The Design and Formal Analysis of the Secure Infrastructure Access Protocol

John G. Underhill

Quantum Resistant Cryptographic Solutions Corporation

Abstract. The Secure Infrastructure Access Protocol (SIAP) is a symmetric authentication system that combines identity structured key material, a memory hard passphrase function, a cSHAKE based derivation interface, and a one time token tree to provide strong guarantees of authentication, replay resistance, and state continuity. This paper gives a formal treatment of SIAP, describes its cryptographic structure, and analyzes its security under classical and post quantum adversarial models.
The analysis confirms that SIAP achieves its intended authentication and token freshness properties when its stated assumptions hold, and identifies the conditions under which confidentiality of the token tree and resistance to offline attacks are maintained. The paper also introduces an engineering level description of SIAP derived directly from the implementation, providing a precise operational model for the protocol as it is realized in practice.

1 Introduction

1.1 Context and Motivation

The Secure Infrastructure Access Protocol (SIAP) is an authentication mechanism designed for controlled access to infrastructure systems, devices, and administrative domains. SIAP replaces traditional password based login methods with a structured combination of identity material, a memory hard passphrase function, a cSHAKE based derivation process, and a one time token tree derived from a single server key. The protocol is intended to provide strong guarantees against impersonation, replay, token reuse, database compromise, and offline attack.

SIAP is designed for environments that require predictable authentication behavior, auditability, and long term security. These environments benefit from a protocol in which authentication tokens evolve deterministically, where state divergence is detected immediately, and where the exposure of stored server side data does not allow the reconstruction of future authentication tokens. The use of a passphrase derived encryption layer further protects the device side key material against theft or physical access attacks.

1.2 Contributions

This paper presents a formal analysis of SIAP, aligned with its specification and its reference implementation. The contributions of this study are as follows.

First, the paper provides a complete engineering level description of SIAP as it is realized in the implementation. This description defines the exact data structures, derivation functions, transcript inputs, and update operations used in the protocol.

Second, the paper formalizes the authentication process using a sequence of well defined games. These games capture impersonation, replay, token misuse, and passphrase based attacks. The resulting framework allows the security of SIAP to be expressed in terms of the hardness of the underlying primitives, including the SCB passphrase function, cSHAKE, and the one time token tree construction.

Third, the paper evaluates SIAP under classical and post quantum adversarial models. The analysis identifies the assumptions required for authentication security, the limits of those assumptions, and the consequences of partial compromise of server or device state.

Finally, the paper provides a cryptanalytic evaluation of the protocol design. This evaluation examines token derivation, identity binding, state progression, and the consequences of tree exposure. It also gives conditions under which offline attacks or state desynchronization are prevented.

1.3 Structure of the Paper

The paper is organized as follows. Section 2 presents an engineering description of SIAP based directly on the reference implementation. Section 3 provides the formal specification used in the rest of the analysis. Section 4 defines the security model and adversarial capabilities. Section 5 introduces the security experiments used to evaluate impersonation and replay resistance. Section 6 gives the provable security analysis and relates SIAP to the assumptions on its primitives. Section 7 presents a cryptanalytic evaluation of the protocol. Section 8 discusses implementation alignment, performance, and operational considerations. Section 9 outlines limitations and directions for future work. The paper concludes with a summary of results.

2 Engineering Description of SIAP

This section provides an implementation faithful description of the Secure Infrastructure Access Protocol (SIAP). The description is derived directly from the reference source code and is expressed in implementation agnostic mathematical terms. This section defines the identities, key materials, internal state structures, and all operational flows used in the protocol. It also establishes the notation and structural foundation for the formal analysis in later sections.

SIAP is an offline symmetric authentication protocol that uses a server side master key, a structured identity string, a memory hard passphrase function, and a one time token tree. Each authentication consumes one token and advances the state on both the device and the server. SIAP provides symmetric forward secrecy through destructive leaf erasure and resists offline attacks through the use of the SCB hardened passphrase function.

2.1 Identities and Key Material

SIAP identities follow a hierarchical fixed width structure represented as a concatenation of integer fields. The primary identity used throughout the protocol is the device identity string

$$\mathrm{Kid} = \mathrm{DID} \parallel \mathrm{idx}$$

which consists of the device identifier DID and an index field that identifies the active token within the device token tree. The index field is incremented after each successful authentication.

The server maintains a single master key

$$K_{\mathrm{srv}} = (kbase, sid, dsalt, ts_{exp})$$

where $kbase$ is a uniformly random secret, sid is the server identifier, $dsalt$ is a server wide diversification salt computed as:

$$dsalt = \mathrm{cSHAKE}(kbase, config, sid),$$

and ts_{exp} is the expiration timestamp.

Each device carries an encrypted device key structure:

$$K_{\mathrm{dev}} = (\mathcal{T}, \mathrm{Kid}, ts_{exp})$$

where $\mathcal{T}$ is a fixed size array of one time tokens.

These tokens are derived as:

$$\mathcal{T}[i] = \mathrm{cSHAKE}(kbase, config, \mathrm{DID} \parallel i).$$

The server stores a corresponding device tag

$$\mathrm{Tag} = (\mathrm{Kid}, khash, phash)$$

where $khash$ is a SHAKE hash of the entire token tree and $phash$ is the SCB hardened hash of the user passphrase.

2.2 State and Data Structures

The following data structures are defined exactly as in the reference source code. Sizes are fixed and all fields have well defined alignment.

- **ServerKey**: $(kbase, sid, dsalt, ts_{exp})$.
- **DeviceKey**: $(\mathcal{T}, \text{Kid}, ts_{exp})$ where $\mathcal{T}$ is an array of 1024 tokens.
- **DeviceTag**: $(\text{Kid}, khash, phash)$.

The device key is stored on the device in encrypted form. The device tag is stored on the server in clear form. The server key is maintained only on the server.
The SCB passphrase function produces *phash*. This value is used for both verification and derivation of the encryption key for the device token tree.

2.3 Provisioning and Enrollment Flow

Enrollment constructs both the device key and the device tag. The operations are as follows.

1. The server generates the server key K_{srv} by selecting a random *kbase*, assigning *sid*, computing *dsalt* through cSHAKE, and setting the expiration timestamp ts_{exp}.
2. The server creates a device identity string DID and embeds it in Kid with the index field set to zero.
3. The server constructs the token tree $\mathcal{T}$ by iterating $i = 0$ to 1023 and computing
$$\mathcal{T}[i] = \text{cSHAKE}(kbase, config, \text{DID} \parallel i).$$
4. The server computes the tree hash
$$khash = \text{SHAKE}(\mathcal{T}).$$
5. The user supplies a passphrase. The SCB function computes a hardened passphrase hash
$$phash = \text{SCB}(\mathsf{passphrase}).$$
6. The server constructs the device tag $(\text{Kid}, khash, phash)$.
7. The server encrypts $\mathcal{T}$ to produce the sealed device key. The encryption key and nonce are derived by
$$rcskey = \text{cSHAKE}(phash, \text{Kid}, dsalt),$$
and applied through RCS in authenticated encryption mode.

The sealed device key and the device tag complete the enrollment process.

2.4 Authentication Flow

A complete authentication run proceeds as follows.

1. The device presents the sealed device key. The user supplies a passphrase. The SCB function computes *phash*.
2. The server loads the stored device tag $(\text{Kid}, khash, phash')$.
3. The server verifies identity consistency:
$$\text{Kid}_{dev} = \text{Kid}_{tag}.$$

4. The server verifies expiration by checking
$$ts^{dev}_{exp} \leq ts^{srv}_{exp}.$$

5. The server verifies the passphrase by a constant time comparison of $phash$ with $phash'$.

6. The server derives the RCS decryption key using $phash$, Kid, and $dsalt$, and decrypts $\mathcal{T}$.

7. The server recomputes $khash$ and verifies that it matches $khash$ in the device tag.

8. The server extracts the current token
$$T_{dev} = \mathcal{T}[idx],$$
then erases it:
$$\text{Erase}(\mathcal{T}[idx]).$$

9. The server derives the expected token:
$$T_{srv} = \text{cSHAKE}(kbase, config, \text{Kid}).$$

10. The server verifies the authentication by checking $T_{dev} = T_{srv}$ in constant time.

11. The server increments the index field inside Kid, regenerates $khash$, and re-encrypts the updated device key.

Each authentication consumes one token, provides symmetric forward secrecy, and advances the shared state.

2.5 Error Handling and State Updates

Authentication fails with explicit rejection under any of the following conditions:

- identity mismatch between the device key and the device tag,
- expiration mismatch with the server key,
- incorrect passphrase,
- authentication tag failure during decryption,
- mismatch between recomputed $khash$ and stored $khash$,
- mismatch between T_{dev} and T_{srv}.

On successful authentication the following updates occur:

1. the consumed token is erased from $\mathcal{T}$,
2. the index is incremented,
3. the device tag is updated with a fresh $khash$,
4. the device key is re-encrypted under the derived RCS key.

These updates must be stored persistently on both the device and the server.

2.6 Pseudo-code Representation

This subsection provides exact Pseudo-code derived directly from the reference implementation. All operations follow the control flow in the SIAP code base and use the same logical ordering.

2.6.1 Server Key Generation

The server key generation routine constructs the global SIAP server key from which all device token trees are derived. The function samples a uniform random base key *kbase*, binds it to a server identifier *sid*, sets a global expiration timestamp, and computes a diversification salt *dsalt* using cSHAKE. The resulting structure

$$K_{\mathrm{srv}} = (kbase, sid, dsalt, ts_{exp})$$

serves as the root of the symmetric hierarchy and is never exported from the server.

Algorithm 1 SIAP__SERVER__GENERATE__SERVER__KEY

```
Require: Output structure $K_{\mathrm{srv}}$, server identifier $sid$, key lifetime $\Delta t$
Ensure: Initialized server key $K_{\mathrm{srv}} = (kbase, sid, dsalt, ts_{exp})$
Zero all fields of $K_{\mathrm{srv}}$
$kbase \leftarrow$ CSPRNG(len_kbase)
$K_{\mathrm{srv}}.kbase \leftarrow kbase$
$K_{\mathrm{srv}}.sid \leftarrow sid$
$ts_{now} \leftarrow$ CurrentTime()
$K_{\mathrm{srv}}.ts_{exp} \leftarrow ts_{now} + \Delta t$
$dsalt \leftarrow$ cSHAKE($kbase, config\_salt, sid$)
$K_{\mathrm{srv}}.dsalt \leftarrow dsalt$
return $K_{\mathrm{srv}}$
```

2.6.2 Device Key Generation

The device key generation routine constructs the per device token tree and identity binding using the server key. The function initializes the device identity string Kid with the provided device identifier DID and an index field set to zero, copies the server expiration timestamp, and then fills the token tree by applying cSHAKE to the concatenation (DID $\|$ *idx*) under the server base key. The index field inside Kid is incremented for each token and reset to zero after the tree is generated. The resulting structure

$$K_{\mathrm{dev}} = (\mathcal{T}, \mathrm{Kid}, ts_{exp})$$

is ready to be encrypted under a hardened passphrase.

Algorithm 2 SIAP_SERVER_GENERATE_DEVICE_KEY

```
Require: Output structure $K_{\text{dev}}$, device identifier DID, server key $K_{\text{srv}}$
Ensure: Initialized device key $K_{\text{dev}} = (\mathcal{T}, \text{Kid}, ts_{exp})$
Zero all fields of $K_{\text{dev}}$
Initialize Kid from DID with index field $idx \leftarrow 0$
$K_{\text{dev}}.\text{Kid} \leftarrow \text{Kid}$
$K_{\text{dev}}.ts_{exp} \leftarrow K_{\text{srv}}.ts_{exp}$
for $i \leftarrow 0$ to KTREE_COUNT $- 1$ do
    Kid.$idx \leftarrow i$
    $K_{\text{dev}}.\mathcal{T}[i] \leftarrow \text{cSHAKE}(K_{\text{srv}}.kbase, config\_tree, \text{DID} \parallel i)$
end for
Kid.$idx \leftarrow 0$
$K_{\text{dev}}.\text{Kid} \leftarrow \text{Kid}$
return $K_{\text{dev}}$
```

2.6.3 Server Encrypt Device Key

This routine takes a provisioned device key record and produces the encrypted form that is stored in the server database. It derives the RCS key and nonce exclusively through a cSHAKE computation over the hardened passphrase hash, the structured device identity, and the server diversification salt. The resulting byte stream is split into the AEAD key and nonce used by RCS. The function then encrypts the device token tree to produce ciphertext together with a MAC. The plaintext token tree remains in memory only for the duration of the operation, and all temporary buffers are securely cleared before the function returns.

Algorithm 3 SIAP__SERVER__ENCRYPT__DEVICE__KEY

```
Require: dkey containing plaintext token tree dkey.ktree
Require: skey containing server diversification salt skey.dsalt
Require: phash the SCB hardened passphrase hash
Ensure: dkey.ktree replaced with ciphertext + MAC
assert dkey ≠ null, skey ≠ null, phash ≠ null
enkt ← new byte array of length SIAP_KTREE_SIZE + SIAP_MAC_SIZE
pkey ← new byte array of length SIAP_SERVER_KEY_SIZE + SIAP_NONCE_SIZE
pkey ← cSHAKE(phash, SIAP_HASH_SIZE,dkey.kid, SIAP_KID_SIZE,skey.dsalt,
  SIAP_SALT_SIZE)
kp.key ← pkey[0..SIAP_SERVER_KEY_SIZE-1]
kp.keylen ← SIAP_SERVER_KEY_SIZE
kp.nonce ← pkey[SIAP_SERVER_KEY_SIZE+SIAP_NONCE_SIZE-1]
kp.info ← null
kp.infolen ← 0
// initialize RCS AEAD cipher in encryption mode
rstate ← zero_initialized_RCS_state()
RCS__Initialize(rstate, kp, true)
// encrypt the token tree
RCS__Transform(rstate, enkt, dkey.ktree, SIAP_KTREE_SIZE)
// store ciphertext + MAC back into the device key record
Copy(dkey.ktree, enkt, SIAP_KTREE_SIZE + SIAP_MAC_SIZE)
// securely erase temporary buffers
Clear(enkt, SIAP_KTREE_SIZE + SIAP_MAC_SIZE)
Clear(pkey, SIAP_SERVER_KEY_SIZE + SIAP_NONCE_SIZE)
RCS__Dispose(rstate)
return
```

2.6.4 Server Decrypt Device Key

The routine SIAP__SERVER__DECRYPT__DEVICE__KEY takes a sealed device key record whose token tree is stored in encrypted form and attempts to recover the plaintext token tree. It first derives the RCS key and nonce by applying cSHAKE to the hardened passphrase hash, the structured device identity `kid`, and the server diversification salt `dsalt`. These derived bytes are split into a symmetric key and nonce and used to initialize the RCS state in decryption mode. The function then calls the RCS transform to authenticate and conditionally decrypt the ciphertext token tree into a temporary buffer. If decryption succeeds, the plaintext token tree is copied back into the device key structure. In all cases the temporary buffers and the derived key material are securely cleared before returning, and the function outputs a Boolean indicating whether decryption and authentication were successful.

Algorithm 4 SIAP_SERVER_DECRYPT_DEVICE_KEY

```
Require: dkey device key record with encrypted token tree dkey.ktree
Require: skey server key containing diversification salt skey.dsalt
Require: phash SCB hardened passphrase hash
Ensure: Returns true on successful decryption and authentication, false otherwise. On
    success, dkey.ktree contains the plaintext token tree.
assert dkey ≠ null, skey ≠ null, phash ≠ null
dect ← new byte array of length SIAP_KTREE_SIZE
pkey ← new byte array of length SIAP_SERVER_KEY_SIZE + SIAP_NONCE_SIZE
res ← false
if dkey = null or skey = null or phash = null then
    return false
end if
// derive RCS key and nonce from (phash, Kid, dsalt) using cSHAKE
pkey ← cSHAKE_KDF(phash, SIAP_HASH_SIZE,dkey.kid, SIAP_KID_SIZE,skey.dsalt, SIAP_SALT_SIZE)
kp.key ← pkey[0 .. SIAP_SERVER_KEY_SIZE - 1]
kp.keylen ← SIAP_SERVER_KEY_SIZE
kp.nonce ← pkey[SIAP_SERVER_KEY_SIZE + SIAP_NONCE_SIZE - 1]
kp.info ← NULL
kp.infolen ← 0
rstate ← zero_initialized_RCS_state()
// initialize RCS in decryption mode
RCS_Initialize(rstate, kp, false)
// authenticate and decrypt token tree into temporary buffer
res ← RCS_Transform(rstate, dect, dkey.ktree, SIAP_KTREE_SIZE)
if res = true then
    // copy plaintext token tree back into device key
    Copy(dkey.ktree, dect, SIAP_KTREE_SIZE)
end if
// securely clear sensitive material
Clear(dect, SIAP_KTREE_SIZE)
Clear(pkey, SIAP_SERVER_KEY_SIZE + SIAP_NONCE_SIZE)
RCS_Dispose(rstate)
return res
```

2.6.5 Device Authentication

The device authentication routine SIAP_SERVER_AUTHENTICATE_DEVICE drives the complete SIAP verification path on the server. It takes as input the device token buffer dtok, a mutable device key record dkey, the corresponding device tag dtag, the server key skey, and a freshly computed passphrase hash phash supplied by the user.

The function enforces identity and expiration checks, verifies the passphrase by constant time comparison, decrypts the device key using the stored passphrase hash from the tag, validates the integrity of the token tree, extracts and burns a device token, generates the expected server token, compares the two tokens, and on success regenerates the device tag and re encrypts the device key. All error conditions are mapped to explicit error codes and no state is updated on failure.

Algorithm 5 SIAP__SERVER__AUTHENTICATE__DEVICE

```
Require: dtok, dkey, dtag, skey, phash
Ensure: Authentication success or a specific error code
if CTEqual(dkey.kid, dtag.kid) = false
then return identity_mismatch
if Expired(dkey, skey) = true
then return key_expired
if CTEqual(dtag.phash, phash) = false
then return passphrase_unrecognized
if DecryptDeviceKey(dkey, skey,
dtag.phash) = false
then return decryption_failure
if VerifyDeviceTag(dtag, dkey) = false
then return token_tree_invalid
if ExtractDeviceToken(dtok, dkey) = false
then return token_invalid
if GenerateServerToken(stok, dtag, skey) = false
then return token_invalid
if CTEqual(dtok, stok) = false
then return authentication_failure
GenerateDeviceTag(dtag, dkey, phash)
EncryptDeviceKey(dkey, skey, phash)
return success
```

Algorithm 6 ExtractDeviceToken

```
Require: dtok, dkey
$idx \leftarrow$ ReadIndex(dkey.kid)
if $idx \geq$ KTREE_COUNT then
    return false
end if
Copy(dtok, dkey.ktree[idx])
Clear(dkey.ktree[idx])
IncrementIndex(dkey.kid)
return true
```

Algorithm 7 GenerateServerToken

```
Require: stok, dtag, skey
$idx \leftarrow$ ReadIndex(dtag.kid)
if $idx \geq$ KTREE_COUNT then
    return false
end if
stok ← cSHAKE(skey.kbase, config, dtag.kid)
return true
```

Algorithm 8 VerifyDeviceTag

```
Require: dtag, dkey
$h \leftarrow$ SHAKE(dkey.ktree)
return CTEqual($h$, dtag.khash)
```

2.6.6 Server Authentication Token Generation

The helper routine SIAP_SERVER_GENERATE_AUTHENTICATION_TOKEN derives the server side authentication token corresponding to the current device state. It reads the current key index from the device tag, checks that it is within range, then applies cSHAKE with the server base key, a fixed configuration string, and the full identity `dtag.kid` to derive the expected token. This function realizes the server view of the next valid token without accessing the device token tree.

Algorithm 9 SIAP_SERVER_GENERATE_AUTHENTICATION_TOKEN

```
Require: Output buffer token, device tag dtag, server key skey
Ensure: On success, token contains the expected authentication token and the function
    returns true
assert token, dtag, skey are not null
res ← false
if token ≠ null and dtag ≠ null and skey ≠ null then
                                              ▷ Read current key index from identity
    kidx ← BEBytesToUint32(dtag.kid + SIAP_DID_SIZE)
    if kidx < SIAP_KTREE_COUNT then
                                   ▷ Derive token from base key, config string, and identity
        token ← cSHAKE_AuthToken(skey.kbase, SIAP_CONFIG_STRING,
dtag.kid)
        res ← true
    end if
end if
return res
```

2.6.7 Device Token Extraction and Leaf Burn

The helper routine SIAP_SERVER_EXTRACT_AUTHENTICATION_TOKEN operates on the device key record. It reads the current index from the device identity, checks bounds, copies the token at that index from the token tree into the caller supplied buffer, then erases the token in place and increments the index field in `dkey.kid`. This function enforces the one time use and forward secrecy of the symmetric token tree.

Algorithm 10 SIAP_SERVER_EXTRACT_AUTHENTICATION_TOKEN

```
Require: Output buffer token, device key dkey, server key skey
Ensure: On success, token contains the device token at the current index, that token is
    erased from the tree, and the device index is incremented
assert token, dkey, skey are not null
res $\leftarrow$ false
if token $\neq$ null and dkey $\neq$ null and skey $\neq$ null then
                              ▷ Read the current key index from the device identity
    kidx $\leftarrow$ BEBytesToUint32(dkey.kid + SIAP_DID_SIZE)
    if kidx $<$ SIAP_KTREE_COUNT then
                                                          ▷ Copy token from the tree
        MemCopy(token, dkey.ktree + kidx $\cdot$ SIAP_TOKEN_SIZE,
SIAP_TOKEN_SIZE)
                                  ▷ Erase consumed token to enforce one time use
        Clear(dkey.ktree + kidx $\cdot$ SIAP_TOKEN_SIZE, SIAP_TOKEN_SIZE)
                                              ▷ Increment key index field in identity
        BEIncrement(dkey.kid + SIAP_DID_SIZE, SIAP_KEY_ID_SIZE)
        res $\leftarrow$ true
    end if
end if
return res
```

2.6.8 Device Tag Generation and Verification

The routine SIAP_SERVER_GENERATE_DEVICE_TAG constructs the server side tag from the current device state and the passphrase hash. It copies the identity `kid` from the device key, copies the passphrase hash, and computes a SHAKE based hash of the entire token tree into `dtag.khash`. This tag serves as the server commitment to the device state and is used to detect tampering or unintended modification of the token tree.

Algorithm 11 SIAP_SERVER_GENERATE_DEVICE_TAG

```
Require: Output tag dtag, device key dkey, passphrase hash phash
Ensure: dtag updated to reflect the current device state and passphrase hash
assert dtag, dkey, phash are not null
if dtag $\neq$ null and dkey $\neq$ null and phash $\neq$ null then
                                              ▷ Copy identity and passphrase hash
    MemCopy(dtag.kid, dkey.kid, SIAP_KID_SIZE)
    MemCopy(dtag.phash, phash, SIAP_HASH_SIZE)
                                               ▷ Compute hash of entire token tree
    tmph $\leftarrow$ SHAKE_TreeHash(dkey.ktree, SIAP_KTREE_SIZE)
    MemCopy(dtag.khash, tmph, SIAP_KTAG_STATE_HASH)
end if
```

The verification routine SIAP_SERVER_VERIFY_DEVICE_TAG recomputes the SHAKE based hash over the current device token tree and compares it in constant time with the stored tag hash `dtag.khash`. If the values differ, the token tree has been altered and authentication must fail. This check detects both accidental corruption and adversarial modification of the device key state.

Algorithm 12 SIAP__SERVER__VERIFY__DEVICE__TAG

```
Require: Device tag dtag, device key dkey
Ensure: Returns true if the tag matches the current device state, false otherwise
assert dtag, dkey are not null
res ← false
if dtag $\neq$ null and dkey $\neq$ null then
                                                        ▷ Recompute token tree hash
    tmph ← SHAKE_TreeHash(dkey.ktree, SIAP_KTREE_SIZE)
                                            ▷ Compare with stored hash in constant time
    res ← (CTEqual(tmph, dtag.khash, SIAP_KTAG_STATE_HASH) = true)
end if
return res
```

3 Formal Protocol Specification

This section formalizes the behavior of the Secure Infrastructure Access Protocol (SIAP) under a precise mathematical model. The definitions that follow use the implementation structures introduced in the engineering description, while abstracting away platform specific details. SIAP is modeled as a symmetric challenge response protocol that consumes one time tokens, enforces state monotonicity, and derives all encryption material from a hardened passphrase function and a server side master key.

3.1 Notation

We use the following notation throughout the formal specification.

- $X \parallel Y$ denotes byte string concatenation.
- $|X|$ denotes the bit length of the string X.
- Kid denotes a structured device identity string.
- idx denotes the index field inside Kid.
- $\mathcal{T}$ denotes the device token tree, an array of 1024 fixed width tokens.
- Erase(x) represents destructive overwriting of the value x.
- cSHAKE(K, S, C) denotes a cSHAKE call with key material K, customization string S, and input C.
- SCB(p) denotes the SCB hardened passphrase function applied to passphrase p.
- RCS_Enc and RCS_Dec represent encryption and decryption under the RCS authenticated encryption construction.
- SHAKE(X) denotes a SHAKE hash applied to a bitstring X.

All random coins are assumed to be sampled from a uniform distribution over the specified domain unless otherwise noted.

3.2 System Entities and Identities

The SIAP system contains three principal components: a server, a provisioned device, and a user who supplies a passphrase. Each entity maintains specific information relevant to authentication.

Server. The server maintains a master key

$$K_{\mathrm{srv}} = (kbase, sid, dsalt, ts_{exp})$$

where $kbase$ is a uniformly random secret, sid identifies the server instance, $dsalt =$ cSHAKE$(kbase, config, sid)$ is a diversification salt, and ts_{exp} is the lifetime of all derived material.

Device. A provisioned device contains an encrypted device key structure:

$$K_{\mathrm{dev}}^{seal} = \mathrm{RCS\_Enc}(rcskey, \mathcal{T}, \mathrm{Kid}, ts_{exp})$$

where $rcskey$ is derived from a hardened passphrase function, $\mathcal{T}$ is the token tree, and Kid contains the device identity DID and the index idx.

Device Tag. The server stores a device tag:

$$\mathrm{Tag} = (\mathrm{Kid}, khash, phash)$$

where $khash = \mathrm{SHAKE}(\mathcal{T})$ and $phash = \mathrm{SCB}(p)$ for the user supplied passphrase p.

Identity Structure. The device identity string is defined as:

$$\mathrm{Kid} = \mathrm{DID} \parallel idx$$

where DID encodes domain, group, server, user, and device fields and idx identifies the next unused token.

3.2.1 Identity Structure

The SIAP identity string is domain oriented and consists of a sequence of fixed width fields that encode the administrative context and the bound device. We model the structured device identifier as

$$\mathrm{DID} = D_{\mathrm{domain}} \parallel D_{\mathrm{group}} \parallel D_{\mathrm{server}} \parallel D_{\mathrm{user}} \parallel D_{\mathrm{device}},$$

where each component is a fixed length bitstring:

$$\begin{aligned} D_{\mathrm{domain}} &\in \{0,1\}^{\ell_{\mathrm{domain}}}, \\ D_{\mathrm{group}} &\in \{0,1\}^{\ell_{\mathrm{group}}}, \\ D_{\mathrm{server}} &\in \{0,1\}^{\ell_{\mathrm{server}}}, \\ D_{\mathrm{user}} &\in \{0,1\}^{\ell_{\mathrm{user}}}, \\ D_{\mathrm{device}} &\in \{0,1\}^{\ell_{\mathrm{device}}}. \end{aligned}$$

The total length of the device identifier is

$$|\mathrm{DID}| = \ell_{\mathrm{domain}} + \ell_{\mathrm{group}} + \ell_{\mathrm{server}} + \ell_{\mathrm{user}} + \ell_{\mathrm{device}},$$

which corresponds to the constant `SIAP_DID_SIZE` in the implementation. All fields are encoded as fixed width big endian integers in the reference code.

The full device key identifier used by SIAP is then defined as

$$\mathrm{Kid} = \mathrm{DID} \parallel idx,$$

where idx is a fixed width counter that encodes the position of the next unused token in the device token tree. In the implementation this counter has length

$$idx \in \{0,1\}^{\ell_{\text{idx}}},$$

with ℓ_{idx} corresponding to the constant `SIAP_KEY_ID_SIZE`.
The hierarchical structure of DID provides the following interpretation.

- D_{domain} identifies a logical administrative domain.
- D_{group} partitions services or roles within a domain.
- D_{server} identifies the SIAP server instance.
- D_{user} identifies the user account.
- D_{device} identifies the physical authentication device bound to that user.

Token derivation uses the structured identity as input to cSHAKE. For each index i in the token tree, the implementation and the formal model both use

$$\mathcal{T}[i] = \text{cSHAKE}(kbase, config, \text{DID} \parallel i),$$

and the server side verification token is computed from the full identity string

$$T_{\text{srv}} = \text{cSHAKE}(kbase, config, \text{Kid}).$$

This binds every token to its domain, group, server, user, device, and index, and ensures separation of token spaces across different identities.

3.3 Token Tree Construction

The token tree $\mathcal{T}$ is a fixed size array of one time tokens defined by

$$\mathcal{T}[i] = \text{cSHAKE}(kbase, config, \text{DID} \parallel i)$$

for each integer $0 \le i < 1024$. Let $\mathcal{T}$ denote the complete set of tokens

$$\mathcal{T} = \{\mathcal{T}[0], \mathcal{T}[1], \ldots, \mathcal{T}[1023]\}.$$

Each token is independent due to the domain separation implicit in the cSHAKE input. Random access to elements of $\mathcal{T}$ is possible without requiring sequential evaluation, which distinguishes SIAP from classical hash chain based OTP schemes.
A token consumed during authentication is immediately erased:

$$\text{Erase}(\mathcal{T}[idx]).$$

This operation provides symmetric forward secrecy since an adversary who compromises the device at a later time cannot reconstruct any previously consumed token.
The server derives the corresponding verification token through the same construction:

$$T_{srv} = \text{cSHAKE}(kbase, config, \text{Kid}).$$

3.4 Passphrase Processing and Encryption

The passphrase p supplied by the user is processed by the SCB hardened password function:

$$phash = \mathrm{SCB}(p).$$

The encryption key for the token tree is derived through cSHAKE by combining the hardened passphrase, the device identity, and the server diversification salt:

$$rcskey = \mathrm{cSHAKE}(phash, \mathrm{Kid}, dsalt).$$

The encrypted device key is defined as:

$$K_{\mathrm{dev}}^{seal} = \mathrm{RCS\_Enc}(rcskey, \mathcal{T}, \mathrm{Kid}, ts_{exp}).$$

The corresponding decryption operation used during authentication is:

$$\mathcal{T} = \mathrm{RCS\_Dec}(rcskey, K_{\mathrm{dev}}^{seal}).$$

The SCB function protects against offline brute force attacks on the device key, since the attacker must invert SCB before deriving $rcskey$.

3.5 Authentication Transcript

An authentication run consists of the following transcript inputs:

- encrypted device key K_{dev}^{seal},
- device tag $\mathrm{Tag} = (\mathrm{Kid}, khash, phash')$,
- SCB hardened passphrase $phash$,
- server key K_{srv}.

The authentication algorithm performs:

1. identity check: $\mathrm{Kid}_{dev} = \mathrm{Kid}_{tag}$,
2. expiration check: $ts_{exp}^{dev} \leq ts_{exp}^{srv}$,
3. passphrase verification: $phash = phash'$,
4. decryption of $\mathcal{T}$ using $rcskey$,
5. verification of $khash$,
6. extraction of $T_{dev} = \mathcal{T}[idx]$,
7. derivation of $T_{srv} = \mathrm{cSHAKE}(kbase, config, \mathrm{Kid})$,
8. equality check of T_{dev} and T_{srv},
9. erasure of $\mathcal{T}[idx]$,
10. increment of idx and update of Kid,
11. regeneration of $khash$ and re encryption of $\mathcal{T}$.

A successful authentication consumes exactly one token and establishes a new shared state.

3.6 State Evolution and Revocation

State evolves through a monotonic increase of the index field:

$$idx \leftarrow idx + 1.$$

The updated device identity becomes:

$$\mathrm{Kid}' = \mathrm{DID} \parallel (idx + 1).$$

The server stores the updated device tag:

$$\mathrm{Tag}' = (\mathrm{Kid}', \mathrm{SHAKE}(\mathcal{T}), phash).$$

The device stores the new sealed device key:

$$K_{\mathrm{dev}}^{seal'} = \mathrm{RCS\_Enc}(rcskey, \mathcal{T}, \mathrm{Kid}', ts_{exp}).$$

A device or account is revoked by removing the associated tag. Since no other secret beyond *kbase* is required for token derivation, revocation does not affect other devices or users.
If an adversary obtains the device key after *idx* has advanced, forward secrecy ensures that previously consumed tokens cannot be recovered, since the erased elements of $\mathcal{T}$ are no longer present.

4 Security Model and Assumptions

This section defines the formal model under which SIAP is analyzed. The model captures the adversarial capabilities, the behavior of the interacting entities, and the assumptions on which the security of the protocol depends. The goal is to provide a precise foundation for the authentication, forward secrecy, replay resistance, and offline attack analyses developed in later sections.

4.1 Threat Model

SIAP is designed for settings where devices are frequently offline or air gapped, and where authentication must rely exclusively on symmetric operations and deterministic state progression. The adversary is modeled as a probabilistic polynomial time algorithm with the following capabilities.

- The adversary may observe any number of authentication attempts between a device and the server.
- The adversary may obtain the server side device tag, which contains Kid, *khash*, and *phash*.
- The adversary may attempt to compromise the device and obtain the sealed device key K_{dev}^{seal} at any point in time.
- The adversary may attempt to guess the user passphrase offline.
- The adversary may initiate chosen input queries in the authentication game, subject to state progression rules.

The adversary is not assumed to observe internal memory of the server or obtain *kbase*, which serves as the root of trust for all token derivation operations. Side channel attacks beyond timing behavior are out of scope.

4.2 Cryptographic Primitives

SIAP relies on the following well defined primitives.

SCB Hardened Passphrase Function. SCB is a memory hard password hashing function with adjustable memory and iteration parameters. It provides protection against offline brute force attacks by forcing an adversary to invest a significant cost for each guess. The SCB output is denoted

$$phash = \mathrm{SCB}(p).$$

cSHAKE Derivation Function. cSHAKE is used as a generalized pseudo-random function. It derives token values, diversification salts, and encryption keys. It is modeled as a pseudo-random function in the sense that

$$\mathrm{cSHAKE}(K, S, X)$$

is indistinguishable from a random function when the key material K is unknown to the adversary.

RCS Authenticated Encryption. RCS provides confidentiality and integrity for the sealed device key. It is modeled as an IND CCA secure authenticated encryption construction.

SHAKE Hash. SHAKE is used to compute the tree hash

$$khash = \mathrm{SHAKE}(\mathcal{T})$$

which binds the server side tag to the contents of the device token tree.

4.3 Security Goals

SIAP aims to achieve the following security properties.

Correctness. If all protocol participants follow the specification and the passphrase is correct, then authentication succeeds.

Authentication Soundness. An adversary should not be able to impersonate a device or a user without either:

- knowledge of the server master key $kbase$,
- knowledge of the hardened passphrase hash $phash$,
- access to an unconsumed token in $\mathcal{T}$.

Replay and Reuse Resistance. Tokens are single use. Any attempt to reuse a previously consumed token must fail.

Symmetric Forward Secrecy. If the adversary compromises the device at time t, then previously consumed tokens cannot be recovered. Formally, if $\mathcal{T}[i]$ has been erased, then the adversary cannot compute $\mathcal{T}[i]$ even with the full state at time t.

Offline Attack Resistance. The adversary should not be able to invert the passphrase from the sealed device key. The memory hardness of SCB ensures that exhaustive guessing remains infeasible.

Server Database Safety. The server side tag contains no information that allows derivation of future tokens or decryption of the device key.

4.4 Adversarial Capabilities

We model the following classes of adversarial interaction.

Observation Oracle. The adversary may observe interactions between the device and the server but does not gain access to ephemeral secrets.

Tag Oracle. The adversary may obtain the device tag for any enrolled identity. This represents a compromise of server side storage.

Sealed Device Key Oracle. The adversary may obtain the sealed device key at any time. This models physical compromise of the device.

Offline Passphrase Attack. Given K_{dev}^{seal}, the adversary may perform offline guesses of the passphrase p by running SCB. The cost of each guess is therefore proportional to the SCB memory and iteration settings.

Desynchronization Attempts. The adversary may attempt to manipulate the index field of the device identity to cause divergence between server and device state. SIAP enforces strict monotonicity of idx, so any stale or advanced index results in rejection.

Forward Secrecy Attack. The adversary may attempt to recover consumed tokens from the sealed device key after compromise.
These models allow the protocol to be analyzed under strong, realistic threat scenarios.

4.5 Assumptions and Trust Anchors

The security arguments for SIAP rely on the following assumptions.

Master Key Secrecy. The value *kbase* remains secret. It enables derivation of both token values and server side verification tokens.

Correctness of SCB. The SCB function is memory hard and resists practical inversion. The cost of offline guessing grows linearly in the memory parameter.

pseudo-randomness of cSHAKE. The cSHAKE based derivation function behaves as a pseudo-random function when keyed with secret material.

Security of RCS. The RCS authenticated encryption scheme provides IND CCA security under derived keys.

Strong Erasure. The device reliably erases consumed tokens. Once Erase($\mathcal{T}[idx]$) is performed, the corresponding token cannot be recovered.

Monotonic State Advancement. Both device and server enforce monotonic progression of the index field. This prevents replay, reuse, or desynchronization attacks.

Integrity of Server Storage. Server side state is assumed to support tamper evident storage of Tag entries, although the confidentiality of Tag is not required.

These assumptions form the trust anchors of the SIAP security model and provide the basis for the formal analysis in the next sections.

5 Security Definitions

This section defines the formal security properties of SIAP using a game based framework. The games model the ability of an adversary to impersonate a device or server, reuse consumed tokens, recover confidential information, or guess the passphrase offline. Each definition is aligned with the protocol behavior described in the engineering and formal specification sections.

5.1 Authentication Security Experiment

The purpose of the authentication experiment is to model an adversary who attempts to authenticate as a legitimate device without access to an unconsumed token.

Experiment $\mathsf{AuthExp}$ The challenger performs:

1. Generates K_{srv}.
2. Generates a device identity Kid with index $idx = 0$.
3. Generates a token tree $\mathcal{T}$ from K_{srv}.
4. Samples a passphrase p and computes $phash = \text{SCB}(p)$.
5. Forms the sealed device key K_{dev}^{seal} and device tag Tag.

The adversary is given $(\text{Tag}, K_{\text{dev}}^{seal})$ and oracle access to the following interfaces:

- $\mathsf{Observe}$: returns transcripts of honest authentication runs without revealing tokens.
- $\mathsf{SealLeak}$: returns K_{dev}^{seal} at any time.

The adversary succeeds if it outputs a tuple (Kid^*, T^*) such that the server would accept T^* as the next valid token for the identity Kid^* under the protocol rules.

$$\text{Adv}_{\mathsf{Auth}}(A) = \Pr[A \text{ wins } \mathsf{AuthExp}].$$

A protocol is authentication secure if $\text{Adv}_{\mathsf{Auth}}(A)$ is negligible for all polynomial time adversaries.

5.2 Replay and Reuse Resistance

Tokens in SIAP are single use. Any reuse of previously consumed tokens must be rejected.

Experiment $\mathsf{ReplayExp}$ The challenger produces $(K_{\text{srv}}, \text{Kid}, \mathcal{T}, \text{Tag})$ and simulates one honest authentication run, consuming token $\mathcal{T}[idx]$.

The adversary is given:

- the consumed token $T_{old} = \mathcal{T}[idx]$,
- the updated tag with index $idx + 1$.

The adversary wins if it produces an authentication attempt accepted by the server using T_{old} or any value derived from T_{old}.

$$\text{Adv}_{\mathsf{Replay}}(A) = \Pr[A \text{ wins } \mathsf{ReplayExp}].$$

Replay security requires $\text{Adv}_{\mathsf{Replay}}(A)$ to be negligible.

5.3 Token Confidentiality

Token confidentiality ensures that an adversary who obtains the sealed device key cannot learn any unconsumed tokens without inverting SCB or breaking RCS.

Experiment $\mathsf{TokenConfExp}$ The challenger:

1. chooses a random index i,
2. samples the token tree $\mathcal{T}$,
3. encrypts it into K_{dev}^{seal} using the derived RCS key,
4. provides $(K_{\text{dev}}^{seal}, \text{Kid}, \text{Tag})$ to the adversary.

The adversary outputs a guess T^* for the hidden token $\mathcal{T}[i]$.

$$\text{Adv}_{\mathsf{Conf}}(A) = \Pr[T^* = \mathcal{T}[i]].$$

Token confidentiality holds when the adversary cannot distinguish $\mathcal{T}[i]$ from a random string without breaking SCB or RCS.

5.4 Server Impersonation Game

The server impersonation game models an adversary who attempts to trick a device into accepting a forged authentication response.
Since SIAP authenticates in one direction only, server impersonation corresponds to generating a fake verification token.

Experiment $\mathsf{SrvImpExp}$ The challenger:

1. generates K_{srv} and $\mathcal{T}$,
2. provides the device identity Kid and index idx to the adversary.

The adversary outputs a value T_{srv}^*.
The adversary wins if:

$$T_{srv}^* = \text{cSHAKE}(kbase, config, \text{Kid}).$$

$$\text{Adv}_{\mathsf{SrvImp}}(A) = \Pr[A \text{ wins } \mathsf{SrvImpExp}].$$

This advantage is negligible when cSHAKE behaves as a pseudo-random function under the secret $kbase$.

5.5 Client Impersonation Game

The client impersonation game models an adversary who attempts to impersonate the device to the server.

Experiment $\mathsf{CliImpExp}$ The challenger:

1. generates $(K_{\text{srv}}, \mathcal{T}, \text{Kid}, \text{Tag})$,
2. erases $\mathcal{T}[idx]$ after performing one authentication.

The adversary is given $(\text{Tag}, K_{\text{dev}}^{seal})$.
The adversary wins if it outputs a token T^* accepted by the server at index $idx + 1$.

$$\text{Adv}_{\mathsf{CliImp}}(A) = \Pr[A \text{ wins } \mathsf{CliImpExp}].$$

Client impersonation resistance follows from the pseudo-randomness of the cSHAKE derived token tree.

5.6 Password Guessing Advantage

The password guessing game models an offline attacker who obtains the sealed device key and attempts to recover the user passphrase.

Experiment $\mathsf{PwdGuessExp}$ The challenger:

1. samples a random passphrase p,
2. computes $phash = \mathrm{SCB}(p)$,
3. generates K_{dev}^{seal} using the derived RCS key,
4. provides K_{dev}^{seal} to the adversary.

The adversary may compute a polynomially bounded number of SCB evaluations with chosen candidate passphrases. It wins if it outputs p^* such that $\mathrm{SCB}(p^*) = phash$.

$$\mathrm{Adv}_{\mathsf{PwdGuess}}(A) = \Pr[A \text{ wins } \mathsf{PwdGuessExp}].$$

Since SCB is memory hard, the adversary must spend at least M units of memory per guess, where M is the SCB parameter. Offline attack resistance requires that the cost of q SCB evaluations is computationally infeasible.

6 Provable Security Analysis

This section analyzes the security of SIAP by relating each security goal to the advantage of an adversary in the corresponding game defined in Section 5. Each subsection provides a reduction style argument that shows how an adversary breaking SIAP can be transformed into an adversary that breaks one of the underlying cryptographic primitives. The analysis is based on the formal specification of SIAP and on the exact engineering behavior described in the implementation.

6.1 Correctness of SIAP

SIAP is correct if an honest device using a correct passphrase always authenticates successfully with an honest server. Correctness follows directly from the construction.

- The server verifies identity matching. For an honest device, $\mathrm{Kid}_{dev} = \mathrm{Kid}_{tag}$.
- The expiration timestamps match since both are set during enrollment.
- The hardened passphrase hash $phash$ computed by the device matches the stored $phash'$.
- The server derives the same RCS decryption key and nonce from $(phash, \mathrm{Kid}, dsalt)$ that were used during enrollment.
- The RCS decryption succeeds and yields the correct token tree $\mathcal{T}$.
- The recomputed tree hash matches the tag hash.
- The server derives the same token $\mathcal{T}[idx]$ that the device derived during enrollment.

Thus the acceptance condition is always met for honest participants, and SIAP satisfies correctness.

6.2 Resistance to Replay and Reuse

Replay resistance requires that a consumed token cannot be used again. This follows from destructive erasure and the strict monotonicity of the index field.
Let $T_{old} = \mathcal{T}[idx]$ be a token used in a successful authentication. After the authentication:

$$\text{Erase}(\mathcal{T}[idx]) \quad \text{and} \quad idx \leftarrow idx + 1.$$

An adversary receives T_{old} in the transcript but cannot use it again, since the server expects the token at index $idx + 1$. The token $\mathcal{T}[idx + 1]$ is independent of $\mathcal{T}[idx]$ due to the pseudo-randomness of cSHAKE.
If an adversary succeeds in ReplayExp, then it can predict the output of cSHAKE at a new input without knowledge of *kbase*. This contradicts the pseudo-random function assumption on cSHAKE.
Thus,

$$\text{Adv}_{\mathsf{Replay}}(A) \leq \text{Adv}_{\text{PRF}}^{\text{cSHAKE}}(B)$$

for a suitable reduction B, and is therefore negligible.

6.3 Impersonation Resistance

SIAP must resist both server impersonation and client impersonation. We treat the two cases separately.

Server Impersonation. The adversary must compute

$$T_{srv} = \text{cSHAKE}(kbase, config, \text{Kid})$$

without knowledge of *kbase*. This is equivalent to computing a pseudo-random function output at a secret input. If an adversary wins SrvImpExp, then it can be used as a subroutine to distinguish cSHAKE from a random function.
Thus,

$$\text{Adv}_{\mathsf{SrvImp}}(A) \leq \text{Adv}_{\text{PRF}}^{\text{cSHAKE}}(B).$$

Client Impersonation. The adversary must produce a valid unconsumed token $\mathcal{T}[idx]$ or its successor without access to *kbase*. If the adversary wins CliImpExp, it can output the value of

$$\text{cSHAKE}(kbase, config, \text{Kid}),$$

which again contradicts the pseudo-randomness of cSHAKE.
Thus,

$$\text{Adv}_{\mathsf{CliImp}}(A) \leq \text{Adv}_{\text{PRF}}^{\text{cSHAKE}}(B).$$

Both impersonation attacks reduce directly to breaking cSHAKE.

6.4 Passphrase Guessing Bounds

Offline passphrase attacks require inverting the SCB hardened password function. SCB forces every evaluation to use a fixed memory cost M and a fixed iteration count ℓ. An adversary that obtains K_{dev}^{seal} must compute SCB for each candidate passphrase.
Let q be the number of guesses the adversary makes. If the adversary wins PwdGuessExp, then it has produced p^* such that

$$\text{SCB}(p^*) = phash.$$

The computational cost of q guesses is at least

$$q \cdot \mathrm{Cost}_{\mathrm{SCB}}(M, \ell),$$

which grows linearly in q and is dominated by the required memory footprint.
Thus,

$$\mathrm{Adv}_{\mathsf{PwdGuess}}(A) \leq q \cdot 2^{-\lambda}$$

where λ is the effective hardness parameter derived from the SCB configuration.
Therefore SIAP inherits its offline guess resistance directly from the hardness of SCB.

6.5 Composition with Underlying Primitives

The security of SIAP results from the composition of three independent cryptographic assumptions:

- cSHAKE behaves as a pseudo-random function when keyed with unknown material,
- RCS provides IND CCA authenticated encryption,
- SCB resists offline inversion due to memory hardness.

We claim that SIAP remains secure under the simultaneous satisfaction of these assumptions. Formally, for every adversary A attacking SIAP, there exist adversaries B_1, B_2, and B_3 such that

$$\mathrm{Adv}_{\mathsf{SIAP}}(A) \leq \mathrm{Adv}^{\mathrm{cSHAKE}}_{\mathrm{PRF}}(B_1) + \mathrm{Adv}^{\mathrm{RCS}}_{\mathrm{IND\text{-}CCA}}(B_2) + \mathrm{Adv}_{\mathrm{SCB}}(B_3).$$

The correctness of this bound follows from examining each attack surface:

- any impersonation attack requires predicting a cSHAKE output,
- any token recovery attack requires decrypting the sealed device key, which requires breaking RCS or SCB,
- any offline passphrase attack requires inverting SCB.

Since each advantage term is negligible under its respective assumption, the combined advantage is negligible. SIAP therefore achieves authentication soundness, replay resistance, token confidentiality, and offline attack resistance under the stated model.

7 Cryptanalytic Evaluation

This section evaluates SIAP from a cryptanalytic perspective, focusing on the separation of key material, resistance to state exposure, protection against online and offline attacks, and the consequences of post compromise access. The analysis is based on both the formal model developed in earlier sections and the precise behavior of the implementation.

7.1 Token Derivation and Key Separation

The SIAP token tree is defined by

$$\mathcal{T}[i] = \mathrm{cSHAKE}(kbase, config, \mathrm{DID} \parallel i).$$

Each token is derived from the master key $kbase$ under domain separated inputs. This provides the following separation properties.

Random Access Derivation. Unlike Lamport chains, where tokens are generated through sequential hashing, SIAP supports random access. The server may compute $\mathcal{T}[i]$ directly without computing any earlier or later tokens. This avoids the linear overhead of hash chains and prevents the adversary from learning structural information when observing token values.

Strict Key Separation. SIAP uses independent uses of cSHAKE for:

- token derivation,
- server side diversification salt,
- RCS encryption key derivation.

These uses are separated by distinct inputs and customization strings. Leakage of any derived value does not compromise the others.

Independence of Tokens. Because cSHAKE behaves as a pseudo-random function, tokens are independent. Knowledge of $\mathcal{T}[i]$ gives the adversary no information about $\mathcal{T}[j]$ for $i \neq j$. This is a strict improvement over HOTP, where predictable counters can reveal structural relationships between tokens or permit selective brute force.

7.2 Tree Exposure and Partial Compromise

The most important question in symmetric OTP systems is whether partial exposure of the secret allows an adversary to derive future or past tokens.

Exposure of the Token Tree. If the adversary compromises the device before authentication, they may obtain $\mathcal{T}$ in decrypted form. The consequences are limited.

- The adversary can impersonate the device until the token tree is exhausted, but cannot derive future tokens beyond the final index because these depend on $kbase$.
- The adversary gains no information about $kbase$.
- The adversary cannot impersonate the server because server tokens depend on $kbase$ alone.

This behavior mirrors the partial compromise model of forward secure MACs, where future keys are safe despite exposure of early material.

Integrity Under Partial Compromise. Even if an adversary observes $\mathcal{T}[i]$, they cannot modify the sealed structure without breaking RCS. Any tampering is detected by the AEAD tag.

Comparison with Lamport Hash Chains. In a classical S Key scheme, a single leak of an intermediate value exposes all future tokens. SIAP avoids this problem, since $\mathcal{T}[i]$ does not permit computation of $\mathcal{T}[i+1]$.

7.3 Server Database Leakage

The server stores only:

$$\mathrm{Tag} = (\mathrm{Kid}, khash, phash).$$

Leakage of Kid**.** The identity string contains no secrets. Leakage provides no computational advantage.

Leakage of *khash*. The tree hash is a one way SHAKE digest of $\mathcal{T}$. Recovering any token from *khash* is as hard as inverting SHAKE. The server learns no token material.

Leakage of *phash*. The hardened passphrase hash *phash* is derived by SCB. Knowledge of phash alone is insufficient to derive the RCS key, since pkey includes skey.kbase, which never leaves the server. Since SCB is memory hard, exhaustive guessing is expensive.

No Server Side Secret Except *kbase*. Server database compromise does not reveal *kbase*, the only value from which new tokens can be derived. This sharply contrasts with HOTP or TOTP deployments, where a leaked server key compromises all associated devices.

7.4 Online and Offline Attack Surfaces

SIAP separates online and offline attacks and designs its primitive composition to protect against both.

Online Attacks. An online attacker attempts authentication queries.

- A wrong token is rejected immediately and increments no state.
- No oracle leaks token prefixes or approximate matches.
- The index value is verified strictly, so incorrect index manipulation results in permanent rejection.

Online forgery requires breaking cSHAKE as a pseudo-random function.

Offline Attacks. The primary offline attack surface is recovery of the passphrase through K_{dev}^{seal}.

An adversary must:

1. guess a passphrase p^*,
2. compute $phash^* = \text{SCB}(p^*)$,
3. derive $rcskey^* = \text{cSHAKE}(phash^*, \text{Kid}, dsalt)$,
4. decrypt K_{dev}^{seal} to test correctness.

Because SCB has memory cost M, each guess requires at least M bytes of memory and a fixed time proportional to the configured iteration count. This prevents large scale GPU based brute force, improving on classical OTP cards that use only PBKDF2 or SHA based derivation.

Comparison with YubiKey AES OTP. Devices that store AES keys allow full offline brute force of the PIN or password. SIAP uses SCB to enforce adversarial cost.

7.5 Post Compromise Scenarios

This subsection analyzes attacks where the adversary compromises the device at time t and obtains K_{dev}^{seal} or the decrypted $\mathcal{T}$.

Case 1: Compromise Before Authentication. If the adversary steals the card before an authentication run:

- they obtain access to all unconsumed tokens,
- they cannot derive future tokens,
- they cannot derive the verification tokens used by the server,
- they cannot break $kbase$.

The adversary obtains the same power as in HOTP if the shared secret is leaked, but is strictly weaker since SIAP's server verification path remains uncompromised.

Case 2: Compromise After Authentication (Forward Secrecy). If a token has already been used and erased:

$$\text{Erase}(\mathcal{T}[idx]),$$

then compromising the device at time t yields no information about past tokens. The adversary cannot reconstruct $\mathcal{T}[idx]$ without inverting cSHAKE or SHAKE.
This provides symmetric forward secrecy, a property usually associated with Diffie Hellman based protocols, not symmetric OTP systems.

Case 3: Compromise of Server Database. Server compromise yields only $(\text{Kid}, khash, phash)$.

- No unconsumed token is revealed.
- No past or future token is revealed.
- Offline guessing of $phash$ is expensive due to SCB.

Case 4: Complete Token Tree Exposure. If the adversary obtains the full tree $\mathcal{T}$:

- future tokens remain secure since they depend on $kbase$ and higher indices,
- server impersonation remains impossible since it requires $kbase$,
- past tokens remain unrecoverable if they were erased.

This is a strictly stronger guarantee than classical HOTP systems.
Overall, SIAP remains secure under strong compromise conditions as long as $kbase$ remains secret and SCB parameters prevent practical offline inversion.

8 Implementation Alignment

This section evaluates the degree to which the SIAP specification and reference implementation match the formal protocol behavior described in earlier sections. The analysis focuses on structural alignment, correctness of derivation paths, constant time behavior, and the handling of state progression and error conditions.

8.1 Alignment with the SIAP Specification

The implementation aligns closely with the normative SIAP specification. All primary derivation paths are consistent with the specification text.

Identity Structure. The specification defines an identity string

$$\mathrm{Kid} = \mathrm{DID} \parallel idx,$$

and the implementation encodes these fields in fixed width segments. The identity comparison in the authentication path is exact and matches the specification.

Token Tree Generation. The specification states that tokens are derived through cSHAKE keyed by *kbase* using identity material and index. The implementation uses

$$\mathcal{T}[i] = \mathrm{cSHAKE}(kbase, config, \mathrm{DID} \parallel i)$$

with an incrementing index embedded directly in the identity structure. This matches the specification's informal description of a deterministic token hierarchy.

Passphrase Hardening. The specification defines a hardened password function. The implementation instantiates this function as SCB, which enforces a memory hard cost profile. The use of the SCB output for both verification and encryption aligns with the documented workflow.

Encryption of Device Key Material. The specification states that the device token tree is encrypted using a key derived from passphrase material. The implementation uses

$$rcskey = \mathrm{cSHAKE}(phash, \mathrm{Kid}, dsalt)$$

with RCS authenticated encryption. This behavior matches the specification while providing stronger security than the abstract description.

State Evolution. The monotonic increase of *idx* and erasure of consumed tokens are correctly implemented. The server and device states evolve exactly as defined in the specification.
Overall, the implementation is consistent with the specification, and any abstract portions of the specification correspond directly to concrete behaviors in the implementation.

8.2 Alignment with the Reference Implementation

The formal model captures the exact logic implemented in the SIAP source code.

Construction of the Token Tree. The implementation uses a loop over 1024 indices and fills $\mathcal{T}[i]$ using cSHAKE. This matches the formal definition and supports random access without sequential dependency.

Derivation of Diversification Salt. The implementation derives

$$dsalt = \mathrm{cSHAKE}(kbase, config, sid)$$

as a server wide salt. The formal specification uses the same definition.

Derivation of RCS Key Material. The decryption and encryption routines use cSHAKE with inputs $(phash, \mathrm{Kid}, dsalt)$. The formal model mirrors this exactly.

Verification of Tree Integrity. The reference implementation recomputes the tree hash with SHAKE and rejects mismatches. This behavior is reflected directly in the formal definition of the authentication transcript.

Index Management and Token Erasure. Upon authentication, the implementation reads $\mathcal{T}[idx]$, overwrites it, and increments idx. This sequence is faithfully represented in the formal specification and impacts all security definitions related to replay and forward secrecy.

Error Conditions. The implementation defines specific error codes for identity mismatch, passphrase mismatch, decryption failure, hash mismatch, and token mismatch. These conditions align exactly with the correctness and safety properties analyzed earlier.

8.3 Constant Time and Side Channel Considerations

The implementation includes several protections against timing and side channel attacks.

Constant Time Comparisons. The reference code uses constant time comparison routines for:

- checking passphrase hashes,
- verifying token equality,
- verifying the tree hash.

This prevents leakage of partial equality information and aligns with best practices for symmetric authentication systems.

RCS Behavior. The RCS authenticated encryption routines are constant time with respect to key dependent operations and avoid data dependent branching. This prevents leakage of token tree structure or identity material during encryption or decryption.

SCB Passphrase Hardening. SCB is intentionally CPU and memory expensive. Its evaluation time is dominated by fixed memory accesses and iteration counts, which reduces the risk of microarchitectural information leakage.

Identity Checks. The comparison of Kid fields does not take place in constant time in the reference implementation. Although the identity string contains no secrets, the formal analysis assumes that only non secret fields may leak through timing behavior. This assumption is valid for SIAP since Kid contains no confidential material.

8.4 State Synchronization and Failure Modes

SIAP has strict state synchronization requirements because the index field identifies the next unused token and must remain synchronized between device and server.

Strict Monotonicity. Both sides enforce the rule that idx must advance by exactly one per successful authentication. Any deviation results in rejection.

No Look Ahead Window. Unlike HOTP, SIAP does not implement a look ahead window. The server never searches ahead for future tokens.

This design:

- improves security by eliminating the acceptance window,
- increases the risk of desynchronization if the device advances state without server confirmation.

Desynchronization Behavior. If the device increments *idx* without the updated tag being stored on the server, the next authentication attempt will fail. Since SIAP preserves complete state on both sides after each operation, recovery requires:

- rolling back the device using a stored copy of K_{dev}^{seal}, or
- regenerating the server side tag from a trusted backup.

Failure Modes. The implementation aborts authentication under the following conditions:

- identity mismatch,
- expired timestamp,
- incorrect passphrase,
- decryption failure,
- tree hash mismatch,
- token mismatch.

Each failure preserves security by preventing advancement of shared state. No partial updates occur on failure, preventing divergence due to incomplete state modification.

In summary, the implementation enforces strong synchronization rules and avoids implicit recovery mechanisms. This simplifies the security model while placing operational responsibility on controlled state management.

9 Parameters and Performance

This section summarizes the key parameter choices in SIAP, evaluates their impact on computational cost and scalability, and describes the operational constraints that arise from the design of the protocol. The analysis reflects the behavior of the reference implementation while presenting performance characteristics at an abstract level suitable for formal evaluation.

9.1 Parameter Choices

SIAP relies on a fixed collection of cryptographic and structural parameters. These values are selected to balance security, performance, and device level constraints.

Token Tree Size. The token tree contains 1024 entries. This provides a fixed number of authentication attempts before the device must be reissued or re enrolled. The choice of 1024 balances storage requirements with operational longevity. Since each authentication consumes exactly one token, the number of permitted sessions per device is bounded by this value.

Key Lengths. The reference implementation uses 256 bit and 512 bit variants of RCS. The cSHAKE keyed derivation uses entropy from *kbase* of corresponding size. These lengths match common post quantum secure parameter recommendations for symmetric primitives.

Passphrase Hardening Parameters. The SCB hardened password function is configured with memory and iteration parameters (M, ℓ) chosen to make each evaluation expensive on commodity hardware. These parameters determine the cost of offline guessing attacks and are selected based on the threat model rather than device capacity.

Hash and PRF Output Sizes. SHAKE and cSHAKE outputs are configured to produce token sizes of either 32 or 64 bytes depending on the security profile. Larger outputs increase resistance to token collisions and improve the difficulty of token forgery.

Expiration Timestamp. The server key and device key share a common expiration timestamp ts_{exp}. The duration is chosen according to the operational environment and does not affect the computational cost of the protocol.

9.2 Complexity and Scaling

The computational cost of SIAP is determined primarily by cSHAKE, SCB, and the RCS authenticated encryption algorithm.

Token Derivation Cost. Token derivation requires a single cSHAKE evaluation. This is efficient relative to Lamport chains, which require sequential hashes to reach position i in the chain. SIAP supports random access without linear cost growth.

Encryption and Decryption. One RCS encryption and one RCS decryption occur per authentication. The cost is linear in the size of the token tree and the associated tag. Performance is dominated by memory movement rather than cryptographic operations. SIAP is therefore suitable for devices with limited compute capacity but reasonable memory bandwidth.

Passphrase Hardening Cost. SCB dominates the cost of authentication on low power devices. Each authentication evaluates SCB once. The cost scales with the memory parameter M and the iteration count. Increasing M raises the security margin against offline attacks but also increases the latency of authentication operations.

State Update Cost. State updates require:

- erasing exactly one token,
- recomputing the tree hash,
- re encrypting the updated device key.

The SHAKE based tree hash requires one pass over $\mathcal{T}$. Although this is linear in the tree size, it is efficient in practice since each token is small and the tree fits easily in memory.

Scalability. SIAP scales well with the number of devices and users because:

- no server side secret other than $kbase$ is required,
- all per device operations are independent,
- server side storage contains only non sensitive tags.

Authentication cost is constant for each device and does not depend on the number of enrolled devices.

9.3 Operational Limits

SIAP exhibits several operational constraints that arise from its design.

Finite Token Supply. Each device supports exactly 1024 authentications. After all tokens are consumed, the device must be reprovisioned. This is a design choice that trades convenience for strong symmetric forward secrecy.

No Look Ahead Window. SIAP does not support a look ahead window for token acceptance. The server accepts exactly the token at index idx. Any deviation results in failure. This simplifies formal analysis and improves security but increases operational sensitivity to state divergence.

Synchronization Requirements. State must be stored reliably on both device and server after each authentication. Temporary loss of state may result in permanent desynchronization. The implementation guarantees that failures do not partially update state, so divergence can only occur through external faults.

Cost of Passphrase Entry. Authentication latency is dominated by SCB. Environments where extremely low latency is required may choose smaller SCB parameters, which reduce security margins. SIAP is best suited to administrative or infrastructure contexts where a slight delay is acceptable.

Device Memory Requirements. The token tree must be decrypted into memory for authentication. Devices must support memory sufficient for the tree size, temporary workspace for SCB, and enough overhead for RCS operations.

In summary, SIAP achieves predictable and bounded performance by using fixed size state structures and symmetric primitives. The cost profile is dominated by SCB and linear scans over the token tree. These characteristics make SIAP suitable for constrained environments where strong symmetric security is required and where predictable behavior is valued over unbounded longevity.

10 Limitations and Future Work

This section summarizes the inherent limitations of the SIAP design and outlines possible extensions and directions for future analysis. Although SIAP satisfies its intended goals in offline and infrastructure controlled environments, its structural properties introduce constraints that are important to acknowledge for practical deployment and for further development of symmetric authentication protocols.

10.1 Modeling Limitations

The formal model presented in this paper captures the essential security properties of SIAP, but it abstracts away several practical concerns.

State Synchronization Assumptions. The model assumes that both server and device maintain perfectly consistent state. SIAP enforces strict monotonicity of the index field and provides no internal mechanisms for recovery from divergence. Real deployments may experience interruptions, power failures, or partial writes that lead to desynchronization. These events are not modeled in the security games.

Abstract Treatment of SCB. The SCB hardened passphrase function is modeled as a memory hard one way function. The analysis does not account for side channel leakage inside SCB, hardware specific behavior, or adversarially induced resource contention. These considerations fall outside the scope of the formal model.

Absence of Multi Factor Composition. The analysis treats SIAP as a two factor mechanism that combines a physical device and a passphrase. Interactions with external authentication systems or broader trust frameworks are not modeled.

No Network or Transport Layer Considerations. SIAP is designed for offline or air gapped environments. The formal model does not incorporate message delivery failures, partial authentication transcripts, or concurrent authentication sessions.
These limitations suggest possible refinements of the model for environments with more complex operational constraints.

10.2 Potential Extensions

Several extensions may strengthen SIAP or broaden its applicability.

Resynchronization Mechanisms. SIAP intentionally avoids look ahead windows to preserve tight security bounds. Introducing a controlled resynchronization mechanism may improve usability while retaining security. Possibilities include a small bounded acceptance window or a separate resynchronization protocol that requires explicit user action.

Adaptive Token Trees. The fixed number of tokens limits the lifetime of a device. Adaptive or renewable token trees, derived from additional layers of cSHAKE and identity material, could extend device longevity without sacrificing forward secrecy.

Hierarchical Identity Structures. The identity structure could be extended to support delegation or role based authentication. This may allow a single device to represent multiple operational contexts while maintaining separation of privileges.

Integration with Asymmetric Authentication. Although SIAP is symmetric, it may be composed with asymmetric schemes, such as FIDO2 or EdDSA, to provide hybrid authentication modes. Such integration can combine symmetric forward secrecy with asymmetric non repudiation and audit guarantees.

Hardware Acceleration. Devices with hardware support for SHAKE and memory hardened KDFs may execute SCB and cSHAKE more efficiently. This could reduce authentication latency and improve performance on constrained devices.

10.3 Directions for Further Analysis

Several research directions follow naturally from the structure of SIAP.

Tighter Bounds for Forward Secrecy. The current analysis provides qualitative and game based guarantees for symmetric forward secrecy. Deriving tighter quantitative bounds, especially under partial compromise conditions, remains an open problem.

Formal Analysis of SCB Parameters. The offline guessing resistance of SIAP depends heavily on the SCB configuration. A deeper study of optimal memory and iteration parameters for various adversarial models would strengthen the theoretical foundation of SIAP.

Comparison with Standard OTP Schemes. Future work may include a more detailed quantitative comparison with HOTP, TOTP, S Key, and FIDO based authentication. This includes token exhaustion rates, attack surfaces, and resilience under large scale compromise.

Robustness Under Faulty Storage. The strict synchronization requirement raises questions about the robustness of SIAP under non ideal storage behavior. Modeling partial writes, corrupted state, or transactional rollback may reveal opportunities for improved resilience.

Post Quantum Considerations. SIAP relies exclusively on symmetric primitives that are believed to resist quantum attacks. A formal treatment of the quantum query model for cSHAKE and RCS may strengthen post quantum confidence in the protocol.

In summary, SIAP achieves a balance between strong symmetric security, forward secrecy, and predictable behavior, but its operational constraints and structural choices offer several opportunities for further refinement and research.

11 Conclusion

11.1 Summary of Results

This paper presented a complete design and analysis of the Secure Infrastructure Access Protocol. The study incorporated the protocol specification, the reference implementation, and a formal model that captures the operational behavior of SIAP with mathematical precision. The engineering section provided an implementation faithful description of the identity structures, token derivation mechanisms, passphrase hardening steps, and authenticated state transitions.

The security model formalized authentication soundness, replay protection, token confidentiality, symmetric forward secrecy, and offline attack resistance. Each property was expressed through a well defined game, and the resulting reductions showed that the security of SIAP depends on the pseudo-random behavior of cSHAKE, the integrity and confidentiality guarantees of RCS, and the memory hardness of the SCB passphrase function. The cryptanalytic evaluation demonstrated that token independence, domain separated key derivation, and destructive token erasure provide strong protection under device compromise and server database leakage.

The implementation alignment section confirmed that the reference code matches the formal specification, including key derivation paths, state updates, and error handling behavior. Performance considerations show that SIAP maintains predictable computational cost and controlled operational limits, with most expense dominated by SCB evaluation and linear scans over the token tree.

11.2 Implications for Deployment

SIAP is suited to environments where symmetric authentication is required without reliance on public key infrastructure or online verification services. The protocol provides forward secrecy through erasure of consumed tokens, strong offline attack resistance through memory hardened passphrase processing, and predictable state evolution through strict monotonic index progression.

Operationally, SIAP benefits deployments where controlled administrative access, offline authentication, and infrastructure level access control are required. The design avoids global secrets shared across users or devices, and server compromise does not expose token material or enable token prediction. The finite token supply and strict synchronization requirements impose clear operational constraints, but these constraints are consistent with environments where security and predictability are prioritized over unbounded token availability.

Future deployments may extend SIAP through improved resynchronization strategies, adaptive token trees, or hybrid compositions with asymmetric mechanisms. Nevertheless, in its current form SIAP provides a clear and robust foundation for symmetric, post quantum resistant authentication in constrained environments.

References

1. National Institute of Standards and Technology (NIST). *SHA 3 Standard: Permutation Based Hash and Extendable Output Functions*. 2015. Available at: https://doi.org/10.6028/NIST.FIPS.202.

2. National Institute of Standards and Technology (NIST). *SP 800 185: SHA 3 Derived Functions*. 2016. Available at: https://doi.org/10.6028/NIST.SP.800-185.

3. M. Bellare, R. Canetti, and H. Krawczyk. *Keying Hash Functions for Message Authentication.* Crypto 1996. Available at: https://doi.org/10.1007/3-540-68697-5_24.

4. National Institute of Standards and Technology (NIST). *HOTP: An HMAC-Based One-Time Password Algorithm.* RFC 4226, 2005. Available at: https://www.rfc-editor.org/rfc/rfc4226.

5. National Institute of Standards and Technology (NIST). *TOTP: Time-Based One-Time Password Algorithm.* RFC 6238, 2011. Available at: https://www.rfc-editor.org/rfc/rfc6238.

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

7. D. Jablon. *S/Key One-Time Password System.* RFC 1760, 1995. Available at: https://www.rfc-editor.org/rfc/rfc1760.

8. N. Provos and D. Mazieres. *A Future-Adaptable Password Scheme.* USENIX 1999. Available at: https://www.usenix.org/legacy/events/usenix99/provos/provos.pdf.

9. C. Percival. *Stronger Key Derivation via Sequential Memory-Hard Functions.* BSDCan, 2009. (The scrypt paper.) Available at: https://www.tarsnap.com/scrypt/scrypt.pdf.

10. J. Bonneau et al. *The Quest to Replace Passwords: A Framework for Comparative Evaluation of Web Authentication Schemes.* IEEE S&P 2012. Available at: https://doi.org/10.1109/SP.2012.44.

11. FIDO Alliance. *FIDO2 Client to Authenticator Protocol (CTAP).* 2018. Available at: https://fidoalliance.org/specifications/.

12. Underhill, J. G. *SIAP Specification.* Quantum Resistant Cryptographic Solutions Corporation, 2025. Available at: https://www.qrcscorp.ca/documents/siap_specification.pdf

13. QRCS Corporation *SIAP Reference Implementation.* Quantum Resistant Cryptographic Solutions Corporation, 2025. Available at: https://github.com/QRCS-CORP/SIAP