S-OTP: Signature-Based One-Time Password

C. Kusuma †

c.kusuma@proton.me

December 28, 2025

Version-001

Abstract. This paper presents a time-expiring, signature-based One-Time Password (OTP) scheme designed to address security limitations of conventional TOTP systems and centralized authentication models. Rather than shared-secret approaches, OTPs are generated as cryptographic signatures over time-bound messages using lightweight OTS and standard cryptographic primitives. Beyond user authentication, the proposed construction mitigates risks inherent in centralized command authorities, where a single compromised controller or verifier constitutes a critical single point of failure. We proposed to enabling independent, cryptographic verification of command or authentication validity, the scheme reduces reliance on a trusted central authority and addresses failure modes analogous to the Byzantine Generals Problem, in which malicious or faulty commanders cannot be distinguished from honest ones without cryptographic guarantees. The proposed scheme achieves $2^{128} - bit$ post-quantum security with low computational overhead.

1. Introduction

Modern authentication systems remain a primary target for cyber attacks, particularly those relying solely on static credentials such as passwords or API keys. Without the use of One-Time Passwords (OTPs), compromised credentials can be replayed indefinitely, enabling unauthorized access through phishing, credential stuffing, database breaches, and man-in-the-middle attacks. In such systems, an attacker who gains access to a valid credential effectively gains persistent control until the secret is rotated or revoked.

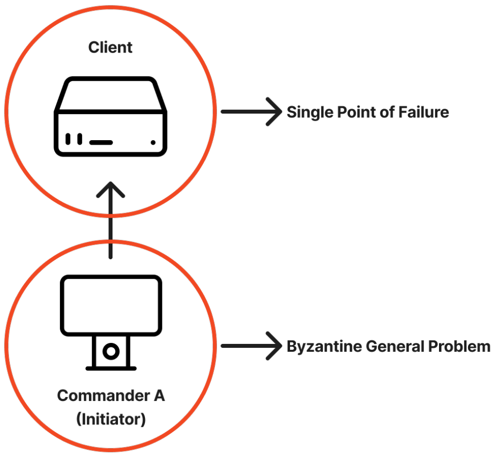
The introduction of OTP mechanisms significantly reduces this attack surface by enforcing single-use or time-limited authentication tokens. Even if an OTP is intercepted, its short validity window and non-reusability prevent long-term exploitation. This property dramatically limits the impact of credential leakage, transforming many high-severity attacks into low-probability, time-constrained events.

However, widely deployed OTP standards—such as Time-based One-Time Passwords (TOTP)—often rely on shared secrets and symmetric verification, which introduce new risks, including server-side secret exposure and phishing relay attacks. These limitations motivate the need for stronger OTP constructions that preserve usability while improving cryptographic assurance.

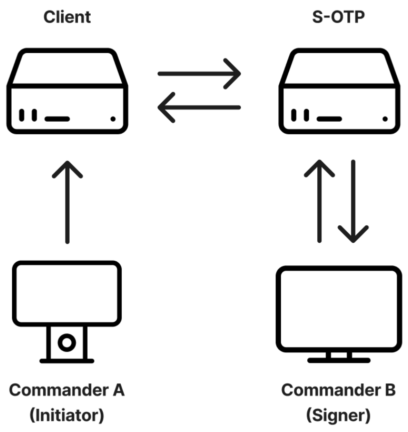
2. Background

Traditionally, OTP mechanisms have been widely deployed in civilian domains such as online banking, payment authorization, and access control systems, where they serve to mitigate password reuse, phishing, and replay attacks by enforcing short-lived authentication factors. In these environments, OTPs are typically bound to shared secrets and centralized verification servers, which is sufficient under assumptions of trusted infrastructure and limited adversarial capability. However, the same OTP concept can be extended beyond financial authentication into adversarial and mission-critical settings. In military and aerospace use cases—such as unmanned aerial vehicle (UAV)/ BOAT command-and-control—authentication must withstand compromised operators, hostile networks, and post-quantum threat models.

If we take a look at a use-case in the critical operations command like a single-controller remote systems for automations, such as handheld-operated UAV/ BOAT platforms, centralizing command authority introduces a critical single point of failure, a compromised or faulty controller effectively acts as a dishonest commander, analogous to the Byzantine Generals Problem, where malicious commands are indistinguishable from legitimate ones. Without cryptographic authentication and temporal guarantees, the UAV/ BOAT and supporting infrastructure cannot reliably verify command integrity, freshness, or authorization.



To mitigate this risk, the proposed model separates command initiation and command authorization. **Commander A** operates the handheld controller and initiates control requests, while **Commander B** independently approves and cryptographically signs each request before it is accepted by the UAV/ BOAT server. This dual-commander architecture prevents unilateral command execution and ensures that no single compromised operator can issue valid control instructions.



Each approved command is protected using a lightweight, time-expiring, signature-based OTP, binding the command to both authorization and strict temporal validity. This transforms command authority from a persistent privilege into a short-lived, verifiable event. Even if Commander **A** is compromised, unauthorized commands cannot propagate without Commander **B**'s approval, significantly reducing replay, command injection, and Byzantine fault impact in safety-critical remote control systems.

2.1 Use-case Dual-Commander

OTP Authorization for UAV/ BOAT Control, Entities;

C_A	: Commander A (controller / initiator)
C_B	: Commander B (authorizer / signer)
$\mathcal{U}$	: UAV/ BOAT Server (Client)
$\mathcal{O}$	: S-OTP Server
pk_B	: Public verification key of Commander B
Δt	: OTP validity window
μ	: OTP challenge message

Step 1: Command Request

Commander A sends a command request to the UAV/ BOAT server:

$$C_A \rightarrow \mathcal{U}: m$$

where $\boldsymbol{m} \in \boldsymbol{M}$ is the requested UAV/ BOAT operation.

Step 2: OTP Challenge Generation

Upon receipt, the UAV/ BOAT server requests a time-bound OTP challenge from the OTP server:

$$\mathcal{U} \rightarrow \mathcal{O}: ReqOTP(m)$$

The OTP server constructs a challenge:

$$\mu = H(m \parallel t \parallel sid)$$

where:

- H is a cryptographic hash function,
- t is the current time epoch,
- sid is a session identifier.

Step 3: Authorization Request

The OTP server forwards the challenge to Commander B:

$$\mathcal{O} \rightarrow C_B: \mu$$

Step 4: Signature-Based OTP Generation

Commander B signs the challenge using a lightweight Lamport one-time signature:

$$\sigma_B = Sign_B(\mu)$$

Commander B returns the signature:

$$C_B \rightarrow \mathcal{O}: \sigma_B$$

Step 5: OTP Verification

The OTP server verifies the signature:

$$Verify_B(\mu, \sigma_B, pk_B) \overset{?}{=} 1$$

and checks time validity:

$$|\ t_{recv} - t\ | \leq \Delta t$$

Step 6: Authorization Decision

If verification succeeds, the OTP server authorizes the request:

$$\mathcal{O} \rightarrow \mathcal{U}: AUTH\_OK(m)$$

Otherwise:

$$\mathcal{O} \rightarrow \mathcal{U}: AUTH\_FAIL$$

Step 7: Command Execution

The UAV/ BOAT server executes the command **if and only if** authorization is granted:

$$\mathcal{U} \xrightarrow{AUTH\_OK} Execute(m)$$

2.2 Security Properties

- Unilateral resistance;

$$\neg\exists\sigma: Verify_B(\mu, \sigma) = 1\ without\ C_B$$

- Reply resistance;

$$Pr[Replay(\mu, \sigma B)] \leq Pr[t > \Delta t]$$

- Byzantine and Failure tolerance;

A compromised C_A cannot cause execution without cooperation from C_B.

2.3 Intuition

- Commander A **cannot self-authorize**
- Commander B **never directly controls UAVs/ BOATs**
- S-OTP server **enforces freshness and validity**
- UAV/ BOAT server **executes only verified, time-bound approvals**

3. Crypto Signature

This work proposes a signature-based, time-expiring OTP model built on lightweight post-quantum and standard cryptographic primitives. By replacing shared secrets with asymmetric, one-time signing keys and binding OTPs to strict temporal validity, the proposed approach significantly reduces replay, impersonation, and credential forwarding risks, providing a robust and future-resilient authentication mechanism.

In the age of quantum adversaries, whose massive parallelism and algorithmic acceleration threaten the security assumptions of widely deployed public-key systems, many conventional cryptographic primitives can no longer be assumed to offer long-term protection. While modern post-quantum signature schemes—such as lattice-based constructions or stateless hash-based schemes like SPHINCS+—provide strong security guarantees, their computational cost, signature size, and verification overhead make them impractical for high-frequency, time-constrained OTP systems. In particular, the large signature and public key sizes inherent to these schemes introduce unacceptable bandwidth, storage, and latency overheads when OTPs are generated and verified repeatedly or embedded into constrained communication channels.

To address these limitations, this work adopts Lamport–Winternitz one-time signatures as the cryptographic foundation for OTP authentication. Unlike general-purpose post-quantum signature schemes, Lamport–Winternitz OTS offers a highly compact and deterministic signing process with predictable performance characteristics, making it well-suited for short-lived, single-use authentication tokens. By leveraging the one-time security property, the OTP itself becomes a cryptographically signed authorization artifact rather than a derived shared secret. This design achieves post-quantum resistance through hash-based security assumptions while avoiding the excessive signature sizes and state-management complexity associated with SPHINCS+ or lattice-based alternatives. As a result, the proposed system strikes a practical balance between quantum resilience, efficiency, and deployability in real-world OTP and command authentication scenarios.

From a systems perspective, signature size and key material overhead are decisive factors in determining the suitability of a post-quantum primitive for OTP-based authentication. Even when selecting the smallest standardized parameter sets, general-purpose post-quantum signature schemes remain costly. For example, compact SPHINCS+ variants (e.g., SPHINCS+-128s) require public keys of approximately 32 bytes and secret keys of 64 bytes, but incur signature sizes on the order of 7–8 KB per authentication event. Similarly, lattice-based schemes optimized for size—such as Dilithium-II or Falcon-512—still require public keys ranging from roughly 0.9 to 1.3 KB, secret keys between 1.2 and 2.5 KB, and signatures between 666 bytes and over 2 KB, depending on the scheme and security margin. While these constructions are well-suited for software distribution, document signing, or infrequent authentication, their per-signature overhead becomes prohibitive in high-frequency, time-expiring OTP systems.

In contrast, Lamport–Winternitz one-time signatures trade reusability for simplicity and efficiency in single-use contexts. A Winternitz-based OTS instantiated with SHA-256 typically produces public keys on the order of 1–4 KB and signatures of comparable size, while relying solely on hash evaluations and avoiding complex algebraic operations. Although the private key material is larger, it is never transmitted and can be deterministically regenerated or derived per OTP instance. When further aggregated using Merkle-tree constructions, many one-time public keys can be amortized under a single compact root hash, reducing the effective public verification key to 32 bytes while preserving post-quantum security guarantees. This asymmetric cost profile aligns naturally with OTP systems, where signatures are short-lived, non-reusable, and verified far more often than keys are distributed.

Consequently, when evaluated under bandwidth constraints, latency sensitivity, and repeated authentication requirements, Lamport–Winternitz one-time signatures offer a more practical design point than SPHINCS+ or lattice-based schemes. They provide quantum-resistant authentication with predictable performance and manageable signature sizes, making them especially suitable for OTP-based banking systems and mission-critical command authorization scenarios such as UAV/ BOAT control, where efficiency and reliability outweigh long-term key reusability.

4. Winternitz One-Time Signature (WOTS)

We propose the Winternitz one-time signature scheme because it is a well-established cryptographic construction whose security is well understood, as it relies solely on the strength of the underlying hash function. The nature it is “stateless”, which simplifies key management and makes database reclamation is straightforward.

The proposed OTP system instantiates a Lamport–Winternitz one-time signature (WOTS) scheme with parameter $w = 16$ and a $256 - bit$ cryptographic hash function H. Let m denote the message to be authenticated (e.g., a time-bound OTP payload). The message is first hashed as:

$$d = H(m) \in \{0,1\}^{256}$$

Interpreting d in base$-\boldsymbol{w}$, the number of required message digits is:

$$t_1 = \lceil \frac{256}{log_2 w} \rceil = 64$$

A checksum is appended to prevent partial forgery and truncation attacks:

$$t_2 = \lfloor log_w(t_1(w-1)) \rfloor + 1 = 3$$

Thus, the total number of hash chains is:

$$\ell = t_1 + t_2 = 67$$

a. Key Generation

- The private key consists of ℓ uniformly random $256 - bit$ values:

$$SK = (x_1, x_2, \dots, x_{67}), x_i \in \{0,1\}^{256}$$

- The public key is derived by iteratively hashing each private element $w - 1 = 15$ times:

$$PK = (H^{15}(x_1), H^{15}(x_2), \dots, H^{15}(x_{67}))$$

- The raw public key size is therefore:

$$67 \times 32 = 2144\ bytes$$

which can be compressed into a single root hash when aggregated using a Merkle tree to $32\ bytes$.

b. Signature Generation

- Let the base $-w$ representation of the message digest and checksum be:

$$(d_1, d_2, \dots, d_{67}), 0 \le d_i \le 15$$

- The signature is computed as:

$$\sigma = (H^{d_1}(x_1), H^{d_2}(x_2), \dots, H^{d_{67}}(x_{67}))$$

- The terminal output show:
  - **Signature size**: 4.32 KB (4419 bytes)
  - **Signing time**: 0.34 ms

```
[SIGNING] Generated Lamport Signature: 📊 Size: 4.32 KB (4419
bytes) ⏱️ Signing Time: 341.243µs (0.34 ms) INFO: 2026/01/31
17:31:55 sign.go:108: Generated signature size: 4.32 KB (4419
bytes), signing time: 341.243µs DEBUG DefaultWOTSParams]
returning w=16 [DEBUG NewWOTSParams] w=16, t1=64, t2=3, n=67 INFO:
2026/01/31 17:31:38 register.go:79: [REGISTRATION] WOTS
Parameters: N=67 (should be 67), W=16, KeyLength=32 [DEBUG
DefaultWOTSParams] returning w=16 [DEBUG NewWOTSParams] w=16,
t1=64, t2=3, n=67 [DEBUG GenerateMasterSeed] userID:
'e227b66ff7759e42fed2bc0882c4061c63ff725a14140421d9dec3133145dee
b772YQS9UVH5T9LXE' (len=80) deviceToken:
'd6fb638e0cc411f304b0cc153062c1136c9de9dda26b450238949f8c17e656e
d' (len=64) userID bytes:
6532323762363666663737353965343266656432626330383832633430363163
3633666637323561313431343034323164396465633331333331343564656562
37373259515339555648355439 4c5845 deviceToken bytes:
```

```
64366662363338653063633431316633303462306363313533303632633313133
36633964653964646132366234353032333839343966386331376536353366564
Argon2 params: t=3, m=65536, p=4 Output length: 64 masterSeed hex
(first 16): eb7d59ef58a1837bc5dcb40eb20aa6c6... masterSeed
length: 64 [DEBUG DefaultWOTSParams] returning w=16 [DEBUG
NewWOTSParams] w=16, t1=64, t2=3, n=67 [DEBUG
GenerateDeterministicKeyPair] masterSeed hex (first 16):
eb7d59ef58a1837bc5dcb40eb20aa6c6... masterSeed length: 64 index:
0 params.W: 16 params.N: 67 keySeed hex (first 16):
eb7d59ef58a1837bc5dcb40eb20aa6c6... keySeed length: 72 private[0]
hex:
848e3af64895c39ee14ca329b2044dc0c6173a06bb1e27df3e2ddcf51417744d
public[0] hex:
9ddb64a2e4487be94795952d0b90d7e3a11d42aea7a027f36d360100e226cf2b
Building hash tree with 67 public segments... [DEBUG
DefaultWOTSParams] returning w=16 [DEBUG NewWOTSParams] w=16,
t1=64, t2=3, n=67 [DEBUG DefaultWOTSParams] returning w=16 [DEBUG
NewWOTSParams] w=16, t1=64, t2=3, n=67 [DEBUG DefaultWOTSParams]
returning w=16 [DEBUG NewWOTSParams] w=16, t1=64, t2=3, n=67
[DEBUG DefaultWOTSParams] returning w=16 [DEBUG NewWOTSParams]
w=16, t1=64, t2=3, n=67 [DEBUG DefaultWOTSParams] returning w=16
[DEBUG NewWOTSParams] w=16, t1=64, t2=3, n=67 [DEBUG
DefaultWOTSParams] returning
w=16……………………………………………………………………………………………………..
```

c. Verification

Verification recomputes the remaining portion of each hash chain:

$$H^{15 - d_i}(\sigma_i) \overset{?}{=} PK_i$$

The signature is accepted if all reconstructed values match the public key.

In the implemented system, the **total verification request time**, including signature verification, OTP lookup, temporal validation, and storage access, is measured as:

$$\text{TotalVerificationTime} \approx 110.23\, ms$$

```
INFO: verify.go:112: OTP verified successfully
INFO: verify.go:113: Total Verification Request Time:
110.229629ms
```

This latency reflects **end-to-end verification cost**, dominated by network communication, database access, and request handling, rather than cryptographic verification alone. The cryptographic verification component—consisting exclusively of hash evaluations—remains computationally lightweight and scales predictably with the parameter ℓ.

d. One-Time Security and Key Freshness

Lamport–Winternitz signatures are strictly one-time secure. Each signature reveals intermediate hash values indexed by the message-dependent digits d_i. Assuming uniformly distributed message digests, the expected fraction of each hash chain revealed is:

$$\mathbb{E}[\frac{d_i}{w-1}] = \frac{1}{2}$$

Thus, a single signature discloses approximately **40–50% of the private key material**. Reusing the same key pair for multiple messages allows an adversary to combine exposed chain prefixes, eventually reconstructing the full private key and forging signatures.

To prevent this, the proposed system enforces **fresh key generation per authentication event**, with each Lamport–Winternitz key pair deterministically derived from a master seed bound to a registered user and device identity. This guarantees that every OTP corresponds to a unique signing key, ensuring resistance to replay, forgery, and Byzantine command injection even under full signature disclosure.

5. Cryptographic key binding

This architecture provides strong binding between user identity, physical device, and cryptographic material while maintaining verifiable integrity through Merkle tree commitments. The hierarchical design ensures that compromise of any single component doesn't invalidate the entire security model, core Identity Components and Merkle Tree Architecture.

5.1 Deterministic Device Fingerprint

The **Device ID** serves as a unique identifier derived from hardware characteristics, ensuring consistent identification across sessions while preserving privacy.

Generation Algorithm:

Given device information $D = \{(\kappa_1, v_1), (\kappa_2, v_2), \dots, (\kappa_n, v_n)\}$, where each κ_i is a hardware attribute name and v_i is its corresponding value:

a. **Normalization**: For each value v_i, apply cleaning:

$$v_i' = trim(remove\_quotes(v_i))$$

$$v_i'' = trim(remove\_quotes(v_i'))$$

b. **Deterministic Ordering**: Sort attribute pairs by key:

$$S = sort(\{(\kappa_i, v_i'')\} \text{ by } \kappa_i)$$

c. **Concatenation**: Create input string:

$$I = concat(\forall(\kappa, v) \in S{:}\kappa + "=" + v,\ separator{:}\ "||")$$

d. **Cryptographic Hashing**: Apply SHAKE256 XOF:

$$H = SHAKE256(\text{LAMPORT\_DEVICE\_ID\_SALT\_V1}| + I)$$

$$DeviceID = hex(H0{:}32)$$

e. **Purpose:**

- Provides consistent identification across registration and authentication
- Resists spoofing through hardware attribute aggregation
- Enables server-side validation of device consistency
- Supports fallback mechanisms when TPM is unavailable

5.2 Device Token: Session-Specific Authorization Token

The Device Token establishes a cryptographic binding between user identity and specific device instance.

a. **Generation Function:**

$$DeviceToken = H(UserID \oplus DeviceID)$$

Where:

- H represents SHAKE256 extended output function
- $\oplus$ denotes concatenation with domain separation
- $UserID$ is the 64-character SHA-256 hash of user identifier
- $DeviceID$ is the deterministic device fingerprint

b. **Properties:**

i. **Unforgeability**: $DeviceToken$, computationally infeasible to find $(UserID', DeviceID')$ such that:

$$H(UserID' \oplus DeviceID') = DeviceToken$$

ii. **Binding:** Token uniquely binds specific user to specific device:

$$Bind = Pr[verify(UserID, DeviceID, DeviceToken) = 1] = 1$$

$$CollisionResistance = Pr[verify(UserID', DeviceID', DeviceToken) = 1] \leq \epsilon$$

iii. **Purpose:**

- Serves as session authorization token
- Enables key storage lookup (combined with Merkle Root)
- Prevents token replay across different devices
- Provides second factor of device authentication

5.3 Merkle Root: Cryptographic State Digest

The Merkle Root provides a compact representation of the complete registration state, enabling efficient verification of component consistency.

Construction:

Given leaves $L = [l_1, l_2, \dots, l_n]$ where:

$$l_1 = UserID, \quad l_2 = DeviceID, \quad l_3 = DeviceToken, \quad l_4 = SeedHash$$

i. **Leaf Hashing**: Each leaf undergoes double hashing:

$$h_i = H\big(H(l_i)\big)$$

ii. **Binary Tree Construction**: For tree levels $j = 1\ to\ \lceil log_2 n \rceil$:

$$h_k^{(j)} = H\left(h_{2k-1}^{(j-1)} \oplus h_{2k}^{(j-1)}\right)$$

For odd nodes at level $j - 1$, $h_{2k}^{(j-1)} = h_{2k-1}^{(j-1)}$

iii. **Root Computation:** Final root hash:

$$RootHash = h_1^{(\lceil log_2 n \rceil)}$$

Verification Properties:

The Merkle Root enables efficient proof of inclusion through Merkle paths. For any leaf l_i, the proof π_i consists of sibling hashes along the path to root. Verification satisfies:

$$Verify(RootHash, l_i, \pi_i) = 1 \Leftrightarrow l_i \in L$$

iiii. Purpose:

- Provides compact state commitment (32 bytes)
- Enables efficient consistency verification
- Supports non-membership proofs for revocation
- Serves as database lookup key component

5.4 Component Integration and Security Guarantees

The identity components form a cryptographic chain where each element verifies the previous:

a. **DeviceID → DeviceToken:**

$$DeviceToken = \mathcal{F}(UserID, DeviceID)$$

b. Components → Merkle Root:

$$RootHash = \mathcal{M}(UserID, DeviceID, DeviceToken, SeedHash)$$

c. Storage Key Derivation:

$$StorageKey = DeviceToken \parallel RootHash$$

5.5 Security Properties

a. Theorem 1 (Device Consistency):

For any two successful authentications A_1, A_2 from the same legitimate user:

$$Pr[DeviceID(A_1) = DeviceID(A_2)] \geq 1 - negl(\lambda)$$

provided hardware attributes remain constant.

Proof Sketch: DeviceID derivation is deterministic function of hardware measurements. Small hardware variations are normalized during value cleaning.

b. Theorem 2 (Token Unforgeability):

Given security parameter λ, for any PPT adversary $\mathcal{A}$:

$$Pr[\mathcal{A}(UserID, DeviceID) \rightarrow DeviceToken' : verify(DeviceToken') = 1] \\ \leq negl(\lambda)$$

Proof Sketch: Reduces to pre-image resistance of SHAKE256. Successful forgery implies SHAKE256 collision.

c. Theorem 3 (State Integrity):

The Merkle Root construction ensures that any unauthorized modification to stored components is detectable:

$$\forall i \in \{1, \dots, n\}, \forall l_i' \neq l_i : RootHash' \neq RootHash$$

with probability $1 - 2^{-256}$.

5.6 Practical Implementation Notes

Hardware Information Collection, the system employs platform-specific methods to gather stable identifiers:

- Windows: wmic commands for CPU, baseboard, system UUID
- macOS: ioreg for platform serial, hardware UUID

- Linux: /sys/class/dmi/id/ for product UUID, udevadm for disk serial
- Mobile: Build fingerprints, model identifiers

a. Fallback Strategies:

When primary identifiers are unavailable, the system employs:

$$FallbackID = \mathcal{H}(platform \parallel arch \parallel MAC \parallel random\_salt)$$

b. TPM Integration (Optional):

When available, the system prioritizes TPM-based identification:

$$DeviceID_{TPM} = \mathcal{H}(TPM\_EK \parallel platform\_info)$$

providing hardware-backed security guarantees.

5.7 Operational Workflow

a. Registration Phase:

i. Signer collects hardware information D
ii. Generates DeviceID via deterministic algorithm
iii. Server computes $DeviceToken = \mathcal{F}(UserID, DeviceID)$
iv. Construct Merkle tree from components
v. Store (DeviceToken, RootHash) as lookup key

b. Authentication Phase:

i. Signer presents DeviceID (recomputed from hardware)
ii. Server retrieves DeviceToken via UserID
iii. Verifies DeviceToken consistency
iv. Uses (DeviceToken, RootHash) to locate stored keys
v. Validates Merkle root matches stored state

c. Revocation and Recovery:

i. **Revocation**: Remove Merkle root from database
ii. **Device Change**: Require re-registration with new DeviceID
iii. **Key Rotation**: New registration generates new Merkle root

5.8 Security Analysis

a. Privacy Considerations:

i. DeviceID reveals only hashed hardware information

ii. No personally identifiable information in Merkle root
iii. Token-based lookup prevents user enumeration

b. Resistance to Attacks:

i. **Device Cloning**: Requires identical hardware measurements
ii. **Token Theft**: Compromises single device only
iii. **Database Leak**: DeviceToken alone insufficient without hardware
iv. **Replay Attacks**: Prevented by OTP mechanism and Merkle verification

5.9 Conclusions

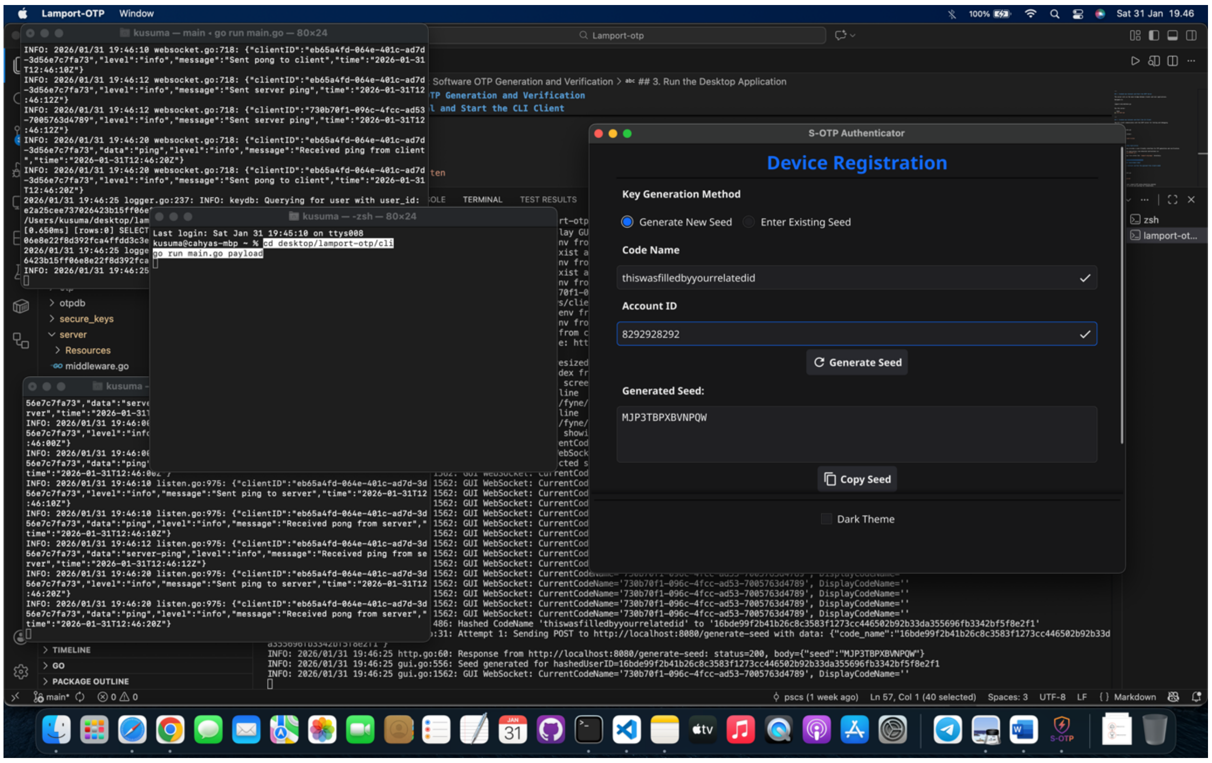
The identity component system provides a layered approach to device authentication:

1. DeviceID establishes hardware-based identity
2. DeviceToken creates cryptographic user-device binding
3. Merkle Root ensures state integrity and efficient verification

Together, these components enable secure, privacy-preserving device authentication while maintaining the one-time property of the Lamport OTP system. The deterministic nature of DeviceID generation ensures consistent device recognition, while the cryptographic binding prevents unauthorized device substitution.

The system's design reflects practical security considerations, including fallback mechanisms for diverse hardware environments and efficient verification through Merkle tree constructions. This approach balances security guarantees with practical deployment considerations across various platforms and device types.

6. Implementation



```
kusuma — main ‹ go run main.go — 80×24
INFO: 2026/01/31 19:46:10 websocket.go:718: {"clientID":"eb65a4fd-064e-401c-ad7d
-3d56e7c7fa73","level":"info","message":"Sent pong to client","time":"2026-01-31
T12:46:10Z"}
INFO: 2026/01/31 19:46:12 websocket.go:718: {"clientID":"eb65a4fd-064e-401c-ad7d
-3d56e7c7fa73","level":"info","message":"Sent server ping","time":"2026-01-31T12
:46:12Z"}
INFO: 2026/01/31 19:46:12 websocket.go:718: {"clientID":"730b70f1-096c-4fcc-ad53
-7005763d4789","level":"info","message":"Sent server ping","time":"2026-01-31T12
:46:12Z"}
INFO: 2026/01/31 19:46:20 websocket.go:718: {"clientID":"eb65a4fd-064e-401c-ad7d
-3d56e7c7fa73","data":"ping","level":"info","message":"Received ping from client
","time":"2026-01-31T12:46:20Z"}
INFO: 2026/01/31 19:46:20 websocket.go:718: {"clientID":"eb65a4fd-064e-401c-ad7d
-3d56e7c7fa73","level":"info","message":"Sent pong to client","time":"2026-01-31
T12:46:20Z"}
2026/01/31 19:46:25 logger.go:237: INFO: keydb: Querying for user with user_id:
e2a25cee737026423b15ff06e
/Users/kusuma/desktop/lam
[0.650ms] [rows:0] SELECT
06e8e22f8d392fca4ffdd3c3e
2026/01/31 19:46:25 logge
6423b15ff06e8e22f8d392fca
INFO: 2026/01/31 19:46:25
```

```
kusuma — -zsh — 80×24
Last login: Sat Jan 31 19:45:10 on ttys008
kusuma@cahyas-mbp ~ % cd desktop/lamport-otp/cli
go run main.go payload
```

```
56e7c7fa73","data":"serve
rver","time":"2026-01-31T
INFO: 2026/01/31 19:46:00
56e7c7fa73","level":"info
:46:00Z"}
INFO: 2026/01/31 19:46:00
56e7c7fa73","data":"ping"
time":"2026-01-31T12:46:00Z"}
INFO: 2026/01/31 19:46:10 listen.go:975: {"clientID":"eb65a4fd-064e-401c-ad7d-3d
56e7c7fa73","level":"info","message":"Sent ping to server","time":"2026-01-31T12
:46:10Z"}
INFO: 2026/01/31 19:46:10 listen.go:975: {"clientID":"eb65a4fd-064e-401c-ad7d-3d
56e7c7fa73","data":"ping","level":"info","message":"Received pong from server","
time":"2026-01-31T12:46:10Z"}
INFO: 2026/01/31 19:46:12 listen.go:975: {"clientID":"eb65a4fd-064e-401c-ad7d-3d
56e7c7fa73","data":"server-ping","level":"info","message":"Received ping from se
rver","time":"2026-01-31T12:46:12Z"}
INFO: 2026/01/31 19:46:20 listen.go:975: {"clientID":"eb65a4fd-064e-401c-ad7d-3d
56e7c7fa73","level":"info","message":"Sent ping to server","time":"2026-01-31T12
:46:20Z"}
INFO: 2026/01/31 19:46:20 listen.go:975: {"clientID":"eb65a4fd-064e-401c-ad7d-3d
56e7c7fa73","data":"ping","level":"info","message":"Received pong from server","
time":"2026-01-31T12:46:20Z"}
```

```
1562: GUI WebSocket: CurrentCodeName='730b70f1-096c-4fcc-ad53-7005763d4789', DisplayCodeName=''
1562: GUI WebSocket: CurrentCodeName='730b70f1-096c-4fcc-ad53-7005763d4789', DisplayCodeName=''
1562: GUI WebSocket: CurrentCodeName='730b70f1-096c-4fcc-ad53-7005763d4789', DisplayCodeName=''
1562: GUI WebSocket: CurrentCodeName='730b70f1-096c-4fcc-ad53-7005763d4789', DisplayCodeName=''
486: Hashed CodeName 'thiswasfilledbyyourrelatedid' to '16bde99f2b41b26c8c3583f1273cc446502b92b33da355696fb3342bf5f8e2f1'
:31: Attempt 1: Sending POST to http://localhost:8080/generate-seed with data: {"code_name":"16bde99f2b41b26c8c3583f1273cc446502b92b33d
INFO: 2026/01/31 19:46:25 http.go:60: Response from http://localhost:8080/generate-seed: status=200, body={"seed":"MJP3TBPXBVNPQW"}
INFO: 2026/01/31 19:46:25 gui.go:556: Seed generated for hashedUserID=16bde99f2b41b26c8c3583f1273cc446502b92b33da355696fb3342bf5f8e2f1
INFO: 2026/01/31 19:46:25 gui.go:1562: GUI WebSocket: CurrentCodeName='730b70f1-096c-4fcc-ad53-7005763d4789', DisplayCodeName=''
```

S-OTP Authenticator

Device Registration

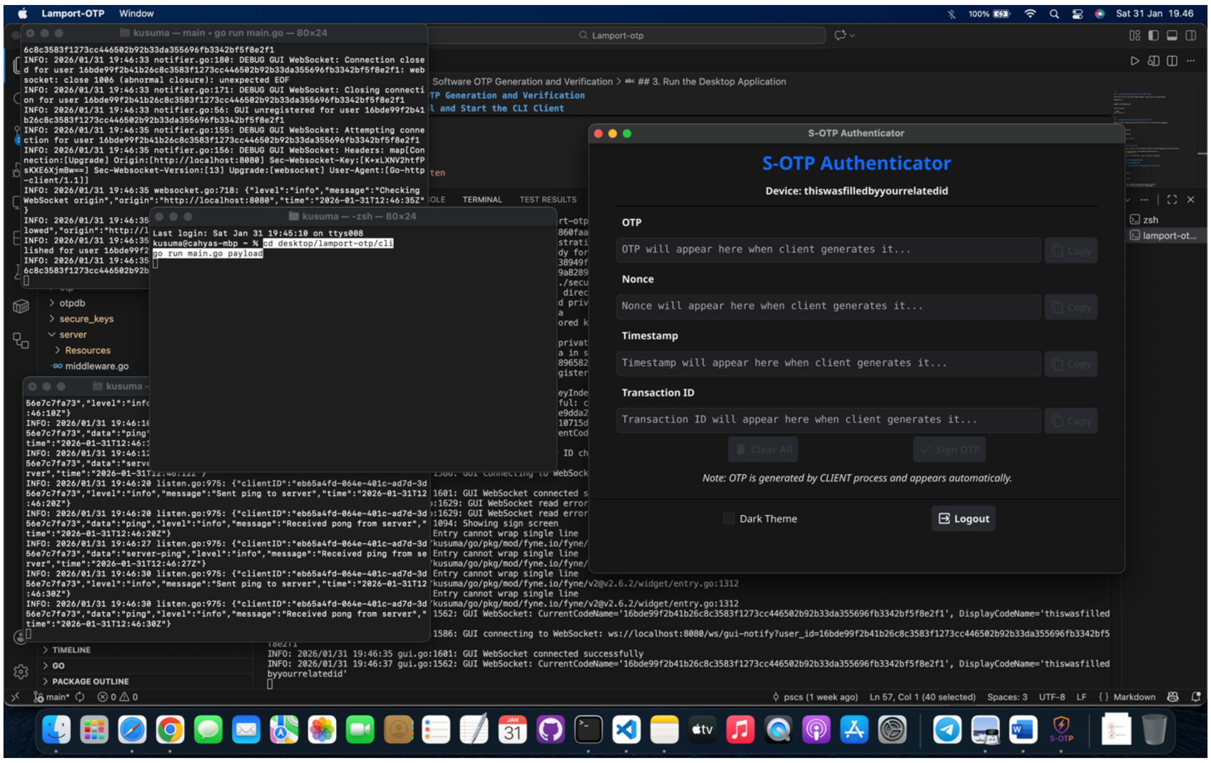
Key Generation Method: [✓ Generate New Seed] Enter Existing Seed

Code Name: thiswasfilledbyyourrelatedid

Account ID: 8292928292

Generated Seed: MJP3TBPXBVNPQW

Dark Theme: ☐



```
kusuma — main ‹ go run main.go — 80×24
6c8c3583f1273cc446502b92b33da355696fb3342bf5f8e2f1
INFO: 2026/01/31 19:46:33 notifier.go:180: DEBUG GUI WebSocket: Connection close
d for user 16bde99f2b41b26c8c3583f1273cc446502b92b33da355696fb3342bf5f8e2f1: web
socket: close 1006 (abnormal closure): unexpected EOF
INFO: 2026/01/31 19:46:33 notifier.go:171: DEBUG GUI WebSocket: Closing connecti
on for user 16bde99f2b41b26c8c3583f1273cc446502b92b33da355696fb3342bf5f8e2f1
INFO: 2026/01/31 19:46:33 notifier.go:56: GUI unregistered for user 16bde99f2b41
b26c8c3583f1273cc446502b92b33da355696fb3342bf5f8e2f1
INFO: 2026/01/31 19:46:35 notifier.go:155: DEBUG GUI WebSocket: Attempting conne
ction for user 16bde99f2b41b26c8c3583f1273cc446502b92b33da355696fb3342bf5f8e2f1
INFO: 2026/01/31 19:46:35 notifier.go:156: DEBUG GUI WebSocket: Headers: map[Con
nection:[Upgrade] Origin:[http://localhost:8080] Sec-Websocket-Key:[K+xLXNV2htfP
sKXE6XjmBw==] Sec-Websocket-Version:[13] Upgrade:[websocket] User-Agent:[Go-http
-client/1.1]]
INFO: 2026/01/31 19:46:35 websocket.go:718: {"level":"info","message":"Checking
WebSocket origin","origin":"http://localhost:8080","time":"2026-01-31T12:46:35Z"
}
```

```
kusuma — -zsh — 80×24
Last login: Sat Jan 31 19:45:10 on ttys008
kusuma@cahyas-mbp ~ % cd desktop/lamport-otp/cli
go run main.go payload
```

S-OTP Authenticator

Device: thiswasfilledbyyourrelatedid

OTP: OTP will appear here when client generates it...

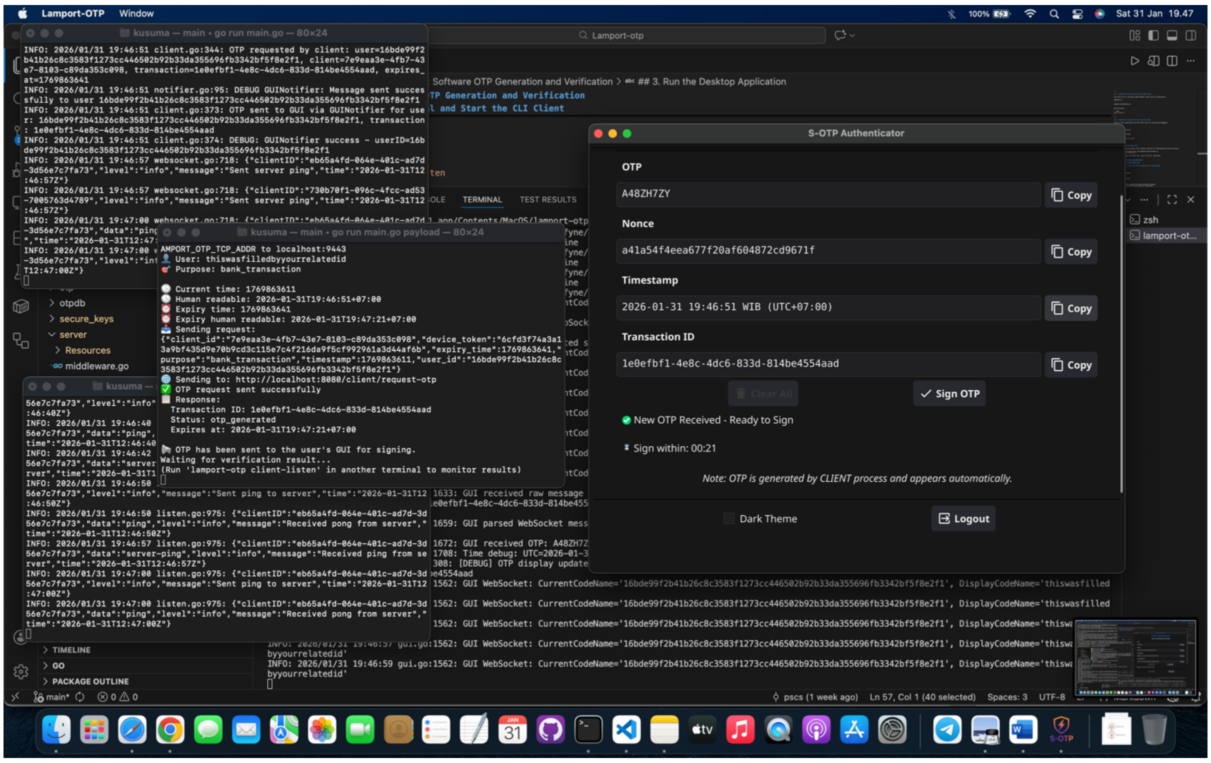
Nonce: Nonce will appear here when client generates it...

Timestamp: Timestamp will appear here when client generates it...

Transaction ID: Transaction ID will appear here when client generates it...

Note: OTP is generated by CLIENT process and appears automatically.

```
1586: GUI connecting to WebSocket: ws://localhost:8080/ws/gui-notify?user_id=16bde99f2b41b26c8c3583f1273cc446502b92b33da355696fb3342bf5
INFO: 2026/01/31 19:46:33 gui.go:1601: GUI WebSocket connected successfully
INFO: 2026/01/31 19:46:37 gui.go:1562: GUI WebSocket: CurrentCodeName='16bde99f2b41b26c8c3583f1273cc446502b92b33da355696fb3342bf5f8e2f1', DisplayCodeName='thiswasfilled
byyourrelatedid'
```



```
kusuma — main ‹ go run main.go — 80×24
INFO: 2026/01/31 19:46:51 client.go:344: OTP requested by client: user=16bde99f2
b41b26c8c3583f1273cc446502b92b33da355696fb3342bf5f8e2f1, client=7e9eaa3e-4fb7-43
e7-8103-c89da353c098, transaction=1e0efbf1-4e8c-4dc6-833d-814be4554aad, expires_
at=1769863641
INFO: 2026/01/31 19:46:51 notifier.go:95: DEBUG GUINotifier: Message sent succes
sfully to user 16bde99f2b41b26c8c3583f1273cc446502b92b33da355696fb3342bf5f8e2f1
INFO: 2026/01/31 19:46:51 client.go:373: OTP sent to GUI via GUINotifier for use
r: 16bde99f2b41b26c8c3583f1273cc446502b92b33da355696fb3342bf5f8e2f1, transaction
: 1e0efbf1-4e8c-4dc6-833d-814be4554aad
INFO: 2026/01/31 19:46:51 client.go:374: DEBUG: GUINotifier success - userID=16b
de99f2b41b26c8c3583f1273cc446502b92b33da355696fb3342bf5f8e2f1
INFO: 2026/01/31 19:46:57 websocket.go:718: {"clientID":"eb65a4fd-064e-401c-ad7d
-3d56e7c7fa73","level":"info","message":"Sent server ping","time":"2026-01-31T12
:46:57Z"}
INFO: 2026/01/31 19:46:57 websocket.go:718: {"clientID":"730b70f1-096c-4fcc-ad53
-7005763d4789","level":"info","message":"Sent server ping","time":"2026-01-31T12
:46:57Z"}
```

```
kusuma — main ‹ go run main.go payload — 80×24
AMPORT_OTP_TCP_ADDR to localhost:9443
👤 User: thiswasfilledbyyourrelatedid
🎯 Purpose: bank_transaction

🕒 Current time: 1769863611
🕒 Human readable: 2026-01-31T19:46:51+07:00
⏰ Expiry time: 1769863641
⏰ Expiry human readable: 2026-01-31T19:47:21+07:00
📤 Sending request:
{"client_id":"7e9eaa3e-4fb7-43e7-8103-c89da353c098","device_token":"6cfd3f74a3a1
3a9bf435d9e70b9cd3c115e7c4f216da9f5cf992961a3d44af6b","expiry_time":1769863641,"
purpose":"bank_transaction","timestamp":1769863611,"user_id":"16bde99f2b41b26c8c
3583f1273cc446502b92b33da355696fb3342bf5f8e2f1"}
🌐 Sending to: http://localhost:8080/client/request-otp
✅ OTP request sent successfully
📋 Response:
  Transaction ID: 1e0efbf1-4e8c-4dc6-833d-814be4554aad
  Status: otp_generated
  Expires at: 2026-01-31T19:47:21+07:00

📢 OTP has been sent to the user's GUI for signing.
Waiting for verification result...
(Run 'lamport-otp client-listen' in another terminal to monitor results)
```

S-OTP Authenticator

OTP: A48ZH7ZY

Nonce: a41a54f4eea677f20af604872cd9671f

Timestamp: 2026-01-31 19:46:51 WIB (UTC+07:00)

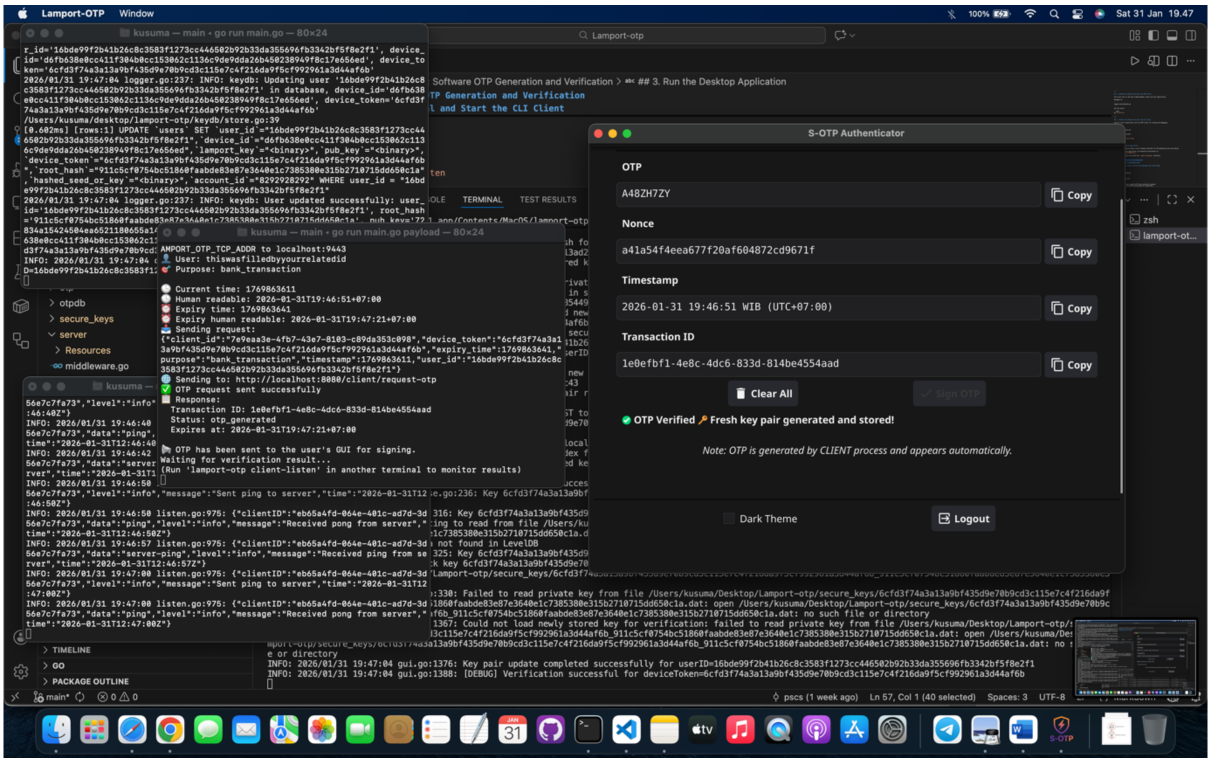
Transaction ID: 1e0efbf1-4e8c-4dc6-833d-814be4554aad

✓ New OTP Received - Ready to Sign

Sign within: 00:21

Note: OTP is generated by CLIENT process and appears automatically.

```
1633: GUI received raw message
1659: GUI parsed WebSocket mess
1672: GUI received OTP: A48ZH7Z
1708: Time debug: UTC=2026-01-3
308: [DEBUG] OTP display update
1562: GUI WebSocket: CurrentCodeName='16bde99f2b41b26c8c3583f1273cc446502b92b33da355696fb3342bf5f8e2f1', DisplayCodeName='thiswasfilled
```



```
r_id='16bde99f2b41b26c8c3583f1273cc446502b92b33da355696fb3342bf5f8e2f1', device_
id='d6fb638e0cc411f304b0cc153062c1136c9de9dda26b450238949f8c17e656ed', device_to
ken='6cfd3f74a3a13a9bf435d9e70b9cd3c115e7c4f216da9f5cf992961a3d44af6b'
2026/01/31 19:47:04 logger.go:237: INFO: keydb: Updating user '16bde99f2b41b26c8
c3583f1273cc446502b92b33da355696fb3342bf5f8e2f1' in database, device_id='d6fb638
e0cc411f304b0cc153062c1136c9de9dda26b450238949f8c17e656ed', device_token='6cfd3f
74a3a13a9bf435d9e70b9cd3c115e7c4f216da9f5cf992961a3d44af6b'
/Users/kusuma/desktop/lamport-otp/keydb/store.go:39
[0.602ms] [rows:1] UPDATE `users` SET `user_id`="16bde99f2b41b26c8c3583f1273cc44
6502b92b33da355696fb3342bf5f8e2f1",`device_id`="d6fb638e0cc411f304b0cc153062c113
6c9de9dda26b450238949f8c17e656ed",`lamport_key`="<binary>",`pub_key`="<binary>",
`device_token`="6cfd3f74a3a13a9bf435d9e70b9cd3c115e7c4f216da9f5cf992961a3d44af6b
",`root_hash`="911c5cf0754bc51860faabde83e87e3640e1c7385380e315b2710715dd650c1a"
,`hashed_seed_or_key`="<binary>",`account_id`="8292928292" WHERE user_id = "16bd
e99f2b41b26c8c3583f1273cc446502b92b33da355696fb3342bf5f8e2f1"
2026/01/31 19:47:04 logger.go:237: INFO: keydb: User updated successfully: user_
id='16bde99f2b41b26c8c3583f1273cc446502b92b33da355696fb3342bf5f8e2f1', root_hash
='911c5cf0754bc51860faabde83e87e3640e1c7385380e315b2710715dd650c1a', pub_key='72
834a15424504ea6521180655a14
638e0cc411f304b0cc153062c11
d3f74a3a13a9bf435d9e70b9cd3
INFO: 2026/01/31 19:47:04 c
D=16bde99f2b41b26c8c3583f12
```

```
AMPORT_OTP_TCP_ADDR to localhost:9443
👤 User: thiswasfilledbyyourrelatedid
🎯 Purpose: bank_transaction

🕒 Current time: 1769863611
🕒 Human readable: 2026-01-31T19:46:51+07:00
⏰ Expiry time: 1769863641
⏰ Expiry human readable: 2026-01-31T19:47:21+07:00
📤 Sending request:
{"client_id":"7e9eaa3e-4fb7-43e7-8103-c89da353c098","device_token":"6cfd3f74a3a1
3a9bf435d9e70b9cd3c115e7c4f216da9f5cf992961a3d44af6b","expiry_time":1769863641,"
purpose":"bank_transaction","timestamp":1769863611,"user_id":"16bde99f2b41b26c8c
3583f1273cc446502b92b33da355696fb3342bf5f8e2f1"}
🌐 Sending to: http://localhost:8080/client/request-otp
✅ OTP request sent successfully
📋 Response:
  Transaction ID: 1e0efbf1-4e8c-4dc6-833d-814be4554aad
  Status: otp_generated
  Expires at: 2026-01-31T19:47:21+07:00

📢 OTP has been sent to the user's GUI for signing.
Waiting for verification result...
(Run 'lamport-otp client-listen' in another terminal to monitor results)
```

```
56e7c7fa73","level":"info"
:46:40Z"}
INFO: 2026/01/31 19:46:40
56e7c7fa73","data":"ping",
time":"2026-01-31T12:46:40
INFO: 2026/01/31 19:46:42
56e7c7fa73","data":"server
rver","time":"2026-01-31T1
INFO: 2026/01/31 19:46:50
56e7c7fa73","level":"info","message":"Sent ping to server","time":"2026-01-31T12
:46:50Z"}
INFO: 2026/01/31 19:46:50 listen.go:975: {"clientID":"eb65a4fd-064e-401c-ad7d-3d
56e7c7fa73","data":"ping","level":"info","message":"Received pong from server","
time":"2026-01-31T12:46:50Z"}
INFO: 2026/01/31 19:46:57 listen.go:975: {"clientID":"eb65a4fd-064e-401c-ad7d-3d
56e7c7fa73","data":"server-ping","level":"info","message":"Received ping from se
rver","time":"2026-01-31T12:46:57Z"}
INFO: 2026/01/31 19:47:00 listen.go:975: {"clientID":"eb65a4fd-064e-401c-ad7d-3d
56e7c7fa73","level":"info","message":"Sent ping to server","time":"2026-01-31T12
:47:00Z"}
INFO: 2026/01/31 19:47:00 listen.go:975: {"clientID":"eb65a4fd-064e-401c-ad7d-3d
56e7c7fa73","data":"ping","level":"info","message":"Received pong from server","
time":"2026-01-31T12:47:00Z"}
```

OTP
A48ZH7ZY
Nonce
a41a54f4eea677f20af604872cd9671f
Timestamp
2026-01-31 19:46:51 WIB (UTC+07:00)
Transaction ID
1e0efbf1-4e8c-4dc6-833d-814be4554aad

✅ OTP Verified 🔑 Fresh key pair generated and stored!

Note: OTP is generated by CLIENT process and appears automatically.

```
e.go:236: Key 6cfd3f74a3a13a9bf
316: Key 6cfd3f74a3a13a9bf435d9
ing to read from file /Users/ku
1c7385380e315b2710715dd650c1a.d
 not found in LevelDB
325: Key 6cfd3f74a3a13a9bf435d9
k key 6cfd3f74a3a13a9bf435d9e70
Lamport-otp/secure_keys/6cfd3f74

:330: Failed to read private key from file /Users/kusuma/Desktop/Lamport-otp/secure_keys/6cfd3f74a3a13a9bf435d9e70b9cd3c115e7c4f216da9f
1860faabde83e87e3640e1c7385380e315b2710715dd650c1a.dat: open /Users/kusuma/Desktop/Lamport-otp/secure_keys/6cfd3f74a3a13a9bf435d9e70b9c
f6b_911c5cf0754bc51860faabde83e87e3640e1c7385380e315b2710715dd650c1a.dat: no such file or directory
1367: Could not load newly stored key for verification: failed to read private key from file /Users/kusuma/Desktop/Lamport-otp/s
3c115e7c4f216da9f5cf992961a3d44af6b_911c5cf0754bc51860faabde83e87e3640e1c7385380e315b2710715dd650c1a.dat: open /Users/kusuma/Des
e or directory
INFO: 2026/01/31 19:47:04 gui.go:1376: Key pair update completed successfully for userID=16bde99f2b41b26c8c3583f1273cc446502b92b33da355696fb3342bf5f8e2f1
INFO: 2026/01/31 19:47:04 gui.go:1388: [DEBUG] Verification successful for deviceToken=6cfd3f74a3a13a9bf435d9e70b9cd3c115e7c4f216da9f5cf992961a3d44af6b
```

7. System Architecture Overview

The implemented system follows a three-tier distributed architecture where signers (ground station personnel) authenticate commands for UAV/ BOAT systems.

COMPONENT 1: SIGNERS GUI (GROUND STATION)

├── Registration interface for signer accounts

├── Real-time OTP display and signing

├── WebSocket connection for command reception

└── Verification status monitoring

COMPONENT 2: CORE REGISTRATION ENGINE

├── Cryptographic key generation (Lamport W-OTS)

├── Device fingerprinting and identification

  ├── Secure key storage and management

└── Server communication for registration

COMPONENT 3: COMMUNICATION INFRASTRUCTURE

├── WebSocket for real-time OTP delivery

```
├── REST API for registration/verification
        ├── Secure session management
└── Automatic reconnection for mission continuity
```

7.1 Two-Path Registration Strategy

The system supports two distinct registration paths for different operational scenarios in UAV/ BOAT command centers:

PATH A: NEW SIGNER REGISTRATION

```
┌── Phase 1: Signer Identification
│   ├── Enter Code Name (min 12 characters)
│   ├── Enter Account ID (9-12 digits, non-sequential)
│   └── Validate signer credentials
│
├── Phase 2: Cryptographic Setup
│   ├── Hash Code Name → 64-character user ID (SHAKE256)
│   ├── Request seed generation from server
│   ├── Receive 256-bit cryptographic seed
│   └── Store seed securely for key generation
│
├── Phase 3: Device Registration
│   ├── Collect device fingerprint (OS, hardware, MAC)
│   ├── Generate consistent device ID
│   ├── Generate Lamport key pair from seed
│   └── Register device with server
│
└── Phase 4: Secure Storage
    ├── Store private key in encrypted LevelDB
    ├── Save registration data locally
    ├── Generate device token for sessions
    └── Initialize key index for OTP signing
```

PATH B: EXISTING SEED LOGIN (SIGNER ROTATION)

```
┌── Phase 1: Seed Recovery
│   ├── Input existing 256-bit seed
│   ├── Validate seed format and integrity
│   └── Proceed to device authentication
│
├── Phase 2: Device Authentication
│   ├── Collect current device fingerprint
│   ├── Generate device ID
│   ├── Authenticate with server using seed
│   └── Receive session token and user ID
│
└── Phase 3: Key Reconstruction
    ├── Regenerate Lamport keys from seed
    ├── Load existing key index
    └── Restore signer session
```

7.2 Cryptographic Key Generation Process

The registration process generates Lamport Winternitz One-Time Signature (W-OTS) keys**:**

Step 1: Seed Processing
Input: Code Name → SHAKE256 → 64-char user ID
Server: Generates 256-bit cryptographic seed
Output: Seed stored securely for key derivation

Step 2: Key Pair Generation
Algorithm: Lamport W-OTS with SHAKE256

Parameters: N=67 segments, W=4, KeyLength=32 bytes
Process: Deterministic key generation from seed + device ID
Output: 67×32-byte private key segments (2144 bytes total)

Step 3: Public Key Derivation
Process: Hash each private key segment to create public key
Root Hash: Merkle tree root of public key segments
Storage: Root hash stored on server for verification

Step 4: Secure Storage
LevelDB: Encrypted private key storage
Local File: Registration data with key index
Server: Public key root hash and device metadata

7.3 OTP Reception and Display Flow

7.3.1 Real-Time OTP Delivery Mechanism

When a UAV/ BOAT requires command authorization, the system follows this flow:

```
Step 1: UAV/ BOAT Command Request
│   ├── UAV/ BOAT generates time-based OTP (8 bytes)
│   ├── System generates cryptographic nonce (16 bytes)
│   ├── Create transaction ID for command tracking
│   └── Timestamp command for audit trail
Step 2: Server Processing
│   ├── Validate UAV/ BOAT credentials
│   ├── Package OTP, nonce, transaction ID, timestamp
│   ├── Route to appropriate signer via WebSocket
│   └── Store command metadata for verification
Step 3: Signer GUI Update
│   ├── WebSocket receives "otp_generated" message
│   ├── Parse OTP, nonce, transaction ID, timestamp
│   ├── Format timestamp for local timezone display
│   └── Update GUI fields with received data
Step 4: Time-Sensitive Display
│   ├── Clear previous OTP data and countdowns
│   ├── Enable "Sign OTP" button for authorization
│   ├── Start 30-second countdown timer
│   └── Display OTP expiration warning
```

7.3.2 Time-Sensitive Authorization Window

Critical for UAV/ BOAT command scenarios where timing is essential:

```
30-SECOND AUTHORIZATION WINDOW:
├── 0-10 seconds: Optimal authorization period
│   ├── OTP fresh and valid
│   ├── Maximum verification time available
│   └── Low risk of timeout
```

```
│
├── 10-20 seconds: Standard authorization period
│   ├── OTP still valid
│   ├── Sufficient time for verification
│   └── Moderate urgency indicated
│
├── 20-30 seconds: Critical authorization period
│   ├── OTP nearing expiration
│   ├── Limited time for verification
│   └── Visual countdown warning signer
│
└── After 30 seconds: Authorization expired
    ├── OTP invalidated
    ├── "Sign OTP" button disabled
    ├── Clear OTP data from display
    └── Require new command request from UAV/ BOAT
```

7.4 OTP Signing and Verification Flow

7.4.1 Signer Authorization Process

When an siggner authorizes a UAV/BOAT command:

```
Phase 1: Signer Action
│   ├── Signer clicks "Sign OTP" button
│   ├── System validates Signer credentials
│   ├── Disable sign button to prevent duplicate clicks
│   └── Display "Verifying..." status
Phase 2: Message Construction
│   ├── Extract OTP from display field (8 bytes)
│   ├── Extract nonce from display field (16 bytes)
│   ├── Concatenate: "OTP:Nonce" (e.g., "a1b2c3d4:e5f6...")
│   └── Prepare message for cryptographic signing
Phase 3: Cryptographic Signing (SIGNER-SIDE)
│   ├── Load private key from local secure storage
│   ├── Generate Lamport signature on message
│   ├── Return 4.32 KB (4,419 bytes) signature
│   └── Zeroize private key copy from memory
Phase 4: Signature Registration
│   ├── Send signature to server for registration
│   ├── Server validates signature against stored public key
│   ├── Store signature in database with metadata
│   └── Mark OTP as "signed and registered"
Phase 5: State Management
│   ├── Store signature in Signer session state
│   ├── Enable "Clear" button for Signer control
│   └── Prepare for verification request by UAV/BOAT
```

7.4.2 The system adheres to the fundamental security principle:

a. Private Key Isolation: The private key SK never leaves the signer's device. Formally:

$$\forall t \in Time, \forall network\ packet\ p: SK \notin content(p)$$

This is ensured through:

- **Local Key Storage**: Private keys stored in LevelDB with device-specific encryption

- **In-Memory Protection**: Key zeroization after use

$$(privateKeyCopy[i][j] \ = \ 0)$$

- **No Server Transmission**: Signing occurs entirely signer-side

b. Signing Process

Message Construction:

Given OTP O and nonce N:

$$M = "OTP:" \parallel O \parallel ":" \parallel N$$

Where ǁ denotes concatenation.

c. Signature Generation (Signer-Side):

The signing function S operates on the signer's device:

$$S: (M, SK, params) \rightarrow \sigma$$

Where:

- SK is the private key loaded from local secure storage
- σ is the Lamport signature (4,419 bytes)
- params includes WOTS parameters ($N = 67, W = 16N = 67, W = 16$)

d. Signature Registration:

The signature is sent to the server for registration:

$$R: (\sigma, M, metadata) \rightarrow storage\_key$$

The server validates σ against the public key PK already stored during registration:

$$Verify(PK, M, \sigma) = 1 \Longrightarrow accept\_signature$$

e. Security Properties of the Workflow

Property 1 (Non-Repudiation):

Once a signature is registered on the server:

$$Pr[Operator\ denies\ signing\ M \mid Verify(PK, M, \sigma) = 1] \leq \epsilon$$

This holds because only the signer possessing SK could generate σ.

Property 2 (Forward Secrecy for Keys):

Even if one OTP signature is compromised:

$$I(SK \mid \sigma_1, \sigma_2, \ldots, \sigma_k) = H(SK) \quad (full\ entropy\ preserved)$$

This is due to the one-time nature of Lamport signatures.

Property 3 (Replay Prevention):

Each signature is uniquely bound to $(M, nonce, timestamp)$:

$$\forall i \neq j: (M_i, N_i, t_i) \neq (M_j, N_j, t_j) \Longrightarrow \sigma_i \neq \sigma_j$$

f. Implementation Details

Signer-Side Signing Process:

```
// Simplified signing workflow

func SignOTP(otp, nonce string) (signature string, error) {

    // 1. Load private key from LOCAL storage

    sk := keyManager.LoadKey(deviceToken, rootHash)
    // 2. Construct message

    message := fmt.Sprintf("OTP:%s:%s", otp, nonce)

    // 3. Generate signature LOCALLY

    signature := lamport.Sign(message, sk)

    // 4. Zeroize key copy from memory

    zeroize(sk)

    // 5. Send signature (NOT private key) to server

    return registerSignature(signature, message)

}
```

7.4.3 Server-Side Verification Process

After signer authorization, the system verifies the command:

```
Step 1: Verification Request
│  ├── Send OTP, nonce, signature, transaction ID to server
│  ├── Include signer credentials (user ID, device token)
│  └── Start verification countdown (30 seconds)
Step 2: Signature Validation
│  ├── Reconstruct message: "OTP:Nonce"
│  ├── Compute SHA3-256 hash of message
│  ├── Extract signature indices from hash (67 segments)
│  ├── Retrieve signer's public key from database
│  └── Verify each signature segment against public key
Step 3: Security Checks
│  ├── Validate key index (prevent replay attacks)
│  ├── Check OTP expiration timestamp
│  ├── Verify transaction ID uniqueness
│  └── Confirm signer authorization permissions
Step 4: Response Processing
│  ├── If valid: Return "OTP Verified on Server"
│  ├── If invalid: Return error with specific reason
│  ├── Include "key_update_allowed" flag if key rotation needed
│  └── Send WebSocket notification to signer GUI
```

a. Server-Side Validation:

The server performs:

- **Signature Verification**: Using stored public key PK

- **Nonce Check**: Ensure nonce hasn't been reused
- **Timestamp Validation**: Check within valid time window
- **Database Registration**: Store $(\sigma, M, metadata)$

b. Storage Schema:

The server stores:

- `transaction_id`: Unique identifier
- `signer_id`: Hashed user identifier
- `otp_code`: The OTP value
- `nonce`: Random nonce
- `signature`: The 4.32KB Lamport signature
- `public_key`: Reference to signer's public key
- `timestamp`: Creation time
- `status`: "signed", "verified", "expired"

7.5 Key Management and Rotation Flow

7.5.1 Dynamic Key Rotation Strategy

The system implements adaptive key rotation for long-term UAV/ BOAT missions:

```
KEY ROTATION TRIGGERS:
├── Usage-Based Rotation
│   ├── After N signatures (configurable threshold)
│   ├── Based on remaining key index capacity
│   └── Automatic trigger from verification response
│
├── Time-Based Rotation
│   ├── Periodic rotation (e.g., every 24 hours)
│   ├── Mission-phase based rotation
│   └── Scheduled maintenance rotations
│
├── Security-Event Rotation
│   ├── After potential compromise detection
│   ├── Signer change or shift rotation
│   └── System anomaly detection
│
└── Manual Rotation
    ├── Signer -initiated key refresh
    ├── Administrative override
    └── Emergency rekeying procedures
```

7.5.2 Key Rotation Execution Flow

When key rotation is triggered:

```
Phase 1: Rotation Trigger
│   ├── Server includes "key_update_allowed": true
│   ├── Signer GUI displays key update status
│   ├── Initiate new key generation process
│   └── Preserve old key for pending verifications
Phase 2: New Key Generation
│   ├── Generate new Lamport key pair from existing seed
│   ├── Create new root hash (Merkle tree)
│   ├── Store new private key in secure storage
│   └── Transmit new public key to server
Phase 3: State Transition
│   ├── Update Signer state with new key index (reset to 0)
│   ├── Maintain old key for backward compatibility
│   ├── Update local registration data
│   └── Confirm successful key storage
Phase 4: Verification and Cleanup
│   ├── Verify new key can be loaded and used
│   ├── Send WebSocket notification of successful update
│   ├── Display success message in Signer GUI
│   └── Schedule old key deletion (after grace period)
```

7.6 Security and Audit Trail Flow

7.6.1 Comprehensive Audit Logging

Every authentication event generates detailed audit records:

```
AUDIT LOG STRUCTURE:
├── Timestamp: Event occurrence time (UTC)
├── Signer ID: 64-character hashed identifier
├── Device ID: Consistent device fingerprint
├── Transaction ID: Unique command identifier
│
```

```
├── OTP: 8-byte one-time password
├── Nonce: 16-byte cryptographic nonce
├── Signature: 4.32 KB (4,419 bytes) Lamport signature
├── Verification Result: Success/Failure status
├── Key Index: Current key usage counter
├── Location: Signer geolocation (if available)
├── Mission ID: Associated UAV/ BOAT mission
├── UAV/ BOAT ID: Specific vehicle identifier
└── Event Type: Registration/Login/Signing/Verification
```

8. Reclaiming disks space:

The advantage of using hash-based one-time signatures such as Winternitz / Lamport OTS lies in their stateless verification and minimal long-term storage requirements.

Each signature is self-contained: verification depends only on the message, the signature, and the corresponding public verification material. No global signer state or counter is required during verification, which simplifies system design and reduces synchronization risk.

From a storage perspective, once a one-time signature has been successfully verified and consumed, the full signature material no longer needs to be retained. It can be safely pruned and replaced by a compact hash commitment (e.g., $H(\mu \parallel \sigma)$) that serves as an immutable audit reference.

This pruning strategy allows the database to:

- Avoid unbounded growth as authorization events accumulate
- Retain verifiable historical integrity
- Preserve correctness for future signature operations, since each OTS key is used exactly once and is cryptographically independent of all others

Because future signatures are derived from **fresh, unlinkable one-time keys**, pruning previously used signatures **does not disrupt** subsequent signing or verification processes.

8.1 Introduction to Stateless Signature Management

The SOTP system produces substantial cryptographic signatures (4,419 bytes each) that present critical storage challenges for UAV/ BOAT systems with extended mission durations. This chapter describes an innovative storage reclamation strategy that replaces 4.32KB signatures with compact 32-byte hash values while maintaining cryptographic integrity and audit trail completeness.

Storage Growth Projection:

```
├── 1 signature = 4.32 KB

├── 100 signatures/day = 432 KB/day
├── 3,000 signatures/month = 12.96 MB/month
├── 36,000 signatures/year = 155.52 MB/year
├── 1,000 users × 100 signatures/day = 432 MB/day
└── 1,000 users × 1 year = 155.52 GB/year
```

8.2 Core Innovation: Stateless Signature Pruning

The implemented system introduces a novel pruning mechanism that reclaims 99.2% of signature storage while maintaining cryptographic integrity and security**:**

Original vs Pruned Storage:

```
├── Original Signature: 4,419 bytes (4.32 KB)
├── Pruned Storage: 32 bytes (0.032 KB)
├── Storage Reduction: 99.2%
└── Space Savings: 138:1 ratio
```

8.3 Pruning Architecture Overview

The pruning system follows a multi-layer architecture that preserves security while optimizing storage:

LAYER 1: Real-Time Signature Tracking

```
│  ├── Bloom filter for rapid duplicate detection
│  ├── Hash-based signature indexing
│  ├── In-memory caching for performance
│  └── Atomic storage operations
```

LAYER 2: Scheduled Pruning Engine

```
│  ├── Background pruning scheduler (5-minute intervals)
│  ├── Expired OTP detection and cleanup
│  ├── Excess signature management (>3 signatures/user)
```

│ └── Hash chain creation for audit trail

LAYER 3: Cryptographic Integrity Preservation

│ ├── SHA-256 signature hash storage (32 bytes)

│ ├── Hash chain for temporal integrity

│ ├── Public key hash for key association

│ └── Transaction ID preservation

LAYER 4: Security Monitoring

│ ├── Signature reuse detection

│ ├── Suspicious activity monitoring

│ ├── Statistical analysis and reporting

│ └── Forensic audit trail generation

8.4 Pruning Workflow

PRUNING CYCLE EXECUTION (EVERY 5 MINUTES):

┌── Phase 1: User Discovery

│ ├── Identify all users with stored signatures

│ ├── Count signatures per user

│ └── Prioritize users with excessive signatures

│

├── Phase 2: Expired OTP Cleanup

│ ├── Identify OTPs beyond validity period (30 seconds)

│ ├── Detect burned OTPs (successfully verified)

│ ├── Preserve metadata while clearing signatures

│ └── Generate hash chain entries

│

```
├── Phase 3: Excess Signature Management
│   ├── Maintain maximum of 3 active signatures per user
│   ├── Remove oldest signatures beyond threshold
│   ├── Preserve signature hashes for security
│   └── Update hash chain
│
└── Phase 4: System Cleanup
    ├── Remove old signature hashes (>90 days)
    ├── Update bloom filters
    ├── Generate pruning statistics
    └── Log pruning activities
```

8.5 Individual Signature Pruning Process

When pruning a single signature, the system executes this **atomic process**:

SIGNATURE PRUNING ALGORITHM:

Step 1: Signature Hash Generation

Input: Original signature (4,419 bytes)

Process: SHA-256 cryptographic hash

Output: 32-byte signature hash (64 hex characters)

Step 2: Duplicate Prevention Check

Process: Check if hash already exists in `UsedSignature` table

Security: Prevents signature reuse attacks

Action: Skip if duplicate detected (already stored)

Step 3: Secure Storage

Store: 32-byte hash in `UsedSignature` table

Metadata: `UserID, OTPCode, PublicKeyHash, TransactionID`

Timestamp: Creation time for audit trail

Step 4: Hash Chain Construction

Input: Previous hash chain entry (if exists)

Process: Concatenate with current OTP data

Algorithm: SHA-256 of `[prev_hash|user_id|otp_code|...]`

Output: New hash chain entry

Step 5: Signature Clearance

Action: Set signature field to empty string

Retention: Keep OTP record with all other metadata

State: Mark OTP as `"pruned"` but retain audit trail

Step 6: Transaction Commit

Ensure: All operations succeed or rollback

Atomicity: Complete transaction or none

Integrity: Maintain database consistency

8.6 Cryptographic Integrity Preservation

The system implements a cryptographic hash chain that preserves temporal relationships between pruned signatures:

HASH CHAIN CONSTRUCTION:

Input Components (Concatenated):

├── Previous Hash: Hash of previous chain entry (if exists)

├── User ID: 64-character user identifier

├── OTP Code: 8-byte one-time password

├── Nonce: 16-byte cryptographic nonce

├── Original Signature: 4,419-byte Lamport signature

├── Transaction ID: Unique command identifier

├── Issued At: Timestamp of OTP generation

├── Expires At: Timestamp of OTP expiration

├── Burned Status: Boolean (true/false)

├── Verified Status: Boolean (true/false)

└── Public Key: 32-byte hash of public key

Hashing Process:

```
Hash = SHA-256(prev_hash + "|" + user_id + "|" + otp_code
+ ...)
```

Chain Properties:

├── Immutability: Each entry cryptographically linked to previous

├── Temporal Order: Chain order reflects signature sequence

├── Integrity: Any tampering detectable through hash mismatch

└── Compactness: Each chain entry = 64 bytes (32-byte hash + metadata)

8.7 Bloom Filter Optimization

For rapid duplicate detection**,** the system uses probabilistic bloom filters**:**

BLOOM FILTER CONFIGURATION:

├── False Positive Rate: 1% (configurable)

├── Expected Items: 10,000 signatures per user

├── Filter Size: Calculated based on false positive rate

├── Hash Functions: Multiple hash indices for each signature

└── Memory Efficiency: ~1.2 MB per user for 10k signatures

Operation Flow:

1. Signature Received → Calculate SHA-256 hash

2. Check Bloom Filter → Rapid probabilistic check

3. If "definitely not present" → New signature (fast path)

4. If "might be present" → Database verification (slow path)

5. Add to Bloom Filter → Update filter bits

Performance Benefits:

├── Query Time: $\boldsymbol{O}(\boldsymbol{k})$ where $\boldsymbol{k}$ = number of hash functions

├── Memory Usage: Minimal compared to full signature storage

├── Scalability: Handles thousands of signatures efficiently

└── Security: Prevents signature reuse attacks

8.8 Security Considerations

Signature Reuse Prevention, The system implements multiple layers of defense against signature reuse attacks:

REUSE PREVENTION MECHANISMS:

Layer 1: Atomic Database Check

├── Database constraint: `UNIQUE(signature_hash, user_id)`

├── Transaction isolation: Prevents race conditions

└── Immediate detection: Reject duplicate signatures

Layer 2: Bloom Filter Screening

├── In-memory probabilistic check

├── Rapid detection of potential duplicates

└── Reduces database load for common cases

Layer 3: Temporal Analysis

├── Timestamp validation

├── OTP expiration enforcement

└── Sequence number verification

Layer 4: Statistical Monitoring

├── Rate limiting per user

├── Pattern analysis for anomalies

└── Alert generation for suspicious activity

8.9 Forensic Audit Trail

Despite pruning, the system maintains a comprehensive forensic audit trail**:**

AUDIT TRAIL COMPONENTS:

1. Hash Chain Entries

├── Cryptographic proof of signature sequence

├── Temporal ordering preserved

├── Immutable record of all pruned signatures

2. `UsedSignature` Records

├── 32-byte signature hashes

├── Associated metadata (user, transaction, timestamp)

├── Public key references for verification

3. Statistical Logs

├── Pruning operations timestamps

├── Storage reclamation metrics

├── Suspicious activity alerts

4. Integrity Verification

├── Periodic hash chain validation

├── Cross-reference with OTP records

├── Consistency checks between systems

8.10 Storage Reclamation Metrics

The pruning system delivers dramatic storage reductions**:**

STORAGE SAVINGS CALCULATION:

For 1,000 users with 100 daily signatures:

Before Pruning:

├── Daily: 1,000 × 100 × 4.32 KB = 432,000 KB (432 MB)

├── Monthly: 432 MB × 30 = 12.96 GB

├── Yearly: 12.96 GB × 12 = 155.52 G

After Pruning:

├── Daily: 1,000 × 100 × 0.032 KB = 3,200 KB (3.2 MB)

├── Monthly: 3.2 MB × 30 = 96 MB

├── Yearly: 96 MB × 12 = 1.15 GB

Storage Reduction:

├── Daily: 432 MB → 3.2 MB (99.26% reduction)

├── Monthly: 12.96 GB → 96 MB (99.26% reduction)

├── Yearly: 155.52 GB → 1.15 GB (99.26% reduction)

└── Space Ratio: 138:1 compression

8.11 Performance Impact

The pruning system maintains high performance despite cryptographic operations:

PERFORMANCE METRICS:

Pruning Operations:

├── Signature Hashing: < 1 ms (SHA-256)

├── Database Insert: 2-5 ms (optimized indexes)

├── Bloom Filter Update: < 0.1 ms

├── Hash Chain Update: 1-2 ms

└── Total Pruning Time: 5-10 ms per signature

System Overhead:

├── CPU Usage: < 1% for pruning operations

├── Memory: ~1.2 MB/user for bloom filters

├── Database: Minimal index overhead

└── Network: No additional traffic

Scaling Characteristics:

├── Linear scaling with user count

├── Constant-time per signature operation

├── Batch processing for efficiency

└── Background scheduling minimizes impact

8.12 Implementation for UAV/ BOAT Systems

For resource-constrained UAV/ BOAT systems, the pruning architecture offers critical advantages:

EMBEDDED SYSTEM BENEFITS:

1. Minimal Storage Requirements

├── Flash memory conservation

├── Extended mission duration

├── Reduced storage hardware costs

2. Power Efficiency

├── Reduced write operations to flash

├── Lower power consumption for storage

├── Extended battery life for UAVs/ BOATs

3. Real-Time Performance

├── Fast signature verification

├── Minimal processing overhead

├── Predictable timing behavior

4. Reliability

├── Reduced wear on flash memory

├── Lower risk of storage exhaustion

├── Graceful degradation under stress

8.13 Recovery and Disaster Planning

RECOVERY MECHANISMS:

1. Hash Chain Reconstruction

├── Cryptographic proof of all signatures

├── Temporal sequence verification

├── Integrity validation

2. Cross-Validation

├── Match hash chain with OTP metadata

├── Verify public key associations

├── Reconstruct audit trail

3. Backup Strategies

├── Periodic hash chain exports

├── Off-site storage of critical hashes

├── Redundant storage of `UsedSignature` table

4. Forensic Analysis

├── Complete audit trail from hashes

├── Timeline reconstruction

├── Evidence preservation for investigations

8.14 Disaster Recovery Scenarios

The system supports multiple disaster recovery scenarios**:**

DISASTER RECOVERY:

Scenario 1: Database Corruption

├── Reconstruct from hash chain backups

├── Validate against OTP metadata

├── Restore signature tracking

Scenario 2: Storage Failure

├── Rebuild bloom filters from database

├── Recalculate hash chains from logs

├── Resume normal operations

Scenario 3: Security Breach

├── Cryptographic proof of all transactions

├── Tamper detection via hash chain

├── Forensic analysis from hashes

Scenario 4: System Migration

├── Export hash chains and signature hashes

├── Import to new system with integrity checks

├── Resume operations with full history

8.15 Conclusions

The Signature Pruning and Storage Reclamation System represents a significant advancement in SOTP storage management. By reducing signature storage by 99.2% while maintaining full cryptographic integrity, the system enables scalable deployment of SOTP for UAV/ BOAT systems.

1. **Massive Storage Reduction**: 4.32 KB → 32 bytes per signature
2. **Cryptographic Integrity Preservation**: Hash chains ensure tamper-evidence
3. **Real-Time Performance**: Minimal impact on command authorization latency
4. **Security Enhancement**: Multiple layers of signature reuse prevention
5. **Forensic Capability**: Complete audit trail despite pruning
6. **Scalability**: Efficient handling of thousands of users and signatures

This system provides a practical solution to the storage challenges of post-quantum signatures challange, making them viable for high-frequency, resource-constrained environments like UAV/ BOAT command systems while maintaining the security properties that make Lamport signatures valuable for post-quantum authentication.

9. References:

1. https://en.wikipedia.org/wiki/Byzantine_fault
2. https://en.wikipedia.org/wiki/Single_point_of_failure
3. https://ieeexplore.ieee.org/document/8806165
4. https://www.navy.mil/DesktopModules/ArticleCS/Print.aspx?PortalId=1&ModuleId=523&Article=2387354
5. https://www.researchgate.net/publication/390486611_Artificial_Intelligence_ATM_OTP_in_Banking_Enhancing_Efficiency_Security_and_Customer_Experience
6. https://www.nist.gov/news-events/news/2024/08/nist-releases-first-3-finalized-post-quantum-encryption-standards
7. https://en.wikipedia.org/wiki/Hash_chain
8. https://en.wikipedia.org/wiki/Merkle_tree
9. https://en.wikipedia.org/wiki/Lamport_signature