. All assumptions are stated in terms of standard post quantum definitions and oracle based adversarial interaction.

3.1 Notation and Conventions

We use uppercase letters for keys and shared secrets, lowercase letters for messages, and bold symbols for algorithms. Byte strings are written as elements of $\{0,1\}^*$. Concatenation is written as $A \parallel B$. For a hash or KDF with domain separation, we write

$$\mathsf{Hash}_{\text{label}}(X), \mathsf{KDF}_{\text{label}}(X),$$

to indicate that the internal function is the same but that a label string is prepended within the domain separation space.
For a peer P, we denote its long term signature key pair by (vk_P, sk_P) and its directional pre-shared keys by

$$\mathsf{psk}_{P\to Q}, \mathsf{psk}_{Q\to P}.$$

Each session has a configuration value config and a session cookie cookie computed from config and the verification keys. The cookie is treated as a public but authenticated value.
Random sampling from a distribution is written as $x \leftarrow \mathcal{D}$.

3.2 Post Quantum Primitives (General KEM and Signature Families)

DKTP supports a range of post quantum key encapsulation and signature schemes. In the formal model, these are treated generically as families of algorithms with standard security properties. The specific implementation may select Kyber, Classic McEliece, or another IND-CCA secure KEM, and may select Dilithium, SPHINCS+, or another EUF CMA secure signature scheme.

Key encapsulation mechanism. A post quantum KEM consists of the algorithms

$$(pk, sk) \leftarrow \mathsf{KEM.KeyGen}(),$$

$$(c, K) \leftarrow \mathsf{KEM.Encap}(pk),$$

$$K \leftarrow \mathsf{KEM.Decap}(c, sk).$$

The security requirement is IND-CCA under classical and quantum adversaries. DKTP uses two encapsulations to derive directional shared secrets:

$$\mathsf{secl}, \mathsf{secr}.$$

Signature scheme. A post quantum digital signature scheme consists of the algorithms

$$(vk, sk) \leftarrow \mathsf{SIG.KeyGen}(),$$

$$\sigma \leftarrow \mathsf{SIG.Sign}(sk, m),$$

$$\mathsf{SIG.Verify}(vk, m, \sigma) \in \{0, 1\}.$$

The security requirement is existential unforgeability under chosen message attack. The Verification message in DKTP relies on this property.

3.3 Hash and KDF with Domain Separation

DKTP uses Keccak based functions for hashing and key derivation. The exact functions (cSHAKE, KMAC, or a KDF built on these primitives) are abstracted in the formal model while maintaining the domain separation structure described in the specification.
Domain separation labels include:

- `DKTP Cookie` for computing the session cookie,
- `DKTP Root IToR` and `DKTP Root RToI` for the directional root keys,
- `DKTP Tunnel` for deriving tunnel keys and initial nonces,
- `DKTP PSK Update` for updating the directional pre-shared keys.

Each label corresponds to a different prefix or customization string inside the KDF.
The KDF is treated as a pseudo-random function family in the quantum random oracle model.

3.4 Pre-Shared Keys in the Formal Model

In the formal model, DKTP uses two pre-shared secrets stored in the peering key structures:

$$\mathsf{pssl}\ \text{(local pre-shared secret)}, \qquad \mathsf{pssr}\ \text{(remote pre-shared secret)}.$$

These values seed tunnel key derivation together with the KEM shared secrets. They are not used as independent authentication mechanisms, and they do not replace the asymmetric exchange.

Use in key derivation. The transmit and receive channel keys are derived as:

$$(tckl,\ nl) \leftarrow \mathsf{KDF}_{\mathrm{DKTP}}(\mathsf{secl},\ \mathsf{pssr}), \qquad (tckr,\ nr) \leftarrow \mathsf{KDF}_{\mathrm{DKTP}}(\mathsf{secr},\ \mathsf{pssl}).$$

Update rule. After establishment confirmation, the pre-shared secrets are updated:

$$\mathsf{pssl} \leftarrow \mathsf{Hash}(\mathsf{pssl} \parallel tckl), \qquad \mathsf{pssr} \leftarrow \mathsf{Hash}(\mathsf{pssr} \parallel tckr).$$

This update is persisted in the peering key structures and influences subsequent handshakes.

Compromise considerations. Compromise of pssl or pssr affects the security of sessions derived using that material, but does not remove the need to break the KEM to reconstruct the full directional tunnel state in sessions where the KEM secrets remain uncompromised.

3.5 Assumptions and Adversarial Interaction

The adversary $\mathcal{A}$ controls all network communication. It may delay, reorder, inject, or modify messages. The adversary interacts with instances of the protocol through oracles that allow the creation of sessions, message delivery, and key exposure.
The main assumptions are:

- KEM security: IND-CCA under quantum adversaries.
- Signature security: EUF CMA under classical and quantum adversaries.
- KDF security: pseudo-randomness of the domain separated Keccak based construction.
- AEAD security: confidentiality and integrity of RCS.
- Pre-shared keys remain secret and uncompromised.

Implementation assumptions, such as secure erasure, constant time KEM decapsulation, and CSPRNG quality, are stated separately in the implementation conformance section.

3.6 Oracle Definitions

The adversary interacts with the protocol through the following oracles. Each oracle operates on a session (P, sid) belonging to party P.

- **Send**(P, sid, m). Delivers message m to the local protocol instance. Used to initiate new sessions and to inject handshake and transport packets.
- **RevealEPH**(P, sid). Reveals the ephemeral KEM private key used in the handshake for that session. This oracle is restricted by freshness conditions in the security games.
- **RevealTunnelKey**(P, sid). Reveals the directional tunnel key for a completed session. Forbidden for sessions that will be tested in confidentiality games.

- **Corrupt**(P). Reveals the long term signature secret key of party P. Permitted in some games and forbidden in others depending on the security goal.
- **Test**(P, sid). Used in confidentiality, forward secrecy, post compromise security, and ratchet security games. Returns either the real key or a random key depending on a hidden bit chosen by the challenger.

These oracle definitions form the basis for all security games in later chapters. They accurately reflect the operations that the implementation performs and the ways in which an adversary might attempt to gain information or influence session state.

4 Formal Protocol Specification

This section presents a precise, specification accurate formal description of the DKTP protocol. Every field, operation, and state component is taken directly from the technical specification and the reference implementation. The handshake, cookie binding, key schedule, tunnel derivation, and ratchet evolution are given in a mathematically explicit form suitable for later security analysis.

4.1 Formal Handshake Messages

This subsection defines the DKTP handshake message flow in a form suitable for security analysis, matching the protocol specified in the DKTP Specification. The handshake consists of six messages exchanged between an initiator I and a responder R.

Long-term keys and header binding. Each peer possesses a long-term signature keypair $(\mathsf{sk}_X^{sig}, \mathsf{vk}_X^{sig})$ for $X \in \{I, R\}$. Handshake authentication binds the serialized packet header Hdr (including sequence number and timestamp) to the message payload by signing a hash of (payload $\|$ Hdr).

Message M1: Connect-Request ($I \to R$). The initiator sends configuration and identification material and authenticates it under its signature key. We model the payload abstractly as

$$M_1 = (\mathsf{kid\_array},\ \mathsf{config},\ \mathsf{vk}_I^{sig}),$$

and the transmitted message includes a signature

$$\sigma_1 = \mathsf{Sig.Sign}(\mathsf{sk}_I^{sig},\ \mathsf{Hash}(M_1 \parallel \mathsf{Hdr})).$$

Message M2: Connect-Response ($R \to I$). The responder verifies σ_1, generates an ephemeral KEM keypair, stores the decapsulation key, and returns the encapsulation key:

$$(\mathsf{dk}_R, \mathsf{ek}_R) \leftarrow \mathsf{KEM.Gen}.$$

The responder also derives and stores a *session cookie state* value sch as defined by the specification (Section 4.2). The payload is modeled as

$$M_2 = (\mathsf{ek}_R,\ \mathsf{sch\_commit}),$$

where $\mathsf{sch\_commit}$ denotes the authenticated commitment value to sch used by the implementation. The responder signs

$$\sigma_2 = \mathsf{Sig.Sign}(\mathsf{sk}_R^{sig},\ \mathsf{Hash}(M_2 \parallel \mathsf{Hdr})).$$

Message M3: Exchange-Request ($I \to R$). The initiator verifies σ_2, performs KEM encapsulation to ek_R obtaining the remote secret secr and ciphertext c_{pta}:

$$(c_{pta},\ \mathsf{secr}) \leftarrow \mathsf{KEM.Encap}(\mathsf{ek}_R).$$

It also generates its own ephemeral KEM keypair $(\mathsf{dk}_I, \mathsf{ek}_I)$ ⍰

Message M4: Exchange-Response ($\mathrm{R} \to \mathrm{I}$). The responder verifies σ_3 and decapsulates $\mathrm{c}_{\mathrm{pta}}$ to obtain its copy of the remote secret secr. The responder encapsulates to ek_{I} to obtain the local secret secl and ciphertext $\mathrm{c}_{\mathrm{ptr}}$:

$$(\mathrm{c}_{\mathrm{ptr}},\ \mathsf{secl}) \leftarrow \mathsf{KEM.Encap}(\mathsf{ek}_{\mathrm{I}}).$$

The responder derives directional tunnel keys and nonces as in Section 4.5, initializes cipher instances, and returns $\mathrm{c}_{\mathrm{ptr}}$ authenticated under its signature:

$$\mathrm{M}_4 = (\mathrm{c}_{\mathrm{ptr}}), \qquad \sigma_4 = \mathsf{Sig.Sign}(\mathsf{sk}_{\mathrm{R}}^{\mathrm{sig}},\ \mathsf{Hash}(\mathrm{M}_4 \parallel \mathsf{Hdr})).$$

The initiator verifies σ_4 and decapsulates $\mathrm{c}_{\mathrm{ptr}}$ to obtain its copy of secl, then derives the same directional tunnel keys and initializes cipher instances.

Message M5: Establish-Request ($\mathrm{I} \to \mathrm{R}$). Let sch denote the session cookie state value held by both peers after processing M_2. The initiator computes a cookie-hash bound to the establish-request header fields:

$$\mathrm{h}_{\mathrm{sch}} \leftarrow \mathsf{Hash}(\mathsf{sch} \parallel \mathsf{Hdr}),$$

and encrypts it under the transmit channel cipher instance with associated data equal to the serialized header. The payload is a ciphertext $\mathrm{cpt}_{\mathrm{I}}$.

Message M6: Establish-Response ($\mathrm{R} \to \mathrm{I}$). The responder decrypts and verifies the establish-request cookie-hash. It then updates the pre-shared secrets as specified, recomputes a cookie-hash bound to the establish-response header fields, encrypts it under the transmit channel cipher instance, and returns ciphertext cpt_R. The initiator decrypts and validates cpt_R. After this point the tunnel is considered established.
This message flow is the basis for the authentication and key indistinguishability games defined in subsequent sections.

4.2 Cookie Based Transcript Binding

DKTP binds the handshake transcript to the establishment confirmation using a session cookie state value sch. The value sch is derived during connect-response processing and is later incorporated into the establish-request and establish-response messages in hashed form, bound to the corresponding establish packet header fields.

Session cookie state derivation. The responder derives and stores sch as a hash over immutable handshake inputs. In the specification, these inputs include the negotiated configuration, the key identification material, and responder-generated values used during the connect-response step. We model this as:

$$\mathsf{sch} \leftarrow \mathsf{Hash}_{\mathrm{DKTP\text{-}SCH}}(\mathsf{kid\_array} \parallel \mathsf{config} \parallel \mathsf{ek}_\mathrm{R} \parallel \mathsf{vk}_\mathrm{I}^{\mathrm{sig}} \parallel \mathsf{vk}_\mathrm{R}^{\mathrm{sig}}).$$

The initiator obtains the necessary commitment information during processing of the connect-response message and reconstructs the corresponding sch value as specified.

Establishment binding. During establishment, the session cookie state is combined with the establish packet header fields to produce a cookie-hash:

$$\mathrm{h}_{\mathrm{sch}} \leftarrow \mathsf{Hash}_{\mathrm{DKTP\text{-}EST}}(\mathsf{sch} \parallel \mathsf{Hdr}).$$

The value $\mathrm{h}_{\mathrm{sch}}$ is then encrypted under the active channel cipher instance with associated data equal to the serialized header. Because Hdr includes sequence number and timestamp fields, this prevents replay and transcript splicing across sessions.
The cookie state sch is not a secret. Its purpose is to provide transcript binding and explicit confirmation that both peers derived identical tunnel state before the tunnel is considered established.

4.3 Formal Key Schedule (Including PSK Mixing)

This subsection defines the DKTP tunnel key derivation in the formal model, matching the DKTP Specification. DKTP derives independent transmit and receive channel keys by applying the DKTP KDF to the KEM shared secrets and the pre-shared secrets stored in the peering key structures.

KEM shared secrets. Let secr denote the shared secret derived from decapsulation of the received ciphertext (the remote-derived secret), and let secl denote the shared secret derived locally during encapsulation to the peer's ephemeral key (the local-derived secret). Both peers obtain identical values of secr and secl after completion of the exchange-response step.

Pre-shared secrets. Each peer maintains:

$$\mathsf{pssl}\text{ (local pre-shared secret)}, \qquad \mathsf{pssr}\text{ (remote pre-shared secret)}.$$

Directional tunnel keys and nonces. The DKTP KDF derives both a tunnel channel key and an initial nonce per direction:

$$(\mathrm{tckl},\ \mathrm{nl}) \leftarrow \mathsf{KDF}_{\mathrm{DKTP}}(\mathsf{secl},\ \mathsf{pssr}), \qquad (\mathrm{tckr},\ \mathrm{nr}) \leftarrow \mathsf{KDF}_{\mathrm{DKTP}}(\mathsf{secr},\ \mathsf{pssl}).$$

The values (tckl, nl) initialize the local transmit cipher instance, and (tckr, nr) initialize the local receive cipher instance.

Establishment and PSK update. After successful establishment confirmation, pre-shared secrets are updated:

$$\mathsf{pssl} \leftarrow \mathsf{Hash}(\mathsf{pssl} \parallel \mathrm{tckl}), \qquad \mathsf{pssr} \leftarrow \mathsf{Hash}(\mathsf{pssr} \parallel \mathrm{tckr}).$$

These updated values persist in the peering key structures and seed subsequent handshakes.
This derivation is the sole mechanism by which DKTP produces initial channel keys and nonces in the formal model.

4.4 Security Margin and Long-Horizon Threat Model

This subsection clarifies the threat model and design rationale underlying DKTP's choice of cryptographic parameters. It does not introduce additional security assumptions beyond those already used in the formal analysis, but explains the intended security horizon and margin considerations.

Long-horizon adversary model. DKTP is designed for scenarios in which confidentiality and integrity of communications are expected to remain robust over extended time horizons, including periods in which cryptanalytic techniques, computational capabilities, or implementation attack surfaces may evolve in unpredictable ways. The design therefore adopts conservative parameter choices that exceed those commonly used in deployed secure channel protocols.

Symmetric security margin. DKTP instantiates its transport channel using RCS-512, a 512-bit keyed authenticated encryption scheme with large nonce space and configurable authentication tag length. Directional channel keys are derived independently for each communication direction, yielding two cryptographically independent channel instances per session.

The intent of this construction is not to claim information-theoretic security, but to provision a substantial security margin against: (i) generic brute-force search, (ii) quantum speedups such as Grover-style attacks, and (iii) unforeseen advances in cryptanalysis affecting symmetric primitives.

Consistent with the RCS security analysis, classical security margins scale with the symmetric key length and sponge capacity, while post-quantum security is conservatively estimated under generic quadratic speedup assumptions.

Integrity margin and misuse resistance. In addition to key length, DKTP emphasizes large authentication margins and strict nonce discipline. Authentication tags and MAC constructions are selected to significantly exceed conventional minimum sizes, reducing the probability of successful forgery even under high query volumes or partial state exposure.

Relation to compromise protection. Extended symmetric margins complement, but do not replace, other compromise-mitigation mechanisms in DKTP, including: ephemeral key exchange, explicit establishment confirmation, directional separation of channel state, evolving persistent secret material across sessions, and mandatory erasure of ephemeral secrets after establishment.

The overall security of DKTP therefore derives from a combination of conservative cryptographic margins and strict state lifecycle management, rather than reliance on any single primitive or assumption.

Scope of claims. All security claims in this document remain computational in nature and are conditioned on the assumed hardness of the underlying primitives and the correct implementation of the protocol. DKTP does not claim detection of passive eavesdropping or information-theoretic secrecy, but aims to approach the upper bound of achievable security guarantees within a purely software-based cryptographic framework.

4.5 RCS Key and Nonce Derivation

DKTP derives the RCS channel keys and initial nonces directly from the KEM shared secrets and the pre-shared secrets in the peering key structures, matching the specification.

Transmit channel derivation. The local transmit channel key and initial nonce are derived from the locally generated KEM shared secret and the remote pre-shared secret:

$$(\mathrm{tckl},\ \mathrm{nl}) \leftarrow \mathsf{KDF}_{\mathrm{DKTP}}(\mathsf{secl},\ \mathsf{pssr}).$$

Receive channel derivation. The local receive channel key and initial nonce are derived from the remotely generated KEM shared secret and the local pre-shared secret:

$$(\mathrm{tckr},\ \mathrm{nr}) \leftarrow \mathsf{KDF}_{\mathrm{DKTP}}(\mathsf{secr},\ \mathsf{pssl}).$$

Cipher initialization. The receive channel cipher is initialized as $\mathsf{RCS}_{\mathrm{rx}}(\mathrm{tckr}, \mathrm{nr})$ and the transmit channel cipher is initialized as $\mathsf{RCS}_{\mathrm{tx}}(\mathrm{tckl}, \mathrm{nl})$.

PSK update. After establishment confirmation, pre-shared secrets are updated as:

$$\mathsf{pssl} \leftarrow \mathsf{Hash}(\mathsf{pssl} \parallel \mathrm{tckl}), \qquad \mathsf{pssr} \leftarrow \mathsf{Hash}(\mathsf{pssr} \parallel \mathrm{tckr}).$$

This update is persisted in the peering key structures for subsequent handshakes.

4.6 Ratchet Chains and State Evolution

DKTP uses a ratchet that updates symmetric state at each handshake. When a new handshake occurs during an established session, the KEM shared secrets, directional PSKs, tunnel keys, and nonces are all refreshed.
Formally, at ratchet step i:

$$\mathsf{secl}_{i+1},\ \mathsf{secr}_{i+1} \quad \text{derived from new KEM operations,}$$

$$K^{I\to R}_{\mathrm{root},i+1}, K^{R\to I}_{\mathrm{root},i+1} \quad \text{derived from the new PSKs and new cookie,}$$

$$\mathrm{tckl}_{i+1},\ \mathrm{nl}_{i+1}, \mathrm{tckr}_{i+1},\ \mathrm{nr}_{i+1} \quad \text{derived from the new root keys.}$$

Old keys, shared secrets, ephemeral private keys, and provisional handshake secrets are erased immediately after use.

4.7 Transport Encryption and Associated Data Structure

After the handshake, all application data is exchanged as encrypted transport packets. Each packet contains a fixed 21 byte header:

$$\text{header} = (\text{flag},\ \text{seq},\ \text{len},\ \text{timestamp}),$$

where:

- flag is a one byte control field,
- seq is an eight byte strictly increasing sequence number,
- len is a four byte payload length,
- timestamp is an eight byte authenticated time value.

The header is authenticated as AEAD associated data under RCS:

$$\mathsf{ciphertext} = \mathsf{RCS.Enc}_{\text{tck, nonce}}(\text{header},\ \text{plaintext}).$$

The receiver verifies:

$$\mathsf{RCS.Dec}_{\text{tck, nonce}}(\text{header},\ \mathsf{ciphertext}) \quad \text{before accepting the packet.}$$

The nonce increments according to the sequence number, ensuring no reuse and guaranteeing replay protection.

4.8 Handshake Pseudo-code Definitions

Client: Connect Request. This function constructs the initial Connect Request message from the client. It verifies that the long term key material has not expired, assembles the verification key and configuration fields, computes the session cookie hash from the configuration and verification keys, and signs the header and payload using the client long term signature key. The resulting packet contains no sensitive secret material and establishes the starting point for the authenticated handshake. The state variable schash is stored for later use in the encrypted verification step.

Algorithm 1 ClientConnectRequest

```
Require: Client kex state kcs, connection state cns
Ensure: Outgoing packet pkt of type connect_request or error
 1: t ← UTC_now()
 2: if t > kcs.expiration then
 3:     cns.exflag ← none
 4:     return key_expired
 5: end if
 6: pkt.pmessage[0..DKTP_KEYID_SIZE) ← kcs.keyid
 7: pkt.pmessage[DKTP_KEYID_SIZE..) ← DKTP_CONFIG_STRING
 8: mtype ← connect_request
 9: mlen ← KEX_CONNECT_REQUEST_MESSAGE_SIZE
10: HeaderCreate(pkt, mtype, cns.txseq, mlen)
11: shdr ← SerializeHeader(pkt)
12: phash ← SHA3_512(shdr || pkt.pmessage[0..DKTP_KEYID_SIZE + DKTP_CONFIG_SIZE))
13: sig ← Sign(phash, kcs.sigkey)
14: Write sig into pkt.pmessage immediately after keyid and config
15: kcs.schash ← SHA3_512(DKTP_CONFIG_STRING || kcs.keyid || kcs.verkey || kcs.rverkey)
16: cns.exflag ← connect_request
17: return none
```

Server: Connect Response. This function processes the client Connect Request and verifies that the configuration and signature fields are valid. It reconstructs the packet hash from the received header and payload and checks it against the client signature. It then computes the session cookie in the same way the client does and generates a fresh ephemeral KEM key pair for the responder. The responder signs its ephemeral public key bound to the cookie and returns this information to the client in the Connect Response. The server also stores its copy of the session cookie hash for later authenticated steps.

Algorithm 2 SERVERCONNECTRESPONSE

```
Require: Server kex state kss, connection state cns, incoming packet $pkt_{in}$
Ensure: Outgoing packet $pkt_{out}$ of type connect_response or error
 1: t ← UTC_now()
 2: if t > kss.expiration then
 3:     cns.exflag ← none
 4:     return key_expired
 5: end if
 6: cfg ← config bytes from $pkt_{in}$.pmessage
 7: if cfg ≠ DKTP_CONFIG_STRING then
 8:     cns.exflag ← none
 9:     return unknown_protocol
10: end if
11: sig || hm ← tail of $pkt_{in}$.pmessage
12: if ¬Verify(sig, hm, kss.rverkey) then
13:     cns.exflag ← none
14:     return authentication_failure
15: end if
16: shdr ← SerializeHeader($pkt_{in}$)
17: hmc ← SHA3_512(shdr || kid || cfg)
18: if VerifyEqual(hm, hmc) ≠ 0 then
19:     cns.exflag ← none
20:     return verify_failure
21: end if
22: kss.schash ← SHA3_512(DKTP_CONFIG_STRING || kss.keyid ||
    kss.rverkey || kss.verkey)
23: (kss.enckey, kss.deckey) ← KEM.KeyGen()
24: mtype ← connect_response
25: mlen ← KEX_CONNECT_RESPONSE_MESSAGE_SIZE
26: HeaderCreate($pkt_{out}$, mtype, cns.txseq, mlen)
27: shdr′ ← SerializeHeader($pkt_{out}$)
28: phash ← SHA3_512(shdr′ || kss.enckey)
29: sig′ ← Sign(phash, kss.sigkey)
30: Write sig′ then kss.enckey into $pkt_{out}$.pmessage
31: cns.exflag ← connect_response
32: return none
```

Client: Exchange Request. This function processes the server Connect Response and authenticates the responder through its long term verification key. It reconstructs the expected header hash, verifies the signature, and then performs encapsulation against the server ephemeral KEM public key to derive the client to server shared secret. The client generates its own ephemeral KEM key pair and sends both the ciphertext and its ephemeral public key to the server. This message includes a signature that authenticates the client contribution to the handshake transcript. The function stores the first KEM shared secret for use in the key derivation and ratchet steps.

Algorithm 3 CLIENTEXCHANGEREQUEST

```
Require: Client kex state kcs, connection state cns, server packet $\mathsf{pkt_{in}}$ (connect response)
Ensure: Outgoing packet $\mathsf{pkt_{out}}$ (exchange request) or error
 1: Parse $\mathsf{pkt_{in}}$.pmessage as
                      $\mathsf{sig} \,\|\, \mathsf{hm} \,\|\, \mathsf{pubk}$
    where pubk is server KEM public key
 2: if $\neg$Verify(sig, hm, kcs.rverkey) then
 3:     cns.exflag ← none
 4:     return authentication_failure
 5: end if
 6: shdr ← SerializeHeader($\mathsf{pkt_{in}}$)
 7: hmc ← SHA3_512(shdr || pubk)
 8: if VerifyEqual(hmc, hm) $\neq 0$ then
 9:     cns.exflag ← none
10:     return verify_failure
11: end if
12: (kcs.secl, cpta) ← KEM.Encapsulate(pubk)
13: (kcs.enckey, kcs.deckey) ← KEM.KeyGen()
14: Write cpta then kcs.enckey into $\mathsf{pkt_{out}}$.pmessage
15: mtype ← exchange_request
16: mlen ← KEX_EXCHANGE_REQUEST_MESSAGE_SIZE
17: HeaderCreate($\mathsf{pkt_{out}}$, mtype, cns.txseq, mlen)
18: shdr′ ← SerializeHeader($\mathsf{pkt_{out}}$)
19: phash ← SHA3_512(shdr′ || cpta || kcs.enckey)
20: sig′ ← Sign(phash, kcs.sigkey)
21: Append sig′ to $\mathsf{pkt_{out}}$.pmessage
22: cns.exflag ← exchange_request
23: return none
```

Server: Exchange Response And Ratchet. This function processes the client Exchange Request and completes the asymmetric key agreement. It verifies the client signature and computes the shared secret secr by decapsulating the ciphertext received from the client. It then encapsulates to the client ephemeral key to compute the second shared secret secl. With both directional shared secrets available, the function computes the directional tunnel keys and initial nonces through cSHAKE based expansion and updates both directional pre shared keys using the DKTP PSK update rule. It initializes the RCS contexts for transmitting and receiving and prepares the Exchange Response containing the second ciphertext and the server signature. This function performs the full server side ratchet and symmetric state initialization.

Algorithm 4 SERVEREXCHANGERESPONSEANDRATCHET

```
Require: Server kex state kss, connection state cns, client packet $\mathsf{pkt_{in}}$ (exchange request)
Ensure: Outgoing packet $\mathsf{pkt_{out}}$ (exchange response) or error
 1: Parse $\mathsf{pkt_{in}.pmessage}$ as
        $$\mathsf{cpta} \parallel \mathsf{pubk_C} \parallel \mathsf{sig}$$
    where cpta is client ciphertext, $\mathsf{pubk_C}$ is client KEM public key
 2: if $\neg\mathsf{Verify}(\mathsf{sig}, \mathsf{khash}, \mathsf{kss.rverkey})$ then
 3:     $\mathsf{cns.exflag} \leftarrow$ none
 4:     return authentication_failure
 5: end if
 6: $\mathsf{shdr} \leftarrow \mathsf{SerializeHeader}(\mathsf{pkt_{in}})$
 7: $\mathsf{phash} \leftarrow \mathsf{SHA3\_512}(\mathsf{shdr} \parallel \mathsf{cpta} \parallel \mathsf{pubk_C})$
 8: if $\mathsf{VerifyEqual}(\mathsf{phash}, \mathsf{khash}) \neq 0$ then
 9:     $\mathsf{cns.exflag} \leftarrow$ none
10:     return hash_invalid
11: end if
12: $\mathsf{secr} \leftarrow \mathsf{KEM.Decapsulate}(\mathsf{cpta}, \mathsf{kss.deckey})$
13: if decapsulation fails then
14:     $\mathsf{cns.exflag} \leftarrow$ none
15:     return decapsulation_failure
16: end if
17: $(\mathsf{secl}, \mathsf{cptb}) \leftarrow \mathsf{KEM.Encapsulate}(\mathsf{pubk_C})$
18: Write cptb into $\mathsf{pkt_{out}.pmessage}$
19: $\mathsf{mtype} \leftarrow$ exchange_response
20: $\mathsf{mlen} \leftarrow$ KEX_EXCHANGE_RESPONSE_MESSAGE_SIZE
21: $\mathsf{HeaderCreate}(\mathsf{pkt_{out}}, \mathsf{mtype}, \mathsf{cns.txseq}, \mathsf{mlen})$
22: $\mathsf{shdr}' \leftarrow \mathsf{SerializeHeader}(\mathsf{pkt_{out}})$
23: $\mathsf{phash}' \leftarrow \mathsf{SHA3\_512}(\mathsf{shdr}' \parallel \mathsf{cptb})$
24: $\mathsf{sig}' \leftarrow \mathsf{Sign}(\mathsf{phash}', \mathsf{kss.sigkey})$
25: Append $\mathsf{sig}'$ to $\mathsf{pkt_{out}.pmessage}$
26:     // Derive symmetric keys and perform PSK ratchet, server view
27: $\mathsf{prnd} \leftarrow \mathsf{cSHAKE512\_XOF}(\mathsf{input} = \mathsf{secr},\ \mathsf{custom} = \mathsf{kss.pssl},\ 2\ \text{blocks})$
28: $\mathsf{tck_{C\to S}.key} \leftarrow \mathsf{prnd}[0..63]$
29: $\mathsf{tck_{C\to S}.nonce} \leftarrow \mathsf{prnd}[64..95]$
30: Initialize cns.rxcpr as RCS with $\mathsf{tck_{C\to S}}$ in receive mode
31: $\mathsf{kss.pssr} \leftarrow \mathsf{cSHAKE512}(\mathsf{kss.pssr},\ \mathsf{tck_{C\to S}.key}[0..31])$
32: $\mathsf{prnd} \leftarrow \mathsf{cSHAKE512\_XOF}(\mathsf{input} = \mathsf{secl},\ \mathsf{custom} = \mathsf{kss.pssr},\ 2\ \text{blocks})$
33: $\mathsf{tck_{S\to C}.key} \leftarrow \mathsf{prnd}[0..63]$
34: $\mathsf{tck_{S\to C}.nonce} \leftarrow \mathsf{prnd}[64..95]$
35: Initialize cns.txcpr as RCS with $\mathsf{tck_{S\to C}}$ in transmit mode
36: $\mathsf{kss.pssl} \leftarrow \mathsf{cSHAKE512}(\mathsf{kss.pssl},\ \mathsf{tck_{S\to C}.key}[0..31])$
37: $\mathsf{cns.exflag} \leftarrow$ exchange_response
38: return none
```

Client: Establish Request And Ratchet. This function processes the server Exchange Response and completes the handshake from the client perspective. It verifies the server signature, decapsulates the ciphertext from the server to derive secr, and then computes the directional tunnel keys and initial nonces through the same cSHAKE based expansion used on the server. It initializes the RCS contexts for transmitting and receiving, updates both directional pre shared keys according to the PSK update rule, and constructs the Establish Request message. This message contains an encrypted confirmation value that binds the sequence number, timestamp, domain identity string, and session cookie hash. This function performs the full client side ratchet and symmetric state initialization.

Algorithm 5 CLIENTESTABLISHREQUESTANDRATCHET

Require: Client kex state kcs, connection state cns, server packet $\mathsf{pkt}_{\mathsf{in}}$ (exchange response)

Ensure: Outgoing packet $\mathsf{pkt}_{\mathsf{out}}$ (establish request) or error

```
 1: Parse pkt_in.pmessage as
                    $\mathsf{cptb} \parallel \mathsf{sig}$
 2: if ¬Verify(sig, hm, kcs.rverkey) then
 3:     cns.exflag ← none
 4:     return authentication_failure
 5: end if
 6: shdr ← SerializeHeader(pkt_in)
 7: hmc ← SHA3_512(shdr || cptb)
 8: if VerifyEqual(hmc, hm) $\neq 0$ then
 9:     cns.exflag ← none
10:     return verify_failure
11: end if
12: secr ← KEM.Decapsulate(cptb, kcs.deckey)
13: if decapsulation fails then
14:     cns.exflag ← none
15:     return decapsulation_failure
16: end if
17:     // Derive symmetric keys and perform PSK ratchet, client view
18: prnd ← cSHAKE512_XOF(input = kcs.secl, custom = kcs.pssr, 2 blocks)
19: $\mathsf{tck}_{C\to S}$.key ← prnd[0..63]
20: $\mathsf{tck}_{C\to S}$.nonce ← prnd[64..95]
21: Initialize cns.txcpr as RCS with $\mathsf{tck}_{C\to S}$ in transmit mode
22: kcs.pssl ← cSHAKE512(kcs.pssl, $\mathsf{tck}_{C\to S}$.key[0..31])
23: prnd ← cSHAKE512_XOF(input = secr, custom = kcs.pssl, 2 blocks)
24: $\mathsf{tck}_{S\to C}$.key ← prnd[0..63]
25: $\mathsf{tck}_{S\to C}$.nonce ← prnd[64..95]
26: Initialize cns.rxcpr as RCS with $\mathsf{tck}_{S\to C}$ in receive mode
27: kcs.pssr ← cSHAKE512(kcs.pssr, $\mathsf{tck}_{S\to C}$.key[0..31])
28:     // Build establish request with encrypted cookie
29: mtype ← establish_request
30: mlen ← KEX_ESTABLISH_REQUEST_MESSAGE_SIZE
31: HeaderCreate(pkt_out, mtype, cns.txseq, mlen)
32: shdr′ ← SerializeHeader(pkt_out)
33: Configure RCS associated data: RCS.SetAD(cns.txcpr, shdr′)
34: st ← SeqTime(pkt_out.sequence, pkt_out.utctime)
35: sch ← SHA3_512(st || DKTP_DOMAIN_IDENTITY_STRING || kcs.schash)
36: ciphertext ← RCS.Encrypt(cns.txcpr, sch)
37: Write ciphertext into pkt_out.pmessage
38: cns.exflag ← establish_request
39: return none
```

Server: Establish Response. This function receives the Establish Request, decrypts it under the newly derived RCS receive context, and verifies the confirmation hash against the expected sequence time value and session cookie. If the values match, it constructs the Establish Response by computing the server confirmation hash, encrypting it under the transmit RCS context, and returning it to the client. This step proves that both sides agree on all asymmetric and symmetric handshake material.

Algorithm 6 SERVERESTABLISHRESPONSE

```
Require: Server kex state kss, connection state cns, client packet $\mathrm{pkt_{in}}$ (establish request)
Ensure: Outgoing packet $\mathrm{pkt_{out}}$ (establish response) or error
 1: shdr ← SerializeHeader($\mathrm{pkt_{in}}$)
 2: RCS.SetAD(cns.rxcpr, shdr)
 3: hm ← RCS.Decrypt(cns.rxcpr, $\mathrm{pkt_{in}}$.pmessage)
 4: if RCS authentication fails then
 5:     cns.exflag ← none
 6:     return decryption_failure
 7: end if
 8: st ← SeqTime($\mathrm{pkt_{in}}$.sequence, $\mathrm{pkt_{in}}$.utctime)
 9: sch ← SHA3_512(st || DKTP_DOMAIN_IDENTITY_STRING || kss.schash)
10: if VerifyEqual(hm, sch) $\neq 0$ then
11:     cns.exflag ← none
12:     return verify_failure
13: end if
14: mtype ← establish_response
15: mlen ← KEX_ESTABLISH_RESPONSE_MESSAGE_SIZE
16: HeaderCreate($\mathrm{pkt_{out}}$, mtype, cns.txseq, mlen)
17: shdr′ ← SerializeHeader($\mathrm{pkt_{out}}$)
18: st′ ← SeqTime($\mathrm{pkt_{out}}$.sequence, $\mathrm{pkt_{out}}$.utctime)
19: sch′ ← SHA3_512(st′ || DKTP_DOMAIN_IDENTITY_STRING || kss.schash)
20: RCS.SetAD(cns.txcpr, shdr′)
21: ciphertext ← RCS.Encrypt(cns.txcpr, sch′)
22: Write ciphertext into $\mathrm{pkt_{out}}$.pmessage
23: cns.exflag ← session_established
24: return none
```

Client: Establish Verify. This function processes the Establish Response and verifies that the server confirmation hash matches the client expected value. If the decryption and hash comparison succeed, the session is fully established and the tunnel keys, nonces, PSKs, and RCS contexts are active. This completes the handshake and transitions the client into the established tunnel state.

Algorithm 7 ClientEstablishVerify

```
Require: Client kex state kcs, connection state cns, server packet $\mathsf{pkt_{in}}$ (establish response)
Ensure: Success or error
 1: shdr ← $\mathsf{SerializeHeader(pkt_{in})}$
 2: $\mathsf{RCS.SetAD(cns.rxcpr, shdr)}$
 3: hm ← $\mathsf{RCS.Decrypt(cns.rxcpr, pkt_{in}.pmessage)}$
 4: if RCS authentication fails then
 5:     return decryption_failure
 6: end if
 7: st ← $\mathsf{SeqTime(pkt_{in}.sequence, pkt_{in}.utctime)}$
 8: sch ← $\mathsf{SHA3\_512}\big(\mathsf{st} \parallel \texttt{DKTP\_DOMAIN\_IDENTITY\_STRING} \parallel \mathsf{kcs.schash}\big)$
 9: if $\mathsf{VerifyEqual(hm, sch)} \neq 0$ then
10:     return verify_failure
11: end if
12: cns.exflag ← session_established
13: return none
```

5 Security Definitions

This section defines the security goals for DKTP in terms of oracle based games between a challenger and an adversary. These games reflect the exact structure of the DKTP handshake and tunnel, including the cookie based binding mechanism, the directional shared secrets from the KEM, the mandatory directional pre-shared keys, and the RCS based authenticated encryption used for transport packets.
All definitions apply to protocol sessions created by calls to the **Send** oracle. A session is denoted by $(\mathsf{P}, \mathsf{sid})$ where P is a party and sid is a locally unique session identifier.

5.1 Partnering

Two sessions $(\mathsf{P}, \mathsf{sid})$ and $(\mathsf{Q}, \mathsf{sid}')$ are partners if:

- each session identifies the other as its intended peer,
- both sessions contain the same session cookie,
- both sessions have verified the signature or encrypted confirmation required by the handshake,
- both sessions complete without error and derive matching directional shared secrets and tunnel keys for their respective directions.

Partnering ensures that both parties agree on:

- configuration parameters,
- verification keys,
- the session cookie,
- directional shared secrets secl and secr,
- directional pre-shared keys after update,
- directional tunnel keys tckl and tckr.

5.2 Freshness

A session is fresh when its key material has not been trivially exposed. The exact conditions differ by game.

Freshness for authentication. A session is fresh for authentication if the long term signature secret key of the peer being authenticated has not been revealed at any time prior to completion of the session.

Freshness for handshake confidentiality. A completed session $(\mathrm{P}, \mathsf{sid})$ is fresh if:

- neither directional tunnel key tckl or tckr for this session has been revealed,
- the ephemeral KEM private keys for this session have not been revealed,
- the adversary has not modified or injected a different cookie into the session,
- the long term signing key of the partner was not corrupted prior to completion.

Freshness for transport confidentiality and integrity. A transport session is fresh if:

- neither tunnel key tckl nor tckr has been revealed,
- the sequence number for the ciphertext has not been used before,
- the corresponding PSK for the direction has not been revealed (which the model forbids entirely).

Freshness for ratchet and PCS. A session is fresh for the ratchet or PCS games if:

- the adversary may corrupt long term keys prior to the handshake but
- must not reveal the new directional tunnel keys generated by the handshake for which the Test query is issued.

5.3 Authentication Game

Authentication captures the requirement that no adversary may cause a party to complete a handshake with a peer that did not participate.

Game structure.

- The adversary interacts with the protocol through **Send**, **Corrupt**, and **RevealEPH**.
- The adversary attempts to cause an honest session $(\mathrm{P}, \mathsf{sid})$ to complete and accept peer identity Q.
- The adversary succeeds if no session at Q partners with $(\mathrm{P}, \mathsf{sid})$.

The adversary's advantage is:

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{auth}}(\lambda) = \Pr[\mathcal{A} \text{ wins the authentication game}].$$

This captures impersonation resistance and reliance on the cookie and signature mechanisms.

5.4 Handshake Confidentiality Game

This game models confidentiality of the initial tunnel keys derived during the handshake.

Game structure.

- The adversary runs the protocol and chooses a fresh completed session $(\mathrm{P}, \mathsf{sid})$.
- The adversary queries **Test**$(\mathrm{P}, \mathsf{sid})$.
- The challenger flips a bit b.
- If $\mathrm{b} = 0$, the challenger returns the real directional tunnel key.
- If $\mathrm{b} = 1$, it returns a random key of the same length.
- The adversary continues interacting with all oracles except **RevealTunnelKey** for that session.
- The adversary outputs a guess b'.

The advantage is:

$$\mathsf{Adv}^{\mathsf{conf}}_{\mathcal{A}}(\lambda) = \left|\Pr[\mathrm{b}' = \mathrm{b}] - \tfrac{1}{2}\right|.$$

5.5 Ciphertext Integrity Game

This game captures the integrity of RCS encrypted transport packets.

Game structure.

- The adversary may send arbitrary ciphertexts to honest receivers.
- The adversary may observe encryptions produced by honest parties through **Send**.
- The adversary succeeds if it causes an honest receiver to accept a ciphertext that was not produced by the sender under the same sequence number and header.

The advantage is:

$$\mathsf{Adv}^{\mathsf{int}}_{\mathcal{A}}(\lambda) = \Pr[\mathcal{A} \text{ produces a valid forgery}].$$

5.6 Replay and Reordering Game

This models protection against replay and out of order delivery.

Game structure.

- The adversary may deliver any encrypted packet with any header.
- Let $\mathsf{seq}_{\max}$ denote the highest accepted sequence number in that direction.
- The adversary wins if it causes acceptance of a ciphertext with sequence number $\mathsf{seq} \leq \mathsf{seq}_{\max}$.

The advantage is:

$$\mathsf{Adv}^{\mathsf{replay}}_{\mathcal{A}}(\lambda) = \Pr[\mathcal{A} \text{ replays or reorders a packet successfully}].$$

5.7 Forward Secrecy Game

Forward secrecy ensures that compromise of long term secret keys after the handshake does not reveal past symmetric tunnel keys.

Game structure.

- The adversary interacts with the protocol normally.
- The adversary selects a completed session $(\mathrm{P}, \mathsf{sid})$.
- After the handshake completes, the adversary obtains the long term signing keys of both parties through **Corrupt**.
- The adversary queries **Test**$(\mathrm{P}, \mathsf{sid})$ on the tunnel key.

Forward secrecy holds if:

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

is negligible.

5.8 Post Compromise Security Game

PCS models recovery from compromise when a new handshake occurs.

Game structure.

- The adversary learns long term signature keys or previous PSKs through **Corrupt** or prior compromise.
- A new handshake is executed.
- The adversary queries **Test** on the tunnel key derived from the new handshake.

- The adversary wins by distinguishing real from random.

Directional PSK updates and new KEM shared secrets support recovery.

5.9 Ratchet Game

The ratchet game models the one way evolution of tunnel keys and PSKs between handshake epochs.

Game structure.

- The adversary may reveal tunnel keys or ephemeral secrets from any ratchet step i.
- The adversary then requests **Test** on the tunnel key at step i + 1.
- The challenger returns either the real or a random key.
- The adversary outputs a guess.

The advantage is:

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

A successful violation would contradict the pseudo-randomness of the KDF and the independence of the new KEM shared secrets.

6 Security Proofs

This section presents formal reduction based proofs for the security goals defined in the previous chapter. All proofs reflect the exact structure of DKTP as described in the specification and implemented in the reference code. The cookie based transcript binding mechanism, mandatory directional pre-shared keys, directional KEM shared secrets, and 512-bit tunnel keys appear explicitly throughout.

6.1 Transcript Integrity Lemma (Including PSK Inputs)

Lemma 6.1 (Transcript Integrity). *If an adversary causes an honest session to accept a handshake in which any field of messages* M_1 *through* M_4 *has been modified, reordered, or replaced, then the adversary either breaks the EUF CMA security of the signature scheme or the INT CTXT security of the RCS encryption used in messages* M_3 *and* M_4.

Proof. The session cookie is computed as

$$\mathsf{cookie} = \mathsf{Hash}_{\mathrm{DKTP\ Cookie}}(\mathsf{config} \parallel \mathsf{vk}_{\mathrm{I}} \parallel \mathsf{vk}_{\mathrm{R}})$$

and appears in cleartext in M_1, authenticated by the responder in M_2 through a signature, and encrypted and authenticated in M_3 and M_4.
If the adversary alters M_1 or M_2, the cookie used by the responder will not match the cookie held by the initiator. The signature σ_{R} covers $\mathrm{pk}_{\mathrm{R}}^{\mathrm{kem}} \parallel \mathsf{cookie}$, so any forged pair implies a signature forgery.
If the adversary alters the encrypted cookie in M_3 or M_4, the modification will be detected by the RCS decryption check. A valid ciphertext that was not produced by the honest peer would constitute an INT CTXT forgery.
Thus, any modification to any handshake message leads to either a signature forgery or an RCS ciphertext forgery. □

6.2 Authentication Proof

Theorem 6.1 (Mutual Authentication). *If the post quantum signature scheme used in DKTP is EUF CMA secure and the cookie binding mechanism functions correctly, then no efficient adversary can cause an honest party to complete a handshake that identifies a peer who did not participate.*

Proof. Assume an adversary $\mathcal{A}$ causes an honest session at party P to complete and accept peer identity Q even though Q never participated.
In M_2, the responder Q must emit a signature

$$\sigma_{\mathrm{Q}} = \mathsf{SIG.Sign}(\mathsf{sk}_{\mathrm{Q}}, \mathrm{pk}_{\mathrm{Q}}^{\mathrm{kem}} \parallel \mathsf{cookie}).$$

If Q did not participate, then $\mathcal{A}$ must have created $(\mathrm{pk}_{\mathrm{Q}}^{\mathrm{kem}}, \sigma_{\mathrm{Q}})$ without access to sk_{Q}.
The cookie is public but fixed, so $\mathcal{B}$ can forward $(\mathrm{pk}_{\mathrm{Q}}^{\mathrm{kem}} \parallel \mathsf{cookie})$ to the EUF CMA challenger as the message. A successful impersonation corresponds to a valid forgery under vk_{Q}.
Therefore,

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{auth}}(\lambda) \leq \mathsf{Adv}_{\mathcal{B}}^{\mathsf{SIG}}(\lambda),$$

which is negligible. □

6.3 Key Indistinguishability Proof with PSK

Theorem 6.2 (Directional Tunnel Key Indistinguishability). *Assume the DKTP KEM is IND-CCA secure and the DKTP KDF is pseudo-random when keyed by* *a secret input unknown to the adversary. Then, for any fresh session in which the adversary does not learn the relevant KEM shared secret or the relevant pre-shared secret prior to derivation, the derived tunnel channel keys* (tckl, tckr) *are computationally indistinguishable from uniform.*

Proof. Fix a fresh session. DKTP derives the local transmit channel key tckl and nonce nl as:

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

and derives the local receive channel key tckr and nonce nr as:

$$(\text{tckr}, \text{nr}) \leftarrow \mathsf{KDF}_{\text{DKTP}}(\mathsf{secr}, \mathsf{pssl}).$$

We analyze tckl; the analysis for tckr is analogous.

By IND-CCA security of the KEM, the shared secret secl produced by encapsulation is indistinguishable from uniform to any adversary not in possession of the corresponding ephemeral decapsulation key during the handshake. Conditioned on freshness, the adversary does not obtain secl.

Given that pssr is a pre-shared secret unknown to the adversary in the fresh session, the KDF input contains at least one secret component (in fact, two in the intended setting). Under the assumption that $\mathsf{KDF}_{\text{DKTP}}$ is pseudo-random when keyed by such a secret input, the output tckl is computationally indistinguishable from uniform.

Directional separation follows because tckl and tckr are derived from different KEM secrets and different pre-shared secrets. Therefore compromise or distinguishability in one direction does not imply compromise in the other direction.

Hence the adversary's advantage in distinguishing the derived channel keys from random is negligible. □

6.4 Ciphertext Integrity Proof

Theorem 6.3 (Ciphertext Integrity). *If the RCS authenticated encryption scheme is INT CTXT secure, then no efficient adversary can cause an honest receiver to accept a forged transport packet.*

Proof. A forged ciphertext must pass RCS decryption under the tunnel key tckl or tckr and the authenticated header. Let $\mathcal{B}$ simulate the protocol and forward all encryption queries to the RCS challenger. If $\mathcal{A}$ produces a valid ciphertext not created by the sender for the same header, $\mathcal{B}$ outputs it as a forgery.

Since the header is part of the associated data, any modification of sequence number, flag, length, or timestamp would invalidate the tag. Thus,

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{int}} \leq \mathsf{Adv}_{\mathcal{B}}^{\mathsf{RCS}},$$

which is negligible. □

6.5 Replay and Ordering Proof

Theorem 6.4 (Replay and Reordering Resistance). *If RCS is INT CTXT secure and sequence numbers are strictly increasing and authenticated, then replayed or reordered packets cannot be accepted by any honest recipient.*

Proof. Let $\mathsf{seq}_{\mathrm{max}}$ be the highest sequence number accepted in a given direction. The receiver rejects any ciphertext with sequence number $\mathsf{seq} \leq \mathsf{seq}_{\mathrm{max}}$. If a stale packet was never produced by the sender, it is a ciphertext forgery, violating INT CTXT. If it was produced earlier, the replay will be rejected based on sequence number. Thus, acceptance of a stale packet contradicts either the authenticated header check or ciphertext integrity. □

6.6 Forward Secrecy

Theorem 6.5 (Forward Secrecy). *If the KEM is IND-CCA secure then compromise of long term signing keys after the handshake does not reveal directional tunnel keys derived in the completed session.*

Proof. Forward secrecy depends on the secrecy of secl and secr, which are erased immediately after use. Long term signing keys do not participate in the derivation of these values. A post handshake corruption of sk_{I} or sk_{R} gives no information about the ephemeral KEM secrets, which are required inputs to the KDF producing the tunnel keys.
Thus the adversary cannot reconstruct the tunnel keys after learning static signing keys. □

6.7 Post Compromise Security

Theorem 6.6 (Post Compromise Security). *If DKTP performs a new handshake with fresh KEM key pairs and updates directional PSKs, then compromise of long term keys or previous tunnel keys does not compromise the new tunnel keys.*

Proof. PCS relies on two factors:

- new KEM shared secrets secl_{i+1} and secr_{i+1},
- new directional PSKs updated from $\mathrm{tckl}_{\mathrm{i}}$ and $\mathrm{tckr}_{\mathrm{i}}$.

Even if the adversary knows all long term keys and previous tunnel keys, it cannot compute the new directional root keys without the new KEM shared secrets. These are protected by IND-CCA security. The cookie ensures that the handshake is not downgraded or redirected.
Thus the new tunnel keys remain confidential. □

6.8 Ratchet Security

Theorem 6.7 (Ratchet Security). *If the KDF is pseudo-random in the quantum random oracle model, then revealing a tunnel key at ratchet step* i *does not allow the adversary to distinguish the tunnel key at step* $i + 1$ *from random.*

Proof. The ratchet update uses fresh KEM shared secrets and updated PSKs:

$$K_{\mathrm{root},i+1} = \mathsf{KDF}(\mathsf{secl}_{i+1} \parallel \mathsf{psk}_{i+1} \parallel \mathsf{cookie}_{i+1}).$$

The adversary may reveal tck_i but this does not reveal secl_{i+1} or psk_{i+1}. The new tunnel key

$$\mathrm{tck}_{i+1} = \mathsf{KDF}_{\mathrm{tunnel}}(K_{\mathrm{root},i+1})$$

is pseudo-random under the QROM.
Any distinguisher breaks the KDF pseudo-randomness. □

6.9 Composition Theorem

Theorem 6.8 (Composition). *Under the assumptions that the KEM is IND-CCA secure, the signature scheme is EUF CMA secure, the RCS AEAD is secure, and the KDF is pseudo-random in the quantum random oracle model, DKTP achieves authentication, handshake confidentiality, ciphertext integrity, replay resistance, forward secrecy, post compromise security, and ratchet security.*

Proof. All components are composed in a black box manner. The authenticated handshake builds on signature security and cookie binding. Tunnel confidentiality and integrity follow from the KEM and RCS properties. Directional separation and PSK mixing produce independent channels. The ratchet provides state evolution and recovery. Summing the negligible advantages yields an overall negligible bound. □

7 Cryptanalysis

This section provides a detailed adversarial analysis of DKTP viewed from the perspective of a capable network level attacker. The goal is to identify all meaningful attack surfaces, characterize the power of adversaries allowed by the formal model, explain why attacks fail under the assumptions on the cryptographic primitives, and quantify the contribution of pre-shared keys, cookies, and ratcheting to the overall security picture. Every vector considered here is derived from or explicitly addressed in the DKTP specification and is evaluated against the formal reductions proven earlier.

7.1 Attack Surfaces in DKTP

DKTP exposes several potential attack surfaces that a network attacker may attempt to exploit. These can be grouped according to the stage of the protocol.

Handshake surfaces.

- Modification of fields in Connect Request or Connect Response.
- Injection of false ephemeral KEM public keys.
- Replacement of KEM ciphertexts.
- Replay of old handshake messages.
- Substitution of a different session cookie.

The signature on $\mathrm{pk}_R^{\mathrm{kem}} \parallel \mathsf{cookie}$ prevents impersonation. The cookie binds M1 through M4 to the correct session configuration and identity. The encrypted cookie in M3 and the encrypted confirmation in M4 prevent manipulation of intermediate handshake state.

Key schedule surfaces.

- Attempt to compute directional root keys without both the shared secret and the correct PSK.
- Attempt to recompute tunnel keys without the KDF output.
- Attempt to derive the next PSK value without access to the corresponding tunnel key.

These fail because the KDF treats each direction separately, applies domain separation through independent secret inputs, and mixes the directional KEM shared secrets with the persisted peering secrets (pssl, pssr).

Transport surfaces.

- Attempting to forge RCS ciphertexts.
- Attempting replay attacks by resending old packets.
- Attempting out of order delivery.
- Attempting to guess the header or tamper with associated data fields.

Transport packets use RCS authenticated encryption with associated data that includes the entire 21 byte header. Nonces never repeat because they are derived from strictly increasing sequence numbers.

Post compromise surfaces.

- Attacks following compromise of static keys.
- Attacks following compromise of previous tunnel keys.
- Attempts to exploit downgraded configuration values.

A new handshake with fresh KEM keys and new PSKs reestablishes full strength security.

7.2 Adversarial Capabilities and Limitations

The adversary controls the network completely. It may reorder, inject, replay, delay, or drop messages. It may interact through the following oracles:

- **Send**: deliver handshake or transport messages and obtain the peer's response.
- **CorruptSig**: reveal long term signing keys.
- **RevealEPH**: reveal ephemeral KEM private keys for a chosen session.
- **RevealPSK**: reveal persisted peering secrets pssl and/or pssr at a chosen time.
- **RevealTunnelKey**: reveal directional tunnel keys for a chosen established session.
- **Test**: challenge the indistinguishability of session keys for a fresh session.

Adversarial limitations stem from the cryptographic assumptions:

- it cannot compute secl or secr for a fresh session without the corresponding ephemeral KEM private key,

- it cannot forge valid handshake signatures over the authenticated transcript hash and serialized header fields,
- it cannot compute DKTP KDF outputs without the required secret inputs for that derivation,
- it cannot modify or forge RCS ciphertexts except with negligible probability under the AEAD security bound,
- it cannot induce nonce reuse within a direction when sequence numbers are enforced and authenticated as associated data.

7.3 Effects of PSK Compromise

This subsection analyzes compromise of the pre-shared secrets stored in the peering key structures:

$$\mathsf{pssl}\text{ (local pre-shared secret)}, \qquad \mathsf{pssr}\text{ (remote pre-shared secret)}.$$

Effect on channel key derivation. DKTP derives channel keys as:

$$(\mathrm{tckl}, \mathrm{nl}) \leftarrow \mathsf{KDF}_{\mathrm{DKTP}}(\mathsf{secl}, \mathsf{pssr}), \qquad (\mathrm{tckr}, \mathrm{nr}) \leftarrow \mathsf{KDF}_{\mathrm{DKTP}}(\mathsf{secr}, \mathsf{pssl}).$$

If an adversary compromises pssl and/or pssr, the adversary removes one secret input from the KDF. Security then depends primarily on the confidentiality of the corresponding KEM shared secret for that direction.

PSK update and persistence. After establishment confirmation, DKTP updates:

$$\mathsf{pssl} \leftarrow \mathsf{Hash}(\mathsf{pssl} \parallel \mathrm{tckl}), \qquad \mathsf{pssr} \leftarrow \mathsf{Hash}(\mathsf{pssr} \parallel \mathrm{tckr}).$$

Therefore, compromise of an old pre-shared secret does not automatically imply compromise of its updated successor after a fresh handshake in which the updated state is computed and persisted.

Interaction with long-term key compromise. Compromise of long-term signature keys enables impersonation at the message authentication layer, but does not by itself reconstruct past channel keys without breaking the KEM secrecy for the corresponding sessions and directions.
This analysis assumes the adversary may compromise stored pre-shared secrets at arbitrary times; the security games should interpret such compromise as removal of the corresponding secret KDF input for sessions after the compromise event.

7.4 KCI and UKS Resistance

Key compromise impersonation resistance. If the adversary compromises sk_I, it cannot impersonate R to I because impersonation requires knowledge of $\mathsf{psk}_{R \to I}$. The cookie binds the configuration and identities, so the attacker cannot introduce a false identity without detection.

Unknown key share resistance. The attacker cannot cause I and R to complete handshakes that share a key but have different peer identities. The cookie includes both verification keys, so the peers must independently compute identical cookies to complete the handshake. A manipulated handshake would violate the transcript integrity lemma.

7.5 Full Analysis of AEAD and Nonce Construction

DKTP uses RCS AEAD for all encrypted handshake messages after M2 and for all transport packets.

Nonce derivation. Nonces are derived from:

$$\mathrm{nonce} = \mathrm{encode}(\mathrm{seq}),$$

where seq is an eight byte strictly increasing counter.
This ensures:

- nonce uniqueness in each direction,
- no possibility of nonce reuse,
- authenticated ordering because the header is part of AEAD associated data.

Associated data. The associated data includes:

$$\mathrm{flag} \parallel \mathrm{seq} \parallel \mathrm{len} \parallel \mathrm{timestamp}.$$

These fields cannot be modified without invalidating the authentication tag.

Integrity consequences. Any tampering of:

- sequence number,
- length,

- timestamp,
- direction flag,
- ciphertext body,

yields rejection under the INT CTXT property of RCS.

7.6 Quantum Random Oracle Considerations

The KDF and cookie hash rely on Keccak based constructions. Keccak is modeled as a quantum random oracle in the formal analysis. Under this model:

- adversarial superposition queries do not reveal input structure,
- the domain separation labels prevent cross protocol interference,
- KDF outputs are pseudo-random even under quantum adversaries,
- collisions on the cookie hash remain negligible.

The security of directional root keys depends on the pseudo-randomness of KDF outputs given secl, psk, and cookie. All three inputs remain unknown to the adversary in fresh sessions.

7.7 Summary of Adversarial Bounds

Combining all reductions and attack surface analyses:

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{auth}} \le \mathsf{Adv}^{\mathsf{SIG}} + \varepsilon,$$
$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{conf}} \le \mathsf{Adv}^{\mathsf{KEM}} + \mathsf{Adv}^{\mathsf{KDF}} + \varepsilon,$$
$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{int}} \le \mathsf{Adv}^{\mathsf{RCS}} + \varepsilon,$$
$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{ratchet}} \le \mathsf{Adv}^{\mathsf{KDF}} + \mathsf{Adv}^{\mathsf{KEM}} + \varepsilon,$$
$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{fs}} \le \mathsf{Adv}^{\mathsf{KEM}} + \varepsilon,$$
$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{pcs}} \le \mathsf{Adv}^{\mathsf{KEM}} + \mathsf{Adv}^{\mathsf{KDF}} + \varepsilon.$$

Directional 512-bit tunnel keys provide a symmetric security level far beyond NIST Category 5. Even quantum adversaries gain no meaningful advantage due to the structure of the KDF, the mandatory use of separate PSKs, and the use of independent KEM shared secrets.

8 Implementation Conformance

This section verifies that the DKTP reference implementation conforms to the formal model presented in the previous chapters. The comparison is performed by examining the source code files that implement the handshake, key schedule, KDF inputs, packet layout, and ratchet update. The goal is to ensure that the operations described in the formal protocol specification are implemented exactly and that no discrepancies exist that would invalidate the proofs or reduce security.

8.1 Mapping Between Model and Code

The handshake logic in the implementation matches the formal protocol description.

Message M1. The Connect Request includes the initiator verification key, configuration identifier, and session cookie. The cookie is computed from the configuration and both verification keys using the Keccak based hash with the correct label. Serialization order matches the formal definition.

Message M2. The Connect Response contains the responder ephemeral KEM public key and a signature covering $(\mathrm{pk}_R^{\mathrm{kem}} \parallel \mathsf{cookie})$ exactly as described in the specification. The cookie in M2 is copied verbatim from the M1 input, and the implementation checks that it matches the expected value.

Message M3. The Exchange message includes the KEM ciphertext and an encrypted copy of the cookie. The implementation derives the provisional encryption key exactly as specified by using the directional shared secret and directional PSK prior to root key derivation.

Message M4. The Verification message contains encrypted confirmation of handshake completion. This matches the formal model where M4 proves the responder derived the same handshake state.

Key schedule. The implementation constructs the root keys using

$$\mathrm{K}_{\mathrm{root}}^{\mathrm{I}\to\mathrm{R}} = \mathsf{KDF}(\mathsf{secl} \parallel \mathsf{psk}_{\mathrm{I}\to\mathrm{R}} \parallel \mathsf{cookie}),$$

and symmetrically for the other direction. The same domain separation labels appear in the implementation. The 512-bit tunnel keys and initial nonces are derived exactly as in the formal specification.

Transport phase. The header fields (flag, sequence, length, timestamp) appear in the exact order and with the exact sizes listed in the specification. The header is passed to RCS as authenticated associated data. The ciphertext is generated using the appropriate directional tunnel key and nonce. The receiver performs the exact same RCS call and checks that the header is authenticated.

State machine. The implementation follows the same state transitions as the formal model: Idle, Handshake, Tunnel Established, Ratchet Ready. Messages out of order cause immediate failure.

8.2 Algorithm Agility in dktp.h

Algorithm agility is encoded directly in the implementation through compile time configuration macros in `dktp.h`. The implementation supports multiple KEM and signature schemes through the following mechanisms.

Configurable KEM. The KEM key generation and decapsulation functions are accessed through wrappers whose implementation depends on the selected KEM family. The formal model treats the KEM abstractly, and the implementation reflects this abstraction correctly.

Configurable signature scheme. The signature functions are selected by configuration. The implementation can choose Dilithium, SPHINCS+, or other supported post quantum signature systems. Key serialization and verification match the exact interface assumed by the formal model.

Invariant protocol structure. Changing the algorithm choices does not change the message formats or state transitions. This property aligns with the protocol description and ensures that the proofs apply for any approved parameter sets.

8.3 Constant Time Behavior

The implementation uses constant time cryptographic operations for all operations that depend on secret data.

- KEM decapsulation uses constant time implementations provided by the vendor library.
- Signature verification uses constant time routines.

- RCS encryption and decryption avoid data dependent branches and table lookups.
- The comparison of authentication tags and cookies uses constant time memory comparison functions.

The implementation contains no branches or memory access patterns that depend on tunnel keys, KEM secrets, PSKs, or any intermediate KDF outputs. This conforms to the assumptions used in the security proofs.

8.4 RNG Requirements and Safety

DKTP requires a cryptographically secure pseudo-random number generator. The implementation uses the system CSPRNG and has the following properties:

- The RNG is invoked to generate ephemeral KEM key pairs.
- Nonces for handshake encryption are derived from KDF outputs and not from the RNG, ensuring determinism and uniqueness.
- Transport nonces are encoded sequence numbers, not RNG output, ensuring uniqueness in the tunnel.
- RNG failures are detected before the protocol continues.

The security proofs assume correct operation of the CSPRNG only for ephemeral KEM key generation.

8.5 Key and State Erasure

The implementation erases sensitive material at the appropriate times:

- ephemeral KEM private keys are zeroed immediately after use,
- directional shared secrets secl and secr are erased after root key derivation,
- tunnel keys are overwritten when ratcheting produces new ones,
- updated PSKs replace old PSKs and overwrite their previous values,
- intermediate buffers used during RCS encryption and decryption are cleared.

The erasure functions used are resistant to compiler optimization removal and match the assumptions required to prevent key recovery by memory inspection.

8.6 Validation of Sequence Number and Nonce Handling

The implementation correctly enforces all sequence and nonce properties required by the formal model.

Sequence numbers. Each direction maintains an unsigned 64 bit sequence counter. The receiver tracks the highest accepted sequence number $\mathrm{seq}_{\max}$ and rejects packets with sequence numbers less than or equal to it. The implementation halts the session on sequence overflow.

Nonces. Nonces are encoded sequence numbers. Since sequences never repeat and always increase, nonces never repeat. This matches the formal model and the assumptions used in the AEAD security reductions.

Associated data. The entire header (flag, sequence, length, timestamp) is passed as associated data to RCS. Any modification breaks the authentication tag, preventing replay and reordering attacks.
This completes the implementation conformance analysis and confirms that the DKTP implementation matches the formal model and specification in all cryptographically relevant details.

9 Concrete Security Estimates

This section provides concrete security estimates for DKTP based on the parameter sets commonly selected for the KEM and signature schemes, the strength added by mandatory directional pre-shared keys, and the 512-bit symmetric tunnel keys derived at the end of the handshake. Estimates rely on NIST post quantum security categories and the theoretical bounds of symmetric and asymmetric cryptographic primitives.

9.1 Security Levels for Select KEM and Signature Choices

DKTP is designed to operate at a consistent 512-bit symmetric security level for all active secure channels. This security level is achieved through the use of 512-bit directional tunnel keys and cSHAKE 512 based key derivation. The authenticated encryption used in the tunnel relies exclusively on RCS 512, which is constructed to match this symmetric strength. The symmetric layer therefore determines the overall channel security and provides a margin significantly above the highest NIST post quantum category.

The asymmetric components of DKTP, namely the key encapsulation mechanism and the digital signature system, are selected through compile time configuration. The protocol supports a variety of post quantum families, including lattice based, code based, and hash based systems. To align with the 512-bit symmetric target, DKTP requires that implementations use only the highest strength asymmetric parameter sets. These provide approximately 256-bit post quantum security, which is then combined with mandatory directional pre shared keys and the 512-bit symmetric layer.
The table below lists the recommended parameter sets and their approximate classical and quantum security strengths. Only the highest category asymmetric sets are shown, since lower ones do not match the intended strength of DKTP.

Primitive	**Parameter Set**	**Type**	**Security Bits**
Kyber	Kyber 1024	KEM	≈ 256
Classic McEliece	mceliece 8192128	KEM	≈ 256
Dilithium	Dilithium V	Signature	≈ 256
SPHINCS+	SPHINCS+ 256f	Signature	≈ 256
RCS	RCS 512	AEAD	512
cSHAKE	cSHAKE 512	KDF	512
SHA3	SHA3 512	Hash	512

Table 1: Recommended post quantum primitives for DKTP. Asymmetric primitives use only their highest category parameter sets, and symmetric components operate exclusively at the 512-bit security level.

In summary, DKTP combines 256-bit post quantum asymmetric primitives with a 512-bit symmetric core. The tunnel keys, nonce derivation, PSK update mechanism, confirmation hashes, and all authenticated encryption rely on 512-bit cSHAKE or RCS. The result is a secure channel construction whose symmetric strength noticeably exceeds NIST Category 5 and ensures long term resistance against quantum capable adversaries.

9.2 Impact of PSK Strength on Overall Security

DKTP uses two persisted pre-shared secrets in the peering key structures:

$$\mathsf{pssl} \text{ (local pre-shared secret)}, \qquad \mathsf{pssr} \text{ (remote pre-shared secret)}.$$

These values are mixed with the directional KEM shared secrets during channel key derivation:

$$(\text{tckl}, \text{nl}) \leftarrow \mathsf{KDF}_{\text{DKTP}}(\mathsf{secl}, \mathsf{pssr}), \qquad (\text{tckr}, \text{nr}) \leftarrow \mathsf{KDF}_{\text{DKTP}}(\mathsf{secr}, \mathsf{pssl}).$$

Security impact. Stronger pre-shared secrets improve robustness against partial compromise scenarios by ensuring that channel key derivation retains secret entropy even if one source is weakened. Conversely, weak or predictable pre-shared secrets reduce this benefit and shift effective security toward reliance on the secrecy of the KEM shared secrets alone.

Update rule and persistence. After establishment confirmation, DKTP updates persisted peering secrets:

$$\mathsf{pssl} \leftarrow \mathsf{Hash}(\mathsf{pssl} \,\|\, \mathsf{tckl}), \qquad \mathsf{pssr} \leftarrow \mathsf{Hash}(\mathsf{pssr} \,\|\, \mathsf{tckr}).$$

The updated values are saved and seed subsequent handshakes.

9.3 Security Margins and Comparison to NIST Categories

NIST defines five categories of post quantum security strength, with Category 5 corresponding to approximately 256-bits of security against classical attacks and 128-bits against quantum Grover style attacks.
In DKTP:

Asymmetric layer. The strength of the handshake depends on the selected KEM and signature scheme and typically ranges between Category 3 and Category 5.

Symmetric layer. The symmetric tunnel keys are 512-bits in each direction. Under quantum attack, Grover style search reduces symmetric security by a square root factor, leaving:

$$\text{Effective quantum symmetric security} \approx 256 \text{ bits.}$$

This exceeds NIST Category 5 requirements.

RCS AEAD. RCS uses a Rijndael based permutation and a cSHAKE based KDF and authentication layer. These have symmetric security levels aligned with the 512-bit tunnel keys and are far above Category 5.

PSK contribution. Because PSKs are included directly as KDF inputs, the symmetric strength is preserved across ratchet updates, even when asymmetric primitives are refreshed with lower category parameters.

Overall channel security. The final security level of a DKTP channel is:

- 512-bit symmetric strength when viewed against classical adversaries,
- approximately 256-bit symmetric strength against quantum adversaries,
- limited by the lower of the KEM and signature scheme during the initial handshake,
- increased after ratchet update because the PSK update is driven by 512-bit tunnel keys.

This provides a significant cushion above the highest NIST post quantum category. DKTP retains strong security margins even when conservative parameter sets are chosen for asymmetric primitives.

10 Conclusion

The Distributed Key Tunnel Protocol provides a comprehensive and high assurance design for establishing authenticated and confidential channels that remain secure against classical and quantum adversaries. This paper has presented a complete formal analysis of the protocol that aligns directly with the DKTP technical specification and the reference implementation. Every handshake field, key derivation input, directional shared secret, tunnel key, ratchet update, and transport packet format has been examined and expressed both in engineering terms and through the formal security model.

The DKTP handshake uses a session cookie to bind configuration values and verification keys into a consistent and authenticated structure. The session cookie ensures that both parties agree on the same protocol parameters and peer identities and prevents any reordering or manipulation of the handshake.

The key schedule uses two independent KEM shared secrets together with directional pre-shared keys. These values are combined with the authenticated cookie through a domain separated KDF to produce directional root keys, tunnel keys, and initial nonces. The resulting tunnel keys are 512-bit symmetric keys that provide long term protection well above the highest NIST post quantum category. This provides strong security margins even when the asymmetric primitives are selected from parameter sets that offer lower categories of security.

The transport phase uses RCS-512 authenticated encryption with sequence based nonces, authenticated headers, and strict replay and ordering rules. These features guarantee confidentiality, integrity, uniqueness of nonces, and resistance to packet manipulation. The ratchet mechanism updates directional

PSKs and tunnel keys after each handshake and allows the protocol to recover security even after compromise of long term keys or previous symmetric keys.
The security proofs show that DKTP achieves mutual authentication, handshake confidentiality, ciphertext integrity, replay resistance, forward secrecy, post compromise security, and ratchet security. Each proof reduces one of these goals to the assumptions of the underlying KEM, signature scheme, KDF, and RCS authenticated encryption. The cookie based transcript binding and mandatory directional PSKs play central roles in these reductions.
The implementation analysis confirms that the code follows the specification and the formal model exactly. All cryptographic operations are invoked in constant time, all sensitive state is erased promptly, and the random number generator is used only for ephemeral KEM keys. The header structure, nonce encoding, sequence number checks, and packet layout match the formal description bit for bit.
Taken together, these results demonstrate that DKTP is a robust, well structured, and conservatively designed secure channel protocol that provides strong protections against both classical and quantum adversaries. Its combination of rigid authenticity, strong symmetric encryption, algorithm agility, and ratchet based state evolution make it a suitable foundation for high security applications that require long term confidentiality and resilience against compromise.

References

1. National Institute of Standards and Technology. *FIPS 203: Module Lattice Based Key Encapsulation Mechanism Standard (ML KEM)*. U.S. Department of Commerce, 2024. Available at: https://csrc.nist.gov/pubs/fips/203/final.
2. National Institute of Standards and Technology. *FIPS 204: Module Lattice Based Digital Signature Standard (ML DSA)*. U.S. Department of Commerce, 2024. Available at: https://csrc.nist.gov/pubs/fips/204/final.
3. Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J. M., Schwabe, P., Seiler, G., and Stehlé, D. *CRYSTALS Kyber: Algorithm Specifications and Supporting Documentation*. 2021. Available at: https://pq-crystals.org.
4. Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., and Stehlé, D. *CRYSTALS Dilithium: Algorithm Specifications and Supporting Documentation*. 2021.
5. National Institute of Standards and Technology. *FIPS 202: SHA 3 Standard, Permutation Based Hash and Extendable Output Functions*. U.S. Department of Commerce, 2015. Available at: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf.
6. Kelsey, J., and National Institute of Standards and Technology. *SP 800 185: SHA 3 Derived Functions, cSHAKE, KMAC, TupleHash, and ParallelHash*. 2016. Available at: https://doi.org/10.6028/NIST.SP.800-185.
7. Bertoni, G., Daemen, J., Peeters, M., and Van Assche, G. *The Keccak Reference*. Submission to the NIST SHA 3 Competition, 2011. Available at: https://keccak.team.
8. Bellare, M., and Namprempre, C. *Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm*. Journal of Cryptology, 2008.
9. Rogaway, P., and Shrimpton, T. *A Provable Security Treatment of the Key Wrap Problem*. Eurocrypt, 2006.
10. Marlinspike, M., and Perrin, T. *The Double Ratchet Algorithm*. Signal Foundation, 2016.
11. Perrin, T. *The Noise Protocol Framework*. Revision 34, 2022. Available at: https://noiseprotocol.org.
12. Canetti, R., and Krawczyk, H. *Analysis of Key Exchange Protocols and Their Use for Building Secure Channels*. Eurocrypt, 2001.

13. LaMacchia, B., Lauter, K., and Mityagin, A. *Stronger Security of Authenticated Key Exchange*. ProvSec, 2007.

14. Dyer, J., Chen, S., and Shrimpton, T. *On the Security of TLS 1.3 Handshake Key Schedule*. IEEE S&P Workshops, 2018.

15. Underhill, J. G. *RCS Specification*. Quantum Resistant Cryptographic Solutions Corporation, 2025. Available at: https://www.qrcscorp.ca.

16. Underhill, J. G. *Dual Key Tunnel Protocol (DKTP) Technical Specification*. Quantum Resistant Cryptographic Solutions Corporation, 2025. Available at: https://www.qrcscorp.ca.

17. QRCS Corporation. *DKTP Reference Implementation*. Source repository, 2025. Available at: https://github.com/QRCS-CORP/DKTP.