Sovereign Key Management in Post-Quantum Local-First Applications

Zach Kelling

Lux Industries

2025

Abstract

We describe a key management architecture for local-first applications on the Lux Network where users hold their own keys and service providers cannot decrypt user data. The architecture uses Argon2id for passphrase-based root key derivation, ML-KEM-1024 (FIPS 203) for post-quantum key encapsulation, AES-256-GCM for symmetric encryption, and ML-DSA-87 (FIPS 204) for post-quantum digital signatures. Vault sharing uses ML-KEM envelope encryption; threshold control uses CGGMP21 (ECDSA) and FROST (EdDSA) with user-controlled signer sets. Fully homomorphic encryption over lattices enables private computation on encrypted data without provider access. We contrast this sovereign model with the regulated custody model where operator decrypt capability is a legal requirement, and show that both models share the same cryptographic primitives but differ in trust assumptions.

Contents

1 The Sovereign App Model

1.1 Local-First Principle

In the sovereign model, the user's device is the primary data store. Data is encrypted at rest on the device; the provider never sees plaintext. The chain provides verifiable checkpoints and ordering, but is not the primary storage layer for user data.

Definition 1.1 (Sovereign Application)**.** *An application is* sovereign *if:*

1. *The user holds the root decryption key.*
2. *The provider cannot derive the root key from any data it holds.*
3. *The application functions offline (reads and writes to local state).*
4. *Sync, storage, and recovery services are replaceable without data loss.*

1.2 Chain Anchors

The blockchain provides three services to sovereign applications:

1. **Ordering**: a total order on state transitions, preventing equivocation.
2. **Availability**: encrypted data blobs can be anchored on-chain for disaster recovery (the chain stores ciphertext, not plaintext).
3. **Capability tokens**: on-chain tokens representing access permissions, revocable by the key holder.

The chain never stores plaintext user data. It stores commitments (hashes), encrypted envelopes, and capability references.

1.3 Provider Agnosticism

The sync layer (cloud backup, multi-device replication) is a replaceable service. The provider receives only encrypted envelopes. Switching providers requires re-uploading encrypted blobs; no re-encryption is needed because the provider never held a key.

2 Key Architecture

2.1 Root Key Derivation

The user's root key is derived from a passphrase using Argon2id [4]:

$$\text{rootKey} = \text{Argon2id}(\text{passphrase}, \text{salt}, t = 3, m = 65536, p = 4) \tag{1}$$

Parameters: 3 iterations, 64 MiB memory, 4 parallel lanes. The salt is generated once and stored alongside the encrypted vault header (the salt is not secret). The output is 32 bytes, used as the master symmetric key.

The root key never leaves the device. It is held in memory only during an active session and zeroed on lock/logout.

2.2 Per-Vault Data Encryption Key

Each vault (a logical container for related data—e.g., a company's cap table, a personal document set) has its own DEK. The DEK is a random 256-bit key, encrypted to the vault owner's ML-KEM-1024 public key:

$$\text{encDEK} = \text{ML-KEM-1024.Encapsulate}(\text{ownerPubKey}) \rightarrow (\text{encapsulated}, \text{sharedSecret}) \tag{2}$$

The shared secret serves as the DEK. The encapsulated key blob is stored alongside the vault metadata.

To share a vault with another user, the DEK is re-encapsulated to the recipient's ML-KEM-1024 public key. Each recipient gets their own encapsulated key blob; the DEK itself is never transmitted in plaintext.

2.3 Per-Document Encryption

Individual documents are encrypted with AES-256-GCM using the vault DEK and a random 12-byte nonce per document. This is implemented in `lux/transfer/pkg/crypto/envelope.go`:

Listing 1: envelope.go: ML-KEM + AES-256-GCM envelope

```
func Seal(plaintext []byte,
          recipientPubKey []byte) (*Envelope, error) {
    pk, err := mlkem.PublicKeyFromBytes(
        recipientPubKey, mlkem.MLKEM1024)
    // ML-KEM encapsulate: shared secret + encapsulated key
    encapsulated, sharedSecret, err := pk.Encapsulate()
    // AES-256-GCM with shared secret as key
    block, err := aes.NewCipher(sharedSecret[:32])
    gcm, err := cipher.NewGCM(block)
    nonce := make([]byte, gcm.NonceSize())
    rand.Read(nonce)
    ciphertext := gcm.Seal(nil, nonce, plaintext, nil)
    return &Envelope{
        Ciphertext:      ciphertext,
        EncapsulatedKey: encapsulated,
        Nonce:           nonce,
        Algorithm:       AlgoMLKEM1024_AES256GCM,
    }, nil
}
```

Decryption requires the recipient's ML-KEM-1024 private key to decapsulate the shared secret, then AES-256-GCM decryption:

Listing 2: envelope.go: decryption path

```
func Open(env *Envelope,
          recipientPrivKey []byte) ([]byte, error) {
    sk, err := mlkem.PrivateKeyFromBytes(
        recipientPrivKey, mlkem.MLKEM1024)
    sharedSecret, err := sk.Decapsulate(env.EncapsulatedKey)
    block, err := aes.NewCipher(sharedSecret[:32])
    gcm, err := cipher.NewGCM(block)
    plaintext, err := gcm.Open(
        nil, env.Nonce, env.Ciphertext, nil)
    return plaintext, nil
}
```

2.4 Post-Quantum Signatures

All envelopes can be signed using ML-DSA-87 (FIPS 204 [2]). The signer's public key is included in the envelope for verification:

Listing 3: envelope.go: ML-DSA-87 signing

```
func Sign(data []byte,
          signerPrivKey []byte) ([]byte, error) {
```

```
    sk, err := mldsa.PrivateKeyFromBytes(
        mldsa.MLDSA87, signerPrivKey)
    return sk.Sign(data)
}
```

ML-DSA-87 provides NIST Security Level 5—equivalent security to AES-256 against quantum adversaries.

2.5 Algorithm Summary

Function	Algorithm	Standard	Security Level
Key derivation	Argon2id	RFC 9106	Memory-hard
Key encapsulation	ML-KEM-1024	FIPS 203	NIST Level 5
Symmetric encrypt	AES-256-GCM	FIPS 197	256-bit
Digital signature	ML-DSA-87	FIPS 204	NIST Level 5
Threshold ECDSA	CGGMP21	—	secp256k1
Threshold EdDSA	FROST	RFC 9591	Ed25519

Table 1: Cryptographic algorithms and their security levels.

3 FHE for Private Computation

3.1 Motivation

Some computations require operating on data from multiple parties without revealing individual values. Examples in the securities domain:

- Accreditation verification: does an investor's income exceed a threshold without revealing the exact amount?
- Private voting: aggregate votes without revealing individual choices.
- Compliance checks: verify aggregate investment limits across multiple accounts.

3.2 Implementation

The FHE module in `lux/transfer/pkg/crypto/fhe.go` uses lattice-based fully homomorphic encryption from the `luxfi/fhe` library:

Listing 4: fhe.go: parameter initialization

```
func NewFHEKey() (*FHEKey, error) {
    params, err := fhe.NewParametersFromLiteral(fhe.PN10QP27)
    intParams, err := fhe.NewIntegerParams(params, 2)
    kg := fhe.NewKeyGenerator(params)
    sk, pk := kg.GenKeyPair()
    return &FHEKey{
        params: params, intParams: intParams,
        sk: sk, pk: pk,
    }, nil
}
```

The parameter set `PN10QP27` provides approximately 128-bit security. Integer operations support 64-bit unsigned values, sufficient for monetary amounts and share counts.

3.3 Encrypted Comparison

The core operation is homomorphic comparison, enabling threshold checks on encrypted data:

Listing 5: fhe.go: encrypted comparison

```
func CompareEncrypted(a, b *FHECiphertext,
                      key *FHEEvalKey) (int, error) {
    // Computed homomorphically, then single-bit
    // results are decrypted.
    // In production the final decryption would
    // be done via MPC.
}
```

The comparison result is a single encrypted bit. In the sovereign model, the final decryption is distributed via MPC among user-controlled signers, so no single party learns the underlying values.

3.4 Integration with Sovereign Keys

FHE ciphertexts are encrypted under the vault's FHE public key (generated alongside the ML-KEM key pair). The FHE secret key is held by the user or split via MPC among user-designated parties. The provider stores ciphertexts and can perform homomorphic operations, but cannot decrypt the results.

4 MPC for Threshold Control

4.1 Same Primitives, Different Trust

The sovereign model uses the same threshold signing protocols as the regulated model (CG-GMP21 for ECDSA, FROST for EdDSA). The difference is who controls the signers:

Property	**Regulated (Liquidity)**	**Sovereign (Lux)**
Shard holders	Operator + HSM + cold	User + trusted parties
Threshold	2-of-3 (operator is 1)	User-defined t-of-n
Operator shards	1 (below threshold)	0
Recovery	Operator cold storage	Social recovery
Key generation	Operator-initiated DKG	User-initiated DKG

Table 2: MPC trust model comparison.

4.2 User-Controlled Signers

In the sovereign model, the user initiates distributed key generation (DKG) and designates the signer set. Typical configurations:

- **Personal**: 2-of-3 with user device, backup device, and a recovery share held by a trusted contact.
- **Organization**: 3-of-5 with board members as shard holders.
- **DAO**: m-of-n with on-chain governance controlling signer rotation.

The provider holds zero shards. The provider cannot sign, cannot veto, and cannot participate in signing.

4.3 Social Recovery

If the user loses their device, recovery uses threshold shares distributed to trusted parties at vault creation time. The recovery protocol:

1. User contacts t of their n designated recovery parties.
2. Each party authenticates the user (out-of-band: video call, in-person, pre-shared secret).
3. Each party submits their recovery share to the new device.
4. The new device reconstructs the root key from t shares.
5. The root key decrypts the vault DEK; all data is accessible.

No operator involvement. No "forgot password" flow that bypasses encryption. If the user cannot reach t recovery parties, the data is permanently inaccessible. This is the intended behavior: data sovereignty means the user bears the responsibility.

5 Contrast with Regulated Model

The regulated custody model (deployed by a regulated financial services provider for the the regulated ATS) and the sovereign model (deployed on Lux Network) use identical cryptographic primitives but make opposite trust assumptions.

Property	**Regulated (Liquidity)**	**Sovereign (Lux)**
Data encryption	AES-256-GCM, per-org DEK from HMAC-SHA256	AES-256-GCM, per-vault DEK from ML-KEM-1024
Master key holder	Operator (Google Cloud KMS, HSM-backed)	User (Argon2id from passphrase)
Can operator decrypt?	Yes (required by SEC 17a-4)	No (by design)
Key encapsulation	Not used (HMAC derivation)	ML-KEM-1024 (FIPS 203)
Signatures	HMAC-SHA256 (migrating to ML-DSA-87)	ML-DSA-87 (FIPS 204)
Threshold signing	CGGMP21/FROST, operator holds 1 shard	CGGMP21/FROST, operator holds 0 shards
Recovery	Operator cold storage + KMS	Social recovery (t-of-n shares)
Data loss on key loss?	No (operator can re-derive DEK)	Yes (if $< t$ recovery parties available)
Regulatory compliance	SEC 17a-4, FINRA 4511, BSA	User responsibility

Table 3: Full comparison of regulated and sovereign key architectures.

The choice between models is a policy decision, not a cryptographic one. The same `luxfi/crypto` library implements both. The difference is in key hierarchy: who holds the root, and whether the provider can derive decryption keys.

Invariant 5.1 (Model Exclusivity)**.** *A deployment is either regulated or sovereign. There is no hybrid where the operator "sometimes" can decrypt. The trust model is determined at deployment time by key hierarchy configuration.*

6 Security Analysis

6.1 Post-Quantum Security

ML-KEM-1024 provides IND-CCA2 security under the Module Learning With Errors (MLWE) assumption at NIST Security Level 5 [1]. This means a quantum adversary with access to

a cryptographically relevant quantum computer cannot distinguish ML-KEM ciphertexts from random, and therefore cannot recover shared secrets or DEKs.

ML-DSA-87 provides EUF-CMA security under the Module Short Integer Solution (MSIS) and MLWE assumptions at NIST Security Level 5 [2].

Theorem 6.1 (Post-Quantum Confidentiality)**.** *Under the MLWE assumption with parameters as specified in FIPS 203 for ML-KEM-1024, the probability that a quantum polynomial-time adversary $\mathcal{A}$ can distinguish the encryption of a chosen message from random is:*

$$Adv^{IND\text{-}CCA2}_{ML\text{-}KEM\text{-}1024}(\mathcal{A}) \leq negl(\lambda)$$

where λ is the security parameter (256 bits for Level 5).

6.2 Forward Secrecy

Each envelope uses a fresh ML-KEM encapsulation, producing an ephemeral shared secret. Compromise of the recipient's long-term private key enables decryption of past envelopes (the encapsulated keys are stored alongside ciphertexts). True forward secrecy requires an interactive protocol (e.g., Signal's X3DH adapted for ML-KEM), which is out of scope for this document but planned for the Lux secure messaging layer [7].

For vault-level forward secrecy, periodic key rotation is supported: the vault DEK is re-generated, all documents are re-encrypted, and the old ML-KEM key pair is destroyed.

6.3 Metadata Protection

The sync protocol transmits encrypted envelopes. The provider observes:

- Envelope size (mitigated by padding to fixed block sizes).
- Timing of uploads/downloads (mitigated by batching).
- Public keys of recipients (inherent to ML-KEM; mitigated by using ephemeral recipient identifiers).

The provider does not observe: plaintext content, file names, document types, or the relationship between documents within a vault.

6.4 Revocation

Access revocation uses on-chain capability tokens. Each vault access grant is represented by an on-chain token. Revoking access burns the token. The revoked party still holds the encapsulated key blob from the previous sharing, but:

1. The vault DEK is rotated after revocation.
2. All documents are re-encrypted with the new DEK.
3. New envelopes are encapsulated only to remaining authorized keys.

The revoked party retains access to data encrypted before revocation (this is inherent to any system where decryption keys were previously distributed). Preventing access to historical data requires re-encryption with key exclusion, which is performed automatically on revocation.

6.5 Security Level Summary

7 Conclusion

The sovereign key architecture on Lux Network ensures that service providers cannot access user data. Users derive root keys from passphrases via Argon2id, encrypt data with ML-KEM-1024 + AES-256-GCM envelopes, and sign with ML-DSA-87—all at NIST post-quantum Security Level 5. Threshold MPC (CGGMP21/FROST) distributes signing authority among user-controlled

Threat	Protection	Assumption
Quantum key recovery	ML-KEM-1024 Level 5	MLWE hardness
Quantum signature forgery	ML-DSA-87 Level 5	MSIS + MLWE hardness
Brute-force root key	Argon2id (64 MiB)	Memory-hard
Threshold signing bypass	CGGMP21/FROST t-of-n	DLP on secp256k1/Ed25519
Provider data access	Envelope encryption	Provider holds no keys
Single point of failure	Social recovery	$\geq t$ parties honest

Table 4: Threat-protection mapping.

parties with zero operator shards. FHE enables computation on encrypted data without decryption. The architecture uses the same cryptographic primitives as the regulated custody model but inverts the trust hierarchy: the user is the root of trust, not the operator.

References

[1] National Institute of Standards and Technology, "FIPS 203: Module-Lattice-Based Key-Encapsulation Mechanism Standard (ML-KEM)," August 2024.

[2] National Institute of Standards and Technology, "FIPS 204: Module-Lattice-Based Digital Signature Standard (ML-DSA)," August 2024.

[3] National Institute of Standards and Technology, "FIPS 197: Advanced Encryption Standard (AES)," November 2001.

[4] A. Biryukov, D. Dinu, D. Khovratovich, "Argon2: New Generation of Memory-Hard Functions for Password Hashing and Other Applications," RFC 9106, September 2021.

[5] R. Canetti, R. Gennaro, S. Goldfeder, N. Makriyannis, U. Peled, "UC Non-Interactive, Proactive, Threshold ECDSA with Identifiable Aborts," in *Proceedings of the 2020 ACM CCS*, 2020.

[6] C. Komlo and I. Goldberg, "FROST: Flexible Round-Optimized Schnorr Threshold Signatures," in *Selected Areas in Cryptography (SAC)*, 2020.

[7] Lux Industries, "Secure Messaging Protocol for Lux Network," Technical Report, 2026.