Symmetric Key Distribution Protocol – SKDP

Revision 1.1a, December 02, 2024

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

This document is an engineering level description of the SKDP encrypted and authenticated network messaging protocol. In its contents, a guide to implementing SKDP, an explanation of its design, as well as references to its component primitives and links to supporting documentation.

Contents **Page**

Foreword 2
Figures 3
Tables 4
1: Introduction 5
2: Scope 8
3: References 9
4: Terms and Definitions 10
5: Structures 11
6: Operational Overview 16
7: Formal Description 28
8: SKDP API 33
9: SKDP Cryptanalysis 44
10: Design Decisions 48

Foreword

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

This document specifies the Symmetric Key Distribution Protocol (SKDP) as an engineering-level standards proposal. It describes the protocol roles, key hierarchy, message formats, packet serialization, key-establishment sequence, authenticated encryption behavior, error handling, and implementation requirements required for interoperable implementations.

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

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

Figures

Contents **Page**

Figure 6.1: SKDP client connect request 17
Figure 6.2: SKDP server connect response 18
Figure 6.3: SKDP client exchange request 19
Figure 6.4: SKDP server exchange response 22
Figure 6.5: SKDP client establish request 24
Figure 6.6: SKDP server establish response 26
Figure 6.7: SKDP client establish verify 27

Tables

Contents **Page**

Table 5.1a: The client key structure 11
Table 5.1b: The device identity structure 11
Table 5.2: The server key structure 11
Table 5.3: The master key structure 12
Table 5.4: The client state structure 12
Table 5.5: The server state structure 13
Table 5.6: The keep alive state 13
Table 5.7: The SKDP packet structure 13
Table 5.8: Packet header flag types 14
Table 5.9: Error type messages 15
Table 8.1a SKDP configuration string. 33
Table 8.1b SKDP error strings. 33
Table 8.1c SKDP packet structure. 33
Table 8.1d SKDP master key structure 34
Table 8.1e SKDP server key structure 34
Table .1f SKDP device key structure 34
Table 8.1g SKDP keep alive state structure 34
Table 8.1h SKDP errors enumeration 35
Table 8.1i SKDP flags enumeration 35
Table 8.1j SKDP constants 37
Table 8.2 SKDP server state structure 40
Table 8.3 SKDP client state structure 42

1: Introduction

Key distribution is one of the most challenging problems in cryptography. The internet has grown at an extraordinary pace since its inception and is now a core communications medium used by billions of people around the globe. The information we send over this public medium must be secured, as the internet has become a primary tool in global commerce and a communications infrastructure connecting people everywhere.

The security mechanisms most widely used today utilize asymmetric cryptography; public/private key cryptography to establish encryption and authentication. These asymmetric primitives use 'trapdoor' functions where a difficult mathematical problem is created using a public key and solved using the private key. The problem with this approach is that the underlying mathematical problems used by these asymmetric ciphers and signature schemes are constantly being challenged by new knowledge and advances in computing technology.

What seems like an intractable problem today could eventually be reduced or even solved at some future time. This is why asymmetric parameters are continually adjusted to make the problem more difficult, and why entire orders of asymmetric cryptography based on large integer factorization and elliptic curves will soon become obsolete due to the emergence of quantum computers. It has been well established that intelligence agencies collect and store encrypted communications streams on a vast scale because, even if the technology to break these encryption technologies does not currently exist, at some future point it may, and all of that stored traffic will become readable.

We could face the same problem with Lattice-Based Encryption (LWE) cryptography in ten or twenty years that we face now with elliptic curves or large integer factorization cryptography: eventually, the technology and the mathematics will evolve, combining to create new threats capable of breaking the cryptographic system. This is further complicated by the choice of parameters used in the design of asymmetric primitives, which are calculated based on projections established only in current knowledge, in a performance-oriented field that often chooses less aggressive parameters to improve performance.

The knowledge that communications are being captured and stored while breakthroughs in technology are unpredictable creates a serious issue that must be addressed. We do not believe that any system based on asymmetric cryptography can guarantee true long-term security absolutely, which must now be considered the lifespan of a human being.

Symmetric cryptography may provide part of the solution. Given sufficiently 'strong' symmetric cryptographic primitives and longer key lengths, symmetric cryptography can be far more computationally expensive to solve and perhaps even impossible to break for an indefinite time. Systems that use pre-shared symmetric keys have traditionally faced challenges of scalability and vulnerability to a single point of failure.

For example, some systems use a single pre-shared key and session counter to key a symmetric cipher and establish an ad hoc encrypted tunnel, some SSH (Secure Shell) implementations use this scheme. The issues with this method are that if a device is ever captured, all past messages

become readable; similarly, if the server's key database is compromised, messages for all hosts on the network—past, present, and future, become instantly readable by an attacker.

SKDP is a symmetric key-distribution scheme that uses a hierarchical pre-shared derivation-key model and per-session random token exchange to create independent transmit and receive encryption states. The implementation derives fresh channel keys and nonces for each session and erases transient token material after establishment. SKDP shall not be described as providing full forward secrecy against later compromise of a long-term device or server derivation key when an attacker has also recorded the key-exchange transcript, because the encrypted token messages are protected by keys derived from those long-term derivation keys. The protocol provides per-session key separation and ephemeral channel state; long-term key compromise remains within the security boundary of the affected deployment scope.

This introduction sets the context for SKDP as a symmetric, authenticated, duplex key-establishment and message-protection protocol. Its security depends on the secrecy of the long-term derivation keys, the unpredictability of the random token generator, correct packet sequencing, timestamp validation, and successful authentication of the exchange and data-phase packet headers.

1.1 Purpose

The SKDP secure messaging protocol, utilized in conjunction with quantum secure symmetric cryptographic primitives, is used to create an encrypted and authenticated duplexed communications channel. This specification presents a secure messaging protocol that creates an encrypted communications channel, in such a way that:

1) The symmetric cipher keys for both the send and receive channels are generated independently for each session from random token material and transcript-derived session hashes. These keys are not persisted by SKDP after session teardown.
2) The capture of a device derivation key or server derivation key compromises the security scope of the corresponding key hierarchy until the affected key material is replaced or disabled by the deployment. SKDP does not implement an in-band revocation list or in-band rekey mechanism in the reviewed code.
3) That each host in the bi-directional communications stream, is responsible for creating the shared secret for the channel they transmit on.

SKDP is a duplexed communications system. It uses a separate shared secret to key both the transmit and receive channels in a communications stream. Each host is responsible for generating the symmetric key that host transmits data on. Symmetric cipher keys are ephemeral, and unique keys are generated for each session. The system works in a client/server model, where a client requests a connection from the server to initiate the key exchange. The server authenticates and encrypts a key sent to the client, and the client encrypts and authenticates a key sent to the server. These keys are used to initialize a quantum secure symmetric cipher for each channel, which encrypts the communications stream. A strong emphasis has been placed on

authentication with SKDP, with the entire key exchange using authentication to guarantee the exchange, and the symmetric stream cipher using KMAC authentication, with additional data parameters (AEAD) that authenticate the SKDP packet headers.

2: Scope

This document describes the SKDP secure messaging protocol, which is used to establish an encrypted and authenticated duplexed message stream between two hosts. This document describes the complete symmetric key exchange, authentication, and the establishment of an encrypted tunnel. This is a complete specification, describing the cryptographic primitives, the derivation functions, and the complete client to server messaging protocol.

2.1 Application

This protocol is intended for institutions that implement secure communication channels used to encrypt and authenticate secret information exchanged between remote terminals.

The key exchange functions, authentication and encryption of messages, and message exchanges between terminals defined in this document must be considered as mandatory elements in the construction of an SKDP communications stream. Components that are not necessarily mandatory, but are the recommended settings or usage of the protocol shall be denoted by the key-words **SHOULD**. In circumstances where strict conformance to implementation procedures is required but not necessarily obvious, the key-word **SHALL** will be used to indicate compulsory compliance is required to conform to the specification.

3: References

3.1 Normative References

The following documents serve as references for key components of SKDP:

3.1.1 NIST FIPS 202: SHA-3 Standard: Permutation-Based Hash and Extendable Output Functions

3.1.2 NIST SP 800-185: Derived Functions cSHAKE, KMAC, TupleHash and ParallelHash

3.1.3 NIST SP 800-90A: Recommendation for Random Number Generation

3.1.4 NIST SP 800-108: Recommendation for Key Derivation using Pseudorandom Functions

3.1.5 NIST FIPS 197 The Advanced Encryption Standard

3.1.6 NIST SP 800-38D: Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC

3.1.7 NIST SP 800-185: cSHAKE and KMAC domain-separation functions used by SKDP key derivation and authentication

3.2 Reference Links

3.2.1 The Keccak Code Package: https://github.com/XKCP/XKCP

3.2.2 NIST AES FIPS 197: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

4: Terms and Definitions

4.1 RCS

The Rijndael-256 Cryptographic Stream (RCS) AEAD authenticated symmetric stream cipher.

4.2 SHA-3

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

4.3 SHAKE

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

4.4 KMAC

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

4.5 AES-GCM

The AES-256-GCM authenticated encryption function used by the QSC AES-GCM interface when SKDP_USE_RCS_ENCRYPTION is not defined. In the implementation, the selected AEAD cipher state is maintained across the session and packet headers are supplied as associated data.

4.6 Selected AEAD Cipher

The cipher abstraction selected at compile time for SKDP packet protection. The default build uses the QSC AES-256-GCM interface. If SKDP_USE_RCS_ENCRYPTION is defined, the implementation uses the QSC RCS AEAD interface.

5: Structures

5.1 Device Key

The device key is an internal structure that stores the device derivation key, the expiration time, and the client identity array.

Parameter	Data Type	Bit Length	Function
Expiration	Uint64	64	Validity Check
CID	Uint8 array	128	Identification
DDK	Uint8 array	256/512	Derivation Key

Table 5.1a: The client key structure.

The expiration parameter is a 64-bit unsigned integer that holds the UTC seconds since the Unix epoch. This value is checked during connection establishment. If the key has expired at connection time, the connection attempt is halted and an error is returned. The implementation does not periodically re-check key expiration during an established session; deployments shall bound session lifetime or re-establish the session according to their operational policy.

The key identity array is a 16-byte array that identifies the derivation scope of the key. The implementation defines SKDP_MID_SIZE as 4 bytes, SKDP_SID_SIZE as 8 bytes, SKDP_DID_SIZE as 12 bytes, SKDP_TID_SIZE as 4 bytes, and SKDP_KID_SIZE as 16 bytes. These constants are used as identity-span values in key derivation and prefix comparison. A master key copies the first 4 bytes of the supplied identity, a server key is derived using the first 12 bytes of the supplied identity, and a device key is derived using the full 16-byte identity.

Master prefix 4 bytes	Server identity span 8 bytes	Device identity span 12 bytes cumulative	Key-set / terminal id 4 bytes

Table 5.1b: The device identity structure.

Server keys are derived from the master derivation key using cSHAKE with the protocol configuration string and the server identity span as customization input. Device keys are derived from the server derivation key using cSHAKE with the protocol configuration string and the full device identity as customization input. The key-set bytes may be used by an application as a key version or provisioning identifier, but the reviewed implementation does not provide a key-store, revocation list, or automatic key-rollover service.

5.2 Server Key

The server key is identical to the client key except for the bit length of the key identification array is ninety-six bits.

Parameter	Data Type	Bit Length	Function
Expiration	Uint64	64	Validity check
SID	Uint8 array	96	Identification
SDK	Uint8 array	256/512	Derivation Key

Table 5.2: The server key structure.

5.3 Master Key

The master key is identical to the client and branch keys except for the bit length of the key identification array is sixty-four bits.

Parameter	Data Type	Bit Length	Function
Expiration	Uint64	64	Validity check
MID	Uint8 array	64	Identification
MDK	Uint8 array	256/512	Derivation Key

Table 5.3: The master key structure.

5.4 Device State

The client state is an internal structure that contains all the variables required by the SKDP operations. This includes elements copied from the client key structure at initialization, send and receive channels symmetric cipher states, session cookies, packet counters, and flags.

Data Name	Data Type	Bit Length	Function
Expiration	Uint64	64	Validity check
DDK	Uint8 array	256/512	Derivation Key
DSH	Uint8 array	128	Session Hash
CID	Uint8 array	Variable	Identification
SSH	Uint8 array	Variable	Session Cookie
RXSEQ	Uint64	64	Packet Counter
TXSEQ	Uint64	64	Packet Counter
Cipher Receive State	Structure	Variable	Symmetric Decryption
Cipher Transmit State	Structure	Variable	Symmetric Encryption
ExFlag	Uint8	8	Protocol Check

Table 5.4: The client state structure.

5.5 Server State

The server state is identical to the client state, except for the additional server identification parameter.

Data Name	Data Type	Bit Length	Function
Expiration	Uint64	64	Validity check
SDK	Uint8 array	256/512	Derivation Key
DID	Uint8 array	128	Identification
DSH	Uint8 array	128	Session Hash
SID	Uint8 array	Variable	Identification
SSH	Uint8 array	Variable	Session Cookie
RXSEQ	Uint64	64	Packet Counter
TXSEQ	Uint64	64	Packet Counter
Cipher Receive State	Structure	Variable	Symmetric Decryption
Cipher Transmit State	Structure	Variable	Symmetric Encryption
ExFlag	Uint8	8	Protocol Check

Table 5.5: The server state structure.

5.6 Keep Alive State

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

Table 5.6: The keep alive state.

5.7 SKDP Packet Header

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

1. The Packet Flag, the type of message contained in the packet; this can be any one of the key-exchange stage flags, an encrypted message flag, a keep-alive flag, or an error flag.
2. The Message Size, this is the size in bytes of the message payload encoded as a 32-bit little-endian integer.
3. The Packet Sequence, this indicates the sequence number of the packet exchange encoded as a 64-bit little-endian integer.
4. The Packet Creation Time, a UTC timestamp of seconds from the Unix epoch encoded as a 64-bit little-endian integer.

The message is a variable-sized array. During the data phase, plaintext messages shall not exceed SKDP_MESSAGE_SIZE bytes. Serialized packet data shall not exceed the validated packet and buffer limits defined by SKDP_MESSAGE_MAX and the selected AEAD tag size.

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

Table 5.7: The SKDP packet structure.

This packet structure is used for both the key exchange protocol, and the encrypted tunnel.

5.8 Flag Types

The following are a preliminary list of packet flag types used by SKDP:

Flag Name	Numerical Value	Flag Purpose
None	0x00	No flag was specified, the default value.
Connect Request	0x01	The key-exchange client connection request flag.
Connect Response	0x02	The key-exchange server connection response flag.
Connection Terminated	0x03	The connection is to be terminated.
Encrypted Message	0x04	The message has been encrypted by the tunnel.
Exchange Request	0x05	The key-exchange client exchange request flag.
Exchange Response	0x06	The key-exchange server exchange response flag.
Establish Request	0x07	The key- exchange client establish request flag.
Establish Response	0x08	The key- exchange server establish response flag.
Establish Verify	0x09	The packet contains an establish verify flag.
Keep Alive Request	0x0A	The packet contains a keep alive request.
Session Established	0x0B	The tunnel is in the established state.
Error Condition	0x0C	The connection experienced an error.

Table 5.8: Packet header flag types.

5.9 Error Types

The following are a preliminary list of error messages used by SKDP:

Error Name	Numerical Value	Description
None	0x00	No error condition was detected.
Authentication Failure	0x01	The symmetric cipher had an authentication failure.
KEX Failure	0x02	The KEX authentication has failed.
Bad Keep Alive	0x03	The keep alive check failed.
Channel Down	0x04	The communications channel has failed.
Connection Failure	0x05	The device could not make a connection to the remote host.
Establish Failure	0x06	The transmission failed at the KEX establish phase.
Invalid Input	0x07	The expected input was invalid.
Keep Alive Expired	0x08	The keep alive has expired with no response.
Key Expired	0x0F	The SKDP public key has expired.
Key Unrecognized	0x09	The key identity is unrecognized.
Packet Un-Sequenced	0x0E	The packet was received out of sequence.
Random Failure	0x0A	The random generator has failed.
Receive Failure	0x0B	The receiver failed at the network layer.
Transmit Failure	0x0C	The transmitter failed at the network layer.
Unknown Protocol	0x0D	The protocol string was not recognized.
General Failure	0x10	The connection experienced an internal error

Table 5.9: Error type messages.

6: Operational Overview

In a multi-tiered distributed topology, a set of branch identification numbers is determined, and the master key is used to create the set of secret branch keys, which are distributed to servers on the network. The servers generate the keys for the client devices associated with each branch, and assign the secret keys to the devices. The method of distribution of secret keys varies with the type of implementation. For example, keys can be imprinted on debit cards issued by financial institutions, embedded on a device, or shared through an encrypted channel with equivalent security to a host device.

Keys should be refreshed periodically by the deployment. The SKDP implementation provides key generation, serialization, deserialization, connection establishment, packet encryption, packet decryption, and connection teardown. It does not implement an internal key-store, an authenticated key-blob format, an in-band revocation list, or an in-band ratchet/rekey exchange. If an application requires revocation, rollback protection, authenticated key storage, or scheduled key rotation, those functions shall be provided by the deployment layer or by an extension that is explicitly specified and tested.

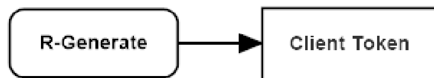
During the implemented key exchange, an error causes the endpoint to send an error packet where possible, close the socket, erase transient key-exchange material, and dispose of cipher state. During the data phase, a packet authentication failure returns an error to the caller. Implementations shall treat a data-phase authentication failure as terminal for the session and close the connection, because the selected AEAD cipher state is sequential and cannot safely continue after a failed in-sequence authentication operation.

The packet header contains a low-resolution packet creation time in the utctime field. This value is the UTC time in seconds since the Unix epoch, written as a 64-bit integer. The timestamp is checked during the exchange and establish portions of the key exchange and during data-phase packet reception to ensure that it is within the valid-time threshold of 60 seconds. The timestamp is part of the serialized packet header, and the serialized header is authenticated by the exchange MAC or supplied as AEAD associated data so that alteration of the header is detected.

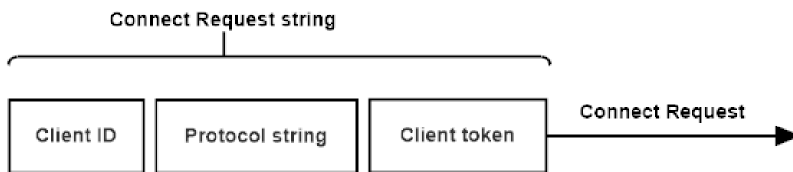
When the key exchange has completed and the encrypted tunnel has been raised, the utctime is checked each time a device receives a packet. The receiver shall require the next expected sequence number, a valid packet time, the encrypted-message flag for data packets, and an authenticated packet header. The packet header is serialized and supplied to the selected AEAD cipher as associated data, ensuring that any alteration to the flag, message length, sequence number, or packet creation time causes authentication failure.

6.1 Connect Request

Generate the random session token



Send the Client ID, protocol string, and session token, in a connection request



Hash the Client ID, protocol string, and session token, and store the session cookie DSH

Figure 6.1: SKDP client connect request.

The client device initializes a key exchange operation, by sending the server a **connection request** packet. The message contains the client's protocol configuration string, key identification array, and a random session token. The client stores a hash of these three values, for use later in the key exchange as the client's session cookie.

6.2 Connect Response

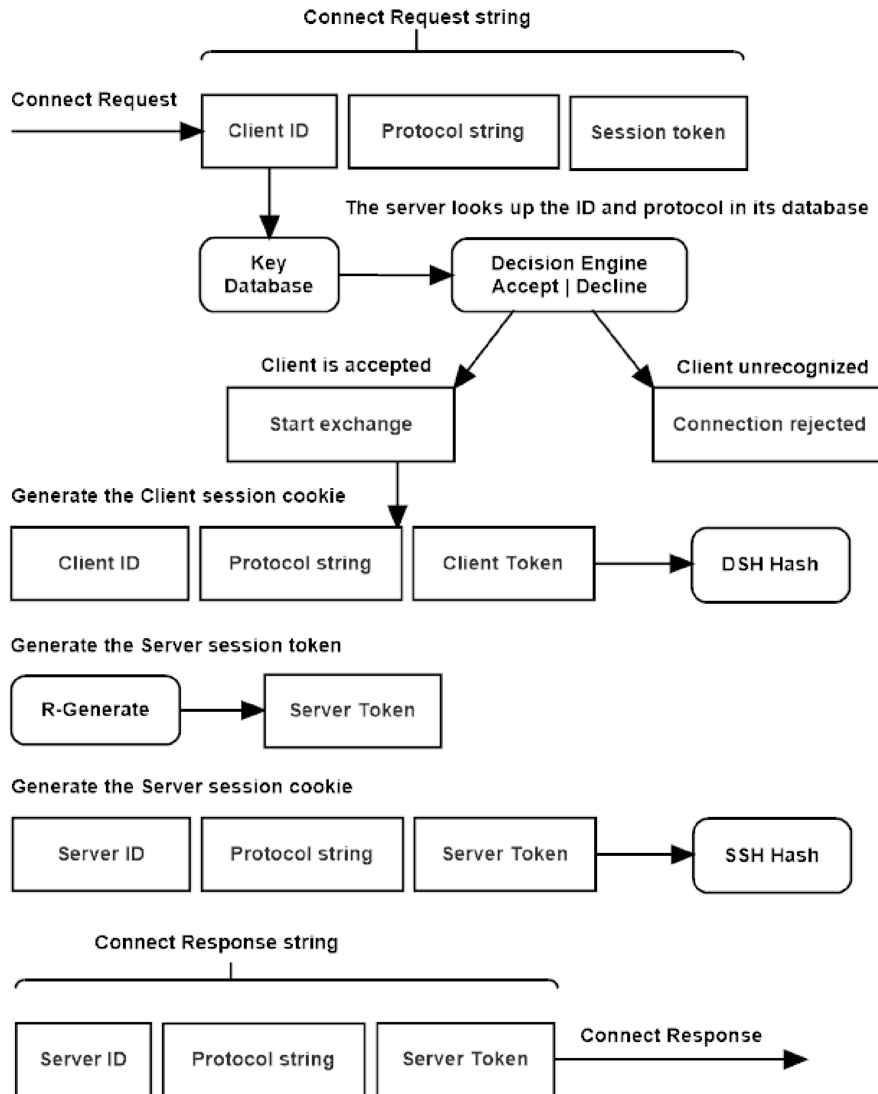


Figure 6.2: SKDP server connect response.

The server receives the **connection request**, checks that the server portion of the key identification array matches its own identity string, and stores the client's identity string in state. The server compares the client's protocol configuration with its own, if either the configuration string or key identification do not match, the connection request is rejected, and the client is sent an error notification.

The server stores a hash of the client id, configuration string, and random token, which will be used as the client's session cookie in the exchange. The server generates a random token, then hashes its own identification array, configuration string, and the random token, and stores this as the server's session cookie. The server then sends a **connect response** message to the client, containing its own identification array, configuration string, and random token.

6.3 Exchange Request

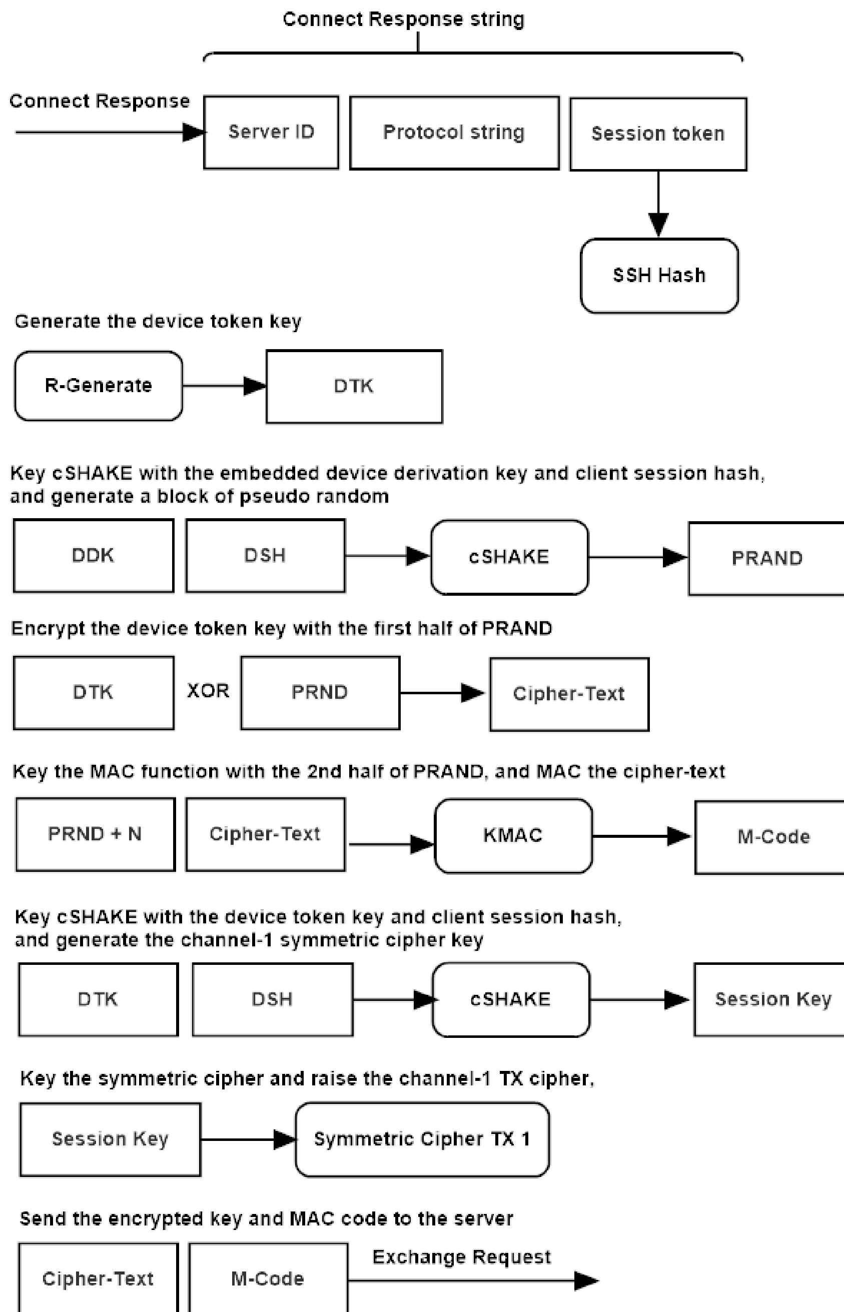


Figure 6.3: SKDP client exchange request.

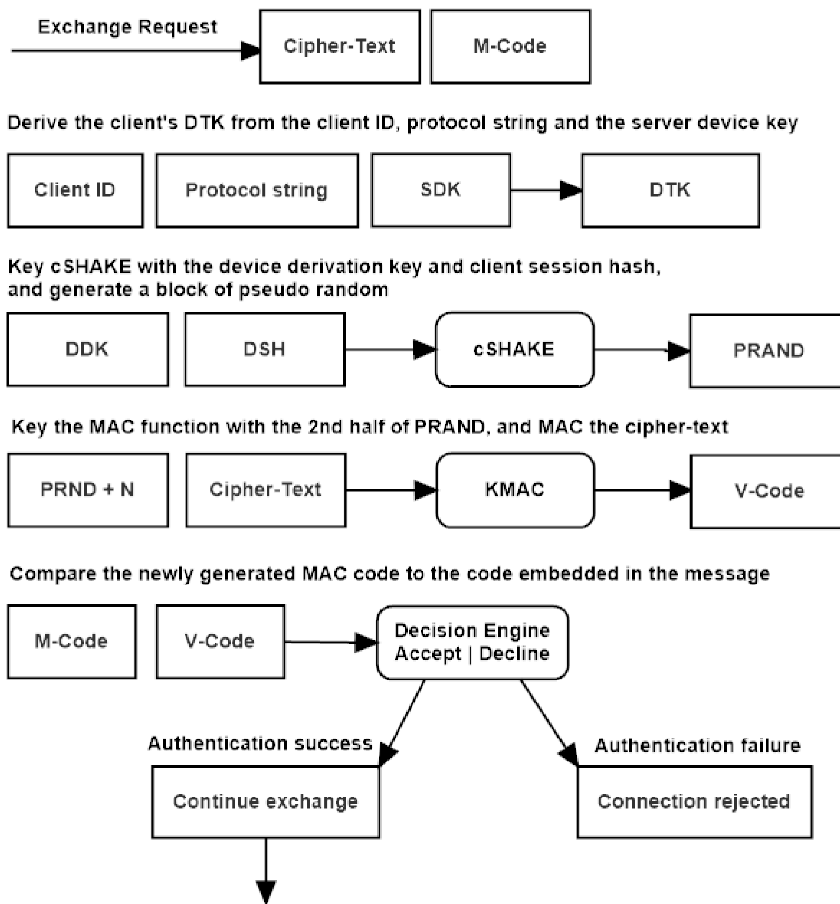
The client receives the **connect response** message from the server, and hashes the server's identity array, protocol configuration string, and random token to create the server session cookie.

The client generates a random device token-key.

The client combines the device token-key and the client session hash to key cSHAKE. The output is used to initialize the channel-1 transmit cipher key and nonce. The client then combines its embedded device derivation key and the client session hash to key cSHAKE, producing the token encryption key material and the token MAC key material.

The client combines its embedded device derivation-key, with the client's session cookie and keys cSHAKE, to generate a pseudo-random block, used as the token encryption and MAC keys. The client encrypts the random token using a bitwise XOR of the first half of the pseudo-random block, then keys KMAC using the second half of the pseudo-random block, the serialized packet header is added to the MAC, along with the cipher-text, generating the MAC tag. The encrypted session token and MAC tag are added to the exchange request message, and sent to the server.

6.4 Exchange Response



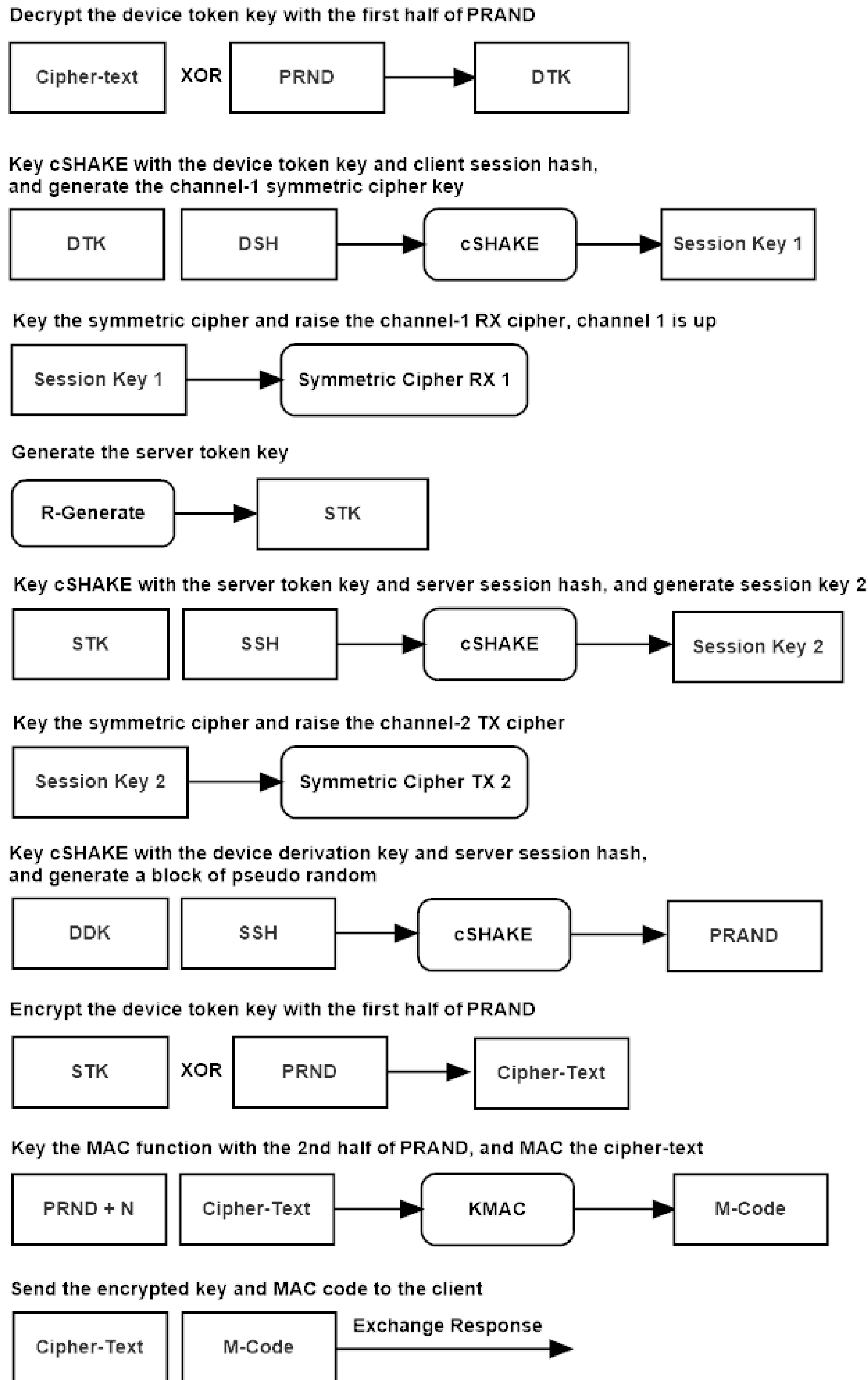


Figure 6.4: SKDP server exchange response.

The server verifies the UTC timestamp in the packet header, to ensure that the packet was sent withing a valid-time window, to prevent replay and re-use of the exchange request packet.

The server combines the server's base derivation key, and the client's key identity array, to key cSHAKE, and derives the client's device derivation key.

The server combines the device derivation-key, and the client's session cookie to key cSHAKE, and generates a pseudo-random block, used as the token encryption and MAC keys. The server uses KMAC to authenticate the cipher-text and the serialized packet header contained in the exchange request message sent by the client, and if that authentication succeeds, the server uses the encryption key to decrypt the session token using a bitwise XOR.

The server combines the recovered device token-key and the device session hash to key cSHAKE. The output is used to initialize the channel-1 receive cipher key and nonce.

The server generates a random session token-key. The server combines the server token-key, and the server session cookie to key cSHAKE, and generates the key and nonce for the server's channel-2 transmit symmetric stream cipher instance.

The server combines the device derivation-key, and the server's session cookie to key cSHAKE, which generates the token encryption and MAC keys. The server encrypts the server session token with the encryption key using a bitwise XOR, then keys KMAC with the MAC key and authenticates the cipher-text. The cipher-text and MAC tag are added to the **exchange response** message and sent to the client.

6.5 Establish Request

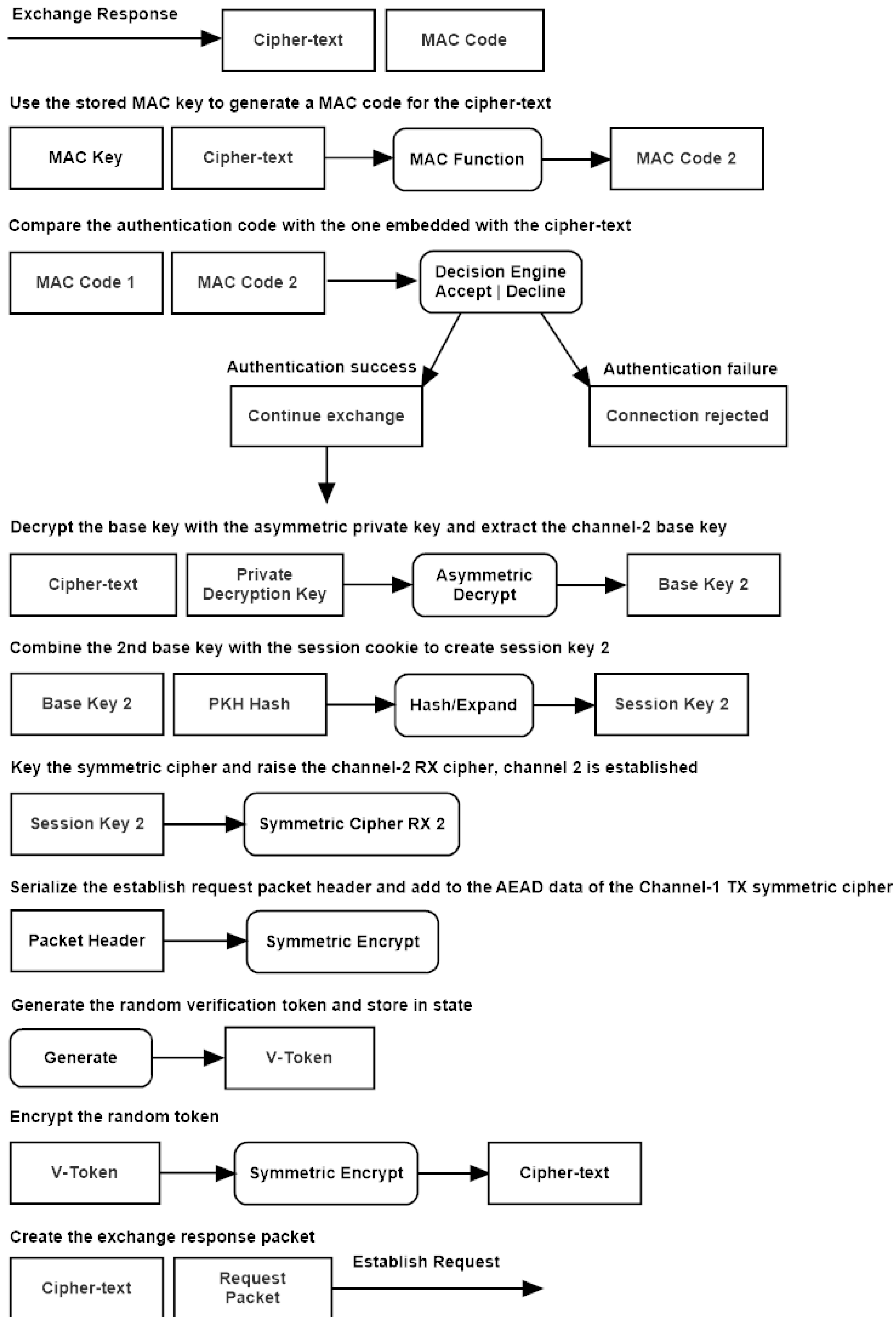


Figure 6.5: SKDP client establish request.

The client verifies the packet headers UTC valid-time, to detect re-use of the packet. The client combines its device derivation key with the server's session hash to key cSHAKE, which generates the token MAC and token encryption keys. The client keys KMAC, and authenticates the serialized packet header and cipher-text contained in the exchange response message sent by the server. If authentication succeeds the client decrypts the server's secret token using the encryption key and a bitwise XOR of the cipher-text.

The client combines the server's session token and the server's session cookie to key cSHAKE, which derives the key and nonce for the channel-2 receive symmetric stream cipher instance.

The client generates a random verification token. The client serializes the establish request packet header and supplies it to the associated data parameter of the channel-1 AEAD cipher. The client encrypts the verification token, appends the AEAD authentication tag, and sends the resulting establish request message to the server.

6.6 Establish Response

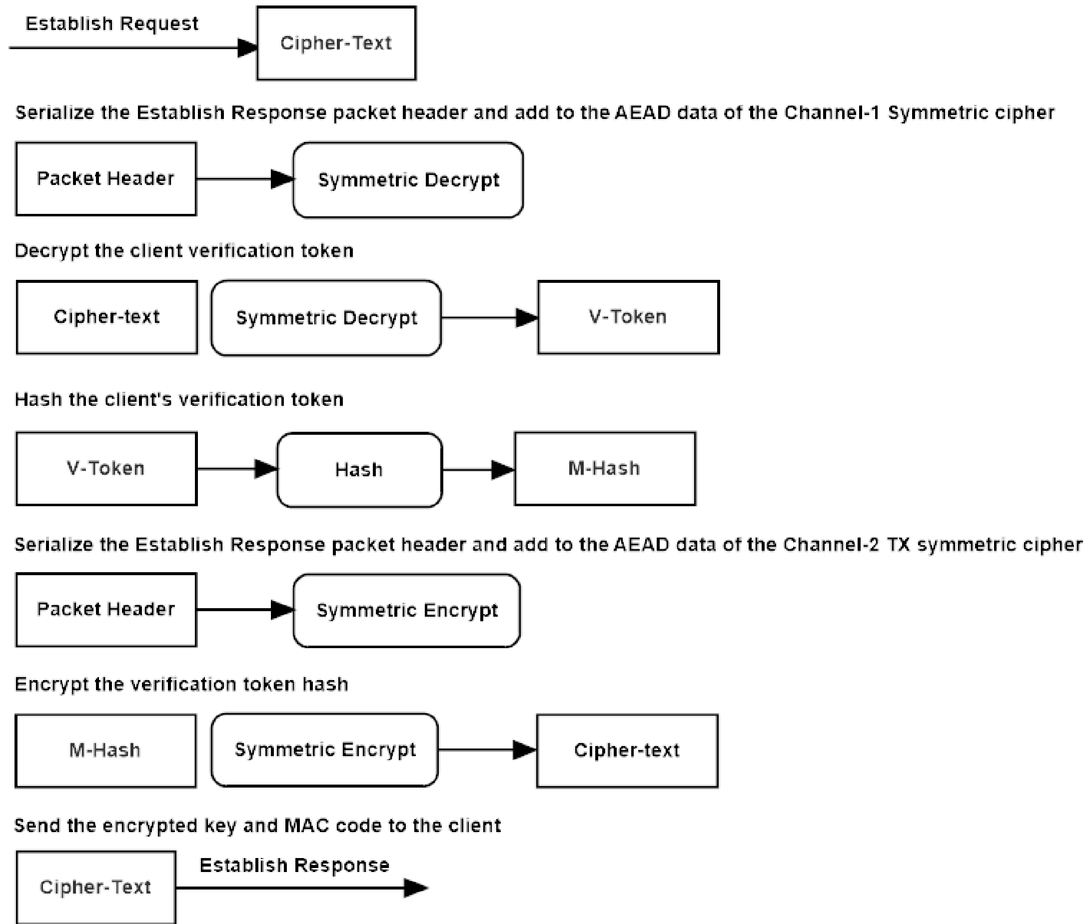


Figure 6.6: SKDP server establish response.

The server verifies that the establish request has the expected flag and message length, serializes the establish request packet header, and supplies it to the associated data parameter of the channel-1 receive AEAD cipher. If authentication and decryption succeed, the server hashes the verification token, supplies the establish response packet header to the channel-2 transmit AEAD cipher as associated data, encrypts the token hash, appends the authentication tag, and sends the establish response packet to the client.

6.7 Establish Verify

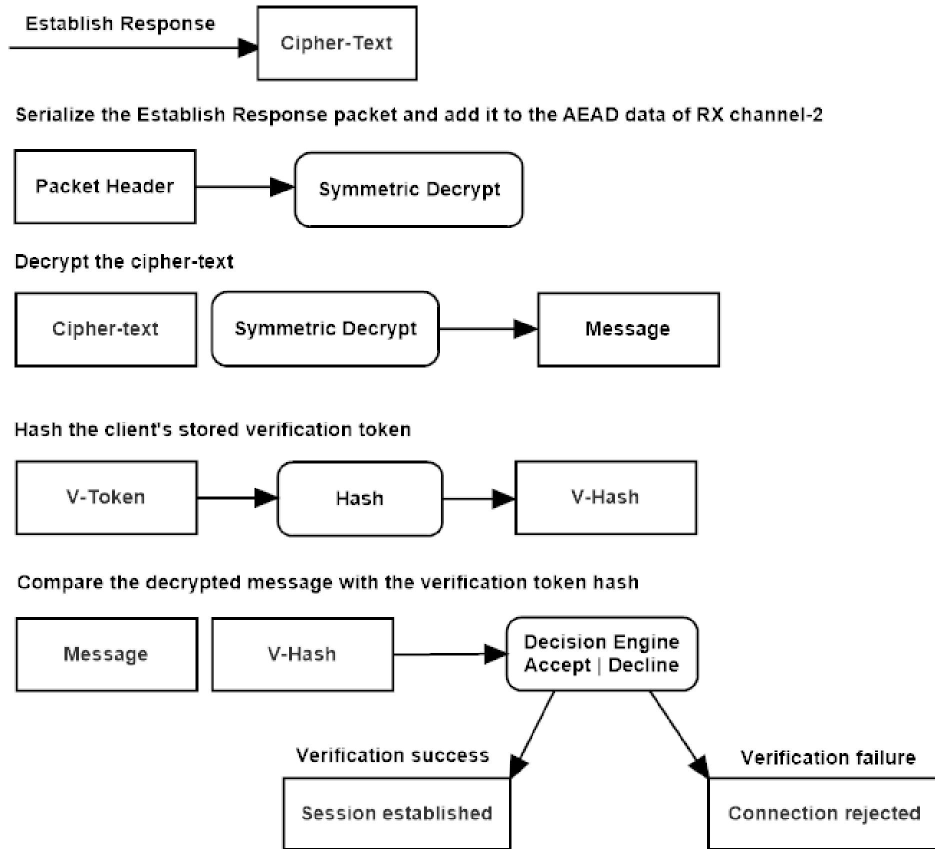


Figure 6.7: SKDP client establish verify.

The client verifies that the establish response has the expected flag and message length, serializes the packet header, and supplies it to the associated data parameter of the channel-2 receive AEAD cipher. If authentication and decryption succeed, the client hashes the stored verification token and compares the hash to the decrypted message. Upon successful comparison, the client raises its session-established flag and is ready to process data.

6.8 Data Phase Packet Processing

A transmitting endpoint shall accept plaintext messages of at most SKDP_MESSAGE_SIZE bytes. It shall increment the transmit sequence number, set the packet flag to Encrypted Message, set msglen to the plaintext length plus SKDP_MACTAG_SIZE, set the packet creation time, serialize the 21-byte header, supply that header to the selected AEAD cipher as associated data, and encrypt the message into the packet payload. A receiving endpoint shall verify that the session is established, the packet flag is Encrypted Message, the packet sequence is exactly rxseq + 1, the packet creation time is within SKDP_PACKET_TIME_THRESHOLD seconds of the receiver time, the message length is at least SKDP_MACTAG_SIZE, the message length does not exceed SKDP_MESSAGE_SIZE + SKDP_MACTAG_SIZE, and the plaintext output buffer is large enough for msglen - SKDP_MACTAG_SIZE bytes. The receive sequence number shall advance only after successful authentication and decryption.

6.9 Serialization and Parsing Requirements

The packet header serialization order is flag, msglen, sequence, and utctime. The msglen field is a 32-bit little-endian integer, and sequence and utctime are 64-bit little-endian integers. A parser shall not read a packet header unless at least SKDP_HEADER_SIZE bytes are available. A parser shall not copy packet payload bytes unless msglen is less than or equal to the remaining stream length, less than or equal to the destination message-buffer capacity, and less than or equal to SKDP_MESSAGE_MAX. Length comparisons shall be performed without addition overflow by first verifying that the stream contains the header and then comparing msglen with streamlen - SKDP_HEADER_SIZE.

6.10 Keep-Alive and Lifecycle Scope

The code contains a server-side keep-alive send function and a keep-alive state structure. The implementation does not define a complete client/server keep-alive request-response state machine and does not integrate keep-alive packets into the data-phase receive sequence. Keep-alive shall therefore be treated as an implementation hook rather than a normative interoperable protocol feature until a complete receive, response, timeout, and sequence-handling procedure is specified. The implementation also does not define an in-band ratchet, rekey, revocation, or key-store protocol.

7: Formal Description

Legend:

← ↔ →	- Assignment and direction symbols
:=, !=, ?=	- Equality operators; assign, not equals, evaluate
C	-The client host
S	-The server host
cnf	-The protocol configuration string
cpr_{rx}	-A receive channels symmetric cipher instance
cpr_{tx}	-A transmit channels symmetric cipher instance
ddk	-The device derivation key
did	-The device identity array
dsh	-The device session hash
dtk	-The device token key
$-E_k$	-Decrypt using the encryption key
E_k	-Encrypt using the encryption key
etk	-The encrypted token
Exp	-The cryptographic key expansion function
H	-The hash function
ke	-The token encryption key
km	-The token MAC key
Exp	-The key expansion function: cSHAKE
M	-The MAC function
mtag	-The MAC authentication output tag
RBG	-The random bytes generator
rtk	-A random token
sdk	-The servers derivation key
sid	-The servers identity array
stk	-The server token

stokd -The device session token

stoks -The server session token

Key Exchange Sequence

7.1 Connect Request

The client generates a random token:

$stokd \leftarrow \mathrm{RBG}(n)$

The client stores the device-id, configuration string, and token in the device session hash:

$dsh \leftarrow \mathrm{H}(did \,||\, cnf \,||\, stokd)$

The client sends its identity string, configuration string, and the generated random token to the server:

$\mathrm{C}\ \{ did \,||\, cnf \,||\, stokd \} \rightarrow \mathrm{S}$

7.2 Connect Response

The server verifies the configuration and client identity, then stores a hash of the message in the device session hash:

$dsh \leftarrow \mathrm{H}(did \,||\, cnf \,||\, stokd)$

The server generates a random token:

$stoks \leftarrow \mathrm{RBG}(n)$

It stores a hash of the server's identity, configuration string, and session token in the server session hash:

$ssh \leftarrow \mathrm{H}(sid \,||\, cnf \,||\, stoks)$

The server then sends its identity, configuration string, and session token to the client:

$\mathrm{S}\ \{ sid \,||\, cnf \,||\, stoks \} \rightarrow \mathrm{C}$

7.3 Exchange Request

The client stores a hash of the server's configuration string, server-id, and server session token:

$sth \leftarrow \mathrm{H}(sid \parallel cnf \parallel stoks)$

It generates a secret random device token key:

$dtk \leftarrow \mathrm{RBG}(n)$

The client combines its embedded device derivation key and the device session hash to produce the token encryption and MAC key material:

$ke, km \leftarrow Exp(ddk, dsh)$

It then encrypts the secret token and computes the MAC for the ciphertext:

$etk \leftarrow \mathrm{E}_{ke}(dtk)$

The client adds the serialized packet header, which includes the packet creation time and sequence number to the MAC along with the ciphertext.

$mtag \leftarrow \mathrm{M}_{km}(sh \parallel etk)$

The client combines its device session hash and the device token key to produce the channel-1 transmit cipher key, initializing the cipher:

$k, n \leftarrow Exp(dtk, dsh)$

$\mathrm{cpr}_{tx}(k, n)$

The client sends the encrypted token and MAC tag to the server:

$\mathrm{C}\ \{\ etk, mtag\ \} \rightarrow \mathrm{S}$

7.4 Exchange Response

The server combines the client's identity string with the server derivation key to derive the client's device derivation key:

$ddk \leftarrow Exp(sdk, did)$

It then combines the device's session hash and the device derivation key to produce the token encryption and MAC keys:

$ke, km \leftarrow Exp(ddk, dsh)$

The server verifies that the UTC valid-time in the packet header is within the timeout threshold. The server verifies the MAC code attached to the client's message:

$\mathrm{M}_{km}(sh \,||\, etk)$?= true = *mtag :* 0

If the MAC is verified, the server decrypts the token and derives the receive channel-1 cipher key:

$dtk \leftarrow \text{-E}_{ke}(etk)$

$k, n \leftarrow Exp(dtk, dsh)$

$\mathrm{cpr}_{rx}(k, n)$

The server generates a secret random token key:

$rtk \leftarrow \mathrm{RBG}(n)$

It combines the server’s session hash and the device's derivation key to produce the token encryption and MAC keys:

$ke, km \leftarrow Exp(ddk, ssh)$

The server encrypts the server token key and computes the MAC:

$etk \leftarrow \mathrm{E}_{ke}(rtk)$

The server adds the serialized packet header, which includes the packet creation time and sequence number to the MAC along with the ciphertext.

$mtag \leftarrow \mathrm{M}_{km}(sh \,||\, etk)$

The server initializes the transmit channel cipher key:

$k, n \leftarrow Exp(rtk, ssh)$

$\mathrm{cpr}_{tx}(k, n)$

The server sends the encrypted token key and MAC tag to the client:

$\mathrm{S}\ \{\ etk, mtag\ \} \rightarrow \mathrm{C}$

7.5 Establish Request:

The client combines the server's session hash, and the device derivation key to produce the token encryption and mac keys.

$ke, km \leftarrow Exp(ddk, ssh)$

The client verifies the UTC valid-time is within the timeout threshold The client verifies the mac code appended to the client message.

$M_{km}(etk) = \text{true} \; ? \; mtag : 0$

If the mac is verified, the client decrypts the servers token-key, and then combines the server token-key and the server's session hash to produce the channel-2 receive cipher key.

$stk \leftarrow -E_{ke}(etk)$

$k, n \leftarrow Exp(stk, ssh)$

$cpr_{rx}(k, n)$

The client generates a random verification token that it stores in state.

$vtok \leftarrow RBG(n)$

It encrypts the verification token and sends the cipher-text to the server.

$cpt \leftarrow E_k(vtok)$

$C \{ cpt \} \rightarrow S$

7.6 Establish Response:

The server authenticates and decrypts the message.

$msg \leftarrow -E_k(cpt)$

The server hashes the decrypted message.

$mhash \leftarrow H(msg)$

The server encrypts the message hash using the channel-2 cipher, and sends it to the client for verification. Both channels of the server's communications stream are now initialized.

$cpt \leftarrow E_k(mhash)$

$S\{ cpt \} \rightarrow C$

7.7 Establish Verify:

The client authenticates and decrypts the message. Both of the client's communication channels are established, the connection is now ready to send and receive data.

$msg \leftarrow \text{-E}_k(cpt)$

The client hashes the verification token stored in state, and compares that hash to the decrypted message for equality. If the check is valid, then the tunnel is ready to process data. If the check fails, the client sends an error message to the server, and tears down the connection.

$vhash \leftarrow \text{H}(msg)$

7.8 Transmission:

All of this is done by using the selected AEAD cipher interface. The default build uses the QSC AES-256-GCM interface. If SKDP_USE_RCS_ENCRYPTION is defined, the selected AEAD cipher is RCS. The associated data for each packet is the canonical serialized SKDP packet header.

$cpt \leftarrow \text{E}_k(m)$

$mc \leftarrow \text{M}_{mk}(sh \parallel cpt)$

If this check fails, the decryption function returns an authentication error and the session shall be treated as failed by the caller. No subsequent data packet shall be accepted on that session after an authentication failure.

$m \leftarrow \text{-E}_k(cpt) \mathrel{?=} \text{true}, m : 0$

8: SKDP API

8.1 Definitions and Shared API

Header:

skdp.h

Description:

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

Structures:

The SKDP_CONFIG_STRING is a static string array containing the fixed protocol configuration string.

Data Set	Purpose
SKDP_CONFIG_STRING	The SKDP configuration string

Table 8.1a SKDP configuration string.

The SKDP_ERROR_STRINGS object is a static string array containing readable SKDP error descriptions used by the error reporting functionality.

Data Set	Purpose
SKDP_ERROR_STRINGS	A string array of readable error descriptions

Table 8.1b SKDP error strings.

The **skdp_packet** contains the SKDP packet structure.

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

Table 8.1c SKDP packet structure.

The **skdp_master_key** contains the SKDP master key state.

Data Name	Data Type	Bit Length	Function
kid	Uint8 Array	0x80	The key identity string
mdk	Uint8 Array	Variable	The master derivation key
expiration	Uint64	0x40	The expiration time, in seconds from epoch

Table 8.1d SKDP master key structure.

The **skdp_server_key** contains the SKDP server key state.

Data Name	Data Type	Bit Length	Function
kid	Uint8 Array	0x80	The key identity string
sdk	Uint8 Array	Variable	The server derivation key
expiration	Uint64	0x40	The expiration time, in seconds from epoch

Table 8.1e SKDP server key structure.

The **skdp_device_key** contains the SKDP device key state.

Data Name	Data Type	Bit Length	Function
kid	Uint8 Array	0x80	The key identity string
ddk	Uint8 Array	Variable	The device derivation key
expiration	Uint64	0x40	The expiration time, in seconds from epoch

Table 8.1f SKDP device key structure.

The **skdp_keep_alive_state** contains the SKDP keep alive state.

Data Name	Data Type	Bit Length	Function
etime	Uint64	0x40	The keep alive epoch time
seqctr	Uint64	0x40	The keep alive packet sequence number
recd	Boolean	0x08	The keep alive response received status

Table 8.1g SKDP keep alive state structure.

Enumerations:

The **skdp_errors** enumeration is a list of the SKDP error code values.

Enumeration	Purpose
skdp_error_none	No error was detected

skdp_error_cipher_auth_failure	The cipher authentication has failed
skdp_error_kex_auth_failure	The kex authentication has failed
skdp_error_bad_keep_alive	The keep alive check failed
skdp_error_channel_down	The communications channel has failed
skdp_error_connection_failure	The device could not make a connection to the remote host
skdp_error_establish_failure	The transmission failed at the KEX establish phase
skdp_error_invalid_input	The expected input was invalid
skdp_error_key_not_recognized	The key was not recognized
skdp_error_random_failure	The random generator has failed
skdp_error_receive_failure	The receiver failed at the network layer
skdp_error_transmit_failure	The transmitter failed at the network layer
skdp_error_unknown_protocol	The protocol version is unknown
skdp_error_unsequenced	The packet was received out of sequence
skdp_error_general_failure	The connection experienced an error
skdp_error_packet_expired	The packet valid-time was exceeded

Table 8.1h SKDP errors enumeration.

The **skdp_flags** enum contains the SKDP packet flags.

Enumeration	**Purpose**
skdp_flag_none	No flag was specified
skdp_flag_connect_request	The SKDP key-exchange client connection request flag
skdp_flag_connect_response	The SKDP key-exchange server connection response flag
skdp_flag_connection_terminate	The connection is to be terminated
skdp_flag_encrypted_message	The message has been encrypted flag
skdp_flag_exchange_request	The SKDP key-exchange client exchange request flag
skdp_flag_exchange_response	The SKDP key-exchange server exchange response flag
skdp_flag_establish_request	The SKDP key-exchange client establish request flag
skdp_flag_establish_response	The SKDP key-exchange server establish response flag
skdp_flag_establish_verify	The packet contains an establish verify
skdp_flag_keep_alive_request	The packet contains a keep alive request
skdp_flag_session_established	The exchange is in the established state
skdp_flag_error_condition	The connection experienced an error

Table 8.1i SKDP flags enumeration.

Constants:

Constant Name	**Value**	**Purpose**

SKDP_PROTOCOL_SEC256	N/A	This flag enables 256-bit security configuration
SKDP_PROTOCOL_SEC512	N/A	This flag enables 512-bit security configuration
SKDP_CONFIG_SIZE	0x1A	The size of the protocol configuration string
SKDP_EXP_SIZE	0x08	The expiration value size
SKDP_HEADER_SIZE	0x15	The SKDP packet header size in bytes
SKDP_KEEPALIVE_STRING	0x14	The keep alive string size in bytes
SKDP_KEEPALIVE_TIMEOUT	Variable	The keep alive timeout in milliseconds (default: 5 minutes)
SKDP_MESSAGE_SIZE	0x400	The message size used during a communications session
SKDP_MESSAGE_MAX	0x415	The maximum serialized packet size for a 1024-byte message and 21-byte header
SKDP_SERVER_PORT	0x899	The default server port address
SKDP_MID_SIZE	0x04	The master id size
SKDP_SID_SIZE	0x08	The server id size
SKDP_DID_SIZE	0x0C	The device id size
SKDP_TID_SIZE	0x04	The session id size
SKDP_KID_SIZE	0x10	The SKDP key identity size
SKDP_SEQUENCE_TERMINATOR	0xFFFFFFFF	The sequence number of a packet that closes a connection
SKDP_CPRKEY_SIZE	0x20/0x40	The SKDP symmetric cipher key size
SKDP_DDK_SIZE	0x20/0x40	The device derivation key size
SKDP_DTK_SIZE	0x20/0x40	The device token key size
SKDP_HASH_SIZE	0x20/0x40	The size of the hash function output
SKDP_MACKEY_SIZE	0x20/0x40	The SKDP mac key size
SKDP_MACTAG_SIZE	0x10/0x20/0x40	The AEAD or MAC tag size. Default AES-GCM uses 0x10; RCS-256 uses 0x20; RCS-512 uses 0x40.
SKDP_MDK_SIZE	0x20/0x40	The size of the master derivation key
SKDP_PERMUTATION_RATE	0x88/0x48	The rate at which keccak processes data
SKDP_SDK_SIZE	0x20/0x40	The server derivation key size
SKDP_STK_SIZE	0x20/0x40	The session token key size
SKDP_STH_SIZE	0x20/0x40	The session token-hash size
SKDP_STOK_SIZE	0x20/0x40	The session token size
SKDP_KEY_DURATION_DAYS	0x16D	The number of days a key remains valid

SKDP_KEY_DURATION_SECONDS	D * 24 * 60 * 60	The number of seconds a key remains valid
SKDP_DEVKEY_ENCODED_SIZE	Variable	The size of the encoded device key
SKDP_MSTKEY_ENCODED_SIZE	Variable	The size of the encoded master key
SKDP_SRVKEY_ENCODED_SIZE	Variable	The size of the encoded server key
SKDP_CONNECT_REQUEST_SIZE	Variable	The kex connect stage request packet size
SKDP_EXCHANGE_REQUEST_SIZE	Variable	The kex exchange stage request packet size
SKDP_ESTABLISH_REQUEST_SIZE	Variable	The kex establish stage request packet size
SKDP_CONNECT_RESPONSE_SIZE	Variable	The kex connect stage response packet size
SKDP_EXCHANGE_RESPONSE_SIZE	Variable	The kex exchange stage response packet size
SKDP_ESTABLISH_RESPONSE_SIZE	Variable	The kex establish stage response packet size
SKDP_ESTABLISH_VERIFY_SIZE	Variable	The kex establish verify stage response packet size
SKDP_ERROR_STRING_DEPTH	0x11	The number of error strings
SKDP_ERROR_STRING_WIDTH	0x80	The length of each error string
SKDP_PACKET_TIME_THRESHOLD	0x3C	The packet valid-time threshold in seconds

Table 8.1j SKDP constants.

Functions:

Packet Clear

Clear a packet's state, resetting the structure to zero.

```
void skdp_packet_clear(skdp_packet* packet)
```

Deserialize Device Key

Deserialize a client device key.

```
void skdp_deserialize_device_key(skdp_device_key* dkey, const uint8_t input[SKDP_DEVKEY_ENCODED_SIZE])
```

Serialize Device Key

Serialize a client device key.

```
void skdp_serialize_device_key(uint8_t output[SKDP_DEVKEY_ENCODED_SIZE], const skdp_device_key* dkey)
```

Deserialize Master Key

Deserialize a master key.

```
void skdp_deserialize_master_key(skdp_master_key* mkey, const uint8_t input[SKDP_MSTKEY_ENCODED_SIZE])
```

Serialize Master Key

Serialize a master key.

```
void skdp_serialize_master_key(uint8_t output[SKDP_MSTKEY_ENCODED_SIZE], const skdp_master_key* mkey)
```

Deserialize Server Key

Deserialize a server key.

```
void skdp_deserialize_server_key(skdp_server_key* skey, const uint8_t input[SKDP_SRVKEY_ENCODED_SIZE])
```

Serialize Server Key

Serialize a server key.

```
void skdp_serialize_server_key(uint8_t output[SKDP_SRVKEY_ENCODED_SIZE], const skdp_server_key* skey)
```

Generate Master Key

Generate a master key-set.

```
bool skdp_generate_master_key(skdp_master_key* mkey, const uint8_t kid[SKDP_KID_SIZE])
```

Generate Server Key

Generate a server key-set.

```
void skdp_generate_server_key(skdp_server_key* skey, const skdp_master_key* mkey, const uint8_t kid[SKDP_KID_SIZE])
```

Generate Device Key

Generate a server key-set.

```
void skdp_generate_device_key(skdp_device_key* dkey, const skdp_server_key* skey, const uint8_t kid[SKDP_KID_SIZE])
```

Error To String

Return a pointer to a string description of an error code.

```
const char* skdp_error_to_string(skdp_errors error)
```

Error Message

Convert a one-byte error message value to the corresponding SKDP error enumeration.

```
skdp_errors skdp_message_to_error(uint8_t message)
```

Message To Error

Populate a packet structure with an error message.

```
void skdp_packet_error_message(skdp_packet* packet, skdp_errors error)
```

Header Deserialize

Deserialize a byte array to a packet header. The input header length shall be at least SKDP_HEADER_SIZE bytes.

```
bool skdp_packet_header_deserialize(const uint8_t* header, size_t headerlen, skdp_packet* packet)
```

Header Serialize

Serialize a packet header to a byte array.

```
void skdp_packet_header_serialize(const skdp_packet* packet, uint8_t* header)
```

Packet To Stream

Serialize a packet to a byte array.

size_t skdp_packet_to_stream(const skdp_packet* packet, uint8_t* pstream)

Stream To Packet

Deserialize a byte array to a packet. The stream length and destination message-buffer capacity shall be supplied and validated before copying payload bytes.

bool skdp_stream_to_packet(const uint8_t* pstream, size_t streamlen, skdp_packet* packet, size_t message_capacity)

8.2 Server API

Header:

skdpserver.h

Description:

Functions used to implement the SKDP server.

Structures:

The **skdp_server_state** contains the SKDP server state structure.

Data Name	Data Type	Bit Length	Function
rxcpr	Selected AEAD cipher state	Variable	The receive channel cipher state
txcpr	Selected AEAD cipher state	Variable	The transmit channel cipher state
did	Uint8 Array	0x10	The device identity string
dsh	Uint8 Array	0x20/0x40	The device session hash
kid	Uint8 Array	0x10	The key identity string
ssh	Uint8 Array	0x20/0x40	The server session hash
sdk	Uint8 Array	0x20/0x40	The server derivation key
exflag	enum	skdp_flags	The KEX position flag

expiration	Uint64	0x40	The expiration time, in seconds from epoch
rxseq	Uint64	0x40	The receive channels packet sequence number
txseq	Uint64	0x40	The transmit channels packet sequence number

Table 8.2 SKDP server state structure.

API:

Connection Close

Close the remote session and dispose of resources.

void skdp_server_connection_close(skdp_server_state* ctx, const qsc_socket* sock, skdp_errors error)

Send Keepalive

Send a keep-alive to the remote host.

skdp_errors skdp_server_send_keep_alive(skdp_keep_alive_state* kctx, const qsc_socket* sock)

Initialize

Initialize the server state structure.

void skdp_server_initialize(skdp_server_state* ctx, const skdp_server_key* skey)

Listen IPv4

Run the IPv4 networked key exchange function. Returns the connected socket and the SKDP server state.

skdp_errors skdp_server_listen_ipv4(skdp_server_state* ctx, qsc_socket* sock, const qsc_ipinfo_ipv4_address* address, uint16_t port)

Listen IPv6

Run the IPv6 networked key exchange function. Returns the connected socket and the SKDP server state.

skdp_errors skdp_server_listen_ipv6(skdp_server_state* ctx, qsc_socket* sock, const qsc_ipinfo_ipv6_address* address, uint16_t port)

Decrypt Packet

Decrypt a packet, authenticate the serialized packet header as associated data, and copy the plaintext to the message output if the output buffer is large enough.

skdp_errors skdp_server_decrypt_packet(skdp_server_state* ctx, const skdp_packet* packetin, uint8_t* message, size_t message_capacity, size_t* msglen)

Encrypt Packet

Encrypt a message and build an output packet. The plaintext message length shall not exceed SKDP_MESSAGE_SIZE.

skdp_errors skdp_server_encrypt_packet(skdp_server_state* ctx, const uint8_t* message, size_t msglen, skdp_packet* packetout)

Ratchet Response

No ratchet response function is declared or implemented in the reviewed code. Ratchet and rekey behavior are outside the implemented SKDP API and shall not be treated as normative behavior in this specification.

No function prototype is defined for this operation in the implemented API.

8.3 Client API

Header:

skdpclient.h

Description:

Functions used to implement the SKDP client.

Structures:

The **skdp_client_state** contains the SKDP client state structure.

Data Name	Data Type	Bit Length	Function
rxcpr	Selected AEAD cipher state	Variable	The receive channel cipher state
txcpr	Selected AEAD cipher state	Variable	The transmit channel cipher state
ddk	Uint8 Array	0x20/0x40	The device derivation key
dsh	Uint8 Array	0x20/0x40	The device session hash
kid	Uint8 Array	0x10	The key identity string
ssh	Uint8 Array	0x20/0x40	The server session hash
exflag	enum	skdp_flags	The KEX position flag
expiration	Uint64	0x40	The expiration time, in seconds from epoch
rxseq	Uint64	0x40	The receive channels packet sequence number
txseq	Uint64	0x40	The transmit channels packet sequence number

Table 8.3 SKDP client state structure.

API:

Send Error

Send an error code to the remote host.

```
void skdp_client_send_error(const qsc_socket* sock, skdp_errors error)
```

Initialize

Initialize the client state structure.

```
void skdp_client_initialize(skdp_client_state* ctx, const skdp_device_key* ckey)
```

Connect IPv4

Run the IPv4 networked key exchange function. Returns the connected socket and the SKDP server state.

```
skdp_errors skdp_client_connect_ipv4(skdp_client_state* ctx, qsc_socket* sock, const qsc_ipinfo_ipv4_address* address, uint16_t port)
```

Connect IPv6

Run the IPv6 networked key exchange function. Returns the connected socket and the SKDP server state.

```
skdp_errors skdp_client_connect_ipv6(skdp_client_state* ctx, qsc_socket* sock, const qsc_ipinfo_ipv6_address* address, uint16_t port)
```

Connection Close

Close the remote session and dispose of resources.

```
void skdp_client_connection_close(skdp_client_state* ctx, const qsc_socket* sock, skdp_errors error)
```

Decrypt Packet

Decrypt a message and copy it to the message output.

```
skdp_errors skdp_client_decrypt_packet(skdp_client_state* ctx, const skdp_packet* packetin, uint8_t* message, size_t message_capacity, size_t* msglen)
```

Encrypt Packet

Encrypt a message and build an output packet.

```
skdp_errors skdp_client_encrypt_packet(skdp_client_state* ctx, const uint8_t* message, size_t msglen, skdp_packet* packetout)
```

Ratchet Request

No ratchet request function is declared or implemented in the reviewed code. Ratchet and rekey behavior are outside the implemented SKDP API and shall not be treated as normative behavior in this specification.

No function prototype is defined for this operation in the implemented API.

8.4 Conformance Requirements

An implementation conforming to this specification shall implement the packet header serialization order flag, msglen, sequence, utctime; shall reject malformed or truncated packet headers; shall reject encrypted data packets whose flag is not Encrypted Message; shall reject plaintext inputs larger than SKDP_MESSAGE_SIZE; shall reject ciphertext packets larger than SKDP_MESSAGE_SIZE + SKDP_MACTAG_SIZE; shall validate packet creation time against

SKDP_PACKET_TIME_THRESHOLD; shall authenticate the canonical serialized packet header as associated data; shall advance the receive sequence only after successful packet authentication; and shall close or otherwise invalidate the session after a data-phase authentication failure. Conforming implementations shall not claim in-band revocation, in-band ratcheting, key-store protection, or complete keep-alive interoperability unless those functions are separately specified and implemented.

9: SKDP Cryptanalysis

9.1 Threat Model and Target Properties

An active, adaptive adversary can control the network, including eavesdropping, modification, replay, reordering, injection, and dropping of packets. The adversary may compromise long-term master, server, or device derivation keys, may record key-exchange transcripts, may observe ciphertext and packet headers, and may attempt to exploit malformed packet encodings. The adversary is not assumed to break the underlying security of SHA3, cSHAKE, KMAC, AES-GCM, RCS, or the approved random-byte generator.

SKDP aims to provide mutual authentication, per-session channel-key separation, authenticated encrypted payload transport, replay resistance through sequence and packet-time validation, and downgrade resistance through the protocol configuration string. SKDP does not provide cryptographic forward secrecy against later compromise of a long-term derivation key if the adversary recorded the key-exchange transcript protected by that key hierarchy.

Goal	Formal requirement
Mutual authentication	Each side accepts ⇔ the peer proved possession of DDK or SDK and all message MACs verified.
Confidentiality & Integrity	Every payload encrypted under a fresh TK is IND-CPA & INT-CTXT.
Session-key separation	Each accepted session uses direction-specific random token material and transcript hashes to derive independent channel keys. Long-term derivation-key compromise with recorded transcripts is outside this property.
Operational key recovery after refresh	Future sessions depend on fresh random tokens and uncompromised current derivation keys. If a long-term key is compromised, the affected scope remains compromised until re-provisioned.
Replay & downgrade defence	UTC field and sequence number inside MAC/AAD reject stale, reordered, or mixed-configuration packets when implementations follow the required state machine.

9.2 Security of the Three-Stage Handshake

Stage	Critical check	Security argument	Result
Connect (Fig. 6.1–6.2)	dsh = H(did‖cnf‖stokd) and ssh = H(sid‖cnf‖stoks) bound IDs & config	Any MitM altering cnf or IDs breaks subsequent MACs	**No silent downgrade / impersonation**
Exchange (Fig. 6.3–6.4)	Token key *dtk* encrypted as etk = dtk ⨁ Keccak(dsh,ddk); tag mtag = KMAC(km, header‖etk)	Forgery ⇒ break UF-CMA of KMAC or predict Keccak output without DDK	**Authentic & confidential token**
Establish (Fig. 6.5–6.7)	Verification token hashed, echoed through channel-2	If either cipher key wrong, MAC fails → abort before data	**Explicit key confirmation**

Key-secrecy argument: channel keys are derived from random token material and transcript-derived session hashes through cSHAKE. The confidentiality of a recorded session therefore depends on the secrecy of the random tokens and on the secrecy of the long-term derivation keys used to protect those tokens during the exchange. If a long-term derivation key is later compromised and the exchange transcript was recorded, the corresponding token-protection material can be recomputed and the recorded session can be exposed. If long-term derivation keys remain secret and random tokens are generated unpredictably and erased after use, the derived channel keys are computationally indistinguishable from random under the assumed PRF behavior of cSHAKE.

9.3 Data-Phase Confidentiality and Integrity

For each direction, the data-phase cipher key and nonce are derived by applying cSHAKE to the direction-specific random token and the corresponding session hash. Packet payloads are processed by the selected AEAD cipher using the canonical serialized SKDP header as associated data.

Confidentiality and integrity depend on correct use of the selected AEAD cipher, uniqueness of the per-session key and nonce material, exact in-order processing, and rejection of malformed or unauthenticated packets. The implementation shall not continue a session after a data-phase authentication failure.

9.4 Replay, Downgrade, and MitM Resistance

- The UTC creation time and sequence number are encoded in the 21-byte header and are authenticated during the exchange and data phases. The receiver rejects packets outside the valid-time threshold and rejects packets whose sequence number is not the next expected value.

- The configuration string is included in the session hashes and is compared during connection establishment. A configuration mismatch causes the exchange to fail before session establishment.
- The exchange MACs cover the encrypted token material and the serialized packet header. Identity and configuration values are bound through the session hashes used as KDF customization input.

9.5 Computational and Bandwidth Costs

Item	256-bit profile	512-bit profile
Handshake Keccak calls	6 × Exp + 4 × P	6 × Exp + 4 × P
Handshake KMAC calls	4	4
Online latency	3 RTT (≈ 2 ms LAN)	3 RTT (≈ 3 ms LAN)
Payload overhead	21-byte header + 16-byte default AES-GCM tag, or 32-byte RCS-256 tag	21-byte header + 64-byte RCS-512 tag

Constant-time reference code (SIMD where available) keeps a full round-trip below 5 ms on a 2 GHz ARM Cortex-A72.

9.6 Comparison with Related Schemes

Dimension	SKDP v1.1	HKDS	DUKPT-AES	Signal Double-Ratchet
Core idea	Symmetric tunnel, dual one-time tokens	Hierarchical key cache	Counter-based derivation	Asym + sym ratchets
Primitives	Keccak cSHAKE/KMAC + selected AEAD (AES-GCM default, RCS optional)	SHAKE/KMAC + XOR	AES + SHA-1	X25519 + AES-GCM
FS	No against later DDK/SDK compromise with recorded exchange; yes for erased ephemeral channel state only	✓	✗	✓
PR	Requires uncompromised current derivation keys and fresh random tokens	✓	✗	✓

Root-key compromise impact	Affected hierarchy compromised until replacement; recorded sessions may be exposed if transcripts were captured	Past safe, future safe after rotation	Past+future exposed	Past safe
Handshake RTT	3	2	0 (pre-shared)	≥ 3
MitM surface	Full-MAC transcript	Full-MAC	None (no handshake)	DH signed only at initial pre-key
Replay defence	UTC + sequence number, authenticated header, exact receive order	UTC + seq#, MACed	Counter only	DH ratchet

Observation – SKDP offers *stronger MitM and replay guarantees* than DUKPT and matches HKDS, while keeping a shorter, fully symmetric handshake than Double-Ratchet (which relies on asymmetric primitives).

9.7 Residual Gaps and Hardening Advice

Long-term derivation-key compromise remains the primary residual risk. If an attacker records the key exchange and later obtains the relevant device or server derivation key, recorded session token material may be recovered. Deployments shall therefore protect derivation keys in hardware or equivalent controlled storage and shall replace affected key material after suspected compromise.

Serialized SKDP keys are raw key records consisting of identity, derivation key material, and expiration metadata. The SKDP wire protocol does not authenticate or encrypt those at-rest records. Deployments shall protect stored or transported key records with an authenticated container or equivalent provisioning mechanism.

The implementation does not provide a complete keep-alive exchange, in-band rekey, in-band ratchet, revocation list, concurrent server framework, or key-store service. Those functions require separate specification, implementation, and conformance tests.

9.8 Conclusion

Under the stated assumptions, SKDP provides authenticated symmetric session establishment and authenticated encrypted payload transport with per-session key separation. Its security boundary includes the secrecy of the long-term derivation keys and the correctness of the packet parsing, sequence handling, timestamp validation, and AEAD authentication logic. The implemented protocol does not include in-band revocation, in-band ratcheting, a complete keep-

alive exchange, or key-store protection; these functions require separate specification before they can be claimed as interoperable SKDP behavior.

10: Design Decisions

SKDP was designed to be flexible and scalable. It can scale to billions of devices using a pyramid hierarchy of client devices connecting to intermediate 'branch' servers which can inter-connect through a master server, or it can be used in a single link between two endpoints. It could be implemented on credit or debit cards as an encrypted transport, in removable media to create pluggable lightweight communications channels, or as the encryption protocol used to connect VPN endpoints. The SKDP protocol can be used anywhere a cryptographically-strong, lightweight, post-quantum secure communications channel is required.

SKDP uses Keccak, the NIST SHA3 secure hash and pseudo-random generation functions. These state-of-the-art functions and protocols, that have been studied extensively and are officially recognized as a strong post-quantum resistant family of cryptographic functions.

SKDP can be configured for 256-bit or 512-bit symmetric security profiles when the corresponding compile-time options and selected cipher support that profile. The default build path uses the QSC AES-256-GCM interface when SKDP_USE_RCS_ENCRYPTION is not defined. The RCS AEAD interface is used when SKDP_USE_RCS_ENCRYPTION is defined. Implementations shall describe the selected cipher configuration precisely and shall not imply that the optional RCS configuration is the default unless that compile-time option is enabled.

The packet creation time field and authenticated header processing provide replay resistance when used with exact sequence validation. The packet header is authenticated as part of the exchange MAC or as AEAD associated data during the data phase. This protects the flag, message length, sequence number, and packet creation time from undetected alteration.