The Design and Analysis of the Symmetric Authenticated Tunneling Protocol

John G. Underhill

Quantum Resistant Cryptographic Solutions Corporation

Abstract. SATP is a symmetric channel protocol designed for post quantum environments where the cost of asymmetric primitives and key encapsulation mechanisms is prohibitive. The protocol uses a deterministic hierarchy of SHAKE based derivations to expand a root secret into branch keys and provisioned client keys, which permits scalable deployment on constrained systems without maintaining per session server state. SATP establishes a secure channel through a two stage exchange that authenticates the server using a hash of provisioned secrets and derives channel keys with a single SHAKE based expansion. All encrypted packets use an authenticated encryption scheme with associated data that binds header fields to each ciphertext. This paper provides a complete engineering description of the protocol derived from the reference implementation, a formal specification of its key hierarchy and message flow, and a security analysis based on game based definitions of channel authenticity, confidentiality, ciphertext integrity, and replay resistance. The analysis includes explicit reductions to the security of SHAKE and the underlying authenticated encryption scheme. The paper also compares SATP with established symmetric trust protocols such as Kerberos, TLS 1.3 in pre shared key mode, and 5G AKA, and explains the use case for SATP in constrained post quantum environments where deterministic symmetric key hierarchies provide practical and efficient security.

1 Introduction

1.1 Context and Motivation

Symmetric trust infrastructures form the basis of many long lived secure systems where endpoints are provisioned with static secrets during manufacturing or enrollment. Such systems include industrial control networks, embedded and aerospace platforms, and distributed sensor deployments. These environments frequently operate under strict computational and memory constraints, and many lack the hardware acceleration or entropy sources required for post quantum public key cryptography. In such settings, protocols based on key encapsulation mechanisms or authenticated key exchange can impose prohibitive overhead and code size. A symmetric alternative with predictable cost and deterministic behavior is often more practical.

SATP is designed for these scenarios. It provides a channel establishment mechanism that relies entirely on symmetric primitives while still delivering strong authentication and confidentiality guarantees. The protocol uses a hash based hierarchy to derive branch and device keys from a root secret, allowing large deployments to be organized without maintaining a central online key distribution infrastructure. Channel keys are derived deterministically from provisioned device keys and a session nonce, and each session is authenticated through a server validation hash computed over device specific secrets. This approach avoids the need for asymmetric operations while ensuring that only devices with correct provisioned material can establish a secure channel.

The purpose of this paper is to present the design and formal analysis of SATP. It consolidates the engineering implementation, the protocol specification, and the mathematical model into a single rigorous study. The goal is not to critique or break the protocol, but to document its structure, justify its security properties, and examine its tradeoffs relative to established symmetric trust mechanisms such as Kerberos, TLS 1.3 in pre shared key mode, and the 5G authentication and key agreement procedure. The analysis is supported by explicit security definitions and game based proofs that capture the authenticity, confidentiality, and replay resistance properties of SATP in constrained post quantum environments.

1.2 Contributions

This paper provides a complete design and analysis of the Symmetric Authentication and Tunneling Protocol (SATP). The main contributions are as follows.

- An engineering level description of SATP that is derived directly from the reference implementation. This description specifies the exact derivation of branch keys, device keys, session keys, and validation hashes, and defines the packet header and message processing rules used by the protocol.
- A formal model of the SATP symmetric key hierarchy together with security notions that capture channel authenticity, confidentiality, ciphertext integrity, replay resistance, and a limited form of forward secrecy appropriate for provisioned symmetric systems.
- Game based security proofs that reduce the authenticity and confidentiality of the protocol to the pseudo-random properties of SHAKE based key derivation and to the security of the authenticated encryption primitive used to protect the data channel.
- A comparative analysis showing how SATP relates to established symmetric trust mechanisms, including Kerberos, TLS 1.3 in pre shared key mode, and the 5G authentication and key agreement procedure. The comparison highlights the tradeoffs

between deterministic symmetric hierarchies and protocols that rely on public key operations.

- A discussion of practical considerations that influence the security of SATP in deployment, including replay windows, sequence handling, denial of service risks, and the role of key rotation epochs in limiting long term exposure.

1.3 Organization of the Paper

The paper is organized as follows. Section 2 presents an engineering level description of SATP based on the reference implementation, including the message flow and derivation of all channel keys. Section 3 provides a review of related symmetric trust protocols and positions SATP within that landscape. Section 4 introduces the cryptographic model, security assumptions, and adversarial capabilities. Section 5 defines the formal execution model and message formats used by SATP. Section 6 presents the security definitions for authenticity, confidentiality, integrity, and replay resistance. Section 7 contains the main security theorems and game based proofs. Section 8 provides a cryptanalytic evaluation of the protocol and discusses robustness considerations. Section 9 describes the operational parameters of SATP and summarizes performance characteristics. Section 10 discusses deployment considerations, key management, and application scenarios. Section 11 outlines limitations and possible directions for future work. Section 12 concludes the paper.

2 Engineering Description of SATP

This section gives an engineering level description of SATP based directly on the reference implementation found in `satp.h`, `satp.c`, `client.c`, `server.c`, and `kex.c`. The description is implementation agnostic in form, but each rule and function corresponds exactly to behavior realized in code. All constants, state transitions, and key derivations follow the reference paths with no abstraction beyond notation.

2.1 Entities and Key Hierarchy

SATP is deployed in a symmetric trust infrastructure with three tiers of long term keys that are represented explicitly in the reference implementation.

Master tier. A provisioning authority holds a master key structure

$$\mathsf{M} = (\mathsf{mdk}, \mathsf{mid}, \mathsf{exp}),$$

where `satp_master_key.mdk` is the master derivation key, `mid` is a master key identifier, and `expiration` records the validity interval. In the code, `satp_generate_master_key` samples a random value into `mdk` and copies `mid` and the expiration time. The master key is never derived from higher level material.

Server or branch tier. Each SATP server holds a server key structure

$$\mathsf{S} = (\mathsf{sdk}, \mathsf{sid}, \mathsf{ST}_c, \mathsf{exp}),$$

corresponding to `satp_server_key`. The field `sdk` is the server derivation key, `sid` is the server identifier, `stc` is a long term validation secret shared with all devices under this server, and `expiration` is inherited from the master key. The function `satp_generate_server_key` first samples `stc` using the system random generator, then derives `sdk` deterministically from the master key and the server identifier by a SHAKE based expansion

$$\mathsf{sdk} \leftarrow G_{\mathsf{srv}}(\mathsf{mdk}, \mathsf{sid}) = \text{Shake256}(\mathsf{mdk} \,\|\, \mathsf{cfg}_s \,\|\, \mathsf{sid}),$$

with the configuration string $\mathtt{cfg}_s$ taken from `SATP_CONFIG_STRING`. This matches the call to `qsc_cshake256_compute` in `satp_generate_server_key`.

Device tier. For each device, provisioning creates a device record

$$\mathsf{D}_i = (\mathsf{KTree}_i, \mathsf{kid}_i, \mathsf{ST}_c, \mathsf{exp}, \mathsf{spass}),$$

implemented as `satp_device_key`. The array `ktree` holds SATP_KEY_TREE_COUNT device keys of length `SATP_DKEY_SIZE`, `kid` is an identity string of length `SATP_KID_SIZE`, `stc` is copied from the server key and equals ST_c for that server, `expiration` is inherited from the server, and `spass` points to the memory hard image of the device authentication secret.

The function `satp_generate_device_key` derives the key tree deterministically from the server derivation key `sdk` and the device identity string `kid`. It first copies the device identity prefix `did` into `kid`, then for each index $j \in \{0, \dots, \mathsf{SATP\_KEY\_TREE\_COUNT} - 1\}$ computes

$$K_{c,i,j} \leftarrow G_{\mathsf{dev}}(\mathsf{sdk}, \mathsf{kid}_i, j) = \text{Shake256}(\mathsf{sdk} \,\|\, \mathsf{kid}_i \,\|\, j),$$

by calling `qsc_cshake256_compute` with key input `skey->sdk` and message input equal to the current `kid` value. After each derivation it increments the trailing key index bytes of `kid` with `qsc_intutils_be8increment`. Once all entries have been filled, the index bytes are reset to zero. Every element $K_{c,i,j}$ in KTree_i is therefore a deterministic function of sdk and the device identity string plus key index.

During protocol execution, the server does not store the full key tree. Instead, it recovers an individual device key on demand using `satp_extract_device_key`. Given a server key sdk and an identity string kid_i whose last bytes encode an index j, the extraction uses

$$K_{c,i} \leftarrow \text{Shake256}(\mathsf{sdk} \,\|\, \mathsf{kid}_i),$$

provided that $j < \mathsf{SATP\_KEY\_TREE\_COUNT}$. This matches the single call to `qsc_cshake256_compute` in `satp_extract_device_key`. In the rest of the paper, $K_{c,i}$ denotes the particular tree element selected by the current index encoded in kid_i and used in the SATP handshake for that device.

2.2 Protocol Messages and Packet Structure

SATP messages consist of a fixed header followed by an optional encrypted payload. The header layout is defined in `satp.h`. Each header contains:

- a flag byte identifying the SATP message type,
- a 32 bit message length,
- a 64 bit sequence number,
- a 64 bit timestamp.

Serialization follows the exact byte order in the reference implementation. These header bytes form the associated data for authenticated encryption.

Let Hdr denote the serialized header. For all packets that are processed through the SATP cipher (the connect response, authentication messages, and established data packets), encryption has the form:

$$(C, T) = \mathsf{AEAD.Enc}(K_{\mathsf{enc}}, N_{\mathsf{enc}}, \mathsf{Hdr}, P),$$

where P is the plaintext payload. Decryption validates the tag before processing any payload, consistent with `satp_decrypt_packet`. The Connect Request is the only message that is sent in plaintext and is not passed through this interface.

2.3 Session Key and Hash Derivation

Session establishment begins when the client generates a 32 byte nonce N_h using the device random generator. All key material for the SATP channel is derived from $K_{c,i}$, the configuration string cfg_s, and N_h.

The client computes a validation hash:

$$H_c = \mathsf{Shake256}(ST_c, K_{c,i}, N_h).$$

This value is stored by the client and used to authenticate the server during the handshake. The server recomputes the same H_c after deriving $K_{c,i}$. The session key tuple is obtained by a single SHAKE expansion

$$X = \mathsf{Shake256}(K_{c,i}, \mathsf{cfg}_s, N_h),$$

and splitting X deterministically into four 32 byte values:

$$(Rk, Rn, Tk, Tn) = X[0{:}32],\ X[32{:}64],\ X[64{:}96],\ X[96{:}128].$$

These values initialize the receive and transmit RCS contexts. The splitting boundaries exactly follow the byte indexing used in `kex_derive_session_keys`.

2.4 Client State Machine and Message Flow

The client follows a deterministic state machine consistent with the logic in `client.c`.

- **Initialization.** The client loads $K_{c,i}$, ST_c, and cfg_s. It initializes replay and timestamp state.
- **Connect Request.** The client generates the session nonce N_h, computes the validation hash H_c, derives the session tuple (R_k, R_n, T_k, T_n), and initializes the SATP cipher contexts. It then constructs the packet header with the current transmit sequence number, writes the device identity string and N_h into the plaintext payload, and sends a Connect Request. No authenticated encryption is applied to the Connect Request, which matches the behavior of `client_connect_request` in the reference implementation.
- **Processing Connect Response.** Upon receiving a Connect Response, the client decrypts the packet, extracts the hash returned by the server, and compares it to the stored H_c. Equality authenticates the server and advances the state to *Connected*.
- **Authentication Request and Response.** After the key exchange completes and the SATP cipher contexts have been initialized, the client performs a password based authentication step. Each device is provisioned with a memory hard authentication verifier $H_{\mathsf{pass},i} \in \{0,1\}^\lambda$ derived from a passphrase and stored in the `spass` field of its device record. The client constructs an authentication request payload

$$\mathsf{AuthReqMsg} = (i, H_{\mathsf{pass},i}),$$

where i is the device identifier of length `SATP_DID_SIZE` and $H_{\mathsf{pass},i}$ has length `SATP_HASH_SIZE`. This payload is encrypted under the established transmit context using the SATP AEAD interface and sent as an `satp_flag_encrypted_message` packet. On the server side, the first encrypted packet received after the key exchange is decrypted and passed to an application level authentication callback, which verifies that the pair $(i, H_{\mathsf{pass},i})$ matches a provisioned device record, typically using a memory hard verifier such as SCB. If verification succeeds, the server sends an Authentication Response containing its server identifier sid encrypted under the SATP channel. The client decrypts this response and accepts the authentication if and only if the recovered identifier equals the expected server identifier. Once this exchange completes successfully, both sides transition to the *Established* state and subsequent packets are handled as application data.

- **Established Operation.** Once established, the client processes SATP packets strictly through the AEAD decrypt function with header bound integrity and monotonic sequence checks.

2.5 Server State Machine and Message Flow

The server logic mirrors the client but includes derivation of device keys from branch level material, as seen in `server.c`.

- **Reception of Connect Request.** The server parses the header, copies the device identity string and client nonce N_h from the plaintext payload, and uses them to derive the device key and session state, as in server_connect_response.
- **Device Key Derivation.** The server recomputes $K_{c,i}$ from K_{br} and i, then recomputes $H_c = \mathsf{Shake256}(N_h, K_{c,i}, ST_c)$.
- **Session Key Computation.** The server derives (Rk, Rn, Tk, Tn) identically to the client and initializes cipher contexts.
- **Connect Response.** The server constructs a packet containing H_c, encrypts it with the receive/transmit roles reversed relative to the client, and sends the response.
- **Authentication Handling.** The server parses the client authentication message, verifies the session tuple, and returns an encrypted acknowledgment. After this phase the state enters *Established*.
- **Failure Handling.** Any invalid packet, failed tag verification, expired timestamp, or sequence rollback transitions the session to a termination state.

2.6 SATP Key Exchange pseudo-code

The following pseudo-code reflects the exact ordering of computations in the reference implementation.

2.7 SATP Key Exchange pseudo-code

The SATP key exchange uses a single client generated nonce N_h and the provisioned device key $K_{c,i}$ to derive a session tuple that seeds the RCS based data channel. The connect request message carries the device identifier and N_h in clear. The connect response contains an encrypted validation hash H_c that authenticates the server to the client.
We write

$$(Rk, Rn, Tk, Tn) \leftarrow \mathsf{Split}(X)$$

to denote splitting a 128 byte string X into four 32 byte segments as

$$Rk = X[0{:}32], \quad Rn = X[32{:}64], \quad Tk = X[64{:}96], \quad Tn = X[96{:}128].$$

The RCS key parameters for a direction are given by a key and nonce pair, and the protocol initializes one transmit and one receive context on each side.

2.7.1 SATP Client Connection Request

The CLIENT_CONNECT_REQUEST function constructs and transmits the initial SATP connection request. It generates the client session nonce N_h, derives the session key material used to initialize the transmit and receive RCS contexts, and computes the session validation hash H_c that authenticates the server in the subsequent response. The function places the device identifier and N_h into the packet payload in clear-text, constructs the packet header with the current transmit sequence number, and initializes both channel directions using the four segments of the derived pseudo-random material. No encryption is applied to the connect request, and the function updates the exchange flag to indicate that a connect request has been sent.

Algorithm 1 CLIENT_CONNECT_REQUEST

```
Require: cls, cns, packetout
 1: if cls = NULL or cns = NULL or packetout = NULL then
 2:     return satp_error_general_failure
 3: end if
 4: Allocate kid, nh, prnd and set to zero
 5: if QSC_CSP_GENERATE(nh, SATP_STOK_SIZE) then
 6:     QSC_MEMUTILS_COPY(kid, cls→kid, SATP_KID_SIZE)
 7:     QSC_CSHAKE256_COMPUTE(prnd, sizeof(prnd), cls→dk, SATP_DKEY_SIZE,
        (uint8_t*)SATP_CONFIG_STRING, SATP_CONFIG_SIZE, nh, SATP_STOK_SIZE)
 8:     QSC_CSHAKE256_COMPUTE(cls→hc, SATP_HASH_SIZE, nh, SATP_STOK_SIZE,
        cls→dk, SATP_DKEY_SIZE, cls→stc, SATP_SALT_SIZE)
 9:     QSC_MEMUTILS_COPY(packetout→pmessage, kid, SATP_KID_SIZE)
10:     QSC_MEMUTILS_COPY(packetout→pmessage + SATP_KID_SIZE, nh,
        SATP_STOK_SIZE)
11:     QSC_MEMUTILS_CLEAR(nh, SATP_STOK_SIZE)
12:     SATP_PACKET_HEADER_CREATE(packetout, satp_flag_connect_request,
        cns→txseq, SATP_CONNECT_REQUEST_MESSAGE_SIZE)
13:     kp.key ← prnd
14:     kp.keylen ← SATP_SKEY_SIZE
15:     kp.nonce ← prnd + SATP_SKEY_SIZE
16:     QSC_RCS_INITIALIZE(&cns→txcpr, &kp, true)
17:     kp.key ← prnd + SATP_SKEY_SIZE + SATP_NONCE_SIZE
18:     kp.nonce ← prnd + SATP_SKEY_SIZE + SATP_NONCE_SIZE + SATP_SKEY_SIZE
19:     QSC_RCS_INITIALIZE(&cns→rxcpr, &kp, false)
20:     QSC_MEMUTILS_CLEAR((uint8_t*)&kp, sizeof(kp))
21:     QSC_MEMUTILS_CLEAR(prnd, sizeof(prnd))
22:     cns→exflag ← satp_flag_connect_request
23:     return satp_error_none
24: else
25:     cns→exflag ← satp_flag_none
26:     return satp_error_random_failure
27: end if
```

2.7.2 SATP Server Connection Response

The server_connect_response function processes an incoming connect request on the server. It copies the device identifier and client nonce from the clear-text payload, derives the device key from the server side key material, initializes the receive and transmit RCS contexts for the session, computes the server validation hash H_c, and returns a connect response that encrypts H_c under the new session keys with the serialized header as associated data.

Algorithm 2 server_connect_response

```
Require: svs, cns, packetin, packetout
Ensure: satp_errors err
uint8_t dk[ SATP_DKEY_SIZE ] ← 0
uint8_t kid[ SATP_KID_SIZE ] ← 0
uint8_t nh[ SATP_STOK_SIZE ] ← 0
uint8_t shdr[ SATP_HEADER_SIZE ] ← 0
if svs  NULL and cns  NULL then
    err ← satp_error_none
    qsc_memutils_copy(kid, packetin→pmessage, SATP_KID_SIZE)
    qsc_memutils_copy(nh, packetin→pmessage + SATP_KID_SIZE, SATP_STOK_SIZE)
    if satp_extract_device_key(dk, svs→sdk, kid) then
        satp_init_session_ciphers(dk, nh, cns)
        qsc_cshake256_compute(svs→hc, SATP_HASH_SIZE, nh, SATP_STOK_SIZE, dk, SATP_SKEY_SIZE, svs→stc, SATP_SALT_SIZE)
        satp_packet_header_create(packetout, satp_flag_connect_response, cns→txseq, SATP_CONNECT_RESPONSE_MESSAGE_SIZE)
        satp_packet_header_serialize(packetout, shdr)
        qsc_rcs_set_associated(&cns→txcpr, shdr, SATP_HEADER_SIZE)
        qsc_rcs_transform(&cns→txcpr, packetout→pmessage, svs→hc, SATP_HASH_SIZE)
        cns→exflag ← satp_flag_connect_request
        err ← satp_error_none
    else
        err ← satp_error_key_expired
    end if
else
    err ← satp_error_general_failure
end if
return err
```

The helper satp_init_session_ciphers implements the RCS keying logic used by the server during the connect response. It derives the pseudo-random session material from the device key, configuration string, and client nonce, and uses it to initialize the receive and transmit cipher contexts, then erases all temporary key material.

Algorithm 3 SATP_INIT_SESSION_CIPHERS

```
Require: dk, nh, cns
 1: uint8_t prnd[ (SATP_SKEY_SIZE + SATP_NONCE_SIZE) * 2 ] ← 0
 2: qsc_rcs_keyparams kp ← {0}
 3: QSC_CSHAKE256_COMPUTE(prnd, sizeof(prnd), dk, SATP_DKEY_SIZE,
    (uint8_t*)SATP_CONFIG_STRING, SATP_CONFIG_SIZE, nh, SATP_STOK_SIZE)
 4: kp.key ← prnd
 5: kp.keylen ← SATP_SKEY_SIZE
 6: kp.nonce ← prnd + SATP_SKEY_SIZE
 7: kp.info ← NULL
 8: kp.infolen ← 0
 9: QSC_RCS_INITIALIZE(&cns→rxcpr, &kp, false)
10: kp.key ← prnd + SATP_SKEY_SIZE + SATP_NONCE_SIZE
11: kp.keylen ← SATP_SKEY_SIZE
12: kp.nonce ← prnd + SATP_SKEY_SIZE + SATP_NONCE_SIZE + SATP_SKEY_SIZE
13: kp.info ← NULL
14: kp.infolen ← 0
15: QSC_RCS_INITIALIZE(&cns→txcpr, &kp, true)
16: QSC_MEMUTILS_CLEAR((uint8_t*)kp, sizeof(qsc_rcs_keyparams))
17: QSC_MEMUTILS_CLEAR(prnd, sizeof(prnd))
```

3 Related Work and Positioning

3.1 Symmetric Key Hierarchies and Kerberos

Kerberos is a classical symmetric authentication system that relies on a central key distribution center which issues tickets that grant access to specific services. Each client and each service shares a long term key with the key distribution center. The server side maintains short lived state in the form of issued tickets, which are bound to timestamps and sequence windows that limit replay attacks. Kerberos provides a flexible trust model within a single administrative domain, but it depends on online ticket generation and a central authority that must remain available.

SATP follows a different design philosophy. It replaces ticket based authentication with a deterministic key hierarchy derived from a single master secret. Branch keys and device keys are derived through SHAKE based expansions, and servers do not maintain per session state outside the active channel. SATP avoids ticket issuance entirely and does not require an online key distribution center once provisioning is complete. The replay window of SATP is enforced by the authenticated encryption layer which binds timestamps and sequence numbers to every ciphertext. These differences make SATP suitable for deployments that cannot rely on online ticket services or central authorities.

3.2 TLS 1.3 PSK and KEM-Based Protocols

TLS 1.3 allows endpoints to resume sessions or establish new sessions using pre shared keys. In most practical settings this mechanism is combined with ephemeral Diffie Hellman exchange, which provides perfect forward secrecy for all parties. Public key operations remain central to the protocol. Modern post quantum deployments of TLS rely on key encapsulation mechanisms such as Kyber to provide forward secrecy under quantum adversaries. These public key operations increase computation and code footprint, and they require support for large keys and structured randomness that is not always available in embedded environments.

SATP removes public key operations entirely. It establishes a channel using only symmetric methods and a single SHAKE based key derivation. The resulting protocol does not achieve full forward secrecy for the server, because compromise of a branch key reveals all device keys in that branch. SATP trades this limitation for predictable cost, small code size, and post quantum resilience in resource constrained environments where KEM based handshakes are impractical. This tradeoff is appropriate for systems that rely on pre provisioned symmetric secrets and that cannot accommodate the overhead of full TLS style negotiation.

3.3 3GPP AKA and Mobile Authentication

The authentication and key agreement procedure used in mobile networks relies on a hierarchical symmetric trust model in which each subscriber identity module stores a long term secret that is known to the home network. Session keys are derived deterministically from this secret and a challenge generated by the network. This approach avoids public key operations and allows large populations of devices to authenticate rapidly with minimal computation.
SATP shares structural similarities with the mobile authentication model. Device keys are provisioned in advance and are deterministically derived from a branch level key. Both systems use symmetric operations to derive fresh session keys and rely on challenge response mechanisms to authenticate the network. The main difference is that SATP integrates channel encryption and replay protected transport directly into the protocol, and binds its session keys to a configuration string and a client generated nonce rather than a network generated challenge. These differences allow SATP to operate in more general environments without relying on cellular infrastructure or the specific message formats used by mobile networks.

3.4 Target Use Cases

SATP is designed for deployment in environments where public key methods are impractical or too costly. These include embedded systems with limited memory, industrial control networks, satellite and aerospace platforms, and legacy infrastructures that require post quantum protection without large code additions. Such systems often rely on symmetric secrets provisioned during manufacturing and cannot accommodate the computational or memory requirements of key encapsulation mechanisms or digital signature verification. SATP provides a channel establishment mechanism that operates entirely within these constraints while offering strong authentication, confidentiality, and replay protection based on standard symmetric primitives.

4 Cryptographic Model and Assumptions

This section introduces the cryptographic model used to analyze SATP. It defines the primitives assumed by the protocol, the key hierarchy and its trust assumptions, the adversarial capabilities, and the security goals that guide the formal proofs in the following sections.

4.1 Primitives and Security Assumptions

SATP relies on three classes of symmetric primitives.

- **SHAKE-based key derivation functions.** The protocol uses SHAKE to derive branch keys, device keys, session keys, and the validation hash H_c. The analysis models SHAKE as a pseudo-random function for all inputs of interest. We assume that for any efficient adversary A the advantage in distinguishing SHAKE from a pseudo-random function is negligible.

- *Memory hard password hashing.* The authentication phase of SATP uses a memory hard function, such as SCB, to derive long term authentication verifiers from device specific passphrases. For each device with identifier i, the provisioning process computes
$$H_{\mathsf{pass},i} = \mathsf{SCB}(\text{passphrase}_i)$$
and stores $H_{\mathsf{pass},i}$ both on the device and in the corresponding server side record. During a session, the client sends $H_{\mathsf{pass},i}$ inside the encrypted authentication request, and the server verifies it against its stored value. The function is assumed to be one way for adversaries with classical, GPU, or moderately parallel hardware. The protocol analysis requires only preimage resistance of this verifier function.
- **Authenticated encryption.** All encrypted SATP packets are protected by an authenticated encryption scheme with associated data. The security analysis assumes ciphertext indistinguishability under chosen ciphertext attack and ciphertext integrity for the AEAD primitive. The protection applies to replay and reordering attacks because the header fields are included in the associated data.

The proofs reduce the security of SATP to the combined security of these primitives, and no other assumptions are required.

4.2 Key Hierarchy Model

SATP uses a deterministic hierarchy of symmetric keys derived from a root secret held by a provisioning authority.

- The root authority holds K_{root}, which is never exposed outside trusted provisioning environments.
- Each branch server receives a branch key
$$K_{\mathsf{br}} = \mathsf{Shake256}(K_{\mathsf{root}} \parallel \mathsf{branch\_id}).$$
- Each client device with identifier i receives a device key
$$K_{c,i} = \mathsf{Shake256}(K_{\mathsf{br}} \parallel i),$$
and a server secret ST_c that is shared only with the branch server.

The model assumes that K_{root} and K_{br} remain secret, and that compromise of a device key $K_{c,i}$ affects only the corresponding device and its established sessions. The security definitions allow the adversary to compromise certain keys as part of the key compromise oracle in order to model limited forward secrecy.

4.3 Adversarial Capabilities

The adversary controls the network and may be classical or quantum. The model assumes the following capabilities.

- **Network control.** The adversary may deliver, drop, delay, reorder, or replay SATP packets. It may inject arbitrary ciphertexts under arbitrary headers.
- **Channel oracles.** The adversary may query encryption oracles that simulate the transmission of application data. It may query decryption oracles that return failure or success depending on whether the ciphertext is valid.

- **Key compromise.** The adversary may request compromise of $K_{c,i}$ for any device that it does not control during a challenge session. After compromise, the adversary learns the device key and may attempt to distinguish previous ciphertexts from random. Compromise of K_{br} or K_{root} is not permitted, because those events break the trust model of the system.
- **Computation limits.** The adversary may be classical or quantum and may perform arbitrary polynomial time computation. The analysis relies on the assumption that no such adversary can violate the security properties of SHAKE or the AEAD primitive.

The model does not limit the adversary's access to packet timing or packet size information, because SATP does not attempt to hide these features.

4.4 Security Goals

The analysis considers the following goals for parties that establish a SATP session.

- **Channel authenticity.** Each participant accepts only if it is communicating with the intended peer and does so only when both sides derive the same session keys. Authenticity requires correct computation and comparison of the validation hash H_c.
- **Confidentiality of application data.** An adversary that cannot compromise $K_{c,i}$ during the session should not distinguish encrypted application data from random strings of the same length.
- **Ciphertext integrity.** An adversary should not cause a participant to accept any ciphertext that was not generated by the peer with the correct session keys. This is guaranteed by the integrity of the AEAD primitive and the binding of header fields to each ciphertext through the associated data.
- **Client side key erasure but no forward secrecy.** If $K_{c,i}$ is revealed at some time t, an adversary that has recorded the SATP handshake for that device can recompute the session keys for any past session, because N_h is transmitted in clear in the Connect Request and the configuration string is public. The implementation erases session keys at termination, which prevents compromise of a device from directly revealing stored session material, but it does not provide forward secrecy against an adversary that both records traffic and later learns $K_{c,i}$.

5 Formal Protocol Specification

This section defines the SATP protocol in formal terms. It introduces the notation used throughout the analysis, specifies all message formats used in the key exchange and data channel, and describes the execution model for honest participants. These definitions align with the engineering description of Section 2 and reflect the exact message flow and key derivations realized in the reference implementation.

5.1 Notation

We use the following notation throughout.

- $\{0,1\}^n$ denotes the set of binary strings of length n. The set of all finite binary strings is $\{0,1\}^*$.
- Concatenation of two strings X and Y is written $X \parallel Y$.

- For a string X, the slice $X[a{:}b]$ denotes the substring consisting of bytes a through $b-1$ inclusive, with zero based indexing.
- Identifiers are strings in $\{0,1\}^*$ and include device identifiers and branch identifiers. We assume that identifiers are unique within their domains.
- Keys are binary strings. The root key is K_{root}, a branch key is K_{br}, and a device key is $K_{c,i}$ for device identifier i.
- Each device stores a server validation secret ST_c known to the corresponding branch server.
- A session nonce N_h is a uniformly random string of length 32 bytes chosen by the client.
- The function $\mathsf{Shake256}(X)$ denotes the SHAKE based expansion of input X into a required number of output bytes.
- The notation $(Rk, Rn, Tk, Tn) \leftarrow \mathsf{Split}(X)$ refers to splitting a 128 byte string X into four 32 byte segments in increasing order of index.
- The authenticated encryption scheme is written

  $$(C, T) = \mathsf{AEAD.Enc}(K, N, A, P)$$

  where A is associated data and P is the plaintext payload, and

  $$P = \mathsf{AEAD.Dec}(K, N, A, C, T)$$

  indicates acceptance and recovery of P, while $\perp$ indicates rejection.
- Packet headers are denoted by Hdr and contain a flag, a message length, a sequence number, and a timestamp. The exact format matches the definition in Section 2.

5.2 Message Formats

We now define the format of every SATP message exchanged during a session.

Connect Request. In the implementation the header flag is set to satp_flag_connect_request and the payload consists of the device identity string and N_H in clear. The Connect Request is not passed through the authenticated encryption layer and therefore has no ciphertext or authentication tag.

$$\mathsf{CR} = (\mathsf{flag}_{\mathsf{cr}},\ i,\ N_h)$$

where $\mathsf{flag}_{\mathsf{cr}}$ denotes the Connect Request flag, N_h is the client generated session nonce, and i is the device identifier.

Connect Response. A Connect Response contains the server computed validation hash:

$$\mathsf{R} = (\mathsf{flag}_{\mathsf{res}},\ H_c).$$

The payload H_c is encrypted under the server's transmit context and is authenticated to the client upon decryption.
Authentication Request. After a client has validated the server using the connect response, it sends an authentication request that proves knowledge of the provisioned

passphrase without exposing it in clear. Each device with identifier i stores an authentication verifier

$$H_{\mathsf{pass},i} = \mathsf{SCB}(\text{passphrase}_i),$$

where SCB denotes a memory hard password hashing function and $H_{\mathsf{pass},i}$ has length `SATP_HASH_SIZE`. The authentication request payload is

$$\mathsf{AR} = (\text{flag}_{\mathsf{ar}}, i, H_{\mathsf{pass},i}),$$

where $\text{flag}_{\mathsf{ar}}$ denotes the authentication request flag and i is encoded as a `SATP_DID_SIZE` byte device identifier. The entire payload $(i, H_{\mathsf{pass},i})$ is encrypted under the client transmit context using the SATP AEAD scheme with the serialized header as associated data. The server decrypts the first encrypted packet received after the key exchange and passes the recovered $(i, H_{\mathsf{pass},i})$ pair to an application level verifier, which compares it against the provisioned verifier for that device.

Authentication Response. If the server side verifier accepts the authentication request for identifier i, the server responds with an authentication response that confirms its identity to the client and signals that the client is authenticated. The authentication response has the form

$$\mathsf{AS} = (\text{flag}_{\mathsf{as}}, \mathsf{sid}),$$

where $\text{flag}_{\mathsf{as}}$ denotes the authentication response flag and sid is the server identifier of length `SATP_SID_SIZE`. The value sid is encrypted under the SATP data channel with the header as associated data. Upon decryption, the client compares the recovered identifier to its expected server identifier and accepts the authentication if and only if they are equal. After a successful authentication response, the session state on both sides is *Established* and subsequent encrypted packets carry application data.

Application Data Packets. Once established, either party may send application data. A data packet has the form:

$$\mathsf{DP} = (\mathsf{flag}_{\mathsf{data}},\ P)$$

where P is arbitrary plaintext data. The header sequence number and timestamp are monotone increasing and form part of the associated data for encryption.

Error Messages. Error messages follow the structure

$$\mathsf{ERR} = (\mathsf{flag}_{\mathsf{err}},\ \mathsf{ErrorCode})$$

where the error code identifies the condition detected by the receiver.

5.3 Execution Model

We formalize the behavior of SATP as a system of interacting sessions. Each session is associated with a participant role, either client or server, and maintains local state throughout its lifetime.

Session State. Each session maintains the following values:

- the device identifier i,
- the device key $K_{c,i}$,
- the server validation secret ST_c (in client state) or its server side copy,
- the session nonce N_h (on the client),

- the session tuple (Rk, Rn, Tk, Tn),
- the current sequence number and replay window state,
- the current state label, one of $\{\mathsf{Init}, \mathsf{Connected}, \mathsf{Auth}, \mathsf{Est}, \mathsf{Terminated}\}$.

Session Start. A client begins a session by generating N_h, computing H_c, deriving the session tuple, and sending a Connect Request. A server begins a session upon reception of a Connect Request, and derives $K_{c,i}$ and H_c based on the request contents.

Acceptance. A client accepts a session if and only if the Connect Response decrypts correctly and yields a value equal to the stored H_c. A server accepts a session once the authentication request validates successfully.

Transition to Established. Both parties transition to the established state once application data is allowed to flow. At this point, both have confirmed that the peer derived the same session keys and that the authentication phase succeeded.

Rejection and Termination. A session is terminated upon any of the following events:

- failure of authenticated decryption,
- violation of the sequence monotonicity rule,
- expiration outside the timestamp window,
- receipt of an error message,
- explicit termination by either participant.

Upon termination, all session keys are erased and the state transitions to $\mathsf{Terminated}$. No further packets are processed.
This execution model defines the traces against which the security definitions of Section 6 are evaluated.

6 Security Definitions

This section defines the adversarial goals relevant to SATP. The definitions follow standard game based formulations and are tailored to the symmetric trust hierarchy and deterministic key schedule used by the protocol. All games are run with respect to the execution model of Section 5.

6.1 Channel Authenticity Experiment

The channel authenticity experiment models an adversary that attempts to cause an honest participant to accept a session without interacting with the intended peer or without deriving matching session keys. On the server, a session is considered accepted only after the authentication request has been decrypted under the SATP channel and the verifier $(i, H_{\mathsf{pass},i})$ has been validated against the provisioned record for i.

The experiment proceeds as follows.

1. The challenger initializes the system parameters, generates K_{root}, and derives all branch and device keys required for the experiment.

2. The adversary is given network control and may deliver, delay, drop, or modify any SATP packet. It interacts with honest participants through oracle access to their message generation and processing functions.
3. The adversary may open any number of sessions and may corrupt any device key except the one used in the challenge session.
4. The adversary succeeds if it causes an honest party to transition to the $\mathsf{Connected}$ or Est state with a peer identity that never participated in the corresponding session, or if two honest parties accept with different session keys.

The advantage of an adversary A in breaking channel authenticity is

$$\mathsf{Adv}^{\mathsf{auth}}_{\mathsf{SATP}}(A) = \Pr[A \text{ wins the authenticity experiment}].$$

The protocol achieves channel authenticity if this advantage is negligible for all efficient adversaries.

6.2 Channel Confidentiality Experiment

Channel confidentiality is modeled by an IND-CCA style experiment for the established data channel. The experiment proceeds as follows.

1. The adversary interacts with honest participants and may perform any number of session initiations, authentications, and transmissions. It may corrupt device keys except within a designated challenge session.
2. Once a challenge session reaches the Est state, the adversary submits two equal length plaintexts P_0 and P_1.
3. The challenger selects a bit $b \in \{0, 1\}$ uniformly at random and returns the encryption of P_b under the active SATP session keys.
4. The adversary continues interacting with all participants, including through decryption oracles which reject malformed ciphertexts. It may not query the challenge ciphertext to the decryption oracle.
5. The adversary outputs a guess b'.

The advantage of A is defined as

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

SATP achieves channel confidentiality if this advantage is negligible for all efficient adversaries. The definition captures confidentiality of all application data sent after the session reaches the established state.

6.3 Ciphertext Integrity Experiment

Ciphertext integrity is defined by an INT-CTXT experiment for the AEAD protected data channel. The experiment models an adversary which attempts to cause an honest party to accept a ciphertext that the peer never generated.

1. The adversary interacts with honest participants and may observe ciphertexts produced by any session through the encryption oracle.
2. The adversary eventually outputs a candidate ciphertext header and payload (Hdr, C, T) for decryption by an honest participant.

3. The adversary wins if the honest participant outputs a plaintext $P \neq \perp$ and the ciphertext was not generated by the corresponding peer during the session.

The advantage of A is:

$$\mathsf{Adv}^{\mathsf{int}}_{\mathsf{SATP}}(A) = \Pr[A \text{ wins the integrity experiment}].$$

SATP achieves ciphertext integrity if this advantage is negligible for all efficient adversaries.

Because header fields are included in the associated data, this definition also covers integrity of sequence and timestamp values.

6.4 Client-Side Forward Secrecy

SATP does not provide forward secrecy under compromise of a device key. Because the Connect Request transmits the session nonce N_h in clear and the session key derivation is fully deterministic in

$$X = \mathrm{Shake256}(K_{c,i} \,\|\, \mathsf{cfg}_s \,\|\, N_h),$$

an adversary that records a session transcript and later compromises the corresponding device key $K_{c,i}$ can recompute the exact session tuple (R_k, R_n, T_k, T_n) for that session. This allows full recovery of all application data protected under that session's keys.
To reflect this property, we define a client-side compromise resilience experiment rather than a forward secrecy experiment. After a session has terminated, the adversary is permitted to obtain $K_{c,i}$ for a target device and is given a challenge string that is either (i) a real ciphertext previously recorded from that session or (ii) a uniformly random string of the same length. Because the adversary can recompute the session keys using the public nonce N_h and the revealed key $K_{c,i}$, it can always distinguish the real ciphertext from random with non-negligible advantage. Hence, the protocol as implemented does not satisfy a forward secrecy definition and instead provides only erasure of session keys from local state once a session terminates.

6.5 Replay and Reordering Security

Replay and reordering security captures the guarantee that an adversary cannot force an honest party to accept packets that were previously processed or that violate the ordered delivery rules of the protocol.

The experiment is defined as follows.

1. The adversary interacts with honest participants and may obtain any number of legitimate ciphertexts from established sessions.
2. The adversary outputs a candidate packet consisting of a header and ciphertext (Hdr, C, T) for processing by an honest participant.
3. The adversary wins if the honest participant accepts the packet and either the timestamp or sequence number in Hdr is not strictly greater than the last accepted values in the session.

The advantage of A in breaking replay or reordering security is:

$$\mathsf{Adv}^{\mathsf{replay}}_{\mathsf{SATP}}(A) = \Pr[A \text{ wins the replay experiment}].$$

Because the header is bound to ciphertexts as associated data, any attempt to reuse old header values produces an invalid authenticated encryption tag. SATP achieves replay and reordering security if the above advantage is negligible.

7 Provable Security Analysis

This section sketches the provable security analysis of SATP in the model and with the goals defined in Sections 4 and 6. The proofs follow a standard game based approach that reduces the advantage of any adversary against SATP to the advantages of related adversaries against SHAKE and the underlying AEAD scheme.

Adversary Model and Assumptions. The security arguments in this section are made under the same threat model and operational assumptions as defined in the SATP specification. The adversary is assumed to have full network visibility and control, including the ability to observe, delay, replay, drop, and inject packets. Compromise of individual client devices or branch servers is considered within scope, with the restriction that compromise occurs after any session keys under analysis have been erased. The adversary may possess quantum computational capabilities limited to generic square-root attacks such as Grover search, and no assumption is made of feasible large-round hidden-shift or state-recovery attacks against the Keccak permutation. All randomness used for nonces and key generation is assumed to be generated by a cryptographically secure random number source, and long-term secrets such as the server authentication secret ST_c are assumed to be stored in a manner resistant to trivial extraction. Unless explicitly stated otherwise, security claims assume correct protocol execution, unique per-session nonces, and strict monotonic enforcement of packet sequence numbers and timestamps.

Authentication Layer Separation. The SATP handshake enforces authentication through three cryptographically independent mechanisms, each serving a distinct security purpose. First, *server authentication* is achieved during tunnel establishment by verification of the session validation hash

$$H_c = \mathrm{SHAKE256}(N_h \parallel K_{c,i} \parallel ST_c),$$

which can be computed only by a server possessing both the correct derived client key $K_{c,i}$ and the long-term server secret ST_c. Second, *client authentication* is performed after tunnel establishment using a passphrase-derived credential hardened via the SCB cost-based key derivation function, ensuring resistance to offline dictionary and brute-force attacks even under partial server compromise. Third, *session authentication and freshness* are enforced by ephemeral per-session keys derived from one-time-use client tree keys $K_{c,i}$, together with authenticated packet sequence numbers and timestamps bound into the AEAD additional authenticated data. These mechanisms are independent by construction: compromise of any single authentication secret does not enable impersonation or retroactive decryption without simultaneous compromise of the remaining layers.

7.1 Overview of Reduction Strategy

The analysis proceeds by defining a sequence of games for each security goal, starting from the real protocol execution and gradually replacing components with idealized objects. At each step we bound the change in the adversary's advantage by the advantage of a related adversary against one of the underlying primitives.

For channel authenticity and confidentiality, the main steps are:

- Replace the SHAKE based derivations of session keys and validation hashes with outputs of independent random functions on the corresponding domains. This step is justified by the pseudo-randomness assumption on SHAKE.
- Replace the AEAD instance with an ideal authenticated encryption oracle that provides perfect confidentiality and ciphertext integrity. This step is justified by the IND CCA and INT CTXT security of the AEAD scheme.

In the resulting ideal game, the adversary has negligible probability of causing an acceptance event that violates the security property of interest. Summing the differences between consecutive games yields the final bound.
For replay and reordering resistance, the argument relies on the binding of timestamps and sequence numbers into the associated data of each ciphertext.

7.2 Authentication Theorem

Theorem 1 (Channel Authenticity). *Let A be an adversary against channel authenticity that makes at most q_h queries to the SHAKE based key derivation functions across all sessions and at most q_e encryption queries to the AEAD channels. Then there exist adversaries B_1 and B_2 such that*

$$\mathsf{Adv}^{\mathsf{auth}}_{\mathsf{SATP}}(A) \leq q_h \cdot \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{Shake}}(B_1) + \mathsf{Adv}^{\mathsf{int}}_{\mathsf{AEAD}}(B_2) + \varepsilon_{\mathsf{auth}},$$

where $\varepsilon_{\mathsf{auth}}$ is negligible in the security parameter and accounts for residual events in the idealized game.

Proof. We define a sequence of games G_0, G_1, G_2.

Game G_0. This is the real channel authenticity experiment from Section 6. The adversary A interacts with honest participants running SATP with SHAKE based key derivations and the real AEAD scheme.

Game G_1. In G_1 the challenger replaces all calls to $\mathsf{Shake256}$ used for branch key, device key, session key, and validation hash derivation with outputs of independent random functions on the corresponding input domains. Specifically, for each derivation type the challenger maintains a table that maps inputs to uniformly random outputs, and answers repeat queries consistently.
Any adversary that can distinguish G_0 from G_1 yields a PRF adversary B_1 against SHAKE. The standard hybrid argument over q_h derivation calls implies

$$|\Pr[A \text{ wins in } G_0] - \Pr[A \text{ wins in } G_1]| \leq q_h \cdot \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{Shake}}(B_1).$$

Game G_2. In G_2 the challenger replaces the AEAD scheme with an ideal authenticated encryption oracle that returns uniformly random ciphertexts and tags on encryption queries and rejects any forged ciphertext with probability one. In particular, in G_2 it is impossible for the adversary to produce a ciphertext that decrypts to a valid message unless it was returned by the encryption oracle.
Any adversary that can distinguish G_1 from G_2 yields an INT CTXT adversary B_2 against the AEAD scheme, because the only difference is the possibility of successful forgeries. Therefore

$$|\Pr[A \text{ wins in } G_1] - \Pr[A \text{ wins in } G_2]| \leq \mathsf{Adv}^{\mathsf{int}}_{\mathsf{AEAD}}(B_2).$$

Analysis of G_2. In G_2, all session keys and validation hashes are uniformly random and independent, and the AEAD oracle does not allow forgeries. For an honest client to accept with a server identity that did not participate in the session, or for two honest parties to accept with different session keys, the adversary must cause the client to accept a Connect Response that contains the correct value of H_c and binds to the correct header fields. Since H_c is a uniformly random value that never appears on the network before encryption, the only way this can happen is if the adversary submits a ciphertext that was previously output by the encryption oracle. This corresponds to replaying a legitimate response, which does not produce an authentication violation in the definition of Section 6.

Thus, in G_2 the probability that A wins the authenticity experiment is bounded by a negligible term $\varepsilon_{\mathsf{auth}}$ that accounts for events such as collisions in identifier encodings and length fields.
Combining the bounds across the three games yields:

$$\mathsf{Adv}^{\mathsf{auth}}_{\mathsf{SATP}}(A) \leq q_h \cdot \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{Shake}}(B_1) + \mathsf{Adv}^{\mathsf{int}}_{\mathsf{AEAD}}(B_2) + \varepsilon_{\mathsf{auth}},$$

which is negligible under the assumed security of SHAKE and the AEAD scheme. □

7.3 Confidentiality Theorem

Theorem 2 (Channel Confidentiality)**.** *Let A be an adversary in the channel confidentiality experiment that makes at most q_h SHAKE based derivation queries and at most q_e encryption queries to the SATP channels. Then there exist adversaries B_1 and B_2 such that:*

$$\mathsf{Adv}^{\mathsf{conf}}_{\mathsf{SATP}}(A) \leq q_h \cdot \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{Shake}}(B_1) + \mathsf{Adv}^{\mathsf{ind\text{-}cca}}_{\mathsf{AEAD}}(B_2) + \varepsilon_{\mathsf{conf}},$$

where $\varepsilon_{\mathsf{conf}}$ is negligible.

Proof. The proof follows the same structure as the authenticity theorem.

Game G_0. G_0 is the real confidentiality experiment from Section 6.

Game G_1. In G_1 we replace SHAKE based derivations with outputs of random functions as in the previous theorem. The distinguishing advantage between G_0 and G_1 is bounded by $q_h \cdot \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{Shake}}(B_1)$.

Game G_2. In G_2 we replace the AEAD scheme with an ideal IND CCA secure encryption oracle that returns uniformly random ciphertexts for new queries and rejects any ciphertext submitted to the decryption oracle that was not produced by the encryption oracle.
The difference in the adversary's view between G_1 and G_2 is bounded by
$\mathsf{Adv}^{\mathsf{ind\text{-}cca}}_{\mathsf{AEAD}}(B_2)$, because any distinguishing strategy can be converted into an IND CCA attack against the AEAD.

Analysis of G_2. In G_2, challenge session keys are uniformly random and independent, and the challenge ciphertext is produced by an ideal encryption oracle. From the adversary's perspective, the challenge ciphertext is independent of the bit b used to select P_b, since the oracle outputs random strings subject to the restriction that decryption is consistent. It follows that the adversary's probability of correctly guessing b is at most one half plus a negligible term, so the advantage in G_2 is bounded by $\varepsilon_{\mathsf{conf}}$. Summing differences across games yields the stated bound. □

7.4 Replay and Reordering Resistance

We conclude with a lemma that relates replay and reordering resistance of SATP to the integrity of the AEAD primitive and the inclusion of header fields in the associated data.

Lemma 1 (Replay and Reordering Resistance)**.** *Let A be an adversary in the replay and reordering experiment that attempts to cause an honest party to accept a packet with timestamp or sequence number not strictly greater than the last accepted values. Then there exists an adversary B against the ciphertext integrity of the AEAD scheme such that*

$$\mathsf{Adv}^{\mathsf{replay}}_{\mathsf{SATP}}(A) \leq \mathsf{Adv}^{\mathsf{int}}_{\mathsf{AEAD}}(B) + \varepsilon_{\mathsf{replay}},$$

where $\varepsilon_{\mathsf{replay}}$ is negligible.

Proof. In SATP, the serialized header, which contains the timestamp and sequence number, is always passed as associated data to the AEAD. For any ciphertext accepted by an honest participant, the tag validation ensures that the header used during decryption matches the header used during encryption.
An adversary that attempts to replay a previously observed packet with the same header cannot violate the replay policy in the protocol model, because the receiver rejects packets whose timestamps or sequence numbers are not strictly increasing. An adversary that attempts to modify the header in order to bypass these checks must produce a new ciphertext and tag pair that the receiver will accept. This is exactly a ciphertext forgery under modified associated data.
Any successful strategy for A thus yields an INT CTXT adversary B that forges a valid ciphertext under new associated data. The probability of such a forgery is bounded by $\mathsf{Adv}^{\mathsf{int}}_{\mathsf{AEAD}}(B)$, plus a negligible term: $\varepsilon_{\mathsf{replay}}$ that accounts for residual events such as counter wraparound. This gives the stated bound. □

8 Cryptanalysis and Robustness

This section discusses the robustness of SATP under realistic threat scenarios that are not fully captured by the idealized security model. The focus is on the effects of compromise within the symmetric key hierarchy, on denial of service risks created by deterministic key derivation, and on attack surfaces that arise from deployment choices and implementation errors.

8.1 Symmetric Hierarchy Compromise and Epoch Rotation

The symmetric key hierarchy used by SATP introduces a strong dependency between the security of the branch keys and the security of all devices attached to that branch. If an adversary compromises K_{br} for a branch, it can recompute every device key $K_{c,i}$ derived from that branch key by evaluating

$$K_{c,i} = \mathsf{Shake256}(K_{\mathsf{br}} \parallel i)$$

for each device identifier i. This reveals all provisioned device keys under that branch and allows the adversary to impersonate any device to the server, and the server to any device, within the branch.

The impact on historical and future sessions must be separated.

- **Historical sessions.** Once K_{br} is known, the adversary can derive all device keys and can recreate the session keys for any session whose nonce N_h and configuration string cfg_s are observable in the transcripts. In the current design, N_h is carried in clear in the Connect Request payload, so a passive adversary that records traffic and later learns K_{br} (or any $K_{c,i}$) can reconstruct the past session keys for that branch.
- **Future sessions.** After compromise of K_{br}, the adversary can actively participate in new handshakes as either party. It can generate its own N_h, derive the session tuple (Rk, Rn, Tk, Tn), and compute the correct validation hash H_c. Future sessions under the compromised branch key therefore provide no authenticity or confidentiality against the adversary.

To limit the long term effects of a branch compromise, the deployment model must include time bounded epochs and key rotation policies.

- In an *epoch based* model, each branch uses a key $K_{\mathsf{br}}^{(j)}$ for a fixed time interval indexed by j, and devices derive $K_{c,i}^{(j)}$ accordingly. The system retires old branch keys and device keys once the epoch ends.
- Compromise of $K_{\mathsf{br}}^{(j)}$ then affects only sessions that were established while epoch j was active, and does not allow recovery of sessions from previous epochs that used unrelated branch keys.
- Rotation policies must ensure that new epochs are provisioned securely and that devices can migrate from $K_{c,i}^{(j)}$ to $K_{c,i}^{(j+1)}$ without leaving residual copies of retired keys.

The protocol design does not prescribe a specific epoch mechanism, but the symmetric hierarchy makes such mechanisms essential for deployments that require strong containment of long term breaches. Without rotation, compromise of K_{br} is catastrophic for the entire lifetime of the branch.

8.2 DoS and State Exhaustion Analysis

The deterministic derivation of device keys from K_{br} and device identifiers introduces a potential denial of service vector on the server. For each incoming connect request, the server must parse the device identifier, derive $K_{c,i}$ using SHAKE, and recompute the validation hash H_c before it can decide whether the request originated from a valid device. An adversary can therefore attempt to flood the server with connect requests containing arbitrary identifiers, forcing it to perform expensive key derivations for non existent devices.
The impact of this vector depends on the cost of SHAKE evaluation on the target platform and the rate at which the attacker can generate requests. In environments where SHAKE is relatively expensive and where network access is inexpensive for the adversary, this attack can increase CPU load and reduce capacity for legitimate clients.

Several engineering strategies can mitigate this risk.

- **Identifier filtering.** Servers can maintain a compact representation of valid device identifiers, such as a Bloom filter or a hash table keyed by identifier, and check membership before deriving $K_{c,i}$. Requests carrying identifiers that are not present in the filter can be rejected early without invoking SHAKE.
- **Rate limiting.** Per source address or per identifier prefix rate limits can prevent individual adversaries from monopolizing server resources. Combined with logging, this allows operators to detect and isolate sources that generate anomalous traffic.
- **Lightweight prechecks.** Servers can require that connect requests conform to rigid length and format constraints for identifiers and other fields. Packets that fail these checks can be dropped without further processing.
- **Tiered processing.** In some deployments, a front end component can perform basic filtering and identity lookup and only pass candidate requests that match provisioned entries to a back end SATP server that performs SHAKE based derivations.

SATP does not attempt to solve denial of service at the protocol level, but the deterministic nature of its key derivation makes these engineering considerations important in practice.

8.3 Attack Surfaces and Misuse Scenarios

Beyond the explicit cryptographic assumptions, SATP inherits several attack surfaces from its operational environment. These do not violate the formal security model directly, but they can weaken effective security if left unaddressed.

Nonce misuse. Each session relies on a fresh, uniformly random N_h generated by the client. Reuse of N_h across different sessions with the same device key $K_{c,i}$ causes reuse of the corresponding session tuple (Rk, Rn, Tk, Tn). This may expose the data channel to attacks that exploit keystream reuse or related key structure in the underlying cipher. Implementations must ensure that the nonce generator is reliable and that failures are detected and handled, for example by aborting the connection if nonce generation fails.

Clock skew and timestamp configuration. Replay protection relies on comparing packet timestamps against local clocks and a configured acceptance window. Incorrect time configuration, large clock skew, or excessively permissive windows can weaken replay resistance and allow delayed packets to be accepted. Conversely, overly strict windows can cause legitimate packets to be rejected in networks with variable latency. Deployments must specify clear policies for clock synchronization and for the size of the allowed window to balance robustness and security.

Weak authentication material. The memory hard authentication layer protects against brute force attacks on the device authentication secret, but it does not compensate for extremely weak passwords or secrets. If devices are provisioned with low entropy authentication material, an adversary that obtains the corresponding SCB image may still recover the secret by exhaustive search. Provisioning processes must therefore enforce minimum entropy requirements and must protect SCB parameters from downgrades that weaken the effective cost of offline attacks.
In summary, the core SATP design withstands adversaries that operate within the assumed cryptographic model, but robust deployments must address these additional risks through careful parameter selection, operational procedures, and defensive implementation techniques.

9 Parameters and Performance

This section summarizes the concrete parameter choices used by SATP and discusses the resulting performance characteristics. The values are drawn from the protocol specification and the reference implementation, and are selected to balance security margin and efficiency in constrained post quantum environments.

9.1 Parameter Choices

SATP fixes the following parameters.

- **Hash and KDF outputs.** All key derivations use SHAKE256. Branch keys K_{br} and device keys $K_{c,i}$ are 256 bit strings. Session derivation produces a 128 byte string that is split into four 32 byte values (Rk, Rn, Tk, Tn), which serve as keys and nonces for the data channel.
- **AEAD key and nonce sizes.** The authenticated encryption scheme uses 256 bit keys and nonces sized to match the underlying cipher. Each direction of the channel maintains its own key and nonce pair. Nonces are derived deterministically from the session tuple and are advanced for each packet to prevent reuse.

- **Replay window threshold.** The protocol defines a maximum time skew between packet timestamps and the local clock, represented by a replay window parameter ΔT. Packets with timestamps outside the interval $[t_{\mathsf{local}} - \Delta T, t_{\mathsf{local}} + \Delta T]$ are rejected. The choice of ΔT depends on deployment latency and clock synchronization guarantees.

These choices ensure that all keys have security levels consistent with 256 bit symmetric primitives and that replay protection remains effective in typical network conditions.

9.2 Implementation Footprint

The reference implementation of SATP is designed to fit within the constraints of embedded and control systems that cannot accommodate large cryptographic libraries.

- **Code size.** The SATP implementation includes the protocol logic, the SHAKE based key derivation, and an authenticated encryption layer. The total code size is significantly smaller than that of typical post quantum key encapsulation suites, because it does not include lattice based arithmetic or large polynomial operations.
- **Memory usage.** The protocol requires storage for a small number of keys per session and for a minimal amount of state, including sequence numbers and timestamp metadata. No per session tickets or long term session caches are required. This keeps RAM usage modest and predictable.
- **Computational cost.** The dominant costs are the SHAKE evaluations used for key derivation and the authenticated encryption operations for the data channel. Both are symmetric operations with predictable constant factors. There are no public key operations, and there is no need for large integer arithmetic or structured lattice operations.

These characteristics make SATP suitable for devices with limited flash, RAM, and CPU resources that still require a secure channel protocol with post quantum resilience.

9.3 Comparison with KEM-Based Designs

Post quantum key exchange protocols based on key encapsulation mechanisms, such as those built from NIST standard candidates, typically require multiple lattice based operations per handshake. These operations involve large public keys and ciphertexts and require several kilobytes of code and data to implement. On constrained devices, such overhead can be prohibitive.

In contrast, SATP relies solely on symmetric primitives. The cost of a handshake is dominated by a small number of SHAKE evaluations and the initialization of the authenticated encryption contexts. The code footprint is correspondingly smaller, since it reuses hash and block cipher components that are often present for other purposes. The protocol also avoids the need to store long term public keys or to handle complex certificate chains.

The tradeoff is that SATP does not provide full forward secrecy under compromise of branch keys. Deployments that can afford the implementation and runtime cost of post quantum key encapsulation may prefer KEM based designs that offer stronger forward secrecy guarantees. SATP is intended for environments where those options are not practical and where a deterministic symmetric hierarchy provides the best balance between efficiency and security.

10 Deployment Considerations

This section discusses practical issues that arise when SATP is deployed in real systems. These considerations extend beyond the formal model and reflect operational, governance, and lifecycle requirements that influence the long term security of the protocol.

10.1 Key Management and Provisioning

SATP depends on a symmetric key hierarchy rooted at a master secret. Secure provisioning and management of these keys is essential.

Root and branch keys. The master key K_{root} is generated and stored in a trusted provisioning environment. Branch keys K_{br} are derived from K_{root} and must be provisioned into branch servers during initialization or during controlled rotation events. Neither K_{root} nor K_{br} should appear in any device level firmware or diagnostics, and both must be protected by hardware or operational controls that provide strong resistance to extraction.

Device keys and server secrets. Each device receives $K_{c,i}$ and ST_c as part of manufacturing or enrollment. Devices must validate that these values are stored in secure memory regions and must not expose them through debugging interfaces. Provisioning systems must ensure that device identifiers are unique, that derivation inputs are encoded consistently, and that the binding between devices and their identifiers cannot be altered post deployment.

Key rotation. The symmetric hierarchy requires well defined rotation policies. Root and branch keys should be rotated on a schedule that reflects organizational risk tolerance. Device keys can be derived dynamically from new branch keys during an epoch transition, but deployments must ensure that devices retire older keys and do not retain multiple sets of credentials indefinitely. Rotation events must be authenticated and logged, and they must be performed out of band to prevent interference from network adversaries.

10.2 Use Cases and Integration Scenarios

SATP is intended for environments where symmetric key provisioning is practical and where public key cryptography imposes unacceptable costs.

Control networks. Industrial control systems and SCADA environments often include devices with limited processing capabilities and predictable communication patterns. SATP provides a lightweight channel protocol that fits within these constraints while offering authenticated encryption and replay resistance. The deterministic key hierarchy simplifies integration with provisioning pipelines.

Embedded systems. Many embedded devices lack hardware acceleration for public key cryptography or have strict limits on code size and memory. SATP requires only hash based derivation and a symmetric authenticated encryption scheme, making it suitable for microcontrollers, sensors, and other embedded components that must operate for long periods with minimal maintenance.

Post quantum constrained environments. Systems that require resistance to quantum adversaries but cannot implement lattice based key encapsulation can use SATP as a symmetric alternative. Examples include aerospace systems, legacy communication platforms, and secure modules that must remain operational for extended periods. SATP provides confidentiality and authenticity based on symmetric primitives that are believed to resist quantum attacks.

10.3 Governance, Audit, and Compliance

Formal security does not replace the need for governance and audit processes. SATP deployments must align with organizational policies and relevant standards.

Configuration and parameter governance. Replay windows, sequence parameters, and authentication settings must be selected carefully and documented. Operators must ensure that these values are configured consistently across devices and that deviations are audited.

Provisioning and lifecycle audit. The key hierarchy must be accompanied by tracking of device enrollment, branch key distribution, and rotation events. Audit logs must record when credentials are provisioned, replaced, or retired. These logs must be protected from tampering and must support forensic analysis if a compromise is suspected.

Compliance alignment. SATP can support deployments that align with existing regulatory and industry standards that emphasize symmetric trust models, such as guidelines for industrial control systems and supply chain security. While SATP does not replicate all features of protocols like TLS, its deterministic structure and symmetric nature make it suitable for environments where certification requires minimal attack surface and predictable cryptographic behavior.
In summary, SATP provides a secure and efficient symmetric channel protocol for constrained environments, but successful deployment requires careful management of provisioning, rotation, configuration, and audit processes. These considerations ensure that the protocol's formal guarantees translate into practical security in real systems.

11 Limitations and Future Work

SATP provides a secure and efficient symmetric channel protocol for constrained post quantum environments, but its design reflects tradeoffs that introduce limitations in certain deployment scenarios. This section summarizes these limitations and identifies directions for future work.

Lack of server side perfect forward secrecy. SATP does not achieve perfect forward secrecy for the server. Compromise of a branch key K_{br} allows an adversary to derive all device keys $K_{c,i}$ within that branch and to compute session keys for future handshakes. The long term effect of a branch compromise remains significant. Deployments must rely on epoch based rotation to bound this exposure. Designing variants of SATP with partial server side forward secrecy while preserving low overhead remains an open research problem.

Dependence on symmetric key provisioning. SATP relies on secure provisioning of the symmetric hierarchy. This requires trusted manufacturing or enrollment pipelines and the ability to protect root and branch keys from disclosure. In distributed environments where provisioning cannot be tightly controlled, symmetric trust hierarchies may be difficult to manage. Future work may explore hybrid designs that combine SATP with lightweight public key mechanisms or secure bootstrapping protocols to enhance provisioning robustness.

Limited authentication expressiveness. Authentication in SATP is based on symmetric secrets and memory hard verifiers. This model is suitable for many embedded and industrial environments, but it does not provide the flexibility of public key based identity frameworks. Integration with external identity providers or with group based authentication mechanisms may require extensions to the SATP handshake.

Formalization scope. The security analysis in this paper focuses on channel authenticity, confidentiality, ciphertext integrity, and replay resistance. It does not cover side channel leakage, denial of service conditions that arise from resource constraints, or multi party composition scenarios. Extending the formal model to cover these aspects and to support automated proof generation through tools such as Tamarin or ProVerif is a promising direction for future work.

Automated verification and code audits. The reference implementation is small and amenable to line by line audit, but comprehensive assurance requires static analysis, symbolic verification, and fuzz testing. Creating a formal reference model that can be linked to the implementation and verified through automated tools would increase confidence in the protocol's correctness. Techniques that support continuous verification during firmware development may be especially valuable for environments where updates are infrequent.

Further performance analysis. While SATP is efficient, a deeper study of performance under varying network conditions, device capabilities, and authentication workloads would help guide parameter choices for large deployments. Experimental evaluations on representative hardware, including microcontrollers and FPGA platforms, would clarify practical limits and influence future optimizations.

Extensions to group communication. SATP is defined as a two party protocol. Some environments require secure group communication channels with symmetric trust assumptions. Extending SATP to support group keys or multicast without losing its deterministic structure and low overhead presents additional research challenges.
In summary, SATP achieves its goals within the assumptions and constraints of a symmetric, hash based channel protocol. Continued research on key rotation, flexible authentication, automated verification, and broader deployment models can further strengthen its applicability and resilience.

12 Conclusion

SATP provides a symmetric channel establishment protocol designed for environments where post quantum public key operations are impractical or too costly. Its design combines a deterministic hash based key hierarchy with a lightweight authenticated encryption layer to produce a secure and efficient alternative to key encapsulation mechanisms. The protocol derives all operational keys from pre provisioned symmetric material and a client generated nonce, and it authenticates the server through a validation hash that depends on device specific secrets. This structure enables SATP to operate in constrained systems while maintaining strong guarantees of authenticity, confidentiality, and replay protection.
The formal analysis presented in this paper defines the cryptographic model and the security goals for SATP and provides game based reductions that tie the protocol's security to the pseudo-randomness of SHAKE and the security of the AEAD primitive. The results show that SATP achieves channel authenticity, channel confidentiality, ciphertext integrity, replay and reordering resistance, and a limited form of forward secrecy for clients. These guarantees hold under classical and quantum adversaries that operate within the symmetric trust assumptions defined by the protocol.

SATP's reliance on symmetric primitives and deterministic key derivation introduces specific tradeoffs compared with public key based protocols. The protocol does not provide forward secrecy for the server under compromise of branch keys, and it depends on secure provisioning of symmetric material. These tradeoffs are acceptable in environments where symmetric trust hierarchies are practical and where the cost of post quantum key encapsulation is prohibitive. Examples include control networks, embedded systems, aerospace applications, and legacy infrastructures that require post quantum protection without significant increases in code size or runtime cost.

In closing, SATP represents a focused design tailored to constrained post quantum deployments. Its deterministic, symmetric foundation allows strong and analyzable security within the intended operating assumptions. Future work on key rotation, extended formal models, and automated verification can further strengthen the protocol and expand its applicability.

References

1. Bertoni, G., Daemen, J., Peeters, M., and Van Assche, G. *The Keccak Reference.* Submission to the NIST SHA-3 Competition, 2011. Available at: https://keccak.team
2. National Institute of Standards and Technology (NIST). *SHAKE: Functions and Applications.* U.S. Department of Commerce, 2015. Available at: https://csrc.nist.gov
3. Dworkin, M. *SP 800-38D: Recommendation for Block Cipher Modes of Operation, Galois/Counter Mode (GCM).* National Institute of Standards and Technology, 2007. Available at: https://csrc.nist.gov
4. Kelsey, J., and Schneier, B. *Cryptographic Key Generation: Theory and Practice.* Springer, 1999. Available at: https://link.springer.com
5. QRCS Corporation. *SATP Technical Specification* Quantum Resistant Cryptographic Solutions Corporation, 2024. Available at: https://www.qrcscorp.ca/documents/satp_specification.pdf
6. QRCS Corporation. *SATP Implementation Analysis* Quantum Resistant Cryptographic Solutions Corporation, 2025. Available at: https://www.qrcscorp.ca/documents/satp_analysis.pdf
7. QRCS Corporation. SATP Library Implementation (GitHub Source Code Repository). https://github.com/QRCS-CORP/SATP