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

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

This document is the second revision of the specification of SKDP (version 1.1), further revisions may become necessary during the pursuit of a standard model, and revision numbers shall be incremented with changes to the specification. The reader is asked to consider only the most recent revision of this draft, as the authoritative expression of the SKDP specification.

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

Tables

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.

What we propose with SKDP is a symmetric scheme that uses pre-shared keys in a way that provides forward secrecy, is scalable, and solves many of the problems associated with existing schemes that use pre-shared keys. In SKDP, capturing a client host's embedded key does not compromise past messages, and capturing the server’s key database does not reveal anything about past encrypted messages because the symmetric ciphers used in the message stream are keyed with ephemeral keys that cannot be derived from the pre-shared key alone.

This introduction sets the stage for SKDP as a secure and scalable key distribution protocol designed to address the limitations of current cryptographic methods by using a symmetric key strategy that incorporates strong forward secrecy and scalable management of secure communication channels.

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 ephemeral, and use shared secrets for each channel that are unique to each session (forward secrecy).
2) The capture of the devices shared key does not directly reveal any information about future sessions (predicative resistance).
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

3.2 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.

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 last epoch (01/01/1900) to the time the key remains valid. This value is checked during the initialization of the client, if the key has expired, the connection attempt is halted and an error returned.

The key identity array is a 16-byte array that uniquely identifies a device key. This identifier can be used to match the key on a branch server. The key identity array, is divided into subsections, 32-bit identification numbers for the master key, branch key, device key, and set instance.

Master ID	**Branch ID**	**Device ID**	**Set ID**
4 bytes	**4 bytes**	**4 bytes**	**4 bytes**

Table 5.1b: The device identity structure.

The master key array, is hashed with a branch and master identification array, to derive the branch key. More than four billion branches may be created from a single master key. The branch key is hashed with the device identification array, as well as the branch and master identification arrays, to derive more than four billion possible device keys. The set identification array, the last four bytes of the key identification array, is the key version counter, the embedded key version, in a set of time-limited keys assigned to the client device.

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, a message, or an error flag.
2. The **Packet Sequence**, this indicates the sequence number of the packet exchange.
3. The **Packet Creation** time, a UTC timestamp of seconds from the epoch.
4. The **Message Size**, this is the size in bytes of the message payload.

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

Packet Flag **1 byte**	**Packet Sequence** **8 bytes**	**Packet Creation** **8 bytes**	**Message Size** **4 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	0xFF	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	0x02	The keep alive check failed.
Channel Down	0x03	The communications channel has failed.

Connection Failure	0x04	The device could not make a connection to the remote host.
Establish Failure	0x05	The transmission failed at the KEX establish phase.
Exstart Failure	0x06	The transmission failed at the KEX exstart phase.
Invalid Input	0x07	The expected input was invalid.
Keep Alive Expired	0x08	The keep alive has expired with no response.
Key Expired	0x09	The SKDP public key has expired.
Key Unrecognized	0x0A	The key identity is unrecognized.
Packet Un-Sequenced	0x0B	The packet was received out of sequence.
Random Failure	0x0C	The random generator has failed.
Receive Failure	0x0D	The receiver failed at the network layer.
Transmit Failure	0x0E	The transmitter failed at the network layer.
Verify Failure	0x0F	The expected data could not be verified.
Unknown Protocol	0x10	The protocol string was not recognized.
General Failure	0xFF	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.

Each key has an expiration-time parameter. The expiration of keys should be determined by the application of this technology. Some applications, for example debit cards, may tolerate longer periods, while other applications like a high-security communications link, might be renewed on a much shorter time period. It is recommended that keys are refreshed periodically, this guarantees that in a worst-case scenario, where either the master or branch keys have been captured, that security is continually restored. A strong post-quantum asymmetric encrypted tunnel, like QSMP, can periodically add entropy to a server's and a device's embedded key, by mixing new entropy with that embedded key, a new shared secret combined with the base key to generate a new base derivations key. These keys are counted as key revisions, in the key-set bytes of the key identification string.

Any error during the key exchange or during the communications operation, causes the client or server to send an error message to the other host, disconnect, and tear down the session. This includes checks for message synchronization, expected size of sent and received messages during the key exchange, authentication failures, and internal errors raised by cryptographic or network functions used by the key exchange and communications stream.

In version 1.1, an anti-replay attack mechanism has been added to the key exchange and the encrypted tunnel. A field has been added to the packet header *utctime*, which contains a low-resolution packet creation time. This is the UTC time in seconds since the epoch, written as a 64-bit integer. This timestamp is checked during the *exchange* and *establish* portions of the key exchange, to be within the valid-time threshold (60 seconds). The timestamp along with the serialized packet header, is added to the MAC function during portions of the client/server key exchange. The *utctime* field is checked when the packet is received, and the header is added to the MAC to ensure it has not been altered.

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 packet header is serialized and added to the AEAD field of the symmetric cipher, ensuring that the packet can not be re-used, and that the packet header has not been altered.

6.1 Connect Request

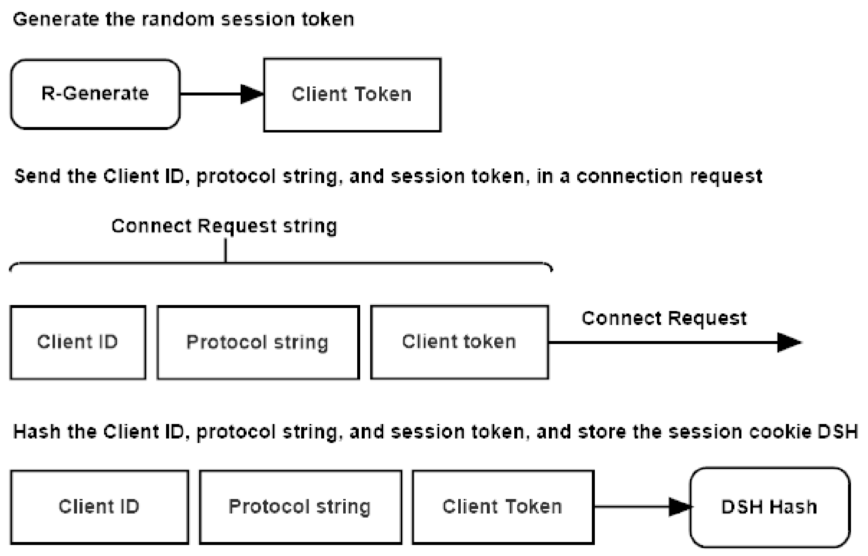

Generate the random session token
R-Generate
Client Token
Send the Client ID, protocol string, and session token, in a connection request
Connect Request string
Client ID
Protocol string
Client token
Connect Request
Hash the Client ID, protocol string, and session token, and store the session cookie DSH
Client ID
Protocol string
Client Token
DSH Hash

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

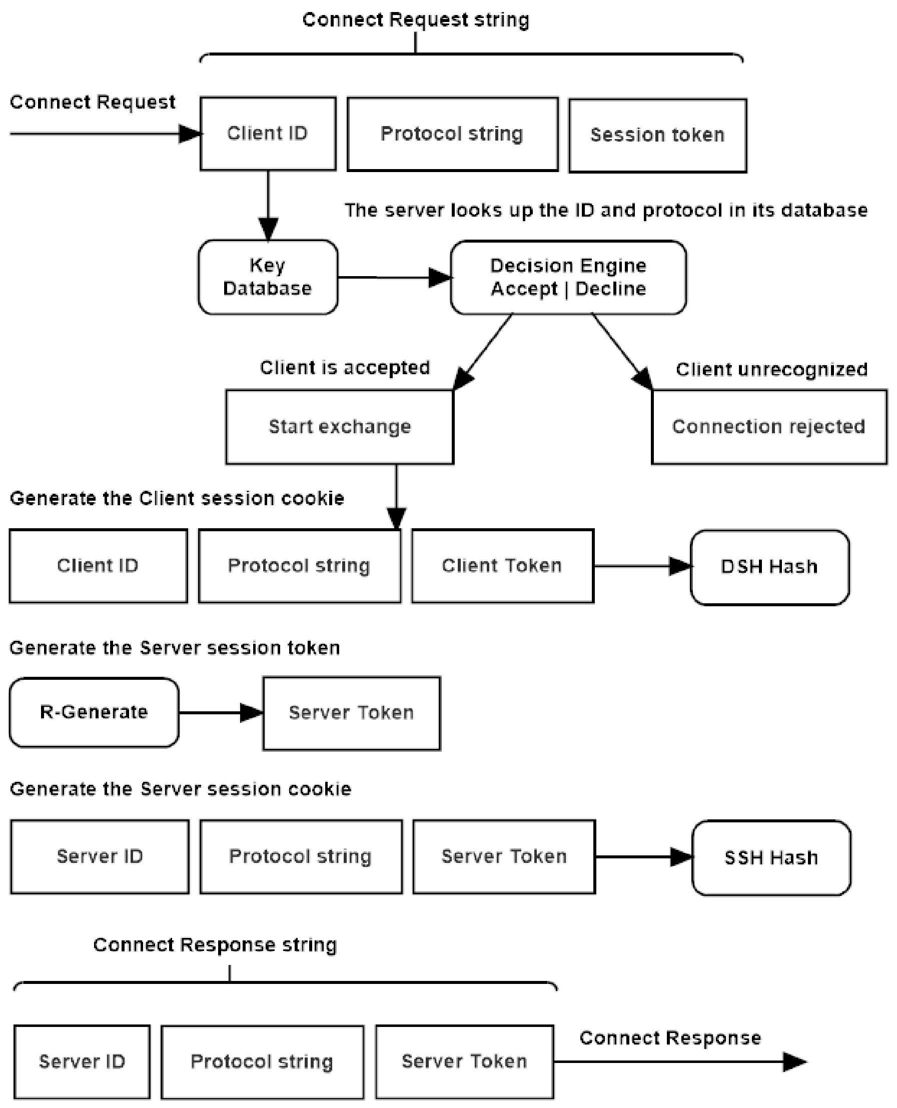


Figure 6.2: SKDP server connect response.

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

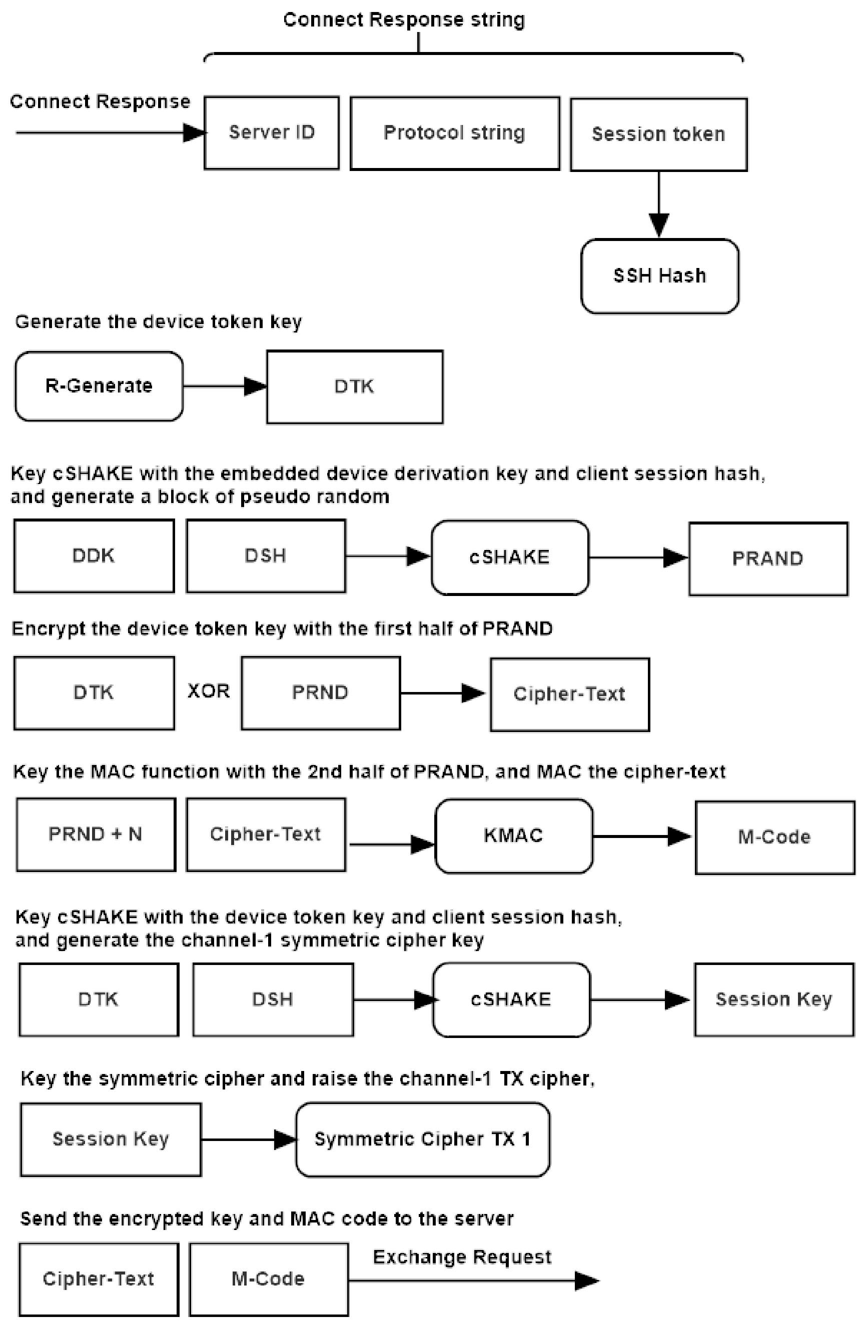

Connect Response string
Connect Response
Server ID
Protocol string
Session token
SSH Hash
Generate the device token key
R-Generate
DTK
Key cSHAKE with the embedded device derivation key and client session hash, and generate a block of pseudo random
DDK
DSH
cSHAKE
PRAND
Encrypt the device token key with the first half of PRAND
DTK
XOR
PRND
Cipher-Text
Key the MAC function with the 2nd half of PRAND, and MAC the cipher-text
PRND + N
Cipher-Text
KMAC
M-Code
Key cSHAKE with the device token key and client session hash, and generate the channel-1 symmetric cipher key
DTK
DSH
cSHAKE
Session Key
Key the symmetric cipher and raise the channel-1 TX cipher,
Session Key
Symmetric Cipher TX 1
Send the encrypted key and MAC code to the server
Cipher-Text
M-Code
Exchange Request

Figure 6.3: SKDP client exchange request.

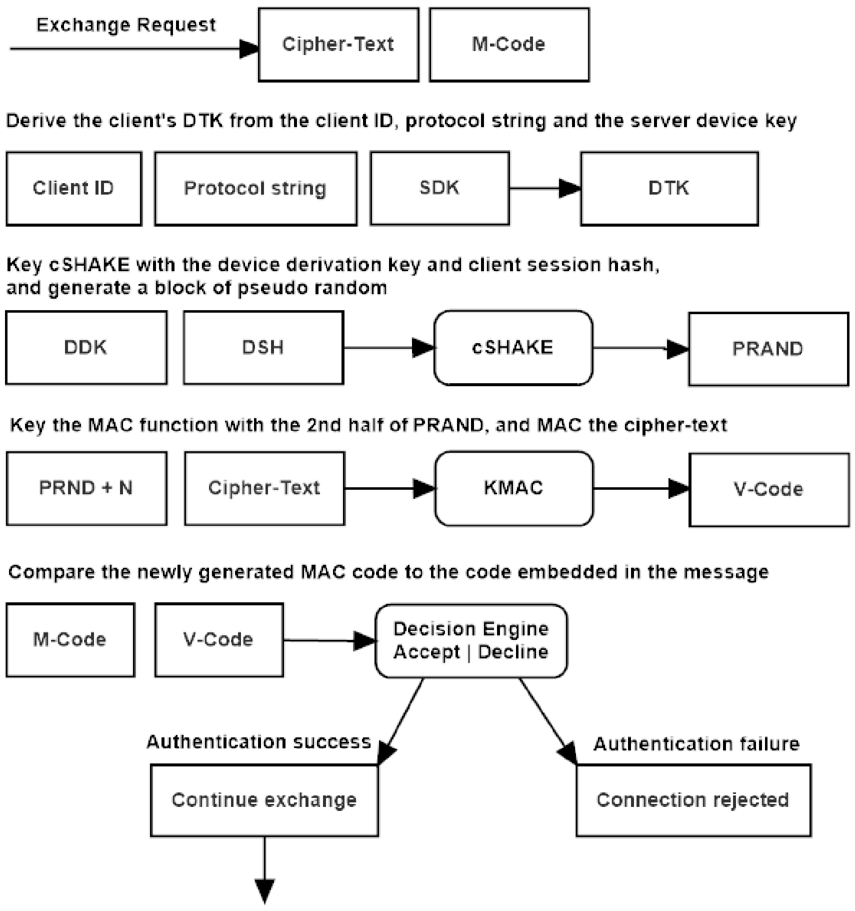
The client receives the **connect response** message from the server, and hashes the server's identity array, protocol configuration string, and random token to create the server session cookie.

The client generates a random device token-key.

The client combines the device token-key, and the client's session cookie to key cSHAKE. The client then generates the symmetric stream cipher's key and nonce from the keyed cSHAKE instance, and initializes the transmit cipher for channel-1.

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 (change in 1.1), 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


Exchange Request
Cipher-Text
M-Code
Derive the client's DTK from the client ID, protocol string and the server device key
Client ID
Protocol string
SDK
DTK
Key cSHAKE with the device derivation key and client session hash, and generate a block of pseudo random
DDK
DSH
cSHAKE
PRAND
Key the MAC function with the 2nd half of PRAND, and MAC the cipher-text
PRND + N
Cipher-Text
KMAC
V-Code
Compare the newly generated MAC code to the code embedded in the message
M-Code
V-Code
Decision Engine Accept | Decline
Authentication success
Authentication failure
Continue exchange
Connection rejected

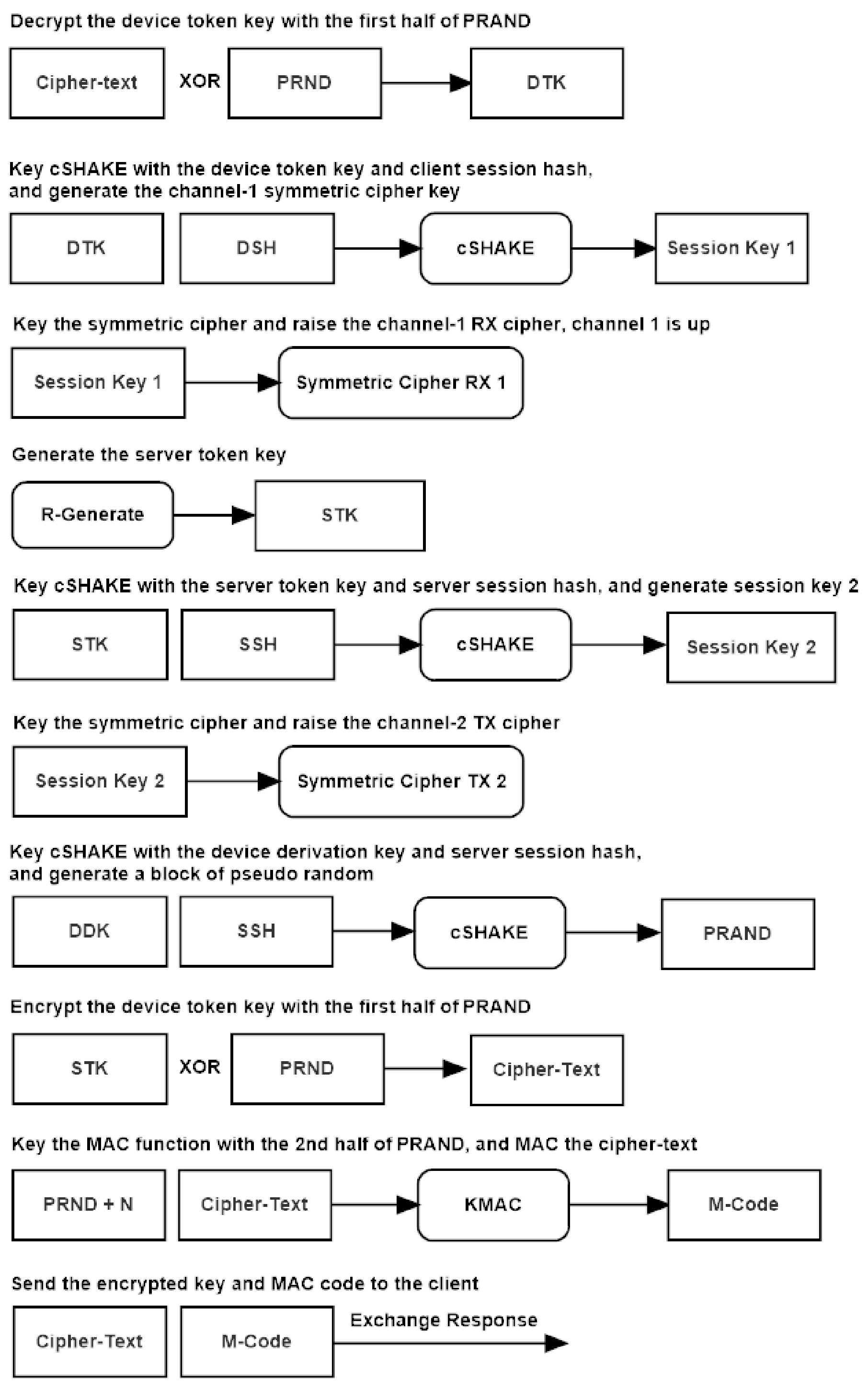

Decrypt the device token key with the first half of PRAND
Cipher-text
XOR
PRND
DTK
Key cSHAKE with the device token key and client session hash, and generate the channel-1 symmetric cipher key
DTK
DSH
cSHAKE
Session Key 1
Key the symmetric cipher and raise the channel-1 RX cipher, channel 1 is up
Session Key 1
Symmetric Cipher RX 1
Generate the server token key
R-Generate
STK
Key cSHAKE with the server token key and server session hash, and generate session key 2
STK
SSH
cSHAKE
Session Key 2
Key the symmetric cipher and raise the channel-2 TX cipher
Session Key 2
Symmetric Cipher TX 2
Key cSHAKE with the device derivation key and server session hash, and generate a block of pseudo random
DDK
SSH
cSHAKE
PRAND
Encrypt the device token key with the first half of PRAND
STK
XOR
PRND
Cipher-Text
Key the MAC function with the 2nd half of PRAND, and MAC the cipher-text
PRND + N
Cipher-Text
KMAC
M-Code
Send the encrypted key and MAC code to the client
Cipher-Text
M-Code
Exchange Response

Figure 6.4: SKDP server exchange response.

The server verifies the UTC timestamp in the packet header (change in 1.1), 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 (change in 1.1) 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 device token-key, and the device session cookie to key cSHAKE. The server then generates the symmetric stream cipher's key and nonce, and initializes the receive symmetric cipher instance for channel-1.

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

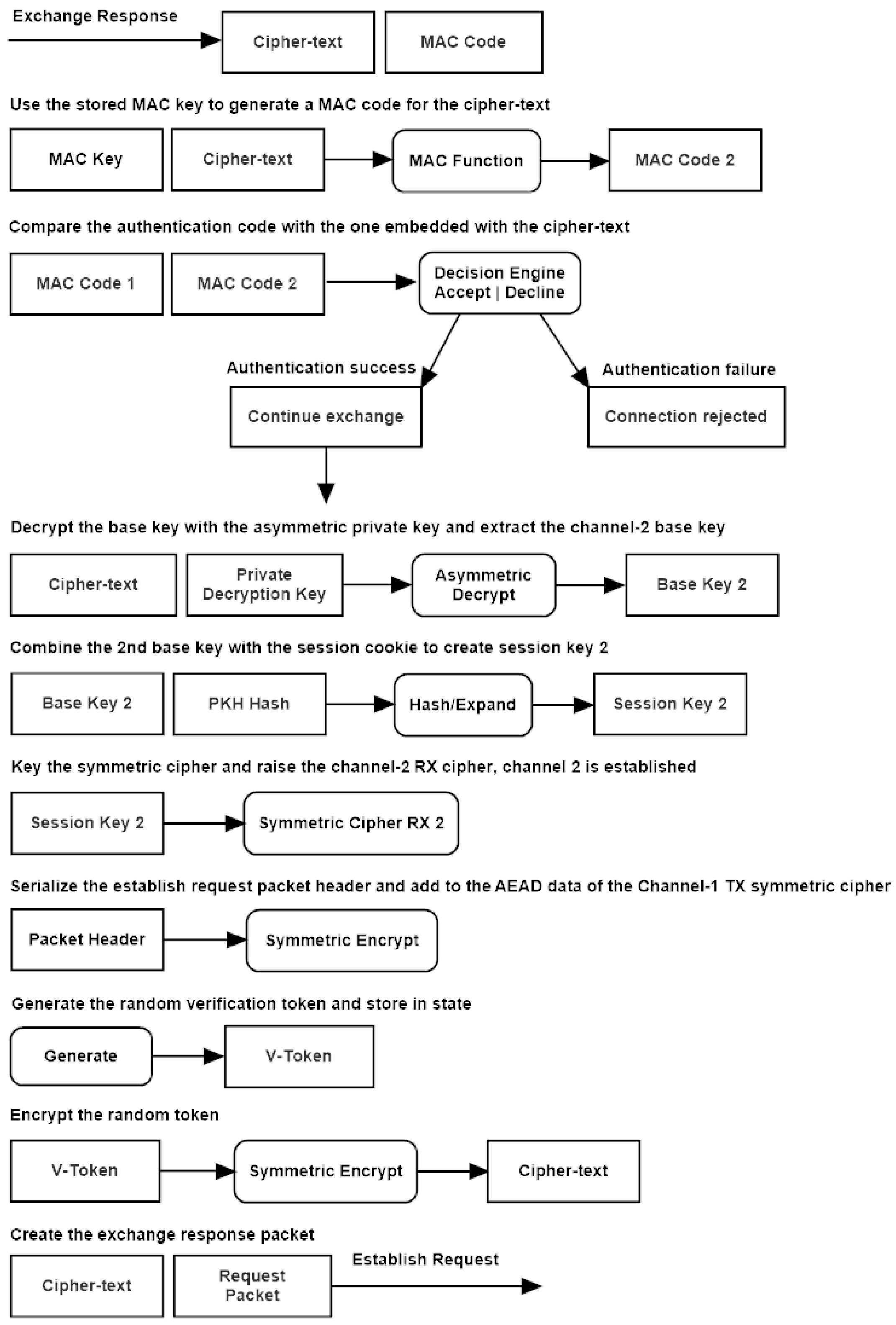

Exchange Response
Cipher-text
MAC Code
Use the stored MAC key to generate a MAC code for the cipher-text
MAC Key
Cipher-text
MAC Function
MAC Code 2
Compare the authentication code with the one embedded with the cipher-text
MAC Code 1
MAC Code 2
Decision Engine Accept | Decline
Authentication success
Authentication failure
Continue exchange
Connection rejected
Decrypt the base key with the asymmetric private key and extract the channel-2 base key
Cipher-text
Private Decryption Key
Asymmetric Decrypt
Base Key 2
Combine the 2nd base key with the session cookie to create session key 2
Base Key 2
PKH Hash
Hash/Expand
Session Key 2
Key the symmetric cipher and raise the channel-2 RX cipher, channel 2 is established
Session Key 2
Symmetric Cipher RX 2
Serialize the establish request packet header and add to the AEAD data of the Channel-1 TX symmetric cipher
Packet Header
Symmetric Encrypt
Generate the random verification token and store in state
Generate
V-Token
Encrypt the random token
V-Token
Symmetric Encrypt
Cipher-text
Create the exchange response packet
Cipher-text
Request Packet
Establish Request

Figure 6.5: SKDP client establish request.

The client verifies the packet headers UTC valid-time (change in 1.1), 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 (change in 1.1) 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 adds it to the additional data parameter of the channel-1 AEAD symmetric stream cipher. The client then encrypts the verification token, and adds the cipher-text to the **establish response** message, and sends it to the server.

6.6 Establish Response

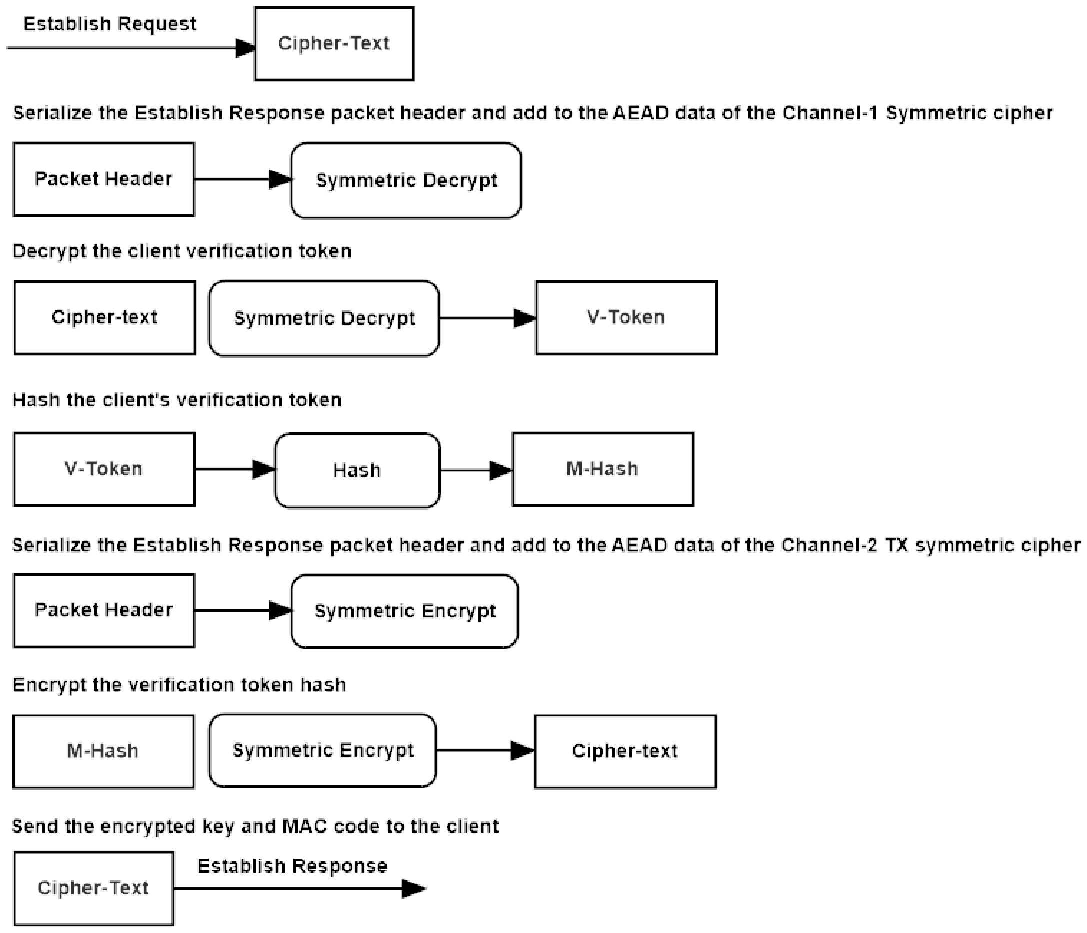

Establish Request
Cipher-Text
Serialize the Establish Response packet header and add to the AEAD data of the Channel-1 Symmetric cipher
Packet Header
Symmetric Decrypt
Decrypt the client verification token
Cipher-text
Symmetric Decrypt
V-Token
Hash the client's verification token
V-Token
Hash
M-Hash
Serialize the Establish Response packet header and add to the AEAD data of the Channel-2 TX symmetric cipher
Packet Header
Symmetric Encrypt
Encrypt the verification token hash
M-Hash
Symmetric Encrypt
Cipher-text
Send the encrypted key and MAC code to the client
Cipher-Text
Establish Response

Figure 6.6: SKDP server establish response.

The server serializes the **establish request** packet header and adds it to the additional data parameter of the AEAD symmetric stream cipher, including the UTC valid-time, it then authenticates and decrypts the cipher-text. The server hashes the message, adds the establish response packet header to the cipher's additional data, and encrypts the message hash. The server then sends the cipher-text back to the client in the establish response packet.

6.7 Establish Verify

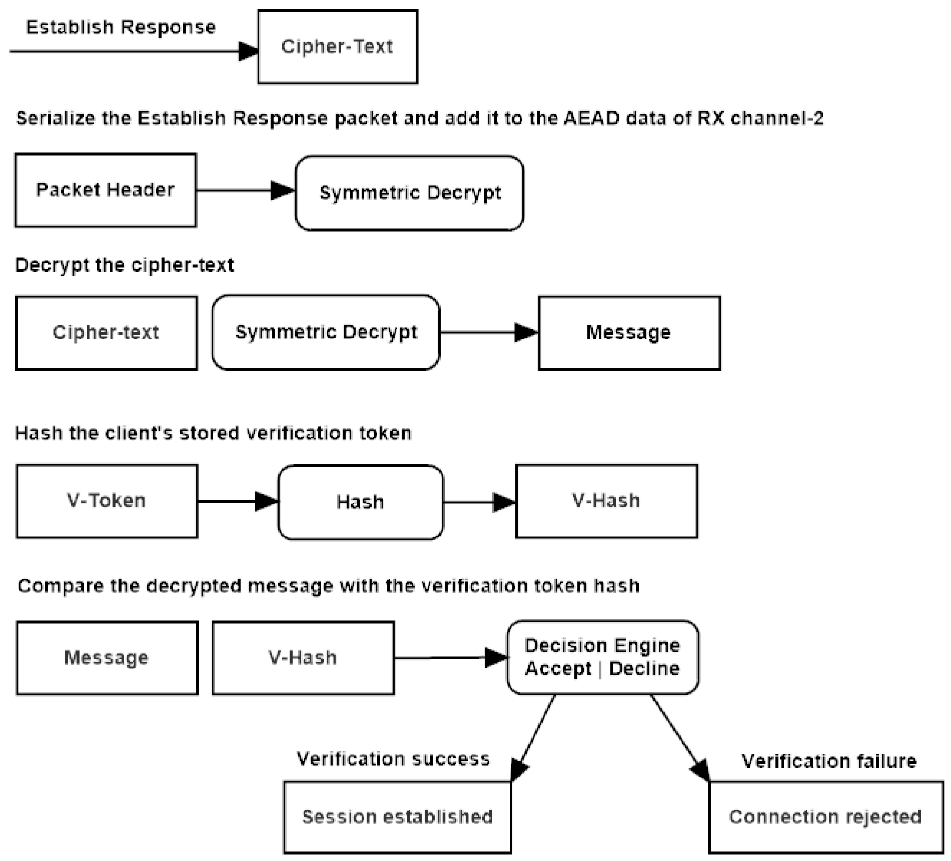


Figure 6.7: SKDP client establish verify.

The client serializes the packet header and adds it to the additional data parameter of the channel-2 RX cipher, including the UTC valid-time (change in 1.1). The client then authenticates and decrypts the cipher-text sent in the establish response message sent by the server. The client hashes the stored copy of the verification token, and compares it to the message for equality. Upon successful decryption and verification of the message, the client raises its session established flag, and is ready to process data.

7: Formal Description

Legend:

$\leftarrow \leftrightarrow \rightarrow$	- 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:

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

It generates a secret random device token key:

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

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

$ke, km \leftarrow \mathrm{Exp}(dsh, ddk)$

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 (version 1.1) to the MAC along with the ciphertext.

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

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

$k, n \leftarrow \mathrm{Exp}(dsh, dtk)$

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

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:

$ddk \leftarrow \mathrm{H}(cid \,||\, sdk)$

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

$ke, km \leftarrow \mathrm{Exp}(dsh, ddk)$

The server verifies that the UTC valid-time in the packet header is within the timeout threshold (version 1.1). The server verifies the MAC code attached to the client's message:

$M_{km}(sh \| etk)$?= true = *mtag :* 0

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

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

$k, n \leftarrow \mathrm{Exp}(dsh, dtk)$

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

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:

$ke, km \leftarrow \mathrm{Exp}(ssh, ddk)$

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

$etk \leftarrow E_{ke}(rtk)$

The server adds the serialized packet header, which includes the packet creation time and sequence number (version 1.1) to the MAC along with the ciphertext.

$mtag \leftarrow M_{km}(sh \| etk)$

The server initializes the transmit channel cipher key:

$k, n \leftarrow \mathrm{Exp}(ssh, rtk)$

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

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

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

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.

$ke, km \leftarrow \mathrm{Exp}(ssh, ddk)$

The client verifies the UTC valid-time is within the timeout threshold (version 1.1) The client verifies the mac code appended to the client message.

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

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.

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

$k, n \leftarrow \mathrm{Exp}(ssh, stk)$

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

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

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

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

$cpt \leftarrow \mathrm{E}_{k}(vtok)$

$\mathrm{C} \, \{ \, cpt \, \} \rightarrow \mathrm{S}$

7.6 Establish Response:

The server authenticates and decrypts the message.

$msg \leftarrow \text{-}\mathrm{E}_{k}(cpt)$

The server hashes the decrypted message.

$mhash \leftarrow \mathrm{H}(msg)$

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.

$cpt \leftarrow \mathrm{E}_{k}(mhash)$

$\mathrm{S}\{ \, cpt \, \} \rightarrow \mathrm{C}$

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.

$msg \leftarrow \text{-}\mathrm{E}_{k}(cpt)$

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.

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

7.8 Transmission:

The host, client or server, transmitting a message, first sets the UTC packet creation time in the packet header (version 1.1), then serializes the packet header and adds it to the symmetric ciphers associated data parameter. The host then encrypts the message, updates the MAC function with the cipher-text, and appends a MAC code to the end of the cipher-text. All of this is done by using the RCS stream cipher's AEAD and encryption functions. The serialized packet header, including the message size, protocol flag, packet creation-time, and sequence number, are added to the MAC state through the additional-data parameter of the authenticated stream cipher RCS. This unique data is added to the MAC function with every packet, along with the encrypted cipher-text.

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

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

The packet UTC creation time is checked to see if it is within the packet valid-time threshold, if it is not, the function returns false and the packet is discarded. The packet is decrypted by serializing the packet header and adding it to the MAC state, then finalizing the MAC on the cipher-text and comparing the output code with the code appended to the cipher-text. If the code matches, the cipher-text is decrypted, and the message passed up to the application. If this check fails, the decryption function returns false, returns an empty message array, and must be handled by the application.

$m \leftarrow -\mathrm{E}_k(cpt)$?= 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_ERROR_STRINGS** is a static string-array containing SKDP configuration string.

Data Set	Purpose
SKDP_CONFIG_STRING	The SKDP configuration string

Table 8.1a SKDP configuration string.

The **SKDP_CONFIG_STRING** is a static string-array containing SKDP error descriptions, used in 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

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	0x0D	The SKDP packet header size
SKDP_KEEPALIVE_STRING	0x14	The keep alive string size
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	0x40D	The maximum message size (may exceed mtu)
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	0x20/0x40	The size of the mac function output
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_CONNECT_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	0x10	The number of error strings
SKDP_ERROR_STRING_WIDTH	0x80	The length of each error string

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

Populate a packet structure with an error message.

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

Packet Error Message

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.

```
void skdp_packet_header_deserialize(const uint8_t* header, 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.

```
void skdp_stream_to_packet(const uint8_t* pstream, skdp_packet* packet)
```

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	RCS state	Variable	The receive channel cipher state
txcpr	RCS 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 message and copy it to the message output.

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

Encrypt Packet

Encrypt a message and build an output packet.

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

Ratchet Response

A ratchet response sends an encrypted token to the client and re-keys the channel. This is useful in a static tunnel configuration, where based on up time or data transferred, additional entropy can be injected into the system on demand..

skdp_errors skdp_server_ratchet_response(skdp_server_state* ctx, skdp_packet* packetout)

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	RCS state	Variable	The receive channel cipher state
txcpr	RCS 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* 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

A ratchet request asks the server fo a token key on demand. This is useful in a static tunnel configuration, where based on up time or data transferred, additional entropy can be injected into the system on demand.

skdp_errors skdp_client_ratchet_request(skdp_client_state* ctx, skdp_packet* packetout)

10. Cryptanalysis of the Symmetric-Key Distribution Protocol

10.1 Threat Model & Target Properties

An **active, adaptive adversary 𝔄** can

- control the network (eavesdrop, modify, replay, reorder, inject, drop);
- compromise any long-term keys: master (MDK), server (SDK), device (DDK);
- read every piece of client or server RAM, except the momentarily-erased one-time tokens;
- obtain chosen-message MAC tags (KMAC) and chosen-ciphertext outputs of the RCS AEAD;
- wield a post-session quantum computer.

SKDP aims to provide:

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.
Forward secrecy (FS)	Compromise of any long-term key after session teardown reveals no past TKs.
Predictive resistance (PR)	Compromise of long-term keys before the next refresh reveals no future TKs.
Replay & downgrade defence	UTC field + seq# inside MAC/AAD rejects stale or mixed-version packets. SKDP Specification

10.2 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 proof sketch TK = SHAKE(ddk ‖ dsh) is PRF-secure; *etk* leaks no min-entropy because Keccak behaves as random permutation; thus TK is indistinguishable from random unless $\mathfrak{A}$ breaks Keccak or extracts DDK.

10.3 Data-Phase Confidentiality & Integrity

$TK_i = SHAKE_\rho(TOK \parallel DDK)$,

$cpt = RCS\frac{AEAD}{TKi}(AAD = header, m)$,

$r = KMAC(TK_{i+1}, cpt)$

IND-CPA RCS is a wide-block cipher keyed with uniform TK_i; the only feasible attack is exhaustive $2^{256}/2^{512}$ search.

- **INT-CTXT** Tags use *independent* TK_{i+1}; forging requires UF-CMA break of KMAC.
- **FS** TK_i erased immediately; later MDK/SDK leak cannot reconstruct it.
- **PR** Future TOK requires fresh random token from peer → unattainable pre-compromise.

10.4 Replay, Downgrade & MitM Resistance

- **UTC + seq#** (21-byte header) is MAC'd during handshake and passed as AEAD AAD during data-phase; any off-window (>|60 s|) or out-of-order packet is dropped.
- **Configuration string (cnf) hash** is folded into every session-cookie and into the tunnel KDF, pinning the algorithm suite.
- **Full-transcript MACs** mean the earliest point a MitM can inject undetected is *before Connect*—but Connect contains no secrets.

10.5 Computational & 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 + 32-byte tag	21-byte header + 64-byte tag

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

10.6 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 SHAKE/KMAC + RCS	SHAKE/KMAC + XOR	AES + SHA-1	X25519 + AES-GCM
FS	✓ (ephemeral TK)	✓	✗	✓
PR	✓	✓	✗	✓
Root-key compromise impact	Past safe, future vulnerable	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 + seq#, MACed	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).

10.7 Residual Gaps & Hardening Advice

1. **SDK capture ⇒ future sessions exposed**: mandate periodic SDK rotation (bind epoch into cnf).
2. **12-round Keccak option** reduces margin; default to 24 rounds for ≥256-bit security.
3. **RCS audit**: commission external cryptanalysis or ship AES-GCM fallback for regulated sectors.
4. **Clock drift**: for battery-powered devices add nonce-based freshness when UTC unavailable.
5. **Side-channel hygiene**: mask Keccak state in embedded HSMs; constant-time XOR token decrypt.

10.8 Conclusion

Under standard assumptions (Keccak behaves as PRP/PRF and KMAC is UF-CMA), SKDP v1.1 achieves mutual authentication, IND-CPA + INT-CTXT confidentiality, forward secrecy and predictive resistance, while resisting MitM, replay and downgrade attacks—even against quantum-capable adversaries—at a cost suitable for low-power appliances. With the recommended key-rotation and side-channel safeguards, SKDP stands as a practical, post-quantum-ready successor to legacy DUKPT-type schemes.

9: 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 use 256-bit or 512-bit symmetric cipher keys. The authenticated symmetric stream cipher RCS, is based on Rijndael, the symmetric cipher used in AES. It has double the internal block size (Rijndael-256), the transformation function is to the Rijndael specification, but the key schedule, used to generate a large set of ‘round keys’ from a small input cipher key, has been changed from the differentially-weak native expansion function to the strong Keccak cSHAKE function. The number of transformation rounds has been increased from 14 used by AES-256 to 22 rounds, and 30 rounds when using the 512-bit key option. These changes strongly mitigate most attacks against AES, as well as setting the number of transformation rounds to at least ***2n*** the best-known attack. RCS uses KMAC the Keccak MAC function to authenticate cipher-text, making RCS oner of the strongest symmetric ciphers available in the world today.

Version 1.1 adds a significant change by introducing an anti-replay attack defence. By adding a packet creation time field to the header, and using the MAC function in the key exchange to guarantee the header integrity, and the AEAD functionality of the RCS stream cipher, the packet creation time can be checked and verified. This strongly mitigates attacks that leverage re-use of a packet, replaying elements of the key exchange or symmetric tunnel. This added attack defense, further reduces the attack surface of the protocol, and guarantees packet headers are not altered in transit.