Secure Infrastructure Access Protocol - SIAP

Revision 1.0a, July 20, 2025

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

This document is an engineering level description of the secure infrastructure access protocol SIAP; a user verification system and token generation and access mechanism.

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 the physical possession of a removable memory card with a passphrase, producing a forward-secret, single-use key that is verified by a host server. Because every authentication event irreversibly destroys the currently active chain key and advances a monotonic counter on the memory access card, compromise of either factor after a successful session yields no usable key material for replay or retrospective decryption.

The construction is intentionally minimalist. All long-lived secrets including the server base key, the user-specific key tree, and the encrypted card payload, are derived from the Keccak SHAKE XOF function, and is protected by a memory-hard key-derivation step using the SCB cost-based key derivation function. No public-key primitives, certificates, or online revocation infrastructure are required. This eliminates exposure to Shor-class quantum computers while keeping the total code footprint below thirty kilobytes, a constraint that allows deployment inside low-cost IoT micro-controllers, secure elements, and Smart-card class memory devices.

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 nonce with the current key-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 offline resilience. During initial provisioning the server encrypts both the user-key structure and a bounded key-chain with SCB, writes the resulting cipher-text to the memory card, and stores only the plain-text identity string on the memory card. At run time the host needs no external registrar: it verifies the plain-text identity header, fetches the user identity data from the server, decrypts the encrypted key data structure and key-chain token with the user’s passphrase, recomputes one SHAKE-derived leaf, and grants or denies access. Consequently, SIAP supports remote kiosks, isolated manufacturing cells, and payment terminals that may remain disconnected from any central authority for prolonged periods.

Time-bounded validity is enforced by embedding start-time and end-time fields directly into the sealed user-key structure, ensuring the host need only interpret two absolute epoch counters to reject stale or pre-dated cards. Because the server replicates these fields inside its own identity record, a mismatch is detected before the costly SCB operation is attempted, conferring a built-in rate-limit against brute-force attacks on the passphrase.

Forward secrecy extends from the one-time-key discipline. Each successful session erases the active key on the card, increments the key-chain index, recalculates the sealing keystream using the same passphrase and the new key-chain index value, and rewrites the encrypted payload in place. Even a complete capture of the token immediately after logout reveals only ciphertext that cannot be decrypted without the passphrase and yields no information about prior leaves, as they are no longer derivable from any retained secret.

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 intentionally compatible to transport protocols. The freshly minted key token may be consumed directly by a symmetric cipher such as RCS-256 to decrypt local storage, injected as a pre-shared key into TLS 1.3 or IKEv2, or hashed once more to sign application-layer requests. By separating authentication from channel establishment, the protocol serves as a drop-in post-quantum upgrade for legacy PSK schemes without altering existing wire formats, thereby enabling incremental adoption across fintech back-offices, critical-infrastructure consoles, and high-frequency trading gateways.

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. By deriving every long-lived value from SHAKE and a memory-hard KDF, the protocol eliminates reliance on public-key infrastructures and remains secure against future quantum adversaries while fitting within the code and memory budgets of smart cards, IoT micro-controllers, and secure elements. SIAP is intended to operate without online trust anchors, enabling secure logins, key releases, and transaction authorizations in environments ranging from tightly regulated fintech data centers to fully offline payment and industrial-control deployments. Because each successful session burns its active key and advances a monotonic counter, the protocol provides forward secrecy, natural replay resistance, and an auditable trail of access events, all without altering existing transport or application protocols.

Below are **eight concrete use-case patterns** that map cleanly onto the capabilities and constraints described in the current SIAP specification (Rev-1.0a). Each entry lists why SIAP is a fit, what elements of the spec it exercises, and any small profile tweaks that make deployment smoother.

#	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* $K_n = 65{,}536 \rightarrow$ ~10 years @ 20 logins/day. Store K_{base} in 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 call. • Start/End-time fields in Uks enforce spend-window.	Keep card BOM low by using MCU+PUF profile; set $\mathbf{K_n \approx 15\,000}$ (1 yr of micro-payments).
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" (64-byte keys) + $\mathbf{K_n}$ **small** (100-500) to force ceremony for large withdrawals.
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.	Write once, lock-down environment; after $K_{idx} == K_n$ line automatically halts for re-issuance.
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.	Provide tiny "TLS-adapter" that domain-separates key: TLS_PSK = SHAKE(0x01‖Ktok).
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.	Pre-cache next leaf on card to avoid SCB decrypt on every order burst.
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: fail closed if K_{idx} jumps >1 (tamper indicator).

2. Scope

This document describes the SIAP secure messaging protocol, which is used to authenticate a client device to a host server. This document describes the complete authentication protocol, card initialization, client authentication, and server response. This is a complete specification, describing the cryptographic primitives, the derivation functions, and the complete client and server messaging protocol.

2.1 Application

This protocol is intended for institutions that implement secure authentication to servers, and exchange of authenticated and encrypted cryptographic tokens.

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 KMAC

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

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 QSTP:

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 a robust set of symmetric cryptographic primitives designed to provide resilience against both classical and quantum-based attacks. The following sections detail the specific cryptographic algorithms and mechanisms that form the foundation of SIAP 's encryption, key exchange, and authentication processes.

4.1 Hash Functions and Key Derivation

Hash functions and key derivation functions (KDFs) are essential to QSTP's ability to transform raw cryptographic data into secure keys and hashes. The following primitives are used:

SHAKE: SIAP employs the Keccak SHAKE XOF function for deriving symmetric keys from shared secrets. This ensures that each session key is uniquely generated and unpredictable, enhancing the protocol's security against key reuse attacks.

4.2 Key Derivation Function

SCB is a cost-based key derivation function, one that can increase the memory usage and computational complexity of the underlying hash function. Suitable for password hashing, key generation, and in cases where brute-force attacks on a derived key must be strongly mitigated.

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

- **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

Domain	Server Group	Server
2 bytes	2 bytes	2 bytes

Structure 5.1a: The server identity string (Sid)

User Group	User	Key Card
2 bytes	4 bytes	4 bytes

Structure 5.1b: The user identity string (Uid)

Domain	Server Group	Server	User Group	User	Key Card ID
2 bytes	2 bytes	2 bytes	2 bytes	4 bytes	4 bytes

Structure 5.1c: The key identity string (Kid)

Key Card ID	Key Index
4 bytes	4 bytes

Structure 5.1d: The key tag string (Ktag)

The server's user data entry (U_{de}) is the searchable structure stored on the server that describes a user profile. This is a hash of the user's state (U_{dh}) and the extended key identity strings ($K_{id}^{(+Kidx)}$) associated with this user.

Parameter	Data Type	Bit Length	Function
Udh	Uint8 array	256	State Verification
Kid + Kidx	Uint8 array	224 * *n*	Key Identification

Structure 5.1e: The key tag string (Ktag)

The user key state (U_{ks}) is the state structure stored on the memory card that holds information about the access tokens. This is the user identity (U_{id}), the server identity (S_{id}), the valid time *from* and *to* fields, the key identity (K_{id}), and the key-chain index (K_{idx}).

Parameter	Data Type	Bit Length	Function

Uid	Uint8 array	80	User Identity
Sid	Uint8 array	48	Server Identity
Start From	Uint64	64	Valid Time
Start To	Uint64	64	Expiration Time
Kid	Uint32	32	Key Card Identity
Kidx	Uint32	32	Key-Chain Index

Structure 5.1f: The user key state 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 16-bit unsigned integer 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 key-chain key-sets for client devices. A key-chain is a set of key tokens assigned to a client (unsigned 8-bit integer arrays of key size length), each key in the key-chain is securely derived using the SHAKE XOF function. The key-chains have a fixed number of keys in the set; each key is 256 or 512 bits, selected with the operational mode constant SIAP_SECURITY_MODE_EXTENDED.
Once the server identity string (S_{id}) has been assigned, and the base key (K_{base}) has been generated, the server is ready to begin loading client memory cards.

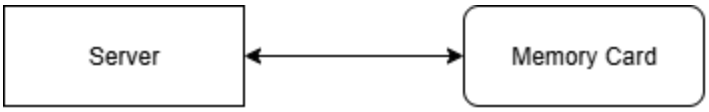
1) The server is assigned an identity string, representing its domain, server-group, and server identity.
$S_{\text{id}} \leftarrow \{\text{domain}, \text{server-group}, \text{server-id}\}$

2) The server writes the truncated key identity string to the memory card header.
$K_{base} \leftarrow G(n)$

5.3 Memory Card Initialization

The memory card is accessed by the server, the server assigns the card to a user group, and a 16-bit domain group. The server generates a unique 32-bit user identification and assigns it to the client completing the user identity (U_{id}) string. The server populates the card's user key state structure (U_{ks}) by assigning the user identity string, the server identity string (S_{id}), the *from* and *to* valid times, and the key tag, which contains a unique 32-bit key card identifier (K_{ci}), and a 32-bit key-chain index (K_{idx}).
The server copies a partial identity string representing the key's full identity to the memory card's plaintext header, consisting of the server identity, user identity, valid *to* and *from* times,

and the key identity. The valid *to* and *from* times are seconds from the epoch that determine the memory card's starting and expiration times. The key identity uniquely identifies the memory card; users can possess more than one memory card for a server, and these values are stored as serialized K_{id} strings in a server database under the user's profile data.
The user's key state structure is hashed, and becomes a field in the server's user profile. The user data entry consists of the hash of the memory card's key state structure, and any key identities assigned to that user. The key state hash is used as a verification stage during the client login, where the memory card's state is decrypted, the state is hashed and that hash is compared to the hash stored in the server's user data entry.
Once the key identity string is written to the memory card's header, and the card's key state is populated, the server generates the card's key-chain; a set of 256-bit or 512-bit keys ranged to a global constant value: SIAP_KEYCHAIN_COUNT (or K_n in the notation), which has a default value of 65,536, but can be increased to match the memory profile of the memory card device. The base key (K_{base}) is a random derivation key of either 256 or 512 bits in length. This base key is hashed using the SHAKE XOR function (universally these are set to either SHAKE-256 or SHAKE-512 depending on the *security mode* setting). The hash is keyed with the base key, a hash of the user's state (U_{dh}) and an incrementing counter. This counter is a monotonic incrementing nonce that matches each key-chain index position between zero and the key-chain maximum keys (K_n). The keys are added consecutively to the key-chain, and pulled from the chain by calculating the point in the contiguous key-chain array that is the index value multiplied by the individual key size, represented globally as: SIAP_KEYCHAIN_KEY_SIZE. Likewise, the server, which reproduces this token key as part of the login authentication, needs only call a single hash function iteration by using the correct key index, to reproduce the key token.



1) The memory card is inserted and the server writes it's identity string, assigns the user to a group, and generates the user and key identity strings and adds them to the memory card state.

$$U_{ks} \leftarrow \{S_{\text{id}}, U_{\text{id}}, \text{from}, \text{to}, K_{tag}\}$$

2) The server writes the truncated key identity string to the memory card header.

$$K_{\text{id}} \leftarrow \{S_{\text{id}}, U_{\text{id}}, \text{from}, \text{to}, K_{ci}\}$$

3) The server adds the key identity to a databse entry that contains a hash of the client's state, and each key identity belonging to the user.

$$\begin{aligned} U_{de} &= \{ \\ &\quad U_{dh} = H(U_{ks}) \\ &\quad K_{\text{id}} \leftarrow \{S_{\text{id}}, U_{\text{id}}, \text{from}, \text{to}, K_{tag}\}\} \end{aligned}$$

4) The server generates the key chain by hashing the server's base key with the user identity hash and the key index.

$$\begin{aligned} &\text{for}_{\ i} = 0 \rightarrow K_n \\ &\quad K^i_{\text{chain}} \leftarrow H(K_{base} || U_{dh} || i) \end{aligned}$$

5) The server generates a unique passphrase, and uses SCB to generate a keystream.

$$\text{ks} \leftarrow SCB(passphrase \ || \ K_{\text{id}x}, n)$$

6) The server encrypts the user key data and the key-chain.

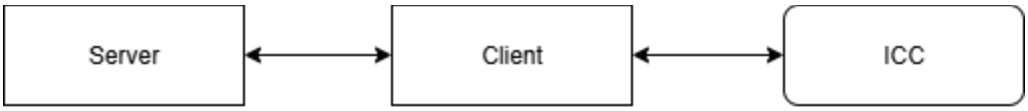
$$ecpt \leftarrow ks \oplus (U_{ks} \ || \ K_{\text{chain}})$$

The memory card is now loaded with the user key state and the key-chain. The server generates a passphrase, and keys the SCB KDF with the passphrase, the key index (initialized at zero), and the serialized user state. The KDF generates a keystream equal in length to the serialized user key state and keychain in length, and performs a bitwise XOR with the keystream, encrypting the card's key state and keychain.

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 data entry. The server loads the data, which includes a hash of the cards key state structure, and any memory card's key identity strings associated with the user.
The client is prompted for the passphrase. The passphrase is combined with the current key index value stored in the key identification string (K_{id}) on the server, and used to key the SCB cost-based KDF function. The SCB function generates a keystream that is used to decrypt the memory card's key state structure and the key-chain containing all of the card's key tokens.

The server stores a copy of the key card's state hash (U_{dh}), the client hashes the decrypted copy of the user key state, and the server compares the two values for equivalence. If the two values match, the client compares the key index value stored in the key identity on the server with the one stored in state, if they match, the key in the key-chain at this index is decrypted.



1) The client initiates the login process, and the server reads the key identity string. The server retieves the user databse entry.

$$U_{de} = \{$$
$$U_{dh} = H(U_{ks})$$
$$K_{\text{id}} \leftarrow \{S_{\text{id}}, U_{\text{id}}, \text{from}, \text{to}, K_{tag}\}\}$$

2) The client is prompted for the passphrase, the passphrase is combined with the key-chain index from the server's copy of the key identity, and used to generate a keystream with the SCB KDF.

$$ks \leftarrow SCB(passphrase \mid\mid K_{\text{idx}}, n)$$

3) The keystream is used to decrypt the user key struct on the memory card. This key structure is serialized and hashed and compared with the hash of the user key structure stored in the server's user database entry.

$$U'_{ks} = ks \oplus ecpt$$
$$U'_{dh} = H(U'_{ks})$$
$$U_{dh} \equiv U'_{dh}$$

4) If the hash is equal, the client decrypts the chain key at the key index.

$$K_{\text{tok}} \leftarrow K^{i}_{\text{chain}} \oplus ecpt^{\text{kidx}\cdot i}$$

The key token is passed to the server for authentication verification.

5.5 Server Authentication Response

The client passes the key token to the server. The server generates a new token using the base derivation key, and the user's key data hash and the current key index taken from the server's user data entry. The server generates the key and compares it with the key token created by the client and compares them for equivalence. If the two values are identical, the authentication has succeeded, and the key token can be used to key a cryptographic function for further processing of data.

1) The client sends the server the key token extracted from the keychain. The server recreates the key token by hashing its base key, the hash of the user state structure, and the current key index.

$$K^{i}_{\text{chain}} \leftarrow H(K_{base}||U_{dh}||K_{\text{idx}})$$

2) The server compares the locally generated key with key token provided by the client for equivalence.

$$K^{i}_{\text{chain}} \equiv K_{\text{tok}}$$

3) If the key generated on the server matches the key token from the client, the client has been authenticated, and the token can be used as a random seed in a secondary transaction.

6. Mathematical Description

Mathematical Symbols

← ↔ →	-Assignment and direction symbols.
:=, !=, ?=	-Equality operators; assign, not equals, evaluate.
C	-The client host.
S	-The server host.
H	-The SHAKE-256 hash function.
K_{base}	-The server's base secret key.
K_{chain}	-The key-chain array.
K_{id}	-The key identity string.
K_{idx}	-The key-chain index.
K_n	-The key maximum count.
K_{ci}	-The key card identifier.
K_{tag}	-The key tag (K_{ci} + K_{idx}).
G(*n*)	-The pseudo-random bytes generator.
SCB	-The SCB cost-based key derivation function.
S_{id}	-The server identity string.
U_{de}	-The server's user database entry.
U_{dh}	-The user key structure hash.
U_{id}	-The user identity string.
U_{ks}	-The user key structure.

6.1 Server Initialization

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

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

The server populates a 64-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.

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

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.

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

The key tag combines the 32-bit integer representing the memory card's unique *key card identifier* value, and a 32-bit integer which is the current *key-chain index*.

$K_{tag} \leftarrow \{ K_{ci} \parallel K_{idx} \}$

The server populates the *user key structure* on the memory card by setting the *valid start-time* and *end-time*; the period (in seconds from the epoch) that the key card is considered valid. The server adds the server identity, the user identity, and adds the 64-bit *key tag*, uniquely identifying the memory card and the current position within the key-chain (initialized at zero).

$U_{ks} \leftarrow \{ U_{id}, S_{id}, \textit{start-time}, \textit{end-time}, K_{tag} \}$

The server hashes the *user key structure*, and stores the hash in the server's *user data hash*.

$U_{dh} \leftarrow H(U_{ks})$

The key identity string is created and stored in plain-text in the key file header. The key identity contains the server hierarchy domain\server-group\server and the user hierarchy user-group\user\key-card, and the start and end valid times.

$K_{id} \leftarrow \{ S_{id} \parallel U_{id} \parallel K_{ci} \parallel \textit{start-time} \parallel \textit{end-time} \}$

The server's user data-set entry consists of the *user data hash*, and the *key identity* strings belonging to that user. The key identity string is extended to include the key-chain index value K_{idx}, by replacing K_{ci} with K_{tag} in the string. The key-chain index value in the server's K_{id} string is later compared with the decrypted key-chain index in the memory card's user key structure for equivalence.

$U_{de} \leftarrow \{ U_{dh}, K_{id}^{i} ... \}$

The server generates the key-chain by hashing the server's base derivation key with the user data hash and a nonce. The nonce parallels the key-chain index; with each call to the hash function producing a single key added to the key-chain that corresponds to the index number of the key-chain index. The number of keys created is set to a global constant key count represented by K_n, if the current key index exceeds K_n in the key identity string, the memory card is rejected.

For i equals 0 to K_n

$K_{chain}^{i} \leftarrow H(K_{base} \,||\, U_{dh} \,||\, i)$

The server prompts the user for a passphrase that is combined with the current key-chain index and passed to the SCB KDF.

$ks \leftarrow SCB(passphrase \,||\, K_{idx}, n)$

The resulting output is used as a keystream to encrypt the user data struct and the key-chain on the memory card.

$ecpt \leftarrow ks \oplus (U_{ks} \,||\, K_{chain})$

The ciphertext is written to the memory card, overwriting the user key structure and key-chain keys.

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 identity information from the data-set.

If the user and key are identified, the user is prompted for a passphrase that is combined with the current key-chain index which acts like a nonce, producing a different keystream each time the keychain counter is incremented. The passphrase and nonce are processed by the cost-based KDF SCB, which generates a key-stream equal in length to the encrypted key identity structure and key-chain.

$ks \leftarrow SCB(passphrase \,||\, K_{idx}, n)$

The server decrypts the user identity structure.

$U_{ks} \leftarrow ks \oplus ecpt$

The server's hash of the user key structure is compared to a hash of the key structure on the memory card.

$U_{tmp} \leftarrow H(U_{ks})$

$Verify(U_{dh},\ U_{tmp})$

If the hash of the decrypted key identity matches the hash stored on the server in the user database entry, and the start and end times are valid, the server decrypts the key at the current key index.

$K_{tok} \leftarrow K_{chain}^{i} \oplus ecpt^{\,(i\,*\,klen)}$

6.4 Server Authentication Response

The server recreates the token by hashing the base key, the user identity hash, and the current key-chain index.

$K_{chain}^{i} \leftarrow H(K_{base} \,||\, U_{dh} \,||\, K_{idx})$

The server compares this to the key token pulled from the client's key-chain.

$Verify(K_{chain}^{i}, K_{tok})$

If the values match, authentication has succeeded, the current key is erased from the key-chain, and the key index is incremented. The K_{id} on the memory card and the server's user data entry are updated to the new key index. A new keystream is generated and used to encrypt the key identity and key-chain using the passphrase and the updated key-chain index value, which overwrites the existing ciphertext on the key card.

$Zeroize(K_{chain}^{Kidx})$

$K_{idx} \mathrel{+}= 1$

$ks \leftarrow SCB(passphrase \,||\, K_{idx}, n)$

$ecpt \leftarrow (U_{ks} \,||\, K_{chain}) \oplus ks$

The client is now authenticated on the server. The key token can be used to key a cipher and decrypt files, elevate system privilege, enable application access, or some other cryptographic process.

7. Security Analysis

The following discussion evaluates SIAP against the threat models typical of regulated-finance, payment-terminal, and offline-value-transfer deployments. It assumes SHA-3 behaves as a random oracle and that the SCB KDF delivers its claimed memory-hard cost amplification.

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-SRV – Compromised host**
  Gains full OS access, including K_{base}*, after some sessions have already completed.*
- **A-QC – Future quantum adversary**
  Possesses a large-scale fault-tolerant quantum computer and all stored ciphertext and traffic.

7.2 Security goals and how SIAP meets them

Goal	Mechanism(s)	Holds against
Two-factor MFA	• SCB-encrypted blob requires passphrase + card possession. • Header check (K_{id}) rejects cloned cards.	A-NET, A-DB
Forward secrecy (future sessions)	• One-time leaf burn on the memory card. • Index monotonically increments on both sides.	A-TOK, A-SRV
Quantum resilience	• Only SHAKE-256, KMAC-256 and SCB (hash-based) are used. • 256-bit capacity ⇒ ≥128-bit PQ margin.	A-QC
Replay & rollback resistance	• Plain K_{idx} in K_{tag}; server holds identical copy. • Any mismatch aborts before SCB.	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 → brute-force SCB.
   Cost factors:

-SCB memory footprint ≥ 64 MiB.
-Time cost ≈ 250 *ms* on a reference desktop.
-10-character lowercase password ⇒ 2^{33} guesses ⇒ ~7 years on a 4 GPU rig.
-*Mitigations:* higher SCB parameters, minimum passphrase entropy policy, or token self-wipe after *N* failures.

2. **Token clone & rollback**
   Path: copy ciphertext, boot from stale image.
   Barrier chain:
   -Header mismatch → fails at Verify(K_{id}^{i}, K_{id}) step.
   -U_{dh} hash check fails if header manually edited.
   -K_{idx} monotone counter in server row prevents accepting older leaf even if header forged.
3. **Server database leak**
   Data revealed: U_{dh}, K_{id}, logfile timestamps.
   Limits: cannot derive leaves without K_{base}, cannot brute U_{id} or S_{id} collisions because SHAKE pre-image ≈ 2^{256} (or 2^{512} in the case of the extended security model).
4. **Full server compromise (post-breach)**
   Impact: future sessions on that server no longer forward-secret.
   Hardening list:
   -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 √ advantage → effective security ≈ 2^{128}. SIAP's symmetric domain separation prevents multi-target quantum attacks from dropping below this boundary.

7.4 Side-channel considerations

- **SCB execution**
  Uniform random strides, deterministic memory schedule → timing and cache traces reveal no passphrase information.
  On low-end MCUs, constant-time implementations remain essential; add dummy row accesses if RAM is limited.
- **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** – Attacker with card & passphrase can deliberately burn leaves; server-side rate-limit and alert when $K_n - K_{idx} <$ threshold.
- **Token/DB desynchronization** – Power cut between card write and DB commit; mandate transactional "card-first, DB-second" persistence or two-phase commit handshake.

7.6 Security posture summary

- **Cryptanalytic robustness** – No shortcut attack is known against the SHAKE-only derivations or the SCB KDF; forward secrecy is information-theoretic once a leaf is destroyed.
- **Post-quantum readiness** – Entire stack remains free of factorization or discrete-log dependencies.

- **Operational controls** – Small, clearly scoped procedures (HSM for K_{base}, card counter rate-limit, encrypted transport where privacy demanded) suffice to address residual threats.

Taken together, SIAP offers a rigorously simple yet high-assurance authentication primitive suitable for fintech consoles, offline payment devices, and critical infrastructure nodes that must survive both modern cloud-borne attacks and future quantum disclosure.

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** ≈ 500 operators × 3 cards each, 50 jump-hosts, < 2 million leaf keys over five years; a global $K_n = 65{,}536$ easily covers the life of each token.
- **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** Pilot batch of 1 million cards, POS terminals already run the bank's TLS stack. Leaf keys are consumed once per payment; setting $Kn = 20{,}000$ supports ~100 spends per month for 15 years.
- **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 technician logs in 10× per week; $Kn = 10{,}000$ spans roughly twenty years. A stolen token self-destructs after five wrong SCB passphrase attempts, meeting PCI device-tamper guidelines.
- **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** 20 production lines × 10 testers, each flashing 60,000 units/day → 12 million logins per month. *Kn = 1,000,000* per card means annual card rotations rather than weekly.
- **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** 30,000 sessions/hour → 720,000 per day; cards issued to gateway racks rotate when K_{idx} approaches *Kn = 4 billion* (~15 years at current load).
- **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** 5 million quotes/day per engine → *Kn = 2 million* per month; cards hot-swapped fortnightly.
- **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** Nation-wide grid with 3,000 kiosks, each engineer averages 5 logins/day. *Kn = 65,536* yields a 35-year lifespan, but cards are reissued every 10 years for hygiene.
- **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 256-bit secret, existing stacks can consume it as a PSK, MAC key, or decryption key without wire-format changes.
- **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 an urgent gap that has opened between the lingering inertia of certificate-based infrastructures and the practical need for forward-secret, post-quantum authentication in devices that must often work without network reach or heavyweight cryptographic engines. By anchoring every long-lived secret in nothing more exotic than SHAKE-256 and a memory-hard passphrase transform, SIAP sheds the complexity, key-lifecycle cost and quantum fragility that accompany public-key systems, yet still delivers the replay resistance, auditability and two-factor assurance demanded by modern payment regulations and critical-infrastructure guidelines.

The security analysis shows that a single compromise vector; loss of the server's base key, defines the protocol's blast radius, and collateral servers remain untouched. Every other realistic adversary, from a roadside pick-pocket to a nation-state traffic collector armed with fault-tolerant quantum hardware, is confined by SCB's memory wall, SHA-3's generous capacity and the monotonic structure of the card's index counter. No feasible shortcut against the sponge functions is known, and every session's leaf key disappears as soon as it has done its work.

Operationally the protocol proves equally frugal. Host integration demands just one new hash call and a table row; card implementation fits in the flash budget of commodity micro-controllers, yet can be hardened inside secure elements when compliance or fault injection concerns dictate. Deployment sketches in the preceding section illustrate how this foundation scales from high-frequency trading engines requiring sub-millisecond re-keying, through PCI-DSS console access, to million-card offline-payment pilots, without disturbing the wire formats or trust assumptions of TLS, SSH, FIX, or EMV.

SIAP is therefore positioned not as a rival to key-exchange standards but as a portable, quantum-safe root-of-trust that can be spliced into any channel that already understands symmetric secrets. Its leaf-burn model turns every authentication into a zero-standing-privilege event; its plaintext identity hierarchy lets operators blacklist a card, user group or entire server with a single update; its offline design enables air-gapped service equipment and disaster-recovery payment rails to remain functional when certificate revocation services and time-stamp authorities are unreachable.

Future revisions will track the maturation of post-quantum MAC and stream-cipher primitives, add conformance profiles for secure-element implementations and refine reference SCB parameters as hardware evolves. Yet the architectural promise is set: a tiny, sponge-only core whose security scales with the width of Keccak and whose utility extends from desktop vaults to the furthest edge of the power grid. SIAP gives system designers a clear, verifiable path to quantum-resilient authentication today, one they can adopt incrementally, without rewriting the fabric of the networks they already trust.