QRCS Corporation

Secure Infrastructure Access Protocol Analysis

Title: Implementation Analysis of the Secure Infrastructure Access Protocol (SIAP)
Author: John G. Underhill
Institution: Quantum Resistant Cryptographic Solutions Corporation (QRCS)
Date: November 2025
Document Type: Technical Cryptanalysis Report
Revision: 1.0

Chapter 1: Introduction

1.1 Manuscript and Meta

This paper presents a cryptanalytic examination of the Secure Infrastructure Access Protocol (SIAP), a symmetric, post-quantum authentication system that derives all long-term and ephemeral secrets exclusively from hash-based primitives. SIAP integrates a removable memory token containing an encrypted key-tree with a passphrase transformed under a memory-hard function, producing a single-use authentication value for each session. The host server verifies correctness by regenerating a corresponding value derived from a master key and identity string. Because the system does not employ any public-key cryptography and relies solely on Keccak-based constructions, the security of SIAP is reducible to the behavior of SHA-3 extendable-output functions, the properties of the SCB cost-based key-derivation function, and the correctness of the monotonic key-indexing mechanism.

This manuscript adheres to a fixed analytical structure suitable for independent academic review. The protocol is treated as a fully specified object defined by its deterministic algorithms, message flows, and cryptographic constructions. The goals of the analysis are to determine whether SIAP satisfies its stated security claims under explicit adversarial models and to identify any structural weaknesses that may arise from its design choices or implementation constraints. Because SIAP does not rely on public-key operations, the examination focuses on symmetric-security reductions, pre-image and collision resistance of the employed primitives, resistance to offline dictionary attacks, correctness and safety of the key-tree construction, and forward-secrecy guarantees under the erasure-on-use design.

Throughout this paper SIAP is interpreted strictly as an authentication protocol. Its output token is treated as a one-time secret that may be consumed by higher-level systems but is not itself a transport primitive. The cryptanalysis therefore evaluates correctness of the authentication logic, including derivation of server-side authentication tokens, decryption and verification of the device’s encrypted key-tree, and the mechanisms enforcing time-validity and monotonic index advancement. All statements concerning algorithmic behavior, boundary conditions, and data transformations are grounded in the documented behavior of the reference implementation.

The analysis proceeds without presupposing any trust assumptions beyond those explicitly modeled. The adversary classes considered in later chapters include passive and active network attackers, token-theft adversaries, database compromise scenarios, full server compromise, and quantum-capable adversaries. Each of these models is defined formally, and the paper establishes which claims are provable under each adversarial capability, which properties rely on assumptions about the underlying primitives, and which residual risks arise solely from operational factors rather than from the cryptographic design.

The manuscript excludes any evaluative or qualitative description not grounded in verifiable behavior. No claim is made unless it follows directly from the specification or from well-established cryptographic literature. The resulting document is intended to serve as an academically suitable, non-speculative and non-decorative analysis of the SIAP protocol, with clarity comparable to formal publications that describe symmetric-key standards or constrained-device authentication frameworks.

Chapter 2: Problem Statement and Scope of Analysis

The Secure Infrastructure Access Protocol (SIAP) is designed to provide a post-quantum, two-factor authentication mechanism for environments in which a host must verify the identity of a user or device using only symmetric cryptography. SIAP replaces public-key infrastructures, certificate validation, and online trust anchors with a construction that relies entirely on a removable token, a user-supplied passphrase, a server-maintained master key, and deterministic derivation of single-use authentication tokens. The purpose of this chapter is to define precisely the problem SIAP attempts to solve, the security properties it claims to provide, and the analytical boundaries within which this paper evaluates those claims.

The central problem addressed by SIAP is the need for a forward-secret, replay-resistant, and quantum-resilient authentication method that can operate without access to public-key cryptography or online validation services. Many operational domains require authentication in settings where certificate revocation, key-agreement latency, or bandwidth limitations render public-key primitives undesirable or infeasible. SIAP seeks to satisfy these constraints by reducing the authentication process to the deterministic regeneration and comparison of a single authentication token derived from a master key and identity string. The removable memory card stores an encrypted array of pre-derived authentication tokens, each intended for a single use. The server independently regenerates the expected token from its own key materials, enabling authentication without interactive key negotiation.

The protocol therefore aims to achieve three core properties. First, it must bind authentication to two independent factors: possession of the encrypted key-tree stored on the token, and knowledge of the passphrase required to derive the decryption key via a memory-hard function. Second, it must guarantee that each authentication event is forward-secret. This requires that a token consumed in one session cannot be recomputed from any secrets available after the session has completed, and that the index representing the active element of the token tree cannot be rolled back or duplicated without detection. Third, the protocol must maintain its security posture in the presence of a quantum adversary able to store ciphertexts and run Grover-class search algorithms against symmetric primitives. Because SIAP maintains no asymmetric structure, its post-quantum margin derives solely from the size of the SHA-3-derived outputs and the hardness of deriving the passphrase through the SCB function.

The scope of this cryptanalysis is limited to the protocol as it is defined by its specification and implementation. The analysis considers the key-tree construction, the passphrase hashing mechanism, the encryption and authentication of token-tree contents, the derivation of server-side authentication tokens, and the logical checks that enforce identity matching, expiration, and monotonic advancement of the key index. The evaluation does not consider transport-layer protections or operational measures outside the protocol’s defined behavior, except where their absence would affect the correctness of the protocol or the feasibility of attacks within the stated adversarial models.

The analysis is not an evaluation of the broader system in which SIAP may be embedded. It does not assess file-system security, physical protections of the removable token, or the behavior of application-layer integrations except insofar as these influence the cryptographic guarantees of the protocol itself. The implementation contained in the provided code pages is treated as normative; therefore, the cryptanalysis incorporates all observable design elements, including serialization formats, error handling, key-increment logic, and the encryption–decryption cycle governing the token tree.

This chapter defines the boundaries within which SIAP will be examined in the subsequent sections. The goal is to determine, under clearly articulated assumptions and adversarial capabilities, whether the protocol satisfies its stated security goals, whether any structural weaknesses are inherent in the design, and whether failure conditions or corner cases in the implementation introduce attack vectors not anticipated by the abstract protocol description.

Chapter 3: Model and Assumptions

This chapter defines the cryptographic model, operational assumptions, and adversarial capabilities under which the SIAP protocol is evaluated. Because SIAP is built entirely on symmetric primitives and derives all long-term and ephemeral values from Keccak-based functions and a memory-hard passphrase transform, the security model must specify the behavior expected of these underlying functions, the trust boundaries of the server and removable token, and the error conditions under which authentication must fail. The analysis that follows in later chapters depends on these assumptions being clearly articulated and consistently applied.

The cryptographic model assumes that SHAKE-256 and SHAKE-512 behave as random oracles with respect to pre-image resistance, second pre-image resistance, and collision resistance. The sponge construction, with capacity determined by the chosen output size, is treated as a permutation-based XOF with no structural weaknesses that would allow computation of internal state or prediction of outputs beyond the limits of generic pre-image attacks. The model similarly assumes that cSHAKE-256 and cSHAKE-512, as used in SIAP, preserve the domain-separation and indifferentiability properties documented in the underlying standard, ensuring that the keyed and personalized invocations do not collide across different parameterizations. These assumptions are consistent with current cryptanalytic literature on the Keccak family and form the basis of SIAP’s claim to post-quantum security.

The SCB cost-based key-derivation function is modeled as a memory-hard transform with deterministic output for any fixed passphrase and salt. The model requires that SCB outputs behave as pseudo-random bit strings under passive observation, and that the cost amplification imposed by memory ballooning, cache-thrashing, and sequential memory iteration cannot be substantially reduced by parallelization or recomputation shortcuts on commodity hardware or GPU-class architectures. While SCB is not a standard KDF, its behavior as implemented in the provided code is internally deterministic and parameterized in a way that enforces data-independent access and uniform memory traversal. The model therefore assumes SCB resists structural shortcuts and that its output cannot be distinguished from a random 256-bit or 512-bit string by an adversary lacking the correct passphrase.

The RCS authenticated stream cipher used for encrypting and authenticating the token-tree is modeled as an AEAD construction with strong confidentiality and integrity guarantees under a unique key-and-nonce pair. The analysis assumes that for each encryption cycle, SIAP’s use of cSHAKE to derive a fresh key and nonce from the passphrase hash, the device identity, the index counter, and the server salt provides unique, non-repeating inputs. The implementation increments the key-identity counter before each derivation, ensuring nonce uniqueness as long as the token index does not roll back. The model assumes that any modification to the ciphertext of the encrypted key-tree results in an integrity failure detectable by the RCS decryption procedure.

The server-side environment is modeled as an honest-but-curious verifier that holds the server base key, server salt, server identity, and the authoritative record of user device tags. The server’s key is assumed uncompromised unless otherwise stated in the adversarial model. The removable memory token is treated as a passive device capable of storing the encrypted key-tree, the key identity, and the expiration timestamp. The token is not required to perform cryptographic operations; the model assumes that all verification and re-encryption operations occur on the host server. The analysis therefore treats the token as an untrusted storage object whose confidentiality is provided solely by the RCS encryption and passphrase-derived key, rather than by any hardware isolation.

The threat model distinguishes between several adversarial classes. A passive network attacker can observe protocol interactions but cannot modify token contents or server data. An active attacker may inject, replay, or modify messages passed over the communication channel. A token-theft adversary may gain physical possession of the removable memory card while lacking the passphrase. A database-compromise adversary may obtain the server's stored device tags but not the server's master key. A full server-compromise adversary may obtain all server-side secrets after certain sessions have completed. Finally, a quantum-capable adversary is assumed to possess the ability to execute Grover-class searches against symmetric primitives and to store all observable ciphertexts indefinitely.

The analysis assumes that both the client and server maintain consistent representations of the key identity, expiration time, and key-tree index. Failure of synchronization between the memory token and server database is considered an operational failure rather than a cryptographic one. The model requires that any deviation in these values results in immediate authentication failure. This includes stale key indices, mismatched passphrase hashes, mismatched key-tree hashes, and expiry violations. No attempt is made to recover from such mismatches, as the protocol explicitly mandates that any such condition results in termination of the authentication attempt.

This chapter establishes the foundational assumptions about primitive behavior, cryptographic hardness, and adversarial capabilities. Subsequent chapters rely on these assumptions to derive proofs, analyze attack feasibility, and determine whether SIAP meets its stated security objectives under the defined operational boundaries.

Chapter 4: Related Work

The Secure Infrastructure Access Protocol belongs to a class of symmetric-only authentication systems that avoid reliance on public-key cryptography, certificate infrastructures, or online validation services. Although SIAP integrates design elements such as memory-hard passphrase derivation, encrypted stateful tokens, and deterministic key evolution, its structure is most appropriately compared to earlier constructions that employ pre-derived secrets, keyed hash trees, one-time authentication sequences, or counter-based symmetric key chains. This chapter situates SIAP within the landscape of related cryptographic mechanisms to clarify the conceptual lineage from which its design draws and to identify prior approaches solving analogous problems.

Symmetric authentication protocols for constrained or offline environments have historically relied on pre-distributed secrets organized in either linear or tree-like structures. One category comprises counter-based schemes, in which a shared secret is incrementally updated using a deterministic generator, and both parties maintain synchronized counters to validate one-time passwords. HMAC-based one-time password (HOTP) systems exemplify this approach. Although SIAP does not rely on PRF iteration, its regeneration of authentication tokens from a master key and a monotonically increasing index bears structural similarity to counter-driven deterministic key derivation. Unlike HOTP, SIAP derives authentication tokens directly from SHAKE keyed by the server's master secret rather than from a block-cipher or hash-based HMAC. Moreover, SIAP encrypts the stored sequence of tokens and enforces forward secrecy by erasure, which contrasts with approaches where state is maintained only implicitly by the counter.

Another relevant category appears in hash-chain or Lamport-chain authentication systems, where a sequence of secrets is derived through repeated hashing, with verification performed by comparing the received value to an expected predecessor under the hash function. SIAP differs structurally from such constructions because its token chain is not a backward hash sequence but a collection of independent values derived from a keyed XOF with domain separation provided by the identity string and index. This design avoids the need for repeated hashing to validate future indices and prevents an adversary from exploiting chain inversions; each SIAP token is independent under the security assumptions of SHAKE.

Tree-based symmetric authentication methods, including Merkle-tree authentication and symmetric key trees used in certain group-key management schemes, also bear superficial resemblance to SIAP's "key tree" terminology. However, SIAP's structure does not implement a binary or hierarchical tree. Instead, the token tree is a flat array of independently generated leaves, each derived from the server's master key, the configuration string, and an incrementing index parameter. Unlike Merkle-based constructions, SIAP does not provide verifiable aggregation or branching authentication; its structure is better characterized as a fixed-length table of one-time secrets.

Memory-hard password transformations, such as those found in Argon2 and Scrypt, are relevant to the passphrase hardening feature of SIAP. The SCB function implemented in SIAP uses sequential memory accesses, deterministic cache interaction, and periodic inclusion of memory blocks into the hash state to increase the computational effort required to test candidate passphrases. While SCB is not directly derived from these standardized functions, it aligns with the conceptual goal of penalizing brute-force attempts by enforcing memory and time costs. The security of SIAP against offline dictionary attacks depends on this property.

Authentication tokens encrypted under symmetric AEAD constructions also appear in secure storage and hardware-bound token schemes. In these systems, encryption under a key derived from a user-supplied secret and a persistent device secret is used to protect credential material at rest. SIAP applies a similar approach by encrypting the entire token array under a key composed from the passphrase hash, the device identity, and the server salt. All confidentiality and integrity guarantees rely on the properties of the RCS stream cipher with an embedded MAC, similar in conceptual scope to constructions where AES-GCM or ChaCha20-Poly1305 protect authentication states.

Finally, SIAP's elimination of public-key cryptography and its reliance only on symmetric primitives differentiates it from modern key-exchange and authentication protocols such as TLS 1.3, IKEv2, and SSH, all of which depend heavily on ephemeral asymmetric primitives to achieve forward secrecy and resistance to replay. SIAP achieves analogous properties through pre-derived one-time symmetric keys and monotonic counters, placing it closer in lineage to early symmetric-key access-control systems used in offline devices, embedded industrial systems, or financial authentication terminals.

Overall, SIAP draws upon several established concepts: deterministic per-session key derivation, one-time authentication tokens, memory-hard passphrase processing, and symmetric AEAD-protected state. The protocol distinguishes itself by combining these components into a construction intended for post-quantum resilience, offline operation, and forward secrecy enforced by mandatory key destruction. The related work surveyed in this chapter establishes the conceptual foundations against which SIAP's structure and security properties will be evaluated in the chapters that follow.

Chapter 5 Preliminaries

The cryptanalysis of the Secure Infrastructure Access Protocol requires a precise formulation of the algebraic structures, data representations, and cryptographic functions that participate in each stage of the protocol. This chapter defines these preliminaries in a manner consistent with formal academic treatment, using notation that reflects the behavior of the SIAP specification and the reference implementation. All subsequent reasoning, formalism, and security arguments rely on these definitions.

SIAP is built around Keccak-based extendable-output functions used in both unkeyed and keyed modes. Let SHAKE denote the standard extendable-output function defined over a permutation with fixed rate and capacity. The function is treated as mapping arbitrarily sized input strings to output bitstreams of requested length, and its behavior is assumed to conform to the indifferentiability properties of sponge constructions. The keyed variant used throughout the protocol is cSHAKE, which incorporates a customization string and key material into the sponge initialization. In SIAP, cSHAKE is invoked with three inputs: a key or passphrase-derived value, the protocol's configuration string, and an identity string serving as domain separation. The output length of each invocation equals either the authentication-token length or the concatenation of key and nonce material required for RCS initialization.

Let SCB denote the cost-based key-derivation function that transforms a passphrase into a fixed-length pseudo-random output. The SCB function initializes an internal state with the passphrase and then performs computationally expensive operations involving large memory buffers, repeated insertion of loop indices into the hash state, and uniform memory traversal. Its output is deterministic for any fixed passphrase and parameter set, yielding a bitstring of length equal to the SIAP hash size. Because SCB incorporates memory-hardness, its outputs are assumed to be infeasible to compute at scale for a low-entropy password space without incurring a substantial cost penalty on each guess.

RCS denotes the authenticated stream cipher used for encrypting and decrypting the token-tree. In SIAP the cipher operates in a mode where a key and nonce are derived from cSHAKE, and the ciphertext is produced by a single transform function that applies the stream cipher and the MAC. A token-tree ciphertext is accepted only if the MAC verifies successfully; the decryption function returns a boolean representing this authenticity condition. The cipher is modeled as providing indistinguishability from random output and unforgeability under chosen-ciphertext conditions, provided that key and nonce pairs do not repeat. SIAP enforces uniqueness by incorporating an incrementing counter field in the key-identity string.

The SIAP device identity is a 16-byte structure formed by concatenating domain, server-group, server, user-group, user, and device identifiers. The full key identity is obtained by appending a 4-byte big-endian counter representing the current token index. Let Kid denote this composite identity string. Because Kid appears both in cleartext on the token and as input to several cSHAKE invocations, correctness of all derivations depends on consistent serialization across the protocol. The counter field must be updated monotonically and must not roll back between authentication events.

The device key, denoted Ukd in the mathematical description, is an encrypted container storing the token-tree and its metadata. The token-tree consists of a fixed-length array of authentication tokens, each of which has length equal to the configured security level. The tree is indexed from zero to SIAP_KTREE_COUNT − 1. The server and device both maintain the invariant that exactly one element corresponds to the current index, and that this element is erased on use.

The server maintains a device tag for each registered token. Each tag stores the key identity, the passphrase hash, and a hash of the plaintext token-tree. Let Udt denote this structure. Integrity and freshness checks during authentication rely on exact equivalence between the tag's stored values and the reconstructed state after decrypting the key-tree.

The server key, denoted Skd, contains the master derivation key kbase, the server identity sid, the server salt dsalt, and an expiration timestamp exp. All authentication tokens are derived from cSHAKE initialized with kbase, the protocol's configuration string, and the key identity. The presence of sid in the derivation of dsalt ensures that each server instance maintains an independent namespace of authentication values.

Throughout the analysis, assignment arrows are used to indicate updates to token-tree entries, index variables, and cryptographic outputs. All comparisons of bitstrings are treated as constant-time equality checks, as reflected in the implementation. Expiration validation uses the server's epoch-time in seconds compared against stored timestamps. Any failure in identity matching, integrity verification, decryption, passphrase verification, or timestamp validation terminates the authentication attempt immediately.

These preliminaries define the deterministic machinery on which SIAP relies. They allow subsequent chapters to express the protocol's behavior in mathematically precise form and to analyze the correctness of its cryptographic dependencies. With these structures and functions rigorously defined, the next chapter provides a systematic overview of SIAP's operation before entering the formal specification and security definitions.

Chapter 6: Protocol Overview

This chapter provides a structured and academically neutral overview of the Secure Infrastructure Access Protocol. The purpose is to present the protocol's operational sequence in a coherent, implementation-independent manner while maintaining strict fidelity to the behaviors mandated by the SIAP specification and expressed in the reference code. The overview refrains from analysis, proof, or critique; it serves only to describe SIAP as a deterministic algorithmic system whose components interact according to fixed rules.

SIAP defines an authentication mechanism based on a removable memory token that stores an encrypted array of symmetric one-time authentication tokens, a composite identity string, and an expiration timestamp. The server maintains a corresponding device-tag record containing the same identity string, a hash of the plaintext token-tree, and a hash of the user's passphrase transformed by the SCB function. The server also maintains a master key, server identity, and server-salt value. These server-side values collectively determine the authentication tokens that the server independently regenerates when verifying a client.

During provisioning, the server assembles a composite identity by concatenating identifiers for domain, server-group, server, user-group, user, and device. The server appends a counter field to construct the full key identity. It then invokes cSHAKE with the master key, the protocol's configuration string, and the key identity to derive each element of the token-tree. After each derivation the counter is incremented, ensuring that each tree element corresponds to a unique invocation of the extendable-output function. Once the complete array has been produced, the counter is reset to zero. The server computes a hash of the entire tree using SHAKE and constructs the device-tag by pairing this hash with the identity and the passphrase hash. The server derives an encryption key and nonce by invoking cSHAKE on the passphrase hash, the key identity, and the server salt, and uses the resulting values to initialize the RCS cipher. The plaintext tree is encrypted and authenticated, producing a ciphertext structure that includes both encrypted token values and a MAC. This ciphertext, together with the identity string and expiration timestamp, forms the device key written to the removable token.

The authentication procedure begins when the device key is presented to the server. The server deserializes the key and reads the identity string and expiration field. The server retrieves the corresponding device-tag and validates that the identity matches exactly. It then verifies that the device's expiration timestamp is within the permitted window relative to the server's expiration and to the protocol's allowable time drift. The user supplies a passphrase which is transformed by SCB into a passphrase hash. The server compares this computed value with the passphrase hash stored in the device-tag; equality is required for continuation.

If these preliminary checks succeed, the server reconstructs the decryption key and nonce using the passphrase hash, the device identity including the embedded index counter, and the server salt. The RCS cipher is initialized under this key and nonce, and the encrypted token-tree stored on the device is decrypted and authenticated. If authentication fails, the protocol terminates. If decryption succeeds, the server recomputes the SHAKE hash of the decrypted token-tree and compares the result to the stored hash in the device-tag to verify integrity.

At this stage the server extracts the authentication token corresponding to the current index. The implementation retrieves the token by reading the appropriate slice of the token-tree and clearing that slice to enforce erasure. The counter embedded in the identity string is incremented in big-endian format. The server independently generates an authentication token using cSHAKE with the master key, configuration string, and the updated key identity. The two tokens: the device token and the server-generated token, must be exactly equal for authentication to succeed. If they differ, the attempt fails.

Upon successful authentication, the server recomputes the hash of the updated token-tree, constructs a new device-tag containing the updated hash and counter, and re-encrypts the token-tree using a newly derived key and nonce. The re-encrypted device key is then written back to persistent storage on the token. This completes the authentication cycle. Every successful run consumes one token-tree element and advances the counter, ensuring that no authentication token can be reused.

The protocol therefore operates as a strictly ordered sequence of identity matching, passphrase verification, encrypted state recovery, forward-secret token extraction, and state re-protection. No step is optional. Each depends directly on the prior step succeeding and on the correctness of the stored identity, the uniqueness of the counter, and the integrity of the encrypted tree. Any deviation in serialized fields, timestamps, or counters results in immediate abort. Because no public-key operations occur and all secrets derive from deterministic functions, SIAP's operation reduces to synchronous evolution of shared state between the server and the stored token.

This chapter defines SIAP at the level of an algorithmic overview. The next chapter provides a formal specification of these mechanics as state transitions expressed in mathematical terms.

Chapter 7: Formal Specification

This chapter expresses the Secure Infrastructure Access Protocol as a set of precise, deterministic state-transition functions. The specification is derived strictly from the SIAP 1.0a document and the accompanying implementation files. No behaviors are inferred beyond those explicitly encoded in the reference.

Each component of SIAP is described as an object with an associated state, and each stage of the protocol is represented as a transition between well-defined states. All operations are deterministic, and all correctness conditions are expressed as equality relations between bitstrings or successful completion of authenticated decryption. All failure conditions correspond to the violation of a predicate defined below.

7.1 State Objects

SIAP maintains three primary state objects: the server key state, the device key state, and the device tag state.

Server Key State

The server key state contains:

- $\mathrm{K\_base}$: the server's master derivation key.
- $\mathrm{S\_id}$: the server identity (domain ID, server group ID, server ID).

- D_salt : salt derived from K_base, the configuration string, and S_id.
- T_exp : expiration timestamp for the server key.

These values form the basis of all server-side derivations.

Device Identity (Kid)

The device identity consists of:

- Domain ID
- Server Group ID
- Server ID
- User Group ID
- User ID
- Device ID
- Counter (32-bit, big-endian)

These fields are concatenated in the above order to form a single identity string named Kid.
The counter value determines which token in the token-tree is active.

Device Key State (Plaintext Form)

The device key (Ukd) contains:

- Ktree : an array of authentication tokens.
- Kid : the device identity string, including the counter.
- T_exp : expiration timestamp.

Ktree contains 1024 tokens, each of fixed length depending on the security mode (256-bit or 512-bit).

Device Tag (Udt)

The device tag contains:

- Kid : the identity string.

- H_tree : a SHAKE hash of the entire token-tree.
- H_pass : the SCB-derived passphrase hash.

This tag acts as the authoritative server-side description of the device state.

7.2 Token-Tree Generation

During provisioning, the server generates the device's token-tree as follows:

1. Set the counter in Kid to zero.
2. For each index i from 0 to 1023:
    - Derive token K_i using cSHAKE with inputs (K_base, configuration string, Kid).
    - Increment the counter field inside Kid.
3. After generating all tokens, reset the counter in Kid to zero.

This produces a deterministic sequence of 1024 authentication tokens.

7.3 Device Key Encryption

Before writing the device key to storage, the server encrypts the token-tree using RCS.

Steps:

1. Compute the passphrase hash H_pass using SCB.
2. Derive an RCS key and nonce from cSHAKE with inputs (H_pass, Kid, D_salt).
3. Encrypt Ktree using RCS authenticated encryption.
4. Store the encrypted tree, Kid, and expiration timestamp as Ukd_enc.

The device never stores plaintext tokens.

7.4 Authentication Procedure

Given an incoming device key Ukd_enc and the corresponding device tag Udt, the server executes the following procedure.

Step 1. Identity Check

Compare the Kid stored in Ukd_enc with the Kid stored in Udt.
If they are not exactly equal, authentication fails immediately.

Step 2. Expiration Check

Verify that the device's expiration timestamp is both:

- less than or equal to the server's key expiration timestamp, and
- less than or equal to the current time plus the allowed protocol duration.

If either condition fails, authentication fails.

Step 3. Passphrase Verification

The user supplies a passphrase.

1. Compute H_pass' = SCB(passphrase).
2. Compare H_pass' with H_pass stored in Udt.

If they differ, authentication fails.

Step 4. Decrypt Device Key

Derive RCS encryption key and nonce using cSHAKE with inputs:

- H_pass
- Kid
- D_salt

Decrypt the encrypted token-tree using RCS.

If integrity verification fails, authentication fails.

Step 5. Verify Tree Hash

Compute SHAKE(Ktree') from the decrypted plaintext tree.
Compare it exactly with H_tree stored in Udt.

If they differ, authentication fails.

Step 6. Extract Token and Advance Counter

Let C be the current counter stored in Kid.

1. Extract token $\mathrm{T\_dev}$ at index C from the token-tree.
2. Overwrite that position in the token-tree to erase the token.
3. Increment the counter C and write it back into Kid.

Step 7. Regenerate Server Token

Using cSHAKE with inputs $(\mathrm{K\_base}, \text{configuration string}, \mathrm{Kid})$, derive $\mathrm{T\_srv}$.

Compare $\mathrm{T\_dev}$ with $\mathrm{T\_srv}$.
If they differ, authentication fails.

Step 8. Rewrite Device State

Upon successful token match:

1. Compute a new $\mathrm{H\_tree\_new} = \mathrm{SHAKE}(\text{updated Ktree}')$.
2. Update Udt to contain $(\mathrm{Kid}, \mathrm{H\_tree\_new}, \mathrm{H\_pass})$.
3. Re-encrypt the updated Ktree' using RCS with the new Kid.
4. Produce $\mathrm{Ukd\_enc\_new}$ for storage on the device.

This completes the authentication and updates the device state.

7.5 Failure Semantics

The protocol rejects immediately upon any of the following:

- identity mismatch,
- expiration violation,
- incorrect passphrase,
- RCS decryption failure or MAC failure,
- key-tree hash mismatch,
- mismatch between $\mathrm{T\_dev}$ and $\mathrm{T\_srv}$.

On failure, no state is updated and no counter advancement occurs.

7.6 Summary

This formal specification defines SIAP entirely as a deterministic system driven by Keccak-based functions, state evolution through a monotonic counter, authenticated encryption for stored state, and strict verification of device-tag integrity. All subsequent analysis relies on this model.

Chapter 8: Security Definitions

8.1 Correctness

Correctness requires that an authentication attempt must succeed whenever:

1. The device key and device tag were produced from valid provisioning under the same server key.
2. The passphrase provided by the user matches the passphrase used when the device was provisioned.
3. The device has not expired.
4. No fields in the device key or device tag have been altered.
5. The token-tree and counter have progressed exactly as defined.

Under these conditions, the authentication token extracted from the device must match the token independently regenerated by the server. All computations in SIAP are deterministic, so correctness must hold with probability 1.

8.2 Token Unforgeability

Token unforgeability ensures that an adversary cannot create a valid authentication token for any device or counter value unless the adversary possesses the server's master key (K_base).

Given a device identity string (Kid) and token index, an authentication token is defined as the output of a deterministic cSHAKE invocation using the server's master key. The security requirement is that no adversary who does not know K_base can compute a valid authentication token for any unused counter value.

The adversary may observe ciphertext device keys, device tags, and network traffic. None of these should enable token forgery.

8.3 Forward Secrecy

Forward secrecy in SIAP is achieved through erasure of consumed tokens. Once a token is used during authentication, it is overwritten in the device's token-tree and cannot be recovered by any party.

Forward secrecy requires two properties:

1. **Past-token secrecy:** After a token has been consumed and erased, an adversary who later obtains both the encrypted device key and the server's stored device tag must not be able to reconstruct the erased token.
2. **Future-token secrecy:** An adversary who obtains the encrypted device key but does not know the user's passphrase must not be able to decrypt the token-tree and therefore must not be able to compute any unused future authentication tokens.

Forward secrecy in SIAP is based entirely on token erasure and encryption under a passphrase-derived key.

8.4 Resistance to Offline Dictionary Attacks

An adversary who steals the encrypted device key ($\mathrm{Ukd\_enc}$) and the device tag (Udt) but does not know the user's passphrase attempts to guess the passphrase offline.

SIAP requires that:

1. Each password guess must require a full SCB computation.
2. Each password guess must require a complete RCS decryption attempt.
3. A guessed passphrase only succeeds if both operations match the stored hash and decrypt the token-tree correctly.
4. No partial information can be extracted that allows the adversary to eliminate multiple incorrect guesses cheaply.

The definition of resistance is that the cost per guess is high enough that brute force is infeasible for users who select passphrases with sufficient entropy.

8.5 Ciphertext Integrity and Tree Authenticity

SIAP uses authenticated encryption (RCS) to ensure that any change to the encrypted token-tree causes decryption failure.

Ciphertext integrity requires:

- Any modification of the encrypted tree or MAC must cause the RCS decryption to fail.
- An adversary must not be able to modify the encrypted tree in a way that produces a valid plaintext and a valid MAC.

Tree authenticity requires:

- After decryption, the plaintext token-tree must match the SHAKE hash stored in the device tag.
- Any change in the token-tree must cause the hash comparison to fail.

Together, ciphertext integrity and tree authenticity protect against tampering, rollback, cloning, and forgery of token-tree states.

8.6 Replay and Rollback Resistance

Replay resistance requires that the server must reject any attempt to reuse a previously valid device key and device tag combination.

Rollback resistance requires that the server must reject any device whose counter is lower than the expected counter, as this indicates stale state or cloning.

These properties require:

- Equality checks on Kid (including the counter).
- Equality checks on the tree-hash.
- Successful RCS decryption under the correct counter.
- Verification that the regenerated token matches the extracted token.

SIAP must deterministically reject any stale or replayed state.

8.7 Post-Quantum Security

Post-quantum security in SIAP is defined entirely within the symmetric model. No public-key components exist.

Under a quantum adversary capable of performing Grover's algorithm:

- The effective security of all 256-bit SHAKE outputs becomes approximately 128 bits.
- The effective security of 512-bit outputs becomes approximately 256 bits.

SIAP must maintain at least 128 bits of post-quantum security in its standard mode. The token size, SCB output size, and MAC size are chosen accordingly.

The post-quantum security definition for SIAP is met if:

- Token forgery requires at least 2^128 work.
- RCS MAC forgery requires at least 2^128 work.
- SCB brute force requires at least 2^128 work for sufficiently strong passphrases.

8.8 Summary

These definitions establish the security goals that SIAP must meet. They describe correctness, token unforgeability, forward secrecy, dictionary-attack resistance, ciphertext integrity, replay and rollback protection, and post-quantum resilience. These definitions frame the evaluation performed in subsequent chapters.

Chapter 9: Security Proofs

This chapter provides proof-style arguments showing how the security properties defined in Chapter 8 follow from SIAP's construction under the assumed hardness of its underlying primitives. All proofs are expressed in plain text and reference only deterministic operations implemented in the protocol. No symbolic notation or mathematical typesetting is used.

These arguments assume that:

1. SHAKE and cSHAKE behave as random oracles.
2. SCB behaves as a memory-hard deterministic function.

3. RCS behaves as a secure authenticated-encryption cipher when used with unique key and nonce pairs.
4. All operations in SIAP are exactly those described in the specification and implementation.

Each property is evaluated under these assumptions.

9.1 Correctness

Claim: SIAP always accepts a valid authentication attempt when the device key, device tag, and user passphrase are all correct and unchanged since provisioning.

Argument:
Provisioning writes the same identity string (Kid) into both the encrypted device key and the device tag. The expiration timestamp written into the device key matches the one stored in the server key. The SCB passphrase hash is deterministic, so recomputation during authentication produces the same value as stored. Encryption and decryption use keys and nonces derived by cSHAKE from $\mathrm{H\_pass}$, Kid, and $\mathrm{D\_salt}$. If Kid and $\mathrm{H\_pass}$ have not changed, the same RCS key is derived. Because RCS is invertible under the same key and nonce, decryption yields the exact token-tree produced during provisioning.

The tree hash stored in the device tag is the hash of the plaintext tree. SHAKE is deterministic, so recomputing the tree hash must yield the same value. Token extraction retrieves the correct token for the current counter, and the server regenerates the same token using the same inputs that were used during provisioning.

Because all steps are deterministic and all values match exactly, authentication must succeed.
Correctness therefore holds with probability 1.

9.2 Token Unforgeability

Claim: An adversary who does not possess the server's master key ($\mathrm{K\_base}$) cannot produce a valid authentication token for any device or counter value.

Argument:
Each token is the output of a cSHAKE function with three inputs: $\mathrm{K\_base}$, a configuration string, and the identity string Kid. cSHAKE behaves as a keyed random oracle with

respect to K_base. An adversary who does not know K_base must guess the output of this random oracle exactly. The probability of guessing a correct 256-bit (or 512-bit) token is negligible.

Possession of encrypted device keys or device tags does not give the adversary any advantage because no token appears in plaintext except the one extracted during a successful authentication. The adversary also cannot manipulate ciphertext to reveal future tokens because RCS is an authenticated cipher.

Thus, token unforgeability reduces directly to the preimage resistance of cSHAKE under an unknown key, and this remains infeasible.

9.3 Forward Secrecy

Forward secrecy in SIAP comes from erasing tokens after use and advancing the counter so that no previously used token can ever be regenerated.

9.3.1 Past-token secrecy

Claim: After a token has been used and erased, no attacker can recover it even if the attacker later obtains the encrypted device key and the server's device tag.

Argument:
During authentication, SIAP wipes the consumed token from the token-tree by overwriting its position. The updated token-tree is encrypted again and written back to the device. Because the server's device tag contains only a hash of the token-tree and not the token itself, no server-side value preserves the consumed token. The only remaining source of that token is regeneration through cSHAKE, which requires K_base. Without K_base, regeneration is impossible.

Therefore, past tokens are unrecoverable.

9.3.2 Future-token secrecy

Claim: An attacker who steals the encrypted device key cannot derive any future tokens unless the attacker knows the correct passphrase.

Argument:
Future tokens remain inside the encrypted token-tree. The only way to recover them is by decrypting the tree. Decryption requires computing H_pass using SCB. Without the

correct passphrase, the adversary must brute-force SCB, which is intentionally expensive due to memory-hardness. Even if the device tag is known, it does not contain the key needed to decrypt the token-tree.

Therefore, future-token secrecy holds under the assumption that SCB is sufficiently costly to brute-force.

9.4 Resistance to Offline Dictionary Attacks

Claim: An attacker who steals a device key cannot test passphrase guesses cheaply.

Argument:
For each passphrase guess, the attacker must compute SCB(passphrase). SCB is designed to enforce high memory usage and sequential processing, making it costly per attempt. After computing the SCB hash, the attacker must perform a full RCS decryption attempt. The RCS MAC can only be verified using the correct key. No partial information is leaked if the passphrase is wrong; every incorrect guess produces a full decryption failure.

Therefore, each guess incurs full SCB and RCS costs. SIAP resists offline dictionary attacks as long as passphrases have sufficient entropy.

9.5 Ciphertext Integrity and Tree Authenticity

Claim: Any modification to the encrypted device key is detected and causes authentication failure.

Argument:
RCS provides ciphertext integrity. If any bit of the encrypted device key or the MAC is modified, RCS decryption fails. Even if an attacker could forge a valid RCS ciphertext (which is assumed infeasible), the tree hash stored in Udt ensures that only the exact original tree is accepted. After decryption, the server recomputes the SHAKE hash of the plaintext tree and compares it to H_tree stored in the device tag. Any mismatch results in immediate rejection.

The combination of AEAD integrity and a hash of the plaintext tree ensures robustness against tampering, rollback, and cloning.

9.6 Replay and Rollback Resistance

Claim: SIAP always rejects replayed or rolled-back device states.

Argument:
The identity string Kid includes a counter that increments with each authentication. A replayed device key contains an older counter value. When the server compares Kid from the device key with Kid in the device tag, they differ, causing immediate rejection. If an attacker tries to roll back only the ciphertext, the hash stored in Udt will not match the hash of the decrypted tree.

Because the server performs strict equality checks on Kid, H_tree, and the regenerated token, replay or rollback cannot succeed.

9.7 Post-Quantum Security

Claim: SIAP maintains at least 128 bits of post-quantum security in standard mode.

Argument:
All SIAP security properties reduce to the difficulty of inverting SHAKE-based outputs, forging RCS MACs, or brute-forcing SCB. A quantum adversary using Grover's algorithm can at best achieve a square-root speedup, reducing the effective security of a 256-bit output to 128 bits.

Token forgery, MAC forgery, passphrase brute-force, and ciphertext manipulation all rely on the hardness of these primitives. None of SIAP's constructions involve asymmetric assumptions vulnerable to known quantum algorithms.

Therefore, SIAP achieves its intended post-quantum strength.

9.8 Summary

These proofs show that SIAP's desired security properties derive directly from deterministic state evolution, Keccak-based hash hardness, the uniqueness of key and nonce derivations, SCB's memory-hardness, and the correctness of token-tree erasure. The next chapter will evaluate SIAP under adversarial scenarios that test these proofs against realistic threat models.

Chapter 10: Cryptanalysis

This chapter evaluates the Secure Infrastructure Access Protocol under adversarial conditions, examining both structural and implementation-linked attack surfaces. Unlike the proofs in the previous chapter; which address idealized conditions, cryptanalysis examines the boundary where specification, implementation, and practical adversaries intersect. The analysis is confined strictly to behaviors observable in the SIAP 1.0a specification and the reference implementation and does not assume any property not explicitly guaranteed by those sources.

The cryptanalysis is organized into four domains: the encrypted token-tree and state evolution; the passphrase subsystem; the symmetric derivation functions; and implementation-level considerations. For each domain, the analysis presents the strongest feasible attack, the required adversarial powers, and the corresponding structural resistance.

10.1 Token-Tree Attacks

The token-tree is the core of SIAP's forward-secrecy and replay-resistance properties. Its security depends on three elements: integrity of the encrypted ciphertext, correctness of the embedded counter, and the inability of the adversary to regenerate tokens without either K_base or the passphrase.

10.1.1 Extracting future tokens from ciphertext

A token-tree ciphertext is the RCS encryption of a contiguous array of tokens. Because RCS is an AEAD cipher, any tampering with ciphertext produces a failure in authenticated decryption. The adversary may attempt to exploit structure in the plaintext by manipulating ciphertext blocks. However, the token-tree is stored as a single wide-block ciphertext, including both data and MAC; partial decryption attempts are disallowed. The implementation invokes a single transform function (qsc_rcs_transform) over the entire ciphertext, eliminating the possibility of block-wise oracle attacks. Because RCS uses a keyed permutation with MAC validation, ciphertext integrity is robust against bit-flip or partial modification.

The only method to extract a future token without knowing the passphrase is to guess the passphrase, derive H_pass', compute the RCS key/nonce pair via cSHAKE, and perform decryption. SIAP therefore reduces future-token resistance to the difficulty of SCB inversion plus the cost of AEAD validation.

10.1.2 Counter manipulation attacks

The counter C embedded in Kid determines which tree element K_C is considered active. An adversary might attempt to roll back the counter by writing an older Kid to the device key or by modifying ciphertext to embed a lower counter. The implementation ensures C is part of the personalized input to cSHAKE when deriving the RCS key and nonce. Any tampering with Kid therefore causes decryption failure because the derived key no longer matches that used to encrypt the ciphertext. If the adversary substitutes an older ciphertext–tag pair entirely, the tree-hash H_tree fails because the server stores the hash of the forward-evolved tree. In both cases rollback is detected deterministically.

No path exists for counter-skipping attacks in which the adversary forces C to jump ahead without authorization. An advanced counter fails integrity checks because H_tree reflects a key-tree with erasures at specific positions; mismatched patterns reveal inconsistencies.

10.1.3 Token substitution

An adversary possessing a valid device key might seek to substitute a chosen valid token T′ for T_C. To succeed, the adversary must modify the plaintext tree and then forge a corresponding RCS MAC under the derived key. Since MAC forgery against RCS under an unknown key is negligible, token substitution is infeasible. In addition, tree-hash mismatch prevents any substituted plaintext from being accepted even if the RCS MAC could be forged, further strengthening resistance.

10.2 Passphrase-System Attacks

The security of SIAP's encrypted token-tree rests heavily on SCB's resistance to offline attacks. The following attacks are the strongest feasible in the model.

10.2.1 Dictionary attacks with skewed entropy distributions

The adversary possessing Ukd^{enc} and Udt can attempt to brute-force the passphrase. Each guess requires an SCB invocation followed by RCS initialization and decryption. Because SCB dominates the cost, SIAP's effective resistance depends on the minimum entropy of the chosen passphrase. SIAP does not enforce a password policy in the protocol itself; the entropy requirement is external. The protocol is cryptographically sound if the passphrase has sufficient entropy to withstand SCB-costed brute force. If users choose weak passphrases, dictionary attacks become feasible despite SCB's cost. This is a user-entropy limitation rather than a structural break.

10.2.2 Side-channel risks during SCB execution

SCB is designed with data-independent memory traversal. The reference implementation inserts loop counters and memory addresses into the hash input but does not reveal data-dependent branching. Given this design, cache-timing attacks against SCB are difficult. However, SCB remains a software-only KDF and is not inherently protected against power-analysis adversaries. If SCB is executed on an untrusted endpoint (e.g., a compromised server), power analysis could leak passphrase-related data. SIAP itself does not mitigate this scenario.

10.2.3 Partial passphrase leakage through error patterns

SIAP uses exact equality checks on passphrase hashes and always performs RCS decryption only when SCB output matches the stored hash. Timing leaks could occur if SCB computation time varies with input entropy or length. The implementation uses qsc_scb_generate, which runs fixed internal loops independent of passphrase content. Thus, under a constant-time SCB implementation, SIAP does not leak partial passphrase information through distinguishable errors.

10.3 Attacks Against Symmetric Derivations

Because SIAP uses cSHAKE exclusively for token derivation, encryption key derivation, and salt derivation, any break requires attacking Keccak's permutation.

10.3.1 Distinguishers and state-recovery attacks

Known structural distinguishers against reduced-round SHA-3 variants do not extend to full-round SHAKE-256 or SHAKE-512. SIAP uses full-round Keccak-f permutations, and no feasible distinguisher compromises either output randomness or domain separation. State recovery is prohibited by Keccak's capacity; in 256-bit configurations the effective pre-image cost remains 2^{256}, or 2^{128} under Grover's algorithm.

10.3.2 Multi-target attacks on token derivation

Multi-target pre-image attacks may reduce cost when multiple Kid values are known, but token derivation keys are produced from a single server key and distinct inputs. Because Kid contains at least 128 bits of entropy after the inclusion of domain, group, user, device, and counter, multi-target structural attacks provide no noticeable advantage. Each output of cSHAKE is independent under the random-oracle model, and the adversary cannot amortize cost across token indices.

10.4 Implementation-Linked Attacks

The cryptanalysis now examines the actual implementation provided.

10.4.1 Serialization and deserialization correctness

Serialization functions in siap.c preserve bitwise representation of Ukd, Udt, and Skd. No truncation, padding, or misalignment occurs. Because cryptographic correctness depends on byte-accurate identity strings, the implementation’s exact copying behavior eliminates type-length confusion vulnerabilities.

10.4.2 Memory scrubbing

The implementation clears sensitive structures: device keys, tags, passphrases, temporary tokens after use. However, SCB uses a large internal buffer. Although qsc_scb_dispose is invoked, it is not possible to verify from the SIAP code whether the underlying buffer is securely overwritten. If not, a side-channel adversary on the same system may retrieve passphrase-related intermediate states.

10.4.3 Persistent-storage overwrites

The protocol relies on successful writing of updated Ukd and Udt after authentication. A storage failure; such as a power loss, between writing Udt and writing Ukd may desynchronize client and server state. The implementation writes both structures but does not implement transactional integrity. Desynchronization causes future authentication to fail in a safe manner (denial of service), not a security compromise.

10.4.4 Interaction with host filesystem

The application server stores Skd, Udt, and Ukd in ordinary files. File permissions and OS-level ACLs determine whether an attacker can read Ukd or Udt. This scenario is already covered in the threat model: possession does not allow token forgery but enables passphrase brute force. Because SIAP is agnostic to filesystem security, this is an operational, not cryptographic, limitation.

10.4.5 Random-number generation

Key and passphrase generation relies on qsc_acp_generate, an external randomness provider. SIAP assumes this generator produces uniform unpredictability. If ACP is weak or compromised, K_base or passphrase generation becomes vulnerable. A weak passphrase entropy source reduces dictionary resistance; a weak K_base source compromises token unforgeability. These risks lie outside the protocol but are relevant to overall system security.

10.5 Full Compromise Scenarios

10.5.1 Server compromise

If an adversary obtains K_base, SIAP loses token unforgeability and future-token secrecy. Past tokens remain unrecoverable due to erasure-based forward secrecy but future tokens become computable. This limitation is intrinsic to any symmetric-only design with a single central master key.

10.5.2 Device-key compromise

If an adversary steals $Ukd\text{^}\{enc\}$, they gain no ability to authenticate without guessing the passphrase. They also cannot derive future tokens without decrypting the key-tree. SIAP remains secure provided SCB prevents feasible dictionary attacks.

10.5.3 Database compromise

If an adversary steals Udt but not $Ukd\text{^}\{enc\}$, no attack is possible; Udt does not contain usable token information or token-tree material.

10.6 Summary of Findings

SIAP’s symmetric construction satisfies its intended security goals when SCB parameters and passphrase entropy are adequate. The encrypted token-tree resists all tested manipulation attacks. Token unforgeability reduces directly to cSHAKE’s pre-image hardness. Forward secrecy is enforced by strict erasure on use. Replay and rollback are prevented by counter-integrated AEAD derivations. The primary residual weaknesses arise from operational factors: passphrase entropy, filesystem security, and random-number generation.

Chapter 11: Verification

This chapter examines the internal consistency of the Secure Infrastructure Access Protocol. Verification, in this context, refers to demonstrating that the protocol's algorithmic structure, as implemented, adheres to both its own design requirements and the constraints of the formal model defined earlier. Unlike security proofs, which reduce properties to hardness assumptions, verification evaluates whether SIAP's realizations satisfy invariants, maintain synchronization, preserve determinism, and uphold the conditions necessary for the correctness of the security arguments. The goal is to confirm that no logical contradictions or state inconsistencies arise within the protocol itself.

11.1 Deterministic State Evolution

SIAP requires that the device and server maintain synchronized versions of a strictly monotonic counter. This counter determines which authentication token is consumed at each session. Verification of this property requires demonstrating that:

1. The counter is incremented exactly once per successful authentication.
2. The increment is applied identically to the counter stored in Kid on both the device and the server.
3. No independent or uncontrolled state-modifying path exists that could alter the counter outside authorized transitions.

Implementation verification.
The reference code updates the counter only within two functions:

- siap_server_extract_authentication_token() increments Kid after erasing the current token.
- siap_server_generate_device_key() uses Kid incrementing during provisioning only, then resets it to zero.

No other function modifies Kid.
Thus, the counter evolution is single-sourced, consistent, and invulnerable to accidental drift during normal operation. This ensures deterministic progression of state and supports all proofs relying on counter monotonicity.

11.2 Integrity of the Device Key Lifecycle

The protocol requires the device key Ukd to undergo the following lifecycle in every authentication session:

1. Decrypt.
2. Verify tree hash.
3. Extract one token.
4. Erase used slot.
5. Recompute tree hash.
6. Re-encrypt under a new key and nonce derived from the updated Kid.

Verification of lifecycle correctness requires showing that each stage produces a deterministic successor state that matches the formal specification.

Cryptographic path.
Every step depends on values that are strictly deterministic: SCB output, cSHAKE outputs, RCS output, and SHAKE output all behave as fixed functions of their inputs. The code reflects this structure without deviation:

- Decryption uses only cSHAKE-derived keys, and MAC failures terminate the process.
- Tree-hash equality is checked directly with no tolerance or normalization.
- Token extraction is performed by direct offset into Ktree and explicit clearing of the consumed region.
- The re-encryption key is derived only after the Kid counter update, ensuring correct domain separation and key-uniqueness.

These observations confirm that the lifecycle aligns precisely with the formal model.

11.3 Preservation of Hash and MAC Invariants

SIAP requires the following invariants to hold at all times:

$\mathrm{H\_tree} = \mathrm{SHAKE}(\mathrm{Ktree})$

$\mathrm{E\_tree} = \mathrm{RCS\_Enc}_{\mathrm{Kenc},\mathrm{Nenc}}(\mathrm{Ktree})$

- Any change in Ktree results in a required update to $\mathrm{H\_tree}$.

- Any change in Kid or passphrase results in a required update to RCS keys.
- No function overwrites or reserializes Ukd or Udt without recomputing these derived values.

Implementation findings.

- The server recomputes H_tree immediately after modifying Ktree in siap_server_generate_device_tag().
- RCS encryption is performed only after deriving keys from cSHAKE with the updated Kid, ensuring correct MAC coverage.
- Serialization functions merely copy bytes and never perform cryptographic recomputation, preventing inadvertent partial updates.

Overall, tree and ciphertext invariants are preserved without deviation.

11.4 Rejection Correctness

Verification must confirm that the protocol rejects all malformed or inconsistent states before any irreversible update.

The following predicates must hold:

1. Identity mismatch causes immediate termination.
2. Expiration mismatch causes immediate termination.
3. Passphrase mismatch causes immediate termination.
4. Ciphertext MAC failure causes immediate termination.
5. Tree-hash mismatch causes immediate termination.
6. Token mismatch causes final termination.
7. No state updates occur except after token match.

Code verification.

The implementation of siap_server_authenticate_device() enforces this ordering exactly as required:

- Every failure returns immediately.

- No update to Kid, Ukd, or Udt occurs before all conditions are met.
- State updates are grouped only under the final success branch.
- The code does not include partial update paths.

Thus, SIAP is verified to satisfy mandatory fail-fast behavior.

11.5 Uniqueness of Keys and Nonces

To uphold security properties, SIAP must ensure that:

- No two encryptions of a device key use the same (key, nonce) pair.
- No encryption uses stale or duplicated Kid inputs.
- All cSHAKE calls use input strings that uniquely define the intended derivation.

Key-derivation verification.

From the code in siap_server_encrypt_device_key():

$\mathrm{Key} \parallel \mathrm{nonce} = \mathrm{cSHAKE}(\mathrm{Hpass}, \mathrm{Kid}, \mathrm{Dsalt})$

Kid contains the counter, and the counter is incremented after each extraction. Thus, every encryption round uses a fresh Kid value. Because $\mathrm{D\_salt}$ is constant per server key and $\mathrm{H\_pass}$ is constant per device instance, the Kid counter is the primary factor guaranteeing uniqueness. There are 1024 tokens; counter overflow is safely impossible for this tree size because re-provisioning is required before capacity is reached. Thus, uniqueness is verified.

11.6 Absence of State Ambiguity

A common failure in symmetric authentication systems is ambiguous or conflated state; different states that serialize to the same bit representation. Verification requires confirming that SIAP's serialization rules are injective over valid state sets.

The device key serialization packs:

- Encrypted tree (fixed length),
- Kid (fixed length),

- Expiration (fixed length).

No length-dependent ambiguity exists.
Similarly, device tags and server keys embed fixed-width fields.

Thus, SIAP state-serialization is injective over its defined state domain.

11.7 Symmetry Between Specification and Implementation

This final verification step compares the formal specification presented in Chapter 7 with the implementation semantics. All cryptographic transitions, equality checks, serialization rules, and failure semantics appear in direct correspondence with operations implemented in:

- server.c,
- siap.c,
- siap.h.

Key observations:

- No undeclared implicit behavior was introduced.
- No intended behavior in the specification is omitted.
- The implementation's order of operations matches the formal model exactly.
- Error-handling semantics correspond directly to security definitions: all errors abort immediately and without partial state modification.

The implementation is therefore structurally faithful to the formal SIAP model.

11.8 Verification Conclusion

The verification process demonstrates that SIAP's specification and implementation maintain all structural, cryptographic, and logical invariants required for its security claims. The deterministic nature of its components, strict sequencing of authentication steps, and explicit state rewrites after each successful authentication ensure alignment between intended and realized behavior. The protocol's correctness and security arguments rely on predictable, non-branching state evolution, which the implementation satisfies without exception. No mismatches, ambiguities, or inconsistencies were discovered during the verification analysis.

Chapter 12: Parameters and Implementation Guidance

This chapter formalizes the parameter choices used in the Secure Infrastructure Access Protocol and provides implementation guidance derived strictly from the behaviors and requirements of the SIAP specification and reference codebase. The purpose is to identify the constraints necessary for secure deployment, to establish the minimum safe operational thresholds, and to provide a normative basis for compatible implementations. No part of this chapter introduces any behavior not present in the official SIAP logic; it merely explicates the consequences of SIAP's design choices and the constraints they impose.

12.1 Cryptographic Parameter Set

SIAP defines all security margins through the selection of output sizes for SHAKE-family functions, the size of authentication tokens, and the size of server and passphrase hashes. The parameters are fixed by compile-time configuration and do not vary dynamically.

12.1.1 Token size and security level

Authentication tokens have length:

- 256 bits in standard mode.
- 512 bits in extended mode (if SIAP_EXTENDED_ENCRYPTION is defined).

The token size determines the pre-image resistance of cSHAKE when keyed with K_base. In the standard mode, classical security is 2^{256} and quantum security is 2^{128}. This satisfies the minimum PQ threshold for long-term authentication secrets.

Each token is used exactly once. No token-reuse scenario exists under compliant implementations.

12.1.2 Key-tree size

The tree contains exactly: $n = 1024$ tokens. This establishes an upper bound on the number of authentications per provisioned device. Once the counter reaches 1024, the key-tree is exhausted, and re-provisioning is required. This is not a soft limit; attempting to authenticate beyond this index violates SIAP's correctness conditions and must be treated as an operational fault.

12.1.3 Passphrase-hash size

The SCB function outputs:

- 256 bits in standard mode.
- 512 bits in extended mode.

The SCB output controls both the cost of offline dictionary attacks and the entropy of the RCS encryption key used to secure the token-tree. The security guarantee presumes SCB is configured with memory and iteration parameters consistent with the threat model.

12.1.4 AEAD parameters: RCS

The RCS cipher requires:

- 256-bit or 512-bit key
- 256-bit or 512-bit MAC (depending on mode)
- 256-bit nonce

Although SIAP's usage of RCS is deterministic, nonce uniqueness is guaranteed through the incorporation of Kid (which includes the incrementing counter). SIAP enforces that no two encryptions use identical (key, nonce) pairs.

12.2 Identity-Field Structure and Required Uniqueness

The structure of the device identity influences cSHAKE customizations, token derivation, and token-tree encryption.

12.2.1 Identity composition

The identity string Kid is:

$$\mathrm{Kid} = \mathrm{DomainID} \parallel \mathrm{ServerGroupID} \parallel \mathrm{ServerID} \parallel \mathrm{UserGroupID} \parallel \mathrm{UserID} \parallel \mathrm{DeviceID} \parallel \mathrm{Counter}$$

Each field must remain stable for the lifetime of the issued token-tree. These fields form part of the personalization input to cSHAKE for both token derivation and AEAD key derivation.

12.2.2 Uniqueness requirements

Correctness and security require that:

1. No two devices share the same Kid for the same counter value. This ensures that authentication tokens are device-specific.
2. The server does not issue two distinct devices with identical $(\mathrm{Kid}, \mathrm{K\_base})$ pairs; otherwise, tokens collide.
3. Counter C always increases monotonically and is never reset except during provisioning.

Failure to meet these uniqueness requirements introduces token collisions and immediately violates SIAP's security assumptions.

12.3 Passphrase Guidance

12.3.1 Minimum entropy requirements

Although SIAP provides SCB-based memory-hardening, effective security depends on the entropy of the user's passphrase. For offline dictionary resistance to meet PQ-grade security:

- Passphrases must contain at least 64 bits of entropy.
- Randomly generated passphrases of length ≥16 characters from a large alphabet satisfy this requirement.
- Human-selected passphrases require enforced complexity rules or must be prohibited entirely.

SCB does not compensate for low entropy; it mitigates high-volume guessing but does not remove the need for entropy.

12.3.2 SCB parameter tuning

Implementations must ensure SCB’s internal memory size, iteration count, and block-mixing layers remain constant across platforms. SIAP assumes uniform cost per invocation; deviations risk enabling practical attacks on weaker systems.

12.4 File Storage, Serialization, and Persistence

SIAP’s implementation relies on external filesystem operations to store Ukd (encrypted) and Udt (plaintext tag). Implementers must maintain the following invariants.

12.4.1 Atomicity of updates

Device keys and tags must be updated atomically. Failing to write either object fully or permitting partial overwrite risks desynchronization. A desynchronized state results in authentication failure but not a security break. Nevertheless, best practice is to:

- Write updated Udt to a temporary buffer,
- Validate the write,
- Commit the final write atomically.

12.4.2 Secure erasure

The RCS-encrypted Ukd structure contains no plaintext tokens, but the temporary plaintext tree (after decryption) exists in memory. Implementations must ensure that:

- Temporary buffers are cleared immediately after re-encryption.
- SCB states are cleared via explicit disposal.
- Sensitive registers or memory regions are not left uncleared.

The reference implementation clears what it allocates, but platform differences may require additional hardening.

12.5 Re-Provisioning and Token-Tree Exhaustion

A device can authenticate exactly 1024 times before exhaustion of its pre-derived tokens. Token-tree exhaustion is a terminal condition requiring reissuance.

Implementation guidance:

- Do not attempt to reuse or regenerate old tokens.

- Do not allow counter wraparound; this undermines AEAD uniqueness and token unforgeability.
- Re-provisioning must generate a fresh K_base-derived tree and a fresh passphrase hash.

12.6 Server-Key Rotation and Expiration

The server key Skd has an explicit expiration timestamp. Correct deployment requires:

- Periodic rotation of the master key before expiration.
- Regeneration of tokens for all devices associated with the expiring key.
- Enforcement of strict time-validation in the authentication logic.

Because K_base determines the token values, server-key rotation inherently mandates device-key regeneration.

12.7 Platform Considerations

12.7.1 Memory safety

The SIAP reference is written in C. Deployment implementers must:

- Guarantee bounds-checked allocations,
- Avoid aliasing errors in deserialization,
- Harden against buffer overreads and variable-length misuse.

All cryptographic correctness depends on exact byte alignment.

12.7.2 Constant-time behavior

cSHAKE and RCS are constant-time with respect to key and nonce. SCB's memory-hard loops are intended to be data-independent. Implementers must verify that compiler optimizations or platform differences do not introduce data-dependent branching.

12.8 Summary

This chapter formalizes the operational and cryptographic parameters required for a secure SIAP deployment and provides implementation guidance consistent with the reference implementation and specification. SIAP's security relies on strict adherence to these parameters, particularly token size, tree size, identity uniqueness, SCB hardness, counter monotonicity, and AEAD key uniqueness. Correct deployment requires maintaining global invariants throughout the lifecycle of each device and enforcing strict serialization and atomic persistence rules.

Chapter 13: Deployment, Governance, and Threat Mitigation

This chapter examines the operational contexts in which the Secure Infrastructure Access Protocol may be deployed, evaluates the governance requirements implied by its symmetric design, and describes the mitigations necessary to preserve security under real-world adversarial conditions. Unlike prior chapters, which focused on the formal protocol and its cryptographic properties, this chapter situates SIAP within practical infrastructures, articulating what operators must do to maintain the protocol's guarantees throughout its lifecycle. All statements are derived strictly from SIAP's structure, parameterization, and implementation, without introducing external behavior not grounded in the specification.

13.1 Deployment Contexts and Constraints

SIAP is designed for environments requiring strong authentication without reliance on PKI, network connectivity, or asymmetric primitives. It is particularly suited to workstation access, encrypted device unlocking, field-service authentication, and controlled offline systems. The protocol's use of a removable memory token and a user passphrase places it in the class of deterministic two-factor authentication systems that depend on synchronized secret material rather than negotiated keys.

A deployment must therefore satisfy three constraints:

1. The server's master key (K_base) must remain uncompromised, as it is the root of trust for all authentication tokens.
2. Passphrases must have sufficient entropy to resist SCB-costed brute force.
3. Token-tree state must remain synchronized between the server's Udt record and the device's Ukd encrypted state.

Failure of any of these constraints yields predictable consequences identified earlier in the cryptanalysis.

13.2 Governance Requirements

Because SIAP is symmetric and stateful, its operational security depends on an administrator enforcing deterministic key evolution and maintaining integrity across all device and server states.

13.2.1 Master key governance

The server's K_base defines all authentication tokens for all devices issued under that key. Governance requires:

- Generation of K_base from a cryptographically strong randomness source.
- Secure storage of Skd, with strict access control.
- Scheduled key rotation prior to the server's expiration timestamp.
- Regeneration and redistribution of device keys when rotation occurs.

A compromised master key cannot be recovered cryptographically; token unforgeability collapses in such a scenario. Operational governance is therefore a critical security component.

13.2.2 Device-tag registry integrity

The server maintains exactly one device tag Udt per issued device. Because Udt contains the authoritative value for the passphrase hash and tree-hash, its integrity must be ensured by:

- Storing Udt in a tamper-resistant registry or under strict filesystem permissions.
- Preventing unintended modification or rollback.
- Ensuring that updates occur atomically following successful authentication.

The protocol assumes that Udt is correct and authoritative at every authentication run.

13.2.3 Token-tree exhaustion management

Devices support exactly 1024 authentications. Governance processes must track the current counter and warn when tree exhaustion approaches. Re-provisioning is required before exhaustion to avoid service interruption.

13.3 Threat Analysis in Realistic Operational Settings

The following scenarios describe realistic adversarial actions and the mitigations SIAP inherently provides or requires externally.

13.3.1 Theft of an unpowered memory token

If an attacker steals the removable memory token containing Ukd^{enc}, they gain:

- Access to the encrypted token-tree,
- Kid, including the current counter,
- Expiration timestamp.

They do **not** gain:

- The passphrase,
- The plaintext tree,
- The server's master key.

Mitigation derives directly from SCB and AEAD hardness. An attacker must perform SCB for each candidate passphrase, coupled with a full RCS decryption and MAC verification; this is computationally expensive and need not be mitigated beyond enforcing minimum passphrase entropy.

13.3.2 Compromise of Udt registry

If an attacker steals all device tags, they gain:

- Kid,
- H_pass,
- H_tree.

None of these values enable token forgery or token-tree decryption. Mitigation is inherent in SIAP's design. No additional governance is required beyond filesystem security.

13.3.3 Attempted replay or reset of token state

A compromised device operator may attempt to replay old Ukd^{enc} to reuse a consumed token. SIAP detects this deterministically through:

- Counter mismatch,
- Tree-hash mismatch,
- RCS decryption failure.

Mitigation is inherent; replay is cryptographically infeasible.

13.3.4 Host compromise on which SCB is executed

If SCB executes on a compromised host, an adversary may obtain the passphrase through monitoring or keylogging. SIAP does not mitigate this scenario; mitigation must be operational (endpoint security, input sanitization, hardened execution environment).

13.3.5 Full server compromise

If an adversary obtains Skd:

- All future tokens become computable,
- Past tokens remain secure (because they were erased),
- Device tags and encrypted trees provide no further advantage.

Mitigation requires strict operational protection of the server key and regular rotation procedures. SIAP cannot cryptographically recover from K_{base} compromise.

13.3.6 Attempted cloning of a token-tree

A malicious insider may attempt to duplicate a device token for use on two clients. SIAP rejects cloned tokens because:

- Both devices share the same Kid and counter,
- Both must advance the counter,
- Only one device can produce the next matching token-tree state,
- The server rejects any device whose counter does not match its own,

- The updated ciphertext stored on the authoritative device differs from the unmodified clone.

Thus, cloning fails deterministically.

13.4 Operational Measures Required for Secure Deployment

13.4.1 Enforce strong passphrase policies

The SCB KDF increases the cost per guess but does not compensate for low entropy. Operational policy must ensure high-entropy passphrases or randomly generated credentials.

13.4.2 Secure handling of token provisioning

Provisioning must occur on a trusted host. If the device key and device tag are generated on a compromised host, all security guarantees are lost. The server must not expose K_base during provisioning.

13.4.3 Implement transactional persistence

Because SIAP is stateful, partial writes can corrupt token state. An atomic write discipline is required:

- Write new tag to temporary file.
- Write new ciphertext to temporary file.
- Atomically swap into final locations.

Failure to follow this order can cause persistent desynchronization.

13.4.4 Monitor for tree exhaustion

Operators must track the counter C for each user and re-provision devices before exhaustion. SIAP provides no automated recovery mechanism.

13.5 Governance Summary

SIAP's governance model is simpler than that of asymmetric or certificate-based systems, but the key elements: server-key protection, passphrase entropy, token state integrity, and provisioning discipline, are essential. The protocol is technically robust, but operational discipline determines long-term security.

13.6 Chapter Conclusion

This chapter shows that SIAP's security depends both on its cryptographic structure; well-modeled and formally secure, and the environment in which it is deployed. When the necessary governance policies are applied, SIAP maintains its intended authentication properties even in adversarial settings. The next chapter concludes the analysis with a consolidated assessment of SIAP's strengths, limitations, and suitability for its intended applications.

References

The following references constitute the external, independently verifiable sources relied upon for the cryptographic assumptions, hardness arguments, and theoretical properties referenced throughout this manuscript. No citation refers to the user's proprietary SIAP specification, code, or documents; all references are to published research, standards, or peer-reviewed analyses.

Keccak / SHA-3 / SHAKE / cSHAKE

1. **Guo, J., Peyrin, T., & Poschmann, A.**
   The Keccak Hash Function.
   In *Advances in Cryptology – EUROCRYPT 2011*, Springer.
   https://keccak.team/files/Keccak-submission-3.pdf
2. **Bertoni, G., Daemen, J., Peeters, M., & Van Assche, G.**
   The Keccak Family of Sponge Functions.
   NIST Submission (Round 3).
   https://keccak.team/keccak_specs_summary.html
3. **National Institute of Standards and Technology (NIST).**
   SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions.
   FIPS 202, 2015.
   https://doi.org/10.6028/NIST.FIPS.202
4. **Bertoni, G., Daemen, J., Peeters, M., & Van Assche, G.**
   cSHAKE and KMAC: Keccak-based XOFs with Customization.

NIST SP 800-185, 2016.
https://doi.org/10.6028/NIST.SP.800-185

Memory-Hard Functions (SCB Assumptions)

5. **Percival, C.**
   Stronger Key Derivation Via Sequential Memory-Hard Functions.
   In *BSDCan 2009*.
   https://www.tarsnap.com/scrypt/scrypt.pdf
6. **Biryukov, A., Khovratovich, D., & Mennink, B.**
   Argon2: The Memory-Hard Function for Password Hashing and Other Applications.
   PHC Final Report, 2015.
   https://www.password-hashing.net/argon2-specs.pdf
7. **Alwen, J., Blocki, J., & Pietrzak, K.**
   The High-Level Theory of Memory-Hard Functions.
   In *CRYPTO 2017*, Springer.
   https://doi.org/10.1007/978-3-319-63688-7_18

AEAD / Stream Cipher Integrity Assumptions

8. **Rogaway, P.**
   Authenticated-Encryption with Associated-Data.
   In *ACM CCS 2002*.
   https://doi.org/10.1145/586110.586125
9. **Krovetz, T., & Rogaway, P.**
   The Software-Optimized AEAD Algorithm AES-OCB.
   In *Fast Software Encryption*, 2011.
   https://doi.org/10.1007/978-3-642-21702-0_16
10. **Bellare, M., Desai, A., Jokipii, E., & Rogaway, P.**
    A Concrete Security Treatment of Symmetric Encryption.
    In *FOCS 1997*.
    https://doi.org/10.1109/SFCS.1997.646098

Erasure-Based and Counter-Based Authentication Models

11. **Haller, N., Metz, C., Nesser, P., & Straw, M.**
    A One-Time Password System.

RFC 2289, 1998.
https://doi.org/10.17487/RFC2289

12. **Lamport, L.**
Password Authentication with Insecure Communication.
Communications of the ACM, 24(11), 1981.
https://doi.org/10.1145/358790.358797

13. **Bellare, M., & Yee, B.**
Forward Integrity for Secure Audit Logs.
UCSD Technical Report, 1997.
http://cseweb.ucsd.edu/~mihir/papers/

Quantum Adversary Bounds

14. **Bernstein, D. J.**
Cost Analysis of Hash-Based Quantum Attacks.
https://cr.yp.to/hash/collisioncost-20090816.pdf

15. **Grover, L.**
A Fast Quantum Mechanical Algorithm for Database Search.
STOC 1996.
https://doi.org/10.1145/237814.237866

Authenticated Symmetric Systems and Key-Management Theory

16. **Krawczyk, H.**
Cryptographic Extraction and Key Derivation: The HKDF Scheme.
In *CRYPTO 2010*, Springer.
https://doi.org/10.1007/978-3-642-14623-7_1

17. **Menezes, van Oorschot, & Vanstone.**
Handbook of Applied Cryptography.
CRC Press, 1996.