

# Secure Infrastructure Access Protocol - SIAP

Revision 1.0a, November 12, 2025

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

This document is an engineering level description of the secure infrastructure access protocol SIAP; a post-quantum secure user verification system and access control mechanism.

Contents	Page
<u>Foreword</u>	2
<u>1: Introduction</u>	3
<u>2: Scope</u>	6
<u>3: Terms and Definitions</u>	7
<u>4: Cryptographic Primitives</u>	10
<u>5: Protocol Description</u>	12
<u>6: Mathematical Description</u>	17
<u>7: Security Analysis</u>	21
<u>8: Use Case Scenarios</u>	24
<u>Conclusion</u>	27

## 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](mailto:john.underhill@protonmail.com)

SIAP, the algorithm constituting the SIAP messaging protocol is patent pending, and is owned by John G. Underhill and Quantum Resistant Cryptographic Solutions Corporation. The code described herein is copyrighted, and owned by John G. Underhill and Quantum Resistant Cryptographic Solutions Corporation.

## 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 token key that is verified by a host server. Because every authentication event irreversibly destroys the currently active authentication token and advances a monotonic counter on the memory access card (mirrored by the server, 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 token 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 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 offline resilience. During initial provisioning the server encrypts the token-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 token-chain token with the user's passphrase and a server salt, recomputes one SHAKE-derived leaf, and grants or denies access based on a reproducible access token match derived from the server's base key. Consequently, SIAP supports remote kiosks, isolated manufacturing cells, and payment terminals that may remain disconnected from any central authority for prolonged periods. SIAP is capable of 256-bit post-quantum security or in enhanced mode, 512-bit post-quantum security, whereby all primitives (SHAKE, SCB, and RCS) run in 512-bit secure operational mode, and keys are scaled to 512-bits length.

Time-bounded validity is enforced by a valid-time field on the sealed user-key structure, ensuring the host need only interpret an absolute epoch counter to reject stale 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 token-chain index, recalculates the sealing keystream

using the same passphrase and server salt at 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 the server salt, 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 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	<b>PCI-DSS 4.0 console MFA for card-data-environment jump-hosts</b>	<ul style="list-style-type: none"> <li>Two-factor (passphrase + memory card) satisfies ‘independent MFA’ clause.</li> </ul>	<i>Default <math>K_n = 65,536 \rightarrow \sim 10</math> years @ 20 logins/day. Store <math>K_{base}</math> in HSM to limit breach radius.</i>

		<ul style="list-style-type: none"> <li>Leaf burn gives replay-proof, audit-friendly logins.</li> </ul>	
2	<b>Offline or low-bandwidth CBDC / e-cash wallets</b>	<ul style="list-style-type: none"> <li>Works with no PKI or network; server derives leaf with single SHAKE call.</li> <li>Start/End-time fields in Uks enforce spend-window.</li> </ul>	Keep card BOM low by using MCU+PUF profile; set $K_n \approx 15\,000$ (1 yr of micro-payments).
3	<b>Technician access to ATMs / POS &amp; PLCs (field service)</b>	<ul style="list-style-type: none"> <li>Card ID (Kci) + Server ID hierarchy (Sid) binds token to device fleet.</li> <li>Immediate failure if stale index or wrong group.</li> </ul>	Hand-held reader app can display remaining keys. Ship spare cards to avoid mid-shift exhaustion.
4	<b>Cold-wallet custody for institutional crypto</b>	<ul style="list-style-type: none"> <li>One-time key token can seed RCS/KMAC to sign withdrawal authorizations.</li> <li>Forward secrecy: key burnt after each signing.</li> </ul>	Use “extended security mode” (64-byte keys) + $K_n$ small (100-500) to force ceremony for large withdrawals.
5	<b>OEM activation / firmware decryption</b> on production lines	<ul style="list-style-type: none"> <li>Device contains server-side <math>K_{base}</math> hash; SIAP card must match to unlock image.</li> <li>No Internet inside secure factory.</li> </ul>	Write once, lock-down environment; after $K_{idx} == K_n$ line automatically halts for re-issuance.
6	<b>Zero-trust VPN pre-shared-key replacement</b>	<ul style="list-style-type: none"> <li><math>K_{tok}</math> outputs 256-bit secret every login; use as TLS-1.3 PSK binder or IKEv2 PSK.</li> <li>Removes long-lived static PSKs vulnerable to harvesting.</li> </ul>	Provide tiny “TLS-adapter” that domain-separates key: $TLS\_PSK = \text{SHAKE}(0x01  K_{tok})$ .
7	<b>High-frequency trading APIs</b> needing sub-millisecond auth	<ul style="list-style-type: none"> <li>Server derives leaf in <math>O(1)</math>; no lattice KEM latency.</li> <li>Memory footprint of 30 kB fits FPGA-adjacent soft-CPU.</li> </ul>	Pre-cache next leaf on card to avoid SCB decrypt on every order burst.
8	<b>Critical-infrastructure kiosks &amp; SCADA HMIs</b> (air-gapped)	<ul style="list-style-type: none"> <li>No network / CA required; Start/End times restrict stolen-card window.</li> <li>Global <math>K_n</math> + monotonic</li> </ul>	Harden reader firmware: fail closed if $K_{idx}$ jumps $>1$ (tamper indicator).

		K <sub>idx</sub> prevents infinite reuse.	
--	--	---	--

## 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**

**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**

**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**

**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**

**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 SIAP'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 Symmetric Cryptographic Primitives

SIAP's symmetric encryption employs the **Rijndael Cryptographic Stream (RCS)**, a stream cipher adapted from the Rijndael (AES) algorithm to meet post-quantum security needs. Key features of the RCS cipher include:

- **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):** RCS integrates with KMAC (Keccak-based Message Authentication Code) to provide both encryption and message authentication in a single operation. This approach ensures that data integrity is maintained alongside confidentiality.

The RCS stream cipher's design is optimized for high-performance environments, making it suitable for low-latency applications that require secure and efficient data encryption. It leverages AVX/AVX2/AVX512 intrinsics and AES-NI instructions embedded in modern CPUs.

### 4.3 Key Derivation Function

**SHAKE Cost Based KDF (SCB)** is a cost-based key derivation function, one that can exponentially 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

#### 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 key ( $K_{base}$ ), the server key salt ( $D_{salt}$ ), the server identity ( $S_{id}$ ), and the keys expiration time as milliseconds 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 ( $U_{de}$ ) is the searchable structure stored on the server that describes a user device profile. This is a hash of the user's token key chain ( $K_{hash}$ ), the SCB passphrase hash ( $P_{hash}$ ), and the key identity strings ( $K_{id}$ ) associated with this user device and token key.

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

Structure 5.1e: The key tag string (Ktag)

### The Device Key

The user device state ( $U_{ds}$ ) is the state structure stored on the memory card that holds information about the key container device and the token key-tree array. This is the key container identity ( $K_{id}$ ), the expiration in seconds from the epoch, and the key-tree array ( $K_{tree}$ ).

Parameter	Data Type	Bit Length	Function
Kid	Uint8 array	128	Key Card Identity
Expiration	Uint64	64	Expiration Time
Ktree	Uint8 Array	256/512 * n	Symmetric Key tree

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 (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 in length. 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 cards expiration time. The server initializes cSHAKE with the base key ( $K_{base}$ ), the SIAP configuration string, and the server's identity string ( $S_{id}$ ), and generates the server salt ( $D_{salt}$ ) used in the key-tree encryption. The server initializes cSHAKE with the server's base key ( $K_{base}$ ), the device's key identity string (including the  $K_{idx}$ ), and the SIAP configuration string. The server generates a key tree member, then increments the key counter ( $K_{idx}$ ), this turns Keccak into a counter mode generator that creates a unique key for every tree member.

The server issues a 256-bit passphrase to the user. This passphrase is hashed with the SCB KDF and stored in the user's database tag structure, along with the full device Kid and a hash of the key-tree.

The passphrase hash ( $P_{hash}$ ) is combined with the server salt ( $D_{salt}$ ), and the key identity ( $K_{id}$ ) to initialize cSHAKE and generate a cipher key and nonce. The cipher key and nonce initialize RCS, which authenticates and encrypts the key tree, adding a MAC to the end of the tree array. In this way, only a combination of the correct key identity (and index), the server salt, and the correct passphrase can authenticate and decrypt the token key tree. The key tree hash ( $K_{hash}$ ), the key identity (including the current key index), and the key tree hash are all stored on the user's key tag and stored in the server's database, 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's key. The user is prompted for the passphrase, which begins the login process. The authentication process is a multi-layered process, and each step must succeed in order to achieve a successful authentication:

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 and must be equal to the expiration time stored on the server's key.
3. The passphrase is hashed with SCB and compared to the passphrase hash in the key tag for equivalence.
4. The cipher encryption key and nonce are derived from the passphrase hash, the server's salt, and the key identity. The RCS cipher is initialized and used to authenticate and decrypt the devices token key tree.
5. The key tree is hashed, and that hash is compared with the key tree hash stored on the user data tag for equivalence.

6. The server generates and authentication token corresponding to the key index.
7. The server compares this authentication token, to the device's token at the same index for equivalence.
8. The device token is erased, and the Kid counter is incremented.
9. The device tag is regenerated; the device's token key tree is hashed and stored, as well as the Kid with the updated index count and the passphrase hash.
10. The key tree is encrypted, and the memory device is updated with the encrypted key tree, and the Kid with the incremented Kidx.

## 5.5 Server Authentication Response

The authentication process is logged at the server, including errors, device identity, and successful access. The server can use this to grant privileges to the user, or use the authentication token for another purpose, like decrypting an asset.

## 6. Mathematical Description

### Mathematical Symbols

$\leftarrow \leftrightarrow \rightarrow$	-Assignment and direction symbols.
$:=, !=, ?=$	-Equality operators; assign, not equals, evaluate.
conf	-The configuration string.
D <sub>salt</sub>	-The server salt array.
exp	-The expiration time in seconds from the epoch.
K <sub>base</sub>	-The server's base secret key.
K <sub>hash</sub>	-The key-tree hash.
K <sub>id</sub>	-The key identity string.
K <sub>idx</sub>	-The key-tree index.
K <sub>tree</sub>	-The key-tree array.
G(n)	-The pseudo-random bytes generator.
P <sub>hash</sub>	-The passphrase hash.
RCS	-The RCS AEAD stream cipher.
SCB	-The SCB cost-based key derivation function.
SHAKE	-The Keccak SHAKE EOF function.
S <sub>id</sub>	-The server identity string.
S <sub>kd</sub>	-The server key.
U <sub>dt</sub>	-The server's user database tag entry.
U <sub>kd</sub>	-The user key structure.
U <sub>id</sub>	-The user identity string.

### 6.1 Server Initialization

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

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

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

## SIAP 1.0a

server is assigned to, and the server field is a unique 16-bit integer identifier representing the server identity.

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

The server creates a salt value by initializing cSHAKE with the base key, the configuration string (a readable string defining the cryptographic primitives and version used in the SIAP implementation), and the server's identity string.

$$D_{salt} \leftarrow \text{SHAKE}(K_{base}, conf, S_{id})$$

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

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

## 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.

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

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*.

$$K_{id} \leftarrow \{ S_{id} \parallel U_{id} \parallel K_{idx} \}$$

The server creates the key tree by initializing SHAKE with the base key, the configuration string, and the device's key identity string, and generating a hash. For every key created, the key index value of the key identity string is incremented, ensuring a unique hash output for every key tree token generated.

where: ( $i = 0, 1, 2, \dots n$ ),

$$K_{tree}^i \leftarrow \text{SHAKE}(K_{base}, conf, U_{id} \parallel K_{idx}^{++})$$

The server hashes the key tree, and adds the key identity string, the passphrase hashed with SCB KDF, and a hash of the key tree to the user data tag, stored on the server.

$$\begin{aligned} P_{hash} &\leftarrow \text{SCB}(passphrase) \\ K_{hash} &\leftarrow \text{SHAKE}(K_{tree}) \\ U_{dt} &\leftarrow \{ K_{id}, K_{hash}, P_{hash} \} \end{aligned}$$

## SIAP 1.0a

The server generates a cipher encryption key by initializing SHAKE with the passphrase hash, the key identity string, and the server salt. The server initializes the RCS cipher and uses it to MAC and encrypt the key tree. The incrementing Kidx ensures a unique key and nonce with every encryption.

$$\begin{aligned} k, n &\leftarrow \text{SHAKE}(P_{\text{hash}}, U_{\text{id}} \parallel K_{\text{idx}}^{++}, D_{\text{salt}}) \\ E_{\text{ktree}} &\leftarrow \text{RCS}_{k,n}(K_{\text{tree}}) \end{aligned}$$

The server adds the key identity string, the encrypted and MAC'd key tree, and the expiration time to the user key device.

$$U_{\text{kd}} \leftarrow \{ K_{\text{id}}, E_{\text{ktree}}, \text{expiration} \}$$

The server key has been created, and along with the user tag are stored on the server. The client is given the user device key and passphrase, and the system is ready for access.

### 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 hashed with SCB KDF to produce the passphrase hash.

$$P_{\text{hash}} \leftarrow \text{SCB}(\text{passphrase})$$

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.

$$U_{\text{kd\_id}} = U_{\text{dt\_id}} ? \text{true} : \text{false}$$

If the key identity strings match, the server compares the expiration time in the user key tag with the expiration time of the device key for equivalence.

$$U_{\text{kd\_exp}} = U_{\text{dt\_exp}} ? \text{true} : \text{false}$$

If the expiration times are equivalent, the server compares the passphrase hash stored in the user key tag with the passphrase hash generated at login.

$$U_{\text{kd\_Phash}} = U_{\text{dt\_Phash}} ? \text{true} : \text{false}$$

If the hashes are equivalent, the server authenticates and decrypts the device key's token key tree. The server generates the unique key and nonce, then authenticates and conditionally decrypts the device's key tree.

```

k, n ← SHAKE(Phash, Uid || Kidx++, Dsalt)
Ktree ← RCSk,n(Ektree)

```

If the key tree ciphertext is authenticated and decrypted, the server extracts the authentication token from the key tree, erasing the key on the tree and incrementing the key index counter.

```

token ← Ktree(kidx)
Erase(Ktree(kidx))
Kidx += 1

```

If the key was extracted successfully, the server generates the corresponding key token for comparison using SHAKE initialized with the server's base key, the configuration string, and the key identity string.

```
token' ← SHAKE(Kbase, conf, Uid || Kidx)
```

The two tokens are compared for equivalency, if they are identical then authentication has succeeded.

```
token = token' ? true : false
```

The server hashes the key tree, and adds the updated hash to the user data tag along with the updated key identity string.

```

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

```

The server then re-encrypts the token key tree, and writes encrypted tree and the updated key identity string to the device key.

```

k, n ← SHAKE(Phash, Uid || Kidx++, Dsalt)
Ektree ← RCSk,n(Ktree)
Ukd ← { Kid, Ektree, expiration }

```

## 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 ( $U_{dh}, K_{id}^1$ ) 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
<b>Two-factor MFA</b>	<ul style="list-style-type: none"> <li>• SCB-encrypted blob requires passphrase + card possession.</li> <li>• Header check (<math>K_{id}</math>) rejects cloned cards.</li> </ul>	A-NET, A-DB
<b>Forward secrecy (future sessions)</b>	<ul style="list-style-type: none"> <li>• One-time leaf burn on the memory card.</li> <li>• Index monotonically increments on both sides.</li> </ul>	A-TOK, A-SRV
<b>Quantum resilience</b>	<ul style="list-style-type: none"> <li>• Only SHAKE-256, KMAC-256 and SCB (hash-based) are used.</li> <li>• 256-bit capacity <math>\Rightarrow \geq 128</math>-bit PQ margin.</li> </ul>	A-QC
<b>Replay &amp; rollback resistance</b>	<ul style="list-style-type: none"> <li>• Plain <math>K_{idx}</math> in <math>K_{tag}</math>; server holds identical copy.</li> <li>• Any mismatch aborts before SCB.</li> </ul>	A-NET, A-TOK
<b>Offline operation</b>	<ul style="list-style-type: none"> <li>• All verification uses local <math>K_{base}</math>; no CA or CRL lookup.</li> <li>• Start/End-time fields enforce validity window.</li> </ul>	All

### 7.3 Detailed attack-surface review

#### 1. Offline dictionary attack on passphrase

*Path:* steal card  $\rightarrow$  brute-force SCB.

*Cost factors:*

-SCB memory footprint  $\geq 64$  MiB.

-Time cost  $\approx 250$  ms on a reference desktop.

**-10-character lowercase password  $\Rightarrow 2^{33}$  guesses  $\Rightarrow \sim 7$  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  $\approx 2^{256}$  (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  $\sqrt{advantage}$  → effective security  $\approx 2^{128}$ . 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  $K_n - K_{idx} < \text{threshold}$ .
- **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  $K_{base}$ ; the operating system calls that HSM for the single SHAKE-256 derivation required by SIAP.

- **Scale**  $\approx 500$  operators  $\times 3$  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  $K_n = 20,000$  supports  $\sim 100$  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  $10\times$  per week;  $K_n = 10,000$  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 ( $\leq 100$ ), but  $K_n$  intentionally tiny ( $\leq 200$ ) 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  $\times$  10 testers, each flashing 60,000 units/day  $\rightarrow$  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  $\rightarrow$  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  $TLS\_PSK = \text{SHAKE}(0x01||K_{tok})$  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  $\mu\text{s}$ . SIAP's single SHAKE call (deterministic 1.2  $\mu\text{s}$  on Intel Ice Lake) meets the budget.

- **Scale** 5 million quotes/day per engine  $\rightarrow Kn = 2$  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  $\mu\text{s}$  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.