Secure Infrastructure Access Protocol - SIAP

Revision 1.1a, May 27, 2026

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

This document is an engineering level description of the Secure Infrastructure Access Protocol; a post-quantum user verification system and access control mechanism based on a removable token state, a passphrase-derived verifier, and one-time symmetric authentication tokens.

Contents **Page**

Foreword

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

This document is the first revision of the specification of SIAP, further revisions may become necessary during the pursuit of a standard model, and revision numbers shall be incremented with changes to the specification. The reader is asked to consider only the most recent revision of this draft, as the authoritative implementation of the SIAP specification.

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

1. Introduction

Secure Infrastructure Access Protocol (SIAP) is a hash-based authentication scheme designed to provide post-quantum, two-factor control over access to computing devices, encrypted data stores, and embedded systems. It combines possession of a removable memory card with knowledge of a passphrase, producing a single-use token key that is verified by a host server. Each successful authentication consumes the active token leaf, advances the key index, regenerates the server-side tag, and reseals the device token tree. Replay resistance and rollback resistance therefore depend on both cryptographic verification and durable, monotonic persistence of the updated server and device state.

The construction is intentionally minimalist. Long-lived derivation state is based on the Keccak SHAKE and cSHAKE functions, while the passphrase path uses a fixed-size cSHAKE seed followed by the SCB cost-based derivation function. The token tree is sealed with the RCS authenticated stream cipher. No public-key primitive, certificate chain, or online validator is required by the SIAP authentication core. The implemented default profile uses 256-bit symmetric security parameters, with an optional extended profile selected by the SIAP_EXTENDED_ENCRYPTION build configuration.

SIAP introduces a structured identity hierarchy that reflects common enterprise and infrastructure deployments. A three-field server identifier distinguishes domain, server-group, and server instance; a two-field user identifier distinguishes group and user; and a memory card identifier pairs a unique device identity with the current token-chain index. This explicit scoping lets administrators revoke a single card, a user, an entire user group, or a compromised server or server group without impacting unrelated assets, all through a table update on the host server.

The protocol is engineered for local and intermittently connected operation. During provisioning the server generates a token tree, derives the passphrase hash through the cSHAKE-to-SCB path, stores a passphrase verifier rather than the direct sealing-key precursor, computes the key-tree hash, and seals the token tree under RCS. At run time the host verifies the plaintext identity fields, loads the corresponding server-side tag, derives the live passphrase hash from the supplied passphrase, authenticates and decrypts the sealed token tree, verifies the tree hash, compares the current token leaf against the server-derived token, and only then erases the leaf and advances the index.

Time-bounded validity is enforced by the expiration field in the device key and the server key. The implementation checks that the device expiration does not exceed the server key expiration, that the device key has not expired, and that the expiration remains within the configured SIAP validity duration. The expiration value is also included in the RCS sealing context, so modification of the sealed-state metadata is detected during decryption.

Forward security is derived from one-time token consumption and state erasure under the assumption that updated state is written durably and atomically. A successful session erases the active token leaf, increments the key index, regenerates the tag over the modified tree, and reseals the tree using a context that includes the current key identity and expiration. The server base key remains the root derivation secret; compromise of that value is outside the forward-

security guarantee for future authentications and requires server-side operational protection such as HSM or equivalent key isolation.

The design anticipates diverse hardware roots of trust. When a secure element is available, the monotonic counter and sealing operation may execute inside the tamper-resistant boundary, offering formal certification paths for payment-card or government deployments. Conversely, cost-constrained mass-market tokens can obtain an unclonable identity by deriving the card nonce from an SRAM-based physically-unclonable function, while still running the entire cryptographic workload in software.

Finally, SIAP is transport neutral. The freshly consumed token may be supplied to an application as an authorization secret, but a conforming integration should derive protocol-specific keys from the token using an explicit domain-separation label before use as a PSK, MAC key, storage-decryption key, or application-layer authorization key. By separating authentication from channel establishment, SIAP can be integrated into existing systems without requiring certificate infrastructure or altering unrelated network wire formats.

1.1 Purpose

The purpose of SIAP is to furnish a lightweight, post-quantum authentication layer that merges possession of a removable memory card with knowledge of a passphrase, generating a single-use secret that is cryptographically verifiable in constant time. The implemented protocol derives passphrase state through cSHAKE and SCB, derives token leaves from the server base key and key identity, verifies token equivalence before mutating device state, and reseals the token tree with authenticated metadata. SIAP is intended to support secure logins, local key release, and transaction authorization in environments where offline operation, small code footprint, and symmetric post-quantum security are design requirements.

Below are eight concrete use-case patterns that map onto the capabilities and constraints described in this SIAP specification. The implemented default token-tree size is 1,024 leaves. Deployments that require larger login volumes shall use explicit re-provisioning, rotation, or a separately specified larger profile rather than assuming an unbounded or implicit tree length.

#	Target domain & problem	Why SIAP fits (spec hooks)	Deployment notes
1	**PCI-DSS 4.0 console MFA** for card-data-environment jump-hosts	• Two-factor (passphrase + memory card) satisfies ‘independent MFA’ clause. • Leaf burn gives replay-proof, audit-friendly logins.	*Default implementation profile uses 1,024 leaves. Schedule rotation or re-provisioning according to login volume; store Kbase in an HSM to limit breach radius.*
2	**Offline or low-bandwidth CBDC / e-cash wallets**	• Works with no PKI or network; server derives leaf with single SHAKE	Use bounded offline windows with re-provisioning or an explicitly specified larger profile for high-volume payment

		call. • Start/End-time fields in Uks enforce spend-window.	use. Keep card BOM low by using MCU+PUF profile where appropriate.
3	**Technician access to ATMs / POS & PLCs** (field service)	• Card ID (Kci) + Server ID hierarchy (Sid) binds token to device fleet. • Immediate failure if stale index or wrong group.	Hand-held reader app can display remaining keys. Ship spare cards to avoid mid-shift exhaustion.
4	**Cold-wallet custody for institutional crypto**	• One-time key token can seed RCS/KMAC to sign withdrawal authorizations. • Forward secrecy: key burnt after each signing.	Use extended security mode where required and intentionally small token counts for high-value ceremonies. Rotate cards as part of the custody procedure.
5	**OEM activation / firmware decryption** on production lines	• Device contains server-side K_{base} hash; SIAP card must match to unlock image. • No Internet inside secure factory.	Use a defined batch or larger-profile extension. After the configured tree is exhausted, the line shall halt or re-provision before further use.
6	**Zero-trust VPN pre-shared-key replacement**	• K_{tok} outputs 256-bit secret every login; use as TLS-1.3 PSK binder or IKEv2 PSK. • Removes long-lived static PSKs vulnerable to harvesting.	Use explicit downstream derivation, for example TLS_PSK = SHAKE(label || Ktok). High-rate use requires a larger profile or automated rotation.
7	**High-frequency trading APIs** needing sub-millisecond auth	• Server derives leaf in O(1); no lattice KEM latency. • Memory footprint of 30 kB fits FPGA-adjacent soft-CPU.	Use a larger specified profile or bounded authorization events. Do not assume an implicit tree larger than the configured implementation profile.
8	**Critical-infrastructure kiosks & SCADA HMIs** (air-gapped)	• No network / CA required; Start/End times restrict stolen-card window. • Global K_n + monotonic K_{idx} prevents infinite reuse.	Harden reader firmware and monitor remaining leaves. Reissue cards before exhaustion according to site policy.

2. Scope

This document describes the Secure Infrastructure Access Protocol (SIAP), which is used to authenticate a device token and passphrase to a host server and to release a single-use symmetric authentication token. This document specifies server initialization, device provisioning, serialized state structures, passphrase hashing and verification, token-tree sealing, authentication, state advancement, and server response behavior. SIAP is an authentication and token-release protocol. It is not itself a transport protocol, packet protocol, or secure messaging channel.

2.1 Application

This protocol is intended for institutions that require local or intermittently connected authentication to servers, embedded systems, encrypted storage, or controlled operational equipment. The key derivation functions, state structures, authentication sequence, sealed-state verification, and state-advancement rules defined in this document are mandatory elements of a conforming SIAP implementation. Components that are not necessarily mandatory, but are recommended settings or usage of the protocol, are denoted by the keyword SHOULD. In circumstances where strict conformance to implementation procedures is required, the keyword SHALL is used to indicate compulsory compliance.

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

3. Terms and Definitions

3.1 Cryptographic Primitives

3.1.1 SCB

The SHAKE Cost Based Key Derivation Function (SCB-KDF) uses advanced techniques such as cache thrashing, memory ballooning, and a CPU intensive core function to mitigate attacks on a hash function by making it more expensive to run dictionary and rainbow attacks to discover a user's passphrase.

3.1.2 RCS

The wide-block Rijndael hybrid authenticated AEAD symmetric stream cipher.

3.1.3 SHA-3

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

3.1.4 SHAKE

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

3.1.5 cSHAKE

The customizable SHAKE function defined in NIST Special Publication SP 800-185. SIAP uses cSHAKE for domain-separated derivations, including server salt generation, token leaf generation, passphrase seed derivation, passphrase verifier derivation, and RCS sealing key generation.

3.2 Network References

3.2.1 Bandwidth

The maximum rate of data transfer across a given path, measured in bits per second (bps).

3.2.2 Byte

Eight bits of data, represented as an unsigned integer ranged 0-255.

3.2.3 Certificate

A digital certificate, a structure that contains a signature verification key, expiration time, and serial number and other identifying information. A certificate is used to verify the authenticity of a message signed with an asymmetric signature scheme.

3.2.4 Domain

A virtual grouping of devices under the same authoritative control that shares resources between members. Domains are not constrained to an IP subnet or physical location but are a virtual group of devices, with server resources typically under the control of a network administrator, and clients accessing those resources from different networks or locations.

3.2.5 Duplex

The ability of a communication system to transmit and receive data; half-duplex allows one direction at a time, while full-duplex allows simultaneous two-way communication.

3.2.6 Gateway: A network point that acts as an entrance to another network, often connecting a local network to the internet.

3.2.7 IP Address

A unique numerical label assigned to each device connected to a network that uses the Internet Protocol for communication.

3.2.8 IPv4 (Internet Protocol version 4): The fourth version of the Internet Protocol, using 32-bit addresses to identify devices on a network.

3.2.9 IPv6 (Internet Protocol version 6): The most recent version of the Internet Protocol, using 128-bit addresses to overcome IPv4 address exhaustion.

3.2.10 LAN (Local Area Network)

A network that connects computers within a limited area such as a residence, school, or office building.

3.2.11 Latency

The time it takes for a data packet to move from source to destination, affecting the speed and performance of a network.

3.2.12 Network Topology

The arrangement of different elements (links, nodes) of a computer network, including physical and logical aspects.

3.2.13 Packet

A unit of data transmitted over a network, containing both control information and user data.

3.2.14 Protocol

A set of rules governing the exchange or transmission of data between devices.

3.2.15 TCP/IP (Transmission Control Protocol/Internet Protocol)

A suite of communication protocols used to interconnect network devices on the internet.

3.2.16 Throughput: The actual rate at which data is successfully transferred over a communication channel.

3.2.17 UDP (User Datagram Protocol)
A communication protocol that offers a limited amount of service when messages are exchanged between computers in a network that uses the Internet Protocol.

3.2.18 VLAN (Virtual Local Area Network)
A logical grouping of network devices that appear to be on the same LAN regardless of their physical location.

3.2.19 VPN (Virtual Private Network)
Creates a secure network connection over a public network such as the internet.

3.3 Normative References

The following documents serve as references for cryptographic components used by SIAP:

3.3.1 FIPS 202: SHA-3 Standard: Permutation-Based Hash and Extendable Output Functions: This standard specifies the SHA-3 family of hash functions, including SHAKE extendable-output functions. https://doi.org/10.6028/NIST.FIPS.202

3.3.2 NIST SP 800-185: SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash, and ParallelHash: This publication specifies four SHA-3-derived functions: cSHAKE, KMAC, TupleHash, and ParallelHash. https://doi.org/10.6028/NIST.SP.800-185

3.3.3 NIST SP 800-90A Rev. 1: Recommendation for Random Number Generation Using Deterministic Random Bit Generators: This publication provides recommendations for the generation of random numbers using deterministic random bit generators. https://doi.org/10.6028/NIST.SP.800-90Ar1

3.3.4 NIST SP 800-108: Recommendation for Key Derivation Using Pseudorandom Functions: This publication offers recommendations for key derivation using pseudorandom functions. https://doi.org/10.6028/NIST.SP.800-108

4. Cryptographic Primitives

SIAP relies on symmetric cryptographic primitives intended to provide resilience against both classical and quantum-based attacks. The following sections describe the hash, KDF, entropy, and authenticated-encryption mechanisms used by the implemented protocol. SIAP does not require public-key encryption, digital signatures, certificates, or certificate validation as part of the authentication core.

4.1 Hash Functions and Key Derivation

Hash functions and key derivation functions are essential to SIAP's ability to transform identity strings, passphrases, and server secrets into deterministic authentication tokens and sealing keys. SIAP uses SHAKE for unkeyed tree hashing and cSHAKE for domain-separated derivations.

SHAKE and cSHAKE: SIAP employs the Keccak SHAKE XOF and the cSHAKE derived function for deriving salts, token leaves, passphrase seeds, passphrase verifiers, and RCS sealing keys. Domain-separation labels identify the derivation purpose so that passphrase seeding, passphrase verification, token derivation, and sealed-state encryption are distinct operations.

4.2 Symmetric Cryptographic Primitives

SIAP's sealed token-tree state is protected by the Rijndael Cryptographic Stream (RCS), an authenticated stream cipher used by the implementation to encrypt the token tree and verify a MAC before plaintext is accepted. The RCS operation is initialized with a key, nonce, and associated-data context derived from SIAP state.

- **Wide-Block Cipher Design**: RCS extends the original AES design with a focus on increasing the block size and number of transformation rounds, thereby enhancing its resistance to differential and linear cryptanalysis.
- **Enhanced Key Schedule**: The key schedule in RCS is cryptographically strengthened using Keccak, ensuring that derived keys are resistant to known attacks, including algebraic-based and differential attacks.
- **Authenticated Encryption with Associated Data (AEAD):** The implemented RCS path authenticates the token tree and associated metadata. The associated data includes the SIAP sealing label, the key identity, and the expiration value. Modification of the key identity or expiration therefore invalidates the sealed state during decryption.

The RCS stream cipher is optimized for high-performance environments and provides authenticated encryption for the device token tree. The SIAP specification relies on the correctness of the underlying QSC RCS implementation and requires verification before plaintext token-tree state is used.

4.3 Key Derivation Function

SHAKE Cost Based KDF (SCB) is a cost-based key derivation function used in SIAP for passphrase hardening. The SIAP implementation first derives a fixed-size passphrase seed using cSHAKE with the SIAP-PHASH-SEED label and then supplies that seed to SCB. This preserves the SCB seed-size contract while allowing passphrases of implementation-defined length within the configured maximum.

The SCB KDF architecture is designed to increase the work required for passphrase guessing by enforcing configurable computational and memory costs. The SIAP implementation shall use the SCB parameters defined by the implementation configuration. Those parameters are not modified by this specification and should be benchmarked for each deployment profile.

- **Brute-Force Attacks:** The high computational and memory costs imposed by SCB exponentially increase the time required to test each key, rendering brute-force attacks infeasible within practical timeframes.
- **Dictionary Attacks:** Memory-hardness ensures that generating and storing comprehensive dictionaries would require exorbitant memory resources, making such attacks impractical.
- **Rainbow Table Attacks:** The iterative and memory-intensive nature of SCB disrupts the feasibility of creating effective rainbow tables, as each table entry would necessitate substantial memory resources and computational effort.
- **Side-Channel Attacks:** The deterministic scattering pattern and uniform memory access intervals obscure access patterns, minimizing timing discrepancies and reducing information leakage through side channels.
- **Parallelized Hardware Attacks:** Each write of a cache line to a memory location, writes the cache position index and loop iterator to the key hash, and the entire buffer is written to the hash at each L2 sized interval (default) 256 KiB, sequential operations that make any significant parallelization impossible.

The SCB KDF's architecture is designed to withstand a variety of cryptographic attacks by enforcing both computational and memory hardness.

5. Protocol Description

5.1 Structures

The Server ID

Domain	Server Group	Server
2 bytes	2 bytes	2 bytes

Structure 5.1a: The server identity string (Sid)

The User ID

User Group	User	Key Card
2 bytes	4 bytes	4 bytes

Structure 5.1b: The user identity string (Uid)

The Device ID

Domain	Server Group	Server	User Group	User	Device
2 bytes	2 bytes	2 bytes	2 bytes	4 bytes	4 bytes

Structure 5.1c: The device identity string (Did)

The Key ID

Device ID	Key Index
16 bytes	4 bytes

Structure 5.1d: The token key identity string (Kid)

The Server Key

The server's master key is stored in the server key structure and contains the server base key (Kbase), the server key salt (Dsalt), the server identity (Sid), and the key expiration time as seconds from the epoch.

Parameter	Data Type	Bit Length	Function
Kbase	Uint8 array	256/512	Server Key
Sid	Uint8 array	48	Identity String
Dsalt	Uint8 array	256/512	Secure Salt
Expiration	Uint64	64	Expiration Time

Structure 5.1e: The server key structure.

The Key Tag
The server's user data entry (Ude) is the searchable structure stored on the server that describes a user device profile. It contains the key identity string (Kid), the hash of the device token tree (Khash), and a passphrase verifier derived from the passphrase hash and Kid. The stored verifier is not the direct RCS sealing-key precursor.

Parameter	Data Type	Bit Length	Function
Kid	Uint8 array	160	Key Identification
Khash	Uint8 array	256/512	Token Chain Hash
Pver	Uint8 array	256/512	Passphrase Verifier

Structure 5.1e: The key tag string (Ktag)

The Device Key
The user device state (Uds) is the state structure stored on the memory card that holds information about the key container device and the token key-tree array. This consists of the key identity (Kid), the expiration in seconds from the epoch, and the sealed key-tree array with its authentication tag.

Parameter	Data Type	Bit Length	Function
Kid	Uint8 array	160	Key Identity
Expiration	Uint64	64	Expiration Time
Ktree	Uint8 Array	256/512 * *n*	Sealed symmetric key tree plus MAC

Structure 5.1f: The user key structure

5.3 Server Initialization

The server identity string is assigned to the server, this consists of the server's domain, the server group, and the server's identity, all 16-bit integers. The server identity can be combined with bits from the server group to extend beyond the 65,535 maximum boundary if necessary to extend the number of servers in an organizational domain. The 48-bits representing the server infrastructure however, provide more than 281 trillion possible device identity assignments in total, or more than 4 billion servers per domain.
A base key is generated for the server, this 256-bit (or optionally 512-bit key), is a random derivation key, used to generate the token key-chain authentication tokens for client devices.
A key-chain is a set of key tokens assigned to a client. Each key in the key-chain is securely derived using cSHAKE from the server base key, the SIAP configuration string, and the token key identity string.

The implemented default key tree contains 1,024 token leaves. Each token is 256 bits in the default profile or 512 bits when the extended security profile is enabled.
The 512-bit secure mode is selected by enabling the SIAP_EXTENDED_ENCRYPTION macro. Once the server identity string (S_{id}) has been assigned, the base key (K_{base}) and the server salt (D_{salt}) have been generated, the server is ready to begin loading client memory cards.

5.3 Memory Card Initialization

The memory card is accessed by the server, the server assigns the card to a user group (16-bit identity), as an extended subdomain of the server's domain group.
The server generates a unique 32-bit user identification and a 32-bit device identification, concatenates this with the user group and the server identity string and assigns it to the client completing the device identity (D_{id}).
The server sets the device card expiration time. The server derives Dsalt from cSHAKE using Kbase, the SIAP configuration string, and Sid. For each token-tree member, the server derives a leaf using cSHAKE with Kbase, the SIAP configuration string, and the current Kid, where Kid contains the device identity and key index. The key index is incremented to derive each subsequent leaf.
The server issues a printable passphrase to the user or accepts a provisioned passphrase. The passphrase is first mapped to a fixed-size seed with cSHAKE using the SIAP-PHASH-SEED label and the SIAP configuration string. The seed is then processed by SCB to produce Phash. The server stores Pver = cSHAKE(Phash, SIAP-PHASH-VERIFIER, Kid), not Phash itself, in the user tag structure.
The live Phash, the server salt, and the seal context are used to derive the RCS key and nonce. The seal context contains the SIAP-RCS-SEAL-v1 label, Kid, and expiration, and is also supplied as associated data to RCS. RCS authenticates and encrypts the token tree, adding a MAC to the end of the tree array. The key tree hash (Khash), the key identity, and the passphrase verifier are stored in the user key tag on the server and used to authenticate the card.

5.4 Client Authentication Request

The authentication process begins when the user inserts the memory card; the server reads the plaintext key identification string and matches it to the server's user tag entry. The server loads the user's key tag and the server key. The user supplies the passphrase, which begins the login process. Each step shall succeed before authentication is accepted:

1. The Kid stored on the device card is compared to the Kid stored in the user's key tag for equivalence.
2. The expiration time of the device key is checked against the server key expiration, the current epoch time, and the configured SIAP validity duration.
3. The passphrase is transformed through the cSHAKE-to-SCB passphrase path to produce Phash; a verifier derived from Phash and Kid is compared to the passphrase verifier stored in the key tag.

4. The cipher encryption key and nonce are derived from Phash, Dsalt, and the seal context. RCS is initialized with the seal context as associated data and is used to authenticate and decrypt the device token key tree.
5. The key tree is hashed, and that hash is compared with the key tree hash stored in the user data tag for equivalence.
6. The server generates the expected authentication token corresponding to the current key index before any token leaf is erased or the key index is advanced.
7. The server compares the expected authentication token to the device token at the same index for equivalence.
8. Only after the token comparison succeeds, the device token is copied to the caller output, the token leaf is erased, and the Kid counter is incremented.
9. The device tag is regenerated; the modified token key tree is hashed, and the tag stores the updated Kid, Khash, and passphrase verifier.
10. The key tree is encrypted and authenticated again with RCS, and the memory device is updated with the sealed key tree and the Kid containing the incremented Kidx. The updated device key and tag shall be persisted durably before the authentication result is relied upon by higher layers.

5.5 Server Authentication Response

The authentication process is logged at the server, including errors, device identity, and successful access. The server may use a successful authentication to grant privileges or may provide the consumed token to another component. When the consumed token is used outside SIAP, the consuming component should derive a purpose-specific key from that token using an explicit domain-separation label.

6. Mathematical Description

Mathematical Symbols

← ↔ →	-Assignment and direction symbols.
:=, !=, ?=	-Equality operators; assign, not equals, evaluate.
conf	-The configuration string.
D_{salt}	-The server salt array.
exp	-The expiration time in seconds from the epoch.
K_{base}	-The server's base secret key.
K_{hash}	-The key-tree hash.
K_{id}	-The key identity string.
K_{idx}	-The key-tree index.
K_{tree}	-The key-tree array.
G(*n*)	-The pseudo-random bytes generator.
P_{hash}	-The passphrase hash.
RCS	-The RCS AEAD stream cipher.
SCB	-The SCB cost-based key derivation function.
SHAKE	-The Keccak SHAKE XOF function.
cSHAKE	-The customizable SHAKE function used for domain-separated derivations.
S_{kd}	-The server key.
U_{dt}	-The server's user database tag entry.
U_{kd}	-The user key structure.
U_{id}	-The user identity string.
Pver	-The server-stored passphrase verifier derived from Phash and Kid.

6.1 Server Initialization

The server generates the base secret (256-bit or 512-bit), from which all client key trees are derived.

$K_{base} \leftarrow G(n)$

The server populates a 48-bit server identity string. The domain is a 16-bit integer representing the server's domain, the server-group is a 16-bit integer representing the server group that the server is assigned to, and the server field is a unique 16-bit integer identifier representing the server identity.

$S_{id} \leftarrow \{ \textit{domain}, \textit{server-group}, \textit{server} \}$

The server creates a salt value by initializing cSHAKE with the base key, the configuration string, and the server identity string.

Dsalt ← cSHAKE(Kbase, conf, Sid)

The server key is constructed from the server identity string, base key, the server salt, and the expiration time.

$S_{kd} \leftarrow \{ S_{id}, K_{base}, D_{salt}, \textit{exp} \}$

6.2 User Card Initialization

The user inserts the memory card into the server, and creates a user account. The server assigns the user to a *user-group* and assigns a user identity string consisting of a 16-bit user group string, and a 32-bit unique *user identification* string, and a 32-bit *device identity* string.

$U_{id} \leftarrow \{ \textit{user-group}, \textit{user-id}, \textit{device-id} \}$

The key identity combines the 48-bit string representing the server's identity, the 96-bit user identity string, and a 32-bit counter which is the current *key-chain index*.

$K_{id} \leftarrow \{ S_{id} \,||\, U_{id} \,||\, K_{idx} \}$

The server creates the key tree by initializing cSHAKE with the base key, the configuration string, and the device key identity string. For every key created, the key index value in Kid is incremented, ensuring a unique derivation context for every token tree member.

where: (i = 0, 1, 2, ... *n*),
Ktreei ← cSHAKE(Kbase, conf, Kid_i)

The server hashes the key tree, derives a passphrase hash through cSHAKE and SCB, derives a server-stored verifier from the passphrase hash, and stores the key identity string, key-tree hash, and verifier in the user data tag.

Seed ← cSHAKE(passphrase, SIAP-PHASH-SEED, conf)
Phash ← SCB(Seed)
Pver ← cSHAKE(Phash, SIAP-PHASH-VERIFIER, Kid)

The server computes the key-tree hash and constructs the server user tag. The tag stores the verifier Pver rather than the direct sealing input Phash.

Khash ← SHAKE(Ktree)
Udt ← { Kid, Khash, Pver }

The server generates a cipher encryption key and nonce from the live passphrase hash, server salt, and seal context. The same seal context is supplied to RCS as associated data.

SealInfo ← { SIAP-RCS-SEAL-v1 || Kid || exp }

k, n ← cSHAKE(Phash, SIAP-RCS-SEAL-v1, Dsalt || SealInfo)
Ektree ← RCSk,n[SealInfo](Ktree)
Ukd ← { Kid, Ektree, expiration }

6.3 Client Authentication Request

The user inserts the memory card and begins the login process. The server reads the key identity string from the memory device. Using the key identity, the server fetches the user's key tag from the data-set.

If the user and key are identified, the user is prompted for the passphrase. The passphrase is transformed through cSHAKE and SCB to produce the live passphrase hash.

Seed ← cSHAKE(passphrase, SIAP-PHASH-SEED, conf)
Phash ← SCB(Seed)

The server loads the user device key and extracts the plaintext key identity string. This string is used to fetch the user key tag, and a comparison is made between the key identity strings on the key tag and the device key for equivalence.

$U_{kd}\_K_{id} = U_{dt}\_K_{id}$? true : false

If the key identity strings match, the server verifies the device expiration against the server key expiration, the current epoch time, and the configured SIAP validity duration.

ValidTime(Ukd_exp, Skd_exp, now) ? true : false

If the time check succeeds, the server derives the passphrase verifier from the live passphrase hash and compares it to the verifier stored in the user key tag.

Pver′ ← cSHAKE(Phash, SIAP-PHASH-VERIFIER, Kid)
Pver′ = Udt_Pver ? true : false

If the verifier is equivalent, the server authenticates and decrypts the device key token tree using the live passphrase hash, the server salt, and the seal context.

SealInfo ← { SIAP-RCS-SEAL-v1 || Kid || exp }
k, n ← cSHAKE(Phash, SIAP-RCS-SEAL-v1, Dsalt || SealInfo)
Ktree ← RCS-Openk,n[SealInfo](Ektree)

If the key tree ciphertext is authenticated and decrypted, the server verifies the key-tree hash before accepting any token state.

Khash′ ← SHAKE(Ktree)
Khash′ = Udt_Khash ? true : false
token′ ← cSHAKE(Kbase, conf, Kid)

The server compares the expected token to the current token-tree leaf before erasing the leaf or advancing the index.

token ← Ktree(Kidx)

The two tokens are compared for equivalence. If they are identical, authentication has succeeded and state advancement may occur.

token = token′ ? true : false

After a successful comparison, the server erases the consumed tree leaf and increments the key index.

Erase(Ktree(Kidx))
Kidx += 1

The server hashes the modified key tree, derives the updated passphrase verifier, and writes the updated tag state.

Khash ← SHAKE(Ktree)
Pver ← cSHAKE(Phash, SIAP-PHASH-VERIFIER, Kid)
Udt ← { Kid, Khash, Pver }
k, n ← cSHAKE(Phash, SIAP-RCS-SEAL-v1, Dsalt || SealInfo)
Ektree ← RCSk,n[SealInfo](Ktree)
Ukd ← { Kid, Ektree, expiration }

7. Security Analysis

The following discussion evaluates SIAP against threat models typical of regulated-finance, payment-terminal, local-console, and offline-value-transfer deployments. It assumes SHAKE and cSHAKE behave as secure sponge-based XOFs, SCB provides the configured cost amplification for the derived passphrase seed, RCS provides authenticated encryption for the sealed token tree, and updated SIAP state is committed durably and atomically after a successful authentication.

7.1 Adversary classes considered

- **A-NET - Network attacker**
  Intercepts, replays, or injects protocol traffic but cannot read the sealed card or server secrets.
- **A-TOK - Token thief**
  Controls the physical memory card but lacks the passphrase.
- **A-DB - Insider database reader**
  Obtains Ude rows (U_{dh}, K_{id}^{i}) *and log files, but neither* K_{base} *nor the card.*
- A-QC – Future quantum adversary
  Possesses a large-scale fault-tolerant quantum computer and all stored ciphertext and traffic.
- **A-SRV - Compromised host**
  Gains full OS access, including Kbase. Such compromise exposes future derivations for the affected server and is outside the protection boundary of the SIAP authentication core.

7.2 Security goals and how SIAP meets them

Goal	Mechanism(s)	Holds against
Two-factor MFA	• Sealed token state requires card possession. • Passphrase verifier requires knowledge of the passphrase. • Kid check rejects wrong device state.	A-NET, A-DB
Forward secrecy (future sessions)	• One-time leaf burn after successful token comparison. • Index monotonically increments when updated state is durably committed.	A-TOK, A-SRV
Quantum resilience	• SHAKE, cSHAKE, SCB, and RCS are symmetric/hash-based. • Default 256-bit profile targets a generic 128-bit post-quantum margin.	A-QC
Replay & rollback resistance	• Kid and Khash are checked. • RCS seal context binds Kid and expiration. • Durable persistence and per-identity serialization are required.	A-NET, A-TOK
Offline operation	• All verification uses local K_{base}; no CA or CRL lookup. • Start/End-time fields enforce validity window.	All

7.3 Detailed attack-surface review

1. **Offline dictionary attack on passphrase**
   Path: steal card and attempt to test candidate passphrases against the sealed token tree.
   Cost factors:
   -The cost is determined by the configured SIAP SCB parameters and the implementation platform. This specification does not alter those parameters.
   -The passphrase is first mapped to a fixed-size seed with cSHAKE and then processed by SCB. Mitigations include a minimum passphrase entropy policy, rate limiting, token lockout policy, or secure-element enforcement for retry counters.
2. **Token clone and rollback**
   Path: copy ciphertext, boot from stale image, or restore an older server tag.
   Barrier chain:
   -The key identity and expiration are authenticated in the RCS seal context, so direct metadata modification invalidates sealed-state authentication.
   -The Khash check rejects token-tree modification, and the current token is compared before a leaf is erased or the index is advanced.
   -Rollback resistance requires durable server-side persistence of the advanced tag and device state, and concurrent authentications for the same identity shall be serialized.
3. **Server database leak**
   Data revealed: Kid, Khash, Pver, and log metadata. The stored verifier is not the direct RCS sealing-key precursor. A database leak alone does not reveal token leaves or permit decryption of the sealed tree without the live passphrase hash and the server salt.
4. **Full server compromise**
   Impact: compromise of Kbase and server-side state permits future token derivation for the affected server and can defeat server-side verification. Prior consumed leaves remain erased from the device tree if state erasure and persistence were completed before compromise.
   -Place K_{base} in an HSM; expose SHAKE via an authorized hash-derivation command.
   -Rotate K_{base} periodically; re-issue cards by exporting new K_{id}.
5. **Quantum adversary**
   Grover's algorithm gives a square-root advantage against generic symmetric search. The default 256-bit symmetric profile therefore targets approximately 128-bit post-quantum security against generic search, while the extended profile increases the symmetric margin.

7.4 Side-channel considerations

- **SCB execution**
  SCB should be implemented without secret-dependent behavior that exposes passphrase information. Deployment profiles shall benchmark the configured SCB parameters on the target platform and enforce rate limits or retry policies appropriate to the threat model.
- **Leaf derivation**
  SHAKE sponge operations are data-independent; EM and power leakage reduced further if executed inside a secure element.

- **Monotonic counter**
  Implement in hardware (SE fuse or TPM NV-index) to survive brown-outs and defeat glitch roll-backs.

7.5 Residual risks and operational mitigations

- **Key-chain exhaustion DoS -** An attacker with both the card and passphrase can deliberately consume leaves. The implemented default tree contains 1,024 leaves; deployments shall monitor remaining leaves and re-provision or rotate cards before exhaustion.
- **Token/DB desynchronization -** Power loss or process failure between device-state and server-tag writes can desynchronize the protocol. Conforming deployments shall use atomic persistence, a two-phase commit, a write-ahead log, or an equivalent recovery mechanism that prevents leaf reuse and rollback acceptance.

7.6 Security posture summary

- **Cryptanalytic robustness -** Under the stated assumptions for SHAKE, cSHAKE, SCB, and RCS, no shortcut attack is specified against the implemented SIAP derivations. Security depends on protecting Kbase, preserving passphrase entropy, using the configured SCB path correctly, authenticating sealed-state metadata, and committing state advancement durably.
- **Post-quantum readiness -** The SIAP authentication core uses symmetric and hash-based primitives and does not depend on integer factorization, discrete logarithms, elliptic-curve discrete logarithms, or certificate-chain validation.
- **Operational controls -** HSM or equivalent protection for Kbase, durable state persistence, per-identity concurrency control, passphrase retry policy, token exhaustion monitoring, and explicit downstream key derivation are required to realize the security claims in deployed systems.

Taken together, SIAP defines a compact symmetric authentication mechanism suitable for environments requiring local verification, offline operation, and post-quantum security margins. Its guarantees are strongest when the cryptographic state transitions specified above are paired with durable storage, rollback protection, and disciplined operational handling of server secrets.

8. Real-World Use-Case Scenarios

8.1 Card-Data-Environment (CDE) Console Access - PCI DSS 4.0

A bank's payment-operations network maintains a few hundred jump-hosts that administrators must reach several times each day. Each jump-host embeds a hardware-security-module slot that protects the server's $\mathbf{K_{base}}$; the operating system calls that HSM for the single SHAKE-256 derivation required by SIAP.

- **Scale** The implemented default profile provides 1,024 leaves per token. Console MFA deployments with frequent administrator logins shall monitor remaining leaves and rotate or re-provision cards on a scheduled basis rather than assuming a larger implicit Kn.
- **Integration** A PAM plug-in on every jump-host invokes SIAP before SSH credentials are accepted, writing an audit line that includes the new K_{idx}. No changes to the bank's existing RADIUS servers or network firewalls are required, because the token merely delivers a one-time symmetric key that the plug-in inserts into the existing SSH "keyboard-interactive" flow.

8.2 Offline Central-Bank Digital - Currency (CBDC) Wallets

A national central bank wants residents to spend up to 200 small transactions while travelling beyond network coverage. Each plastic card contains a low-cost MCU with an SRAM-PUF that derives K_{ci}, so no per-device secret must be injected at personalization.

- **Scale** The default 1,024-leaf profile supports limited offline spending windows. Larger payment pilots require explicit re-provisioning intervals, a defined larger-profile extension, or multiple token epochs.
- **Integration** When terminals are online, they push the spent K_{idx} list to the CBDC ledger; reconciliation rules match those indices to the core ledger and flag double-spend attempts. No public-key certificates are stored on the card, keeping BOM below US $ 1.50.

8.3 Field-Technician Access to ATMs, POS and SCADA Nodes

Equipment vendors issue SIAP USB-C tokens to 2,500 technicians world-wide. Each machine (ATM, PLC, router) holds its own K_{base} in firmware and exposes a SIAP-over-UART service that runs before any OEM back-door shell.

- **Scale** Typical field-service deployments shall size rotation intervals around the implemented 1,024-leaf default or define a larger profile. A stolen token should be subject to a retry or lockout policy implemented by the reader or secure element.
- **Integration** Existing maintenance laptops load a 20-kB SIAP shared library; the rest of the tool-chain (serial console, SSH, vendor GUI) is unchanged. Roll-out requires no VPN because authentication is device-local.

8.4 Institutional Cold-Wallet Custody

A digital-asset custodian stores Bitcoin, Ether, and tokenized securities in FIPS-140-3 vault HSMs. A SIAP smart-card sits in a Faraday bag and must be tapped by two officers ("four-eyes") to release the daily withdrawal batch.

- **Scale** Very small card fleet (≤ 100), but *Kn* intentionally tiny (≤ 200) so the institution must rotate cards at most every six months, enforcing regular audits.

- **Integration** The HSM uses K_{tok} as input key to KMAC-256, signing an approval object that the existing Multisig contract understands. No change to on-chain scripts; SIAP stays entirely in the signing workflow.

8.5 OEM Secure-Boot & Firmware Licensing

An IoT chip maker wants each production-line tester to decrypt firmware only once per device. A fixture holds a SIAP card whose K_{idx} matches the station's progress counter.

- **Scale** High-volume production-line use requires explicit token rotation, larger-profile support, or automated re-provisioning. The default 1,024-leaf profile is suitable for controlled batches, not indefinite production-line operation.
- **Integration** A 100-line stub, compiled into the existing flashing tool, feeds the derived leaf into an AES-CTR decrypt step; no further MES or SAP modifications are needed.

8.6 Zero-Trust PSK Replacement for TLS and IKEv2

A payment gateway terminates 30,000 inbound POS sessions. Instead of static pre-shared keys stored in text files, the gateway calls SIAP to obtain a fresh PSK for each TLS 1.3 connection.

- **Scale** High-rate gateway use requires a larger specified profile, multiple device tokens, or frequent re-provisioning. The implemented default profile shall not be described as supporting millions of sessions without an explicit extension.
- **Integration** A reverse-proxy plug-in inserts TLS_PSK = SHAKE(0x01‖Ktok) into OpenSSL's PSK callback, leaving certificates untouched for browsers.

8.7 High-Frequency Trading API Auth

A trading engine running beside an exchange's matching engine must re-key in under 50 µs. SIAP's single SHAKE call (deterministic 1.2 µs on Intel Ice Lake) meets the budget.

- **Scale** High-frequency trading use requires careful profile definition and rotation policy. The default 1,024-leaf token tree is appropriate only for bounded authorization events unless a larger tree profile is specified.
- **Integration** K_{tok} directly keys RCS-256 for message encryption; the FIX over-TLS stack is bypassed, shaving one RTT and 90 µs median latency.

8.8 Kiosk & Critical-Infrastructure Human-Machine Interfaces

Power-substation HMIs accept SIAP tokens locally; the central SCADA server sees only a boolean permit.

- **Scale** Kiosk and HMI deployments shall monitor remaining token leaves and rotate cards before exhaustion. The default 1,024-leaf profile supports periodic administrative access, not unlimited long-term reuse.
- **Integration** An ARM-TrustZone applet verifies SIAP and toggles a GPIO line that enables the HMI keyboard; the legacy control software remains unchanged.

Cross-cutting integration guidance

- **Transport neutrality** Because SIAP produces a symmetric secret, existing stacks can consume a derived key as a PSK, MAC key, or decryption key. The integration should derive protocol-specific keys from the SIAP token with explicit domain separation.
- **Key-management continuity** The server retains its native account and logging infrastructure; SIAP adds exactly one table (Ude) and one HSM secret (K_{base}).

- **Hardware flexibility** Tokens range from US $0.30 MCU+PUF cards to CC-certified Secure-Elements; all share the same on-card file format, simplifying field upgrades and mixed-fleet management.

These deployment sketches demonstrate how SIAP scales from single-token cold-wallet ceremonies to millions of offline CBDC payments, and how it bolts onto existing TLS, SSH, or proprietary tunnels with minimal code, in every case providing forward-secret, quantum-resistant authentication without a certificate authority.

Conclusion

Secure Infrastructure Access Protocol responds to a practical need for post-quantum authentication in devices and systems that must often operate without network reach or public-key infrastructure. By anchoring authentication in SHAKE, cSHAKE, SCB, and RCS, SIAP provides a compact symmetric construction in which possession of sealed token state and knowledge of a passphrase are both required for successful access.

The security analysis shows that the server base key defines the principal derivation boundary. Loss of Kbase compromises the affected server domain and requires re-provisioning. Other attack classes are constrained by the sealed token tree, the passphrase-derived verifier, constant-time token comparison, authenticated sealed-state metadata, and one-time leaf consumption, provided that persistence and concurrency controls enforce monotonic state advancement.

Operationally the protocol is deliberately small. Host integration requires loading the server key, reading the device key and tag, deriving the live passphrase hash, authenticating and decrypting the sealed tree, comparing the current token, and committing the advanced state. These operations can be integrated into login, local key-release, secure-storage, or controlled-device authorization workflows without adding certificate infrastructure.

SIAP is therefore positioned as a portable symmetric authentication primitive rather than as a replacement transport protocol. Its consumed token may be converted into a PSK, MAC key, or application authorization key through a domain-separated derivation. Its plaintext identity hierarchy allows revocation or replacement at the card, user, group, or server scope when supported by the surrounding account and persistence system.

Future profiles may define larger token trees, secure-element storage profiles, stronger at-rest protection for server state, and conformance tests for serialization and state transitions. The architectural core remains a sponge-based authentication mechanism whose security depends on disciplined implementation of its derivations, authenticated state sealing, and durable monotonic state management.