Symmetric Key Distribution Protocol

SKDP Integration Guide

Revision: 1.0
Date: October 15, 2025

Introduction

The **Symmetric Key Distribution Protocol (SKDP)** is a next-generation post-quantum secure messaging and key management system developed by **Quantum-Resistant Cryptographic Solutions** (QRCS).
Unlike traditional key-exchange schemes that rely on public-key primitives, SKDP uses only symmetric cryptography built upon the **Keccak** family (SHA3, **cSHAKE** and **KMAC**) and QRCS's **RCS** stream cipher.
It enables scalable, authenticated, duplex communication channels with forward secrecy and predictive resistance, eliminating dependence on fragile public-key infrastructures.

This guide provides practical instructions for integrating SKDP into payment networks, cloud applications, industrial control systems, and IoT devices.
It summarizes the SKDP architecture, explains its cryptographic primitives, details the API exposed by the reference implementation, and offers step-by-step integration guidelines.

Protocol Overview

Motivation

Conventional key-exchange mechanisms depend on asymmetric cryptography (RSA, ECC, or lattice-based) which are vulnerable to quantum computers and impose significant operational complexity.
SKDP demonstrates that **strong symmetric cryptography**, when architected around robust key derivation, authentication, and lifecycle management, can replace asymmetric infrastructures altogether.
Removing PKI simplifies deployment, reduces computational overhead and certificate maintenance, and enhances long-term security.

Architecture

SKDP's handshake is divided into three authenticated stages: **Connect, Exchange**, and **Establish**, to derive session-unique keys without ever transmitting a reusable secret.

Each endpoint independently generates random tokens and uses cSHAKE and KMAC to derive **duplexed cipher keys**: one for transmit and one for receive.
The handshake produces a symmetric **transmit key** and **receive key** per direction, ensuring **bidirectional forward secrecy**.

Hierarchical Key Derivation

SKDP implements a tiered model that scales from a single master key to potentially billions of devices:

1. **Master Key (MDK):** Root key held by a central authority.
2. **Server Key (SDK):** Derived from the master key and bound to a server identity.
3. **Device Key (DDK):** Derived from a server key and bound to a device identity.
4. **Token Keys:** Ephemeral secrets exchanged during the handshake; each token derives per-session cipher and MAC keys.

This hierarchy allows institutions to embed keys in cards or devices at manufacturing time and revoke or refresh branches independently.
Tokens are authenticated and encrypted using the derivation keys, ensuring that even compromise of long-term keys does not leak session secrets.

Packet Format and Replay Protection

All SKDP messages are encapsulated in a 21-byte header followed by a variable-length payload. The header includes:

- **Flag (1 byte):** Indicates message type (connect request/response, exchange request/response, establish request/response, encrypted data, keep-alive, error, etc.).
- **Sequence number (8 bytes):** Monotonically increasing packet counter per direction.
- **UTC timestamp (8 bytes):** Seconds since epoch; used to detect replayed packets.
- **Message length (4 bytes):** Size of the payload.

The timestamp and sequence are included in the KMAC/AEAD authentication to reject stale or reordered packets.
Packets outside the valid time window (default 60 seconds) are dropped to prevent replay.

Anti-Replay

Version 1.1 of SKDP introduces an anti-replay mechanism: each packet carries a UTC timestamp, which is MAC'd during the key exchange and included in AEAD for tunnel data.

Endpoints verify that the timestamp is within a 60-second window and that sequence numbers increment correctly.

Security Model and Post-Quantum Posture

SKDP's security is defined under IND-CPA, INT-CTXT, UF-CMA and forward-secrecy models. Key properties include:

- **Mutual authentication:** Both client and server must possess valid derivation keys; MAC failures abort the handshake.
- **Confidentiality & integrity:** Payloads are encrypted with the wide-block **RCS** cipher in AEAD mode and authenticated with KMAC.
- **Forward secrecy (FS):** Session keys are erased immediately after use; compromise of long-term keys yields no past session keys.
- **Predictive resistance (PR):** Even future compromise of derivation keys does not reveal future token keys.
- **Replay & downgrade defense:** UTC timestamps and configuration strings are MAC-bound to prevent replays or algorithm substitution.
- **Quantum resilience:** All security derives from Keccak-family primitives and RCS; 256-bit configurations offer 2^{256} security margin.

SKDP Structures

SKDP defines a set of key and state structures accessible via skdp.h.
Key sizes vary by security level (default 256 bits).
Important structures include:

Master Key (skdp_master_key)

Holds the **master derivation key (MDK)**, key identity (kid), and expiration timestamp.
Master keys are used only by trusted authorities to derive server keys.

Server Key (skdp_server_key)

Stores the **server derivation key (SDK)**, server identity string (kid), and expiration.
This key is used by servers to derive device keys and to participate in the handshake.

Device Key (skdp_device_key)

Represents the **device derivation key (DDK)**, device identity, and expiration.
Embedded in client devices or tokens; used to derive session keys during handshake.

Client State (skdp_client_state)

Maintains all variables required for client operations:

- **DDK** (device derivation key), **dsh** (device session hash), **kid**, **ssh** (server session hash), expiration, receive/transmit packet counters, and flags.
- **RCS cipher contexts**: rxcpr (receive) and txcpr (transmit).
- Updated during handshake; erased on session close.

Server State (skdp_server_state)

Similar to client state but holds the **server derivation key (SDK)** and device identity strings. Maintains RCS cipher contexts, expiration, sequence counters, and flags.

Network Packet (skdp_network_packet)

Represents an SKDP packet with fields for flag, sequence, timestamp, payload length and pointer to message buffer.
Used by both client and server for handshake and encrypted data exchange.

API Summary

The reference implementation, written in MISRA-compliant C, exposes a well-structured API. Common naming conventions include verbs like **generate**, **initialize**, **connect**, **listen**, **encrypt**, **decrypt**, **serialize**, **deserialize**, **send_error**, **close**.
Functions return an skdp_errors enumeration or bool to indicate success/failure.

Key Management Functions (skdp.h)

- **skdp_generate_master_key(mkey, kid):** Derives a new master key using the provided key identity.
- **skdp_generate_server_key(skey, mkey, kid):** Derives a server key from a master key.
- **skdp_generate_device_key(dkey, skey, kid):** Derives a device key from a server key.
- **skdp_serialize_* / skdp_deserialize_* :** Convert master, server or device keys to/from byte arrays for storage or transmission.
- **skdp_error_to_string(error):** Return a human-readable description of an error code.
- **Packet helpers:** skdp_packet_header_serialize, skdp_packet_header_deserialize, skdp_packet_to_stream, skdp_stream_to_packet, skdp_packet_set_utc_time,

skdp_packet_time_valid provide serialization and timestamp management for skdp_network_packet.

Client Functions (skdpclient.h)

Initialization and Connection:

- **skdp_client_initialize(ctx, ckey):** Initialize the client state with the device key; set cipher contexts and sequence counters.
- **skdp_client_connect_ipv4(ctx, sock, address, port) / skdp_client_connect_ipv6(...):** Connect to a server over IPv4/IPv6, perform the three-stage handshake and populate ctx and sock.
- **skdp_client_connection_close(ctx, sock, error):** Send an error notification and tear down the session.

Messaging:

- **skdp_client_encrypt_packet(ctx, message, msglen, packetout):** Encrypt a plaintext message into an SKDP packet; fills header fields and MAC tag.
- **skdp_client_decrypt_packet(ctx, packetin, message, msglen):** Decrypt a received packet and output the plaintext.
- **skdp_client_send_error(sock, error):** Send an error code to the remote host.

Server Functions (skdpserver.h)

Initialization and Listening:

- **skdp_server_initialize(ctx, skey):** Initialize the server state with the server key; set cipher contexts and counters.
- **skdp_server_listen_ipv4(ctx, sock, address, port) / skdp_server_listen_ipv6(...):** Listen on a socket and perform the handshake with a client; returns an error code on failure.
- **skdp_server_connection_close(ctx, sock, error):** Close the connection and free resources.

Messaging:

- **skdp_server_encrypt_packet(ctx, message, msglen, packetout):** Encrypt a plaintext message to send to the client.
- **skdp_server_decrypt_packet(ctx, packetin, message, msglen):** Decrypt an incoming packet from the client.

- **skdp_server_send_keep_alive(kctx, sock):** Send periodic keep-alive messages.

Implementation Guidance

Key Generation and Provisioning

1. **Generate a master key** for your institution with a unique key identifier (kid), e.g., using skdp_generate_master_key().
   Store the master key securely offline; it is used only to derive branch or server keys.
2. **Generate server keys** for each server or branch by deriving from the master key (skdp_generate_server_key()) and assign a server identifier (SID).
3. **Generate device keys** for each client device using the relevant server key (skdp_generate_device_key()).
   Device keys are embedded in devices or stored on tokens (smartcards, hardware modules).
4. **Serialize/deserialize** keys using the provided functions for storage or transmission over management channels.

Client Integration Steps

1. **Initialize client state:** Call skdp_client_initialize(&client_state, &device_key) before any network operations.
   This copies the device derivation key, zeroes counters, and prepares cipher contexts.
2. **Connect to server:** Use skdp_client_connect_ipv4() or skdp_client_connect_ipv6() with a socket pre-configured for IPv4 or IPv6.
   The function performs the three-stage handshake:
   - **Connect Request:** Send kid, configuration string and a random token; compute device session hash dsh.
   - **Connect Response:** Receive server's id, configuration and token; compute server session hash ssh.
   - **Exchange:** Send an encrypted token key; derive transmit cipher key and MAC keys; authenticate with KMAC and include serialized header to defend replay.
   - **Exchange Response:** Server validates MAC, decrypts token, derives device derivation key and receive cipher key.

    - **Establish:** Client verifies server's token hash; both sides echo verification tokens to confirm both cipher keys are correct.

3. **Transmit data:** Call skdp_client_encrypt_packet() to encrypt application data into SKDP packets.
   Send the resulting packet via your network stack.
   Sequence numbers and UTC timestamps are set automatically.
4. **Receive data:** Upon receiving an SKDP packet, call skdp_client_decrypt_packet() to decrypt the payload and verify the MAC and timestamp.
5. **Close session:** When finished, call skdp_client_connection_close() to send an error code (if needed) and release resources.

Server Integration Steps

1. **Initialize server state:** Call skdp_server_initialize(&server_state, &server_key) to copy the server derivation key and clear counters.
2. **Listen for clients:** Use skdp_server_listen_ipv4() or skdp_server_listen_ipv6() to bind a socket to the appropriate port (default 2201) and handle handshake messages.
3. **Handle handshake:** The server side of the handshake mirrors the client process:
    - **Connect Response:** Verify client configuration and kid; generate server session hash and random token.
    - **Exchange Response:** Verify the MAC and decrypt the token using the server's derivation key; derive receive cipher keys.
    - **Establish Response:** Echo verification tokens and confirm channel keys.
4. **Encrypt/decrypt data:** Use skdp_server_encrypt_packet() and skdp_server_decrypt_packet() to send and receive application data.
5. **Keep alive:** Periodically call skdp_server_send_keep_alive() to send keep-alive messages and update sequence numbers.
6. **Close connection:** Call skdp_server_connection_close() to gracefully terminate the session, sending an error code if necessary.

Key Configuration Strings

SKDP supports different **configuration strings** identifying security level and cipher options. Default configuration uses 256-bit keys and RCS encryption.

To select 512-bit security (if compiled with SKDP_PROTOCOL_SEC256 undefined), adjust the configuration string accordingly and ensure both peers agree.

Error Handling

The skdp_errors enumeration defines error codes such as skdp_error_authentication_failure, skdp_error_random_failure, skdp_error_packet_unsequenced, skdp_error_key_expired and skdp_error_unknown_protocol.
Functions return an error code; applications should check the return value and log or send error messages via skdp_client_send_error() or skdp_server_send_error().

Building and Integration Environment

SKDP depends on the **QSC** cryptographic library.
It builds on Windows (MSVC), macOS, and Linux using CMake.
Ensure that both QSC and SKDP are configured to use the same instruction set (AVX, AVX2, AVX-512) for optimal performance.
Projects for Visual Studio and Eclipse are provided in the repository; follow the README instructions to set include directories and references.
For embedded and IoT platforms, SKDP can be compiled with minimal features: disable RCS encryption by defining SKDP_USE_RCS_ENCRYPTION at compile time and supply an alternative cipher if required.

Integration Scenarios

Payment Networks and Financial Systems

SKDP is designed as a **post-quantum successor to symmetric-only tunneling protocols**.
It secures ATM/POS terminals and transaction networks with forward secrecy and zero PKI dependency.
Integration steps:

1. Provision each card or terminal with a device key during manufacturing (DDK).
2. Equip transaction servers with server keys derived from a master.
3. On each transaction, the terminal connects to the server via SKDP, performs the handshake, derives session keys, and encrypts transaction data.
4. Servers verify packets, process transactions, and respond via SKDP.

Cloud Services and Enterprise Applications

SKDP can replace VPN tunnels or TLS in private cloud environments.
Deploy SKDP servers as access gateways and embed device keys in micro-services or client applications.
Leverage keep-alive to maintain long-lived connections with automatic key refresh.
Combine SKDP with QSMP or QSTP to integrate certificate-anchored or messaging-oriented functionality.

Industrial Control Systems and SCADA

Industrial networks require deterministic timing, anti-replay enforcement, and zero-downtime key rotation.
SKDP's symmetric design and small memory footprint (per session < 4 kB) make it ideal for SCADA controllers, sensors and actuators.
Embed device keys in PLCs or control units; run SKDP servers on gateway nodes to handle secure telemetry.
Use the keep-alive mechanism to monitor link health.

IoT and Embedded Devices

With a lightweight footprint and deterministic key management, SKDP scales to billions of constrained devices.
Device keys can be imprinted into microcontrollers at production.
Because SKDP avoids expensive asymmetric operations, energy consumption remains low, enabling long operational life on battery-powered devices.
In connected fleets, branch keys allow grouping devices; compromised devices can be revoked by regenerating the server key and associated device keys.

Best Practices and Security Considerations

- **Synchronize clocks:** Anti-replay checks rely on synchronized UTC time; ensure client and server clocks are within the configured threshold.
- **Secure random number generation:** Token keys and derivation keys depend on high-quality randomness; use hardware RNG or a strong system provider (e.g., qsc_acp_generate()) where available.
- **Key expiration and rotation:** Each key carries an expiration timestamp; enforce periodic refresh of master, server and device keys.
  Consider injecting additional entropy via QSMP or other PQC channels as recommended.
- **Handle errors gracefully:** Always check return codes and send error notifications; drop sessions on authentication failures or out-of-sequence packets.

- **Audit logs and counters:** Monitor sequence numbers and timestamps to detect anomalies.
- **Parallelization:** SKDP uses per-connection cipher states; design your application to handle hundreds of thousands of concurrent tunnels.
- **Combine with post-quantum tunnels:** For additional assurance, run SKDP over QSTP or QSMP to layer symmetric and certificate-based protections.

Conclusion

The **Symmetric Key Distribution Protocol (SKDP)** offers a robust, scalable, and quantum-resistant alternative to traditional symmetric key-distribution schemes.
By relying exclusively on symmetric primitives: **cSHAKE** for derivation, **KMAC** for authentication, and **RCS** for encryption, SKDP provides forward secrecy, predictive resistance and replay protection without PKI overhead.
Its hierarchical key model allows secure management of millions of devices with minimal administrative burden, while the API exposes a clear, consistent set of functions for key generation, connection management, encryption, decryption, and error handling.
Through careful implementation and adherence to the best practices outlined here, developers can integrate SKDP into payment systems, cloud infrastructures, industrial networks and IoT ecosystems to achieve sovereign, post-quantum security for decades to come.