

Cowboy Secrets: CBSS

Technical Whitepaper

Charles DePue, Cowboy Foundation

July 14, 2026 — Draft v0.1.1

Abstract

This document specifies the **Cowboy Secret Service (CBSS)**: the chain-native subsystem that lets account owners store API keys, model credentials, webhook tokens, and other application secrets on Cowboy and have them released, on demand, into the address space of a runner dispatched to execute a specific job for a specific actor. The authorized runner and its host can observe plaintext during execution; the chain, dispatcher, and any individual proxy cannot decrypt it on their own.

CBSS is a **threshold identity-based encryption (IBE)** scheme over BLS12-381, operated by a permissioned set of staked **CBSS Proxies** (the `cbssd` daemon). Each account has a per-account **release key** generated by a Distributed Key Generation (DKG) ceremony. To wrap a secret, the owner's CLI computes an identity from the secret's (`account`, `key_hash`, `version`, `wrap_epoch`) tuple, samples a fresh Boneh-Franklin encryption ephemeral r (publishing $U = r \cdot G2_gen$ in the envelope), derives a one-time AES key from $e(I, MPK)^r$, and AES-256-GCM-encrypts a fresh Data Encryption Key (DEK) into a chain-anchored `WrappedDek`. The plaintext payload itself is encrypted under the DEK and stored as a CBFS object. To unwrap, a runner constructs a signed release request, fans out partial-sign RPCs to t of n committee proxies, Lagrange-combines the partial BLS signatures into the threshold signature $\sigma = MSK \cdot I$, and recovers the same AES key via $e(\sigma, U)$ to decrypt the DEK and (separately) the CBFS payload. The ephemeral r is what makes K secret: without it, K would be a public function of the (public) MPK and identity, and the threshold release would gate availability but not confidentiality.

Together with CBFS and the runner marketplace, CBSS gives Cowboy three properties that prior decentralized-secrets systems each provide one of: (1) **chain-derived authorization** for release requests (proxies check the responsible runner, actor, job, and policy against current chain state); (2) **off-chain plaintext** (no validator, dispatcher, or single proxy sees the DEK or payload); (3) **runner-scoped delivery** (secrets are rendered for one runner and one job rather than stored in a shared secret cache). The critical release path requires no runner transaction, but participating proxies asynchronously attempt receipt transactions after serving. Receipt submission is currently best-effort and is not a completeness guarantee. This specification is a complement to the [Cowboy Technical Whitepaper](#) and the [Cowboy Storage Whitepaper](#); where this document and the Technical Whitepaper conflict, the Technical Whitepaper prevails.

Introduction

Autonomous agents on Cowboy routinely call third-party APIs — model providers, search APIs, payment rails, observability backends, MCP tool servers — that authenticate by bearer token.

Without CBSS, a Cowboy actor can get that bearer token in front of a runner only by putting it in calldata, which puts it in front of every validator and every block-explorer scraper for the rest of consensus history. Equivalently, an actor can put the secret in clear in CBFS and trust a single runner not to leak it. Both options collapse to “you trusted somebody,” and the *somebody* is the wrong shape for an autonomous, self-funding, self-scheduling program.

The fundamental problem is that smart-contract platforms have no native notion of conditional plaintext release. Ethereum’s storage is public; even private rollups make every encrypted byte equally available to whoever decrypts the rollup. Validators must agree on state; if validators must agree on plaintext secrets, plaintext secrets are not secret. Conversely, off-chain secret stores (Hashicorp Vault, AWS Secrets Manager, doppler.com) do release plaintext conditionally, but their authorization model is “an HTTPS bearer token from a long-lived service account,” which has no link to the chain’s notion of who is dispatched to do what.

CBSS sits between these two regimes. The **authorization source** is chain state: an actor’s manifest declares which secret keys it may read, the chain records the job and assigned runner, and the runner constructs and signs a release request against that state. The **plaintext path** lives off chain: the runner fans out partial signatures from a threshold committee, combines them locally, and decrypts the DEK and payload entirely in its own process. The **audit mechanism** is on chain when receipt submission succeeds: each participating proxy may submit a chain-verifiable receipt that pairs the partial signature against the proxy’s published Verifiable Secret Sharing (VSS) commitment. Because daemon receipt work can be dropped when its worker pool is saturated, submission errors are only logged, and transaction admission is not execution confirmation, absence of a receipt is not proof that no release occurred. At no point during release does the chain or any single proxy hold the secret.

This document specifies the cryptographic construction, the on-chain instruction set (opcodes 68–84, 152–153, and 158), the **cbssd** proxy daemon, the runner-side **secrets-manager** client, the owner-facing CLI flows, the gas pricing, the slashing model, and the parameter table.

Architectural Overview

This Architectural Overview and **Related Work** are descriptive and non-binding. **Cryptographic Construction** and §§1–15 contain normative requirements. The key words **MUST**, **MUST NOT**, **REQUIRED**, **SHOULD**, **SHOULD NOT**, and **MAY** in this document are to be interpreted as described in RFC 2119 and RFC 8174 when, and only when, they appear in all capitals.

Terminology

- **Secret:** A named, account-owned, versioned byte string (e.g. `OPENAI_API_KEY`). The chain never sees the plaintext.
- **Owner:** The Cowboy account that owns one or more secrets. The owner’s wallet key authorizes `SetSecret`, `UpdateSecretPolicy`, and `Delete*` transactions.
- **Actor / Runner / Job:** As in the Technical Whitepaper. An actor declares secrets it intends to read in its manifest; a job is dispatched to a specific runner; that runner is the one entitled to pull the declared secrets for the duration of the job.
- **CBSS Proxy:** A staked off-chain participant running the `cbssd` daemon. Proxies hold one BLS scalar share per release-key scope they serve and produce partial threshold signatures on demand.
- **Committee:** The ordered set of proxies serving a particular release key. The account default is 4-of-5, matching Commonware’s `N3f1` quorum. The wire format carries override `n` and `t` values, but the production Commonware ceremony derives its polynomial quorum from `N3f1`; an override is usable only when its requested `t` equals that derived quorum.
- **DKG (Distributed Key Generation):** The multi-round protocol that produces the committee’s Master Public Key (MPK), per-proxy shares, and per-proxy VSS commitments without any single proxy ever holding the Master Secret Key (MSK).
- **PSS Reshare:** Proactive Secret Sharing. A protocol that replaces every committee share with a fresh share for the same MSK, optionally over a different committee, without ever materialising the MSK.
- **Identity (I):** A point in G_1 derived deterministically from `(account, key_hash, version, wrap_epoch)`. Each secret version under each wrap epoch has a unique identity.
- **MPK / MSK:** Master Public Key ($MPK \in G_2 = MSK \cdot G_{2_gen}$) and Master Secret Key ($MSK \in \text{Scalar}$, never instantiated). Preserved across reshares.
- **Wrapped DEK:** AES-256-GCM ciphertext of a Data Encryption Key, encrypted under the symmetric key derived from the IBE pairing $e(I, MPK)^r$, where `r` is the owner’s per-wrap encryption ephemeral. The envelope (serialized format version 2) carries a leading 1-byte `version` tag, `ciphertext`, `nonce`, `ephemeral_u` ($U = r \cdot G_{2_gen}$, 96-byte compressed G_2), and `aad`; readers reject unknown tag values.
- **CBFS pointer:** A CBFS object descriptor pointing to the AES-256-GCM ciphertext of the actual secret payload, encrypted under the DEK.
- **Release request:** A runner-signed structure naming the secret, version, actor, runner, job, release nonce, request block, and serve epoch. Hashed and signed under domain separator `cbss/release-request/v1`.
- **Partial signature (σ_i):** $s_i \cdot I$ where `s_i` is a proxy’s share. Verified individually via the pairing $e(\sigma_i, G_{2_gen}) == e(I, S_i)$ against the on-chain VSS commitment `S_i`.

- **Threshold signature (σ):** Lagrange combination of t partial signatures, equal to $MSK \cdot I$.
- **Receipt:** A best-effort chain submission of (`release_request`, `sigma_i`, `proxy_id`, `committee_epoch_at_serve`, `served_at_block`, `proxy_sig`). An accepted receipt earns the proxy a per-receipt fee and becomes a canonical audit record; missing receipts do not prove non-release.
- **CBSS System Actor (0x04):** The Cowboy system actor that owns secret metadata, release keys, proxy registry, DKG-pending records, release receipts, liveness challenges, and slashing.

Key Features

CBSS provides six core technical features:

- **Off-chain plaintext data path.** Wrap is local to the owner; unwrap is local to the runner. The chain stores ciphertext (`WrappedDek` and `cbfs_pointer`). DEKs and bundle entries use Rust `Zeroizing` wrappers, and mutable rendered strings are wiped after use. Checked codec containers are rebuilt and dropped rather than overwritten in place, so the implementation does not guarantee active zeroization of every plaintext allocation.
- **Threshold IBE over BLS12-381.** The same primitive used by drand for time-locked encryption: identity-based encryption where the “identity” is a per-version tuple, encryption uses only the public MPK, and decryption requires t of n proxies to produce partial BLS signatures on the identity. No single proxy holds enough information to decrypt; collusion requires $\geq t$ proxies.
- **Per-account release keys with optional per-secret override.** The default path generates one DKG per account and reuses its MPK for every secret the account wraps. High-value secrets can opt into a separate DKG and custom committee selection. The current production ceremony supports only the Commonware `N3f1` threshold for the selected committee size; arbitrary T/N ratios are not implemented safely.
- **Chain-derived authorization.** Every release is preceded by a runner-signed `SignedReleaseRequest`. Each proxy queries current chain state before serving, and an accepted receipt is revalidated at receipt-submission head. Receipt recording is asynchronous and best-effort, not a prerequisite for returning a partial.
- **Proactive secret sharing.** Reshares are scheduled (every `RESHARE_INTERVAL_BLOCKS`) and triggered by churn; PSS preserves MPK across reshares so historic ciphertexts remain decryptable, but every share is replaced — a previously-compromised share is invalidated by the next reshare.

- **Independent per-proxy receipt submission.** No coordinator runs the receipt phase. Each participating proxy attempts a per-proxy receipt independently, with up to 10 blocks of jitter to smooth chain throughput. Accepted receipts emit `cbss.release.receipt_recorded`; `cbss.secret.released` is emitted when the accepted set reaches threshold. The daemon does not retry failed receipt submissions and may drop work when the receipt worker pool is saturated.

Differences vs. Existing Secret Stores

Aspect	Hashicorp Vault / AWS Secrets Manager	NuCypher / Lit Protocol	CBSS
Authorization root of trust	Service-account token	TaCo-style ABE / external policy	Cowboy chain (runner job dispatch)
Plaintext custodian	The vault server	Threshold KFrag holders	Owner at write, runner at read
Release latency	Single HTTPS RT	Multi-RT to KFrag holders	Parallel QUIC fanout; 5 s per-proxy timeout
Audit trail	Server logs	Off-chain	Best-effort chain receipts + daemon logs
Revocation	Token revoke; immediate	Re-encryption on policy change	Metadata-only; checked on each proxy authorization and accepted receipt
Re-wrap on policy change	None	Yes (by design)	Never (per-account release key + ACL at read)
Committee rotation	N/A	Manual	Block-scheduled PSS reshare
Proxy slashing	None	None	On-chain SlashCbssProxy evidence

Aspect	Hashicorp Vault / AWS Secrets Manager	NuCypher / Lit Protocol	CBSS
Native to actor model	No	No	Yes — <code>secrets=[KEY]</code> kwarg on runner ops

Three properties are jointly load-bearing for Cowboy and jointly absent from every alternative above:

1. **No re-encryption on ACL change.** CBSS uses a fixed per-account release key. Proxies validate `policy.actors` before serving, and the node validates it again for any submitted receipt. Adding or removing an actor is a metadata-only `UpdateSecretPolicy` instruction; no DEK is re-wrapped and no payload is re-uploaded.
2. **Scope-locked request.** A release request names one (actor, runner, job, secret, version, serve epoch) tuple with a release nonce. A successfully recorded receipt encodes that scope. If receipt submission fails, the daemon audit log is the only release-side record.
3. **No chain transaction on the critical read path.** The owner submits lifecycle transactions at write time; the runner submits zero transactions at read time. Participating proxies attempt receipt transactions asynchronously after serving, so chain inclusion is not on the runner’s latency path.

Related Work

Centralised Secret Stores (Vault, AWS Secrets Manager, Doppler, 1Password Secrets Automation)

Hashicorp Vault is the canonical model: a single service that stores ciphertext, authenticates clients via short-lived tokens, and releases plaintext over HTTPS. AWS Secrets Manager and Doppler are operationally equivalent. They are excellent at what they do — bearer-token authorization, zero-config rotation, fine-grained policy — and absolutely unsuitable for a public blockchain’s autonomous-agent layer:

- **Single point of compromise.** A vault breach exposes every secret it holds.
- **No on-chain authorization link.** A Vault token says “this account has read access”; it has no way to say “this runner address is dispatched to this job at this block height.”
- **No native audit on the consumer chain.** Vault audit logs live in the vault.

- **Operationally Web2.** A Cowboy actor that pulled secrets from Vault would need to bridge a service-account token into chain identity, defeating the point.

Threshold Cryptography for Decentralised Secrets (Lit Protocol, NuCypher / TACo, drand timelock)

NuCypher (now Threshold Network’s TACo) and Lit Protocol both use threshold cryptography to release secrets contingent on policy. They are conceptually closest to CBSS:

- **Lit Protocol** uses a permissioned BLS threshold network. A user encrypts under the network’s MPK with an attached “access control condition” (an EVM `eth_call`-style predicate); to decrypt, a client requests partial decryptions from t of n Lit nodes, which each evaluate the condition before contributing. The trust model is similar to CBSS but the on-chain anchoring is weaker — the access condition is evaluated by each Lit node independently against any chain it can read, with no on-chain receipt that the release happened.
- **NuCypher / TACo** uses proxy re-encryption (ECIES-based KFrags). Granting access requires generating new re-encryption keys and distributing them; revoking access requires re-encryption. The model is elegant for sharing-style workloads (Alice grants Bob access to a document) but expensive for autonomous-agent workloads where an actor’s manifest may decide moment-to-moment which secrets it needs and the access list changes frequently.
- **drand timelock** uses tlock (Cryptology ePrint 2023/189, presented at Real World Crypto 2023) — threshold IBE keyed on future round numbers — to encrypt to a future time. CBSS reuses the same construction (BLS12-381 hash-to-curve identity + pairing-based KEM + AEAD) but keys identities on per-version-tuple rather than future rounds, and adds chain-anchored authorization, audit, and slashing.

CBSS adopts the threshold-IBE part of drand’s tlock and the chain-policy part of Lit’s design, but anchors authorization to Cowboy’s job-dispatch + runner-stake model rather than to an arbitrary `eth_call` predicate. Receipts and slashing are first-class.

Multi-Party Computation Secret Sharing (TSS Vaults, Pyrofex, ZenGo)

Threshold Signature Schemes (TSS) used by custody products like Fireblocks, ZenGo, and Coinbase MPC are mechanically related: they share a private key across nodes and produce signatures via threshold protocols. They are designed for *signing*, not for *encryption-decryption-of-arbitrary-payloads*. Re-purposing a TSS vault for secret release would require either re-encrypting payloads under a derived symmetric key the TSS produces (essentially CBSS’s construction in different clothing) or signing each payload-release request individually (functionally equivalent

but with worse parallelism, since signing requires interactive rounds where IBE decryption can pre-commit identities).

CBSS uses BLS-IBE rather than TSS-ECDSA because the algebra is simpler (single pairing equation per partial verification, non-interactive after DKG), the verification is one pairing per partial (vs. multi-round MPC for ECDSA), and reshare is a clean PSS protocol on Shamir polynomials rather than a custom TSS resharing dance.

Hardware-Rooted Approaches (TEE-Sealed Vaults, Intel SGX SealedData)

Sealing secrets in a TEE (Intel SGX, AMD SEV-SNP, AWS Nitro) gives strong custody guarantees: the secret is unsealed only inside attested hardware. Cowboy already has a TEE attestation primitive; combining it with a centralized vault is straightforward. The reasons CBSS does not stop there:

- **TEE attacks.** Repeated side-channel breakthroughs against SGX (Foreshock, $\mathcal{A}$ PIC, Downfall, etc.) make any “TEE-only” custody model fragile. Threshold cryptography is a structural defense — compromising any $t-1$ proxies still discloses zero plaintext.
- **No vendor lock-in.** A multi-vendor proxy fleet running pure software is a deliberate property; threshold IBE is hardware-agnostic.
- **TEE composes additively.** A secret may carry `policy.tee_required = true` to request a runner attestation gate; the proxies themselves are not required to run in TEEs. The current gate is hardware-rooted for Nitro but still has a self-asserted soft-fail binding path for SGX, TDX, and SEV-SNP (§12.9).

Tabular Summary

System	Custody model	Per-release chain action	Audit on chain	Re-wrap on ACL change	Native to L1 actor model
CBSS	Threshold IBE, t -of- n staked proxies	Best-effort proxy receipts after serve	Accepted per-proxy + threshold events	Never	Yes
Hashicorp Vault	Centralised	N/A (off-chain)	No	No	No

System	Custody model	Per-release chain action	Audit on chain	Re-wrap on ACL change	Native to L1 actor model
AWS Secrets Manager	Centralised	N/A (off-chain)	No (CloudTrail only)	No	No
Lit Protocol	Threshold BLS (PoA committee)	None enforced	Off-chain	No	No
NuCypher / TACo	Proxy re-encryption (KFrag holders)	KFrag generation per grant	Optional	Yes (re-encryption)	No
drand timelock	Threshold IBE (round identities)	None	None	N/A	No
Fireblocks / ZenGo MPC	TSS (key sharded for signing)	None	Off-chain	N/A (signing primitive)	No
TEE-sealed vault	Single TEE	Attestation per release	Optional	Optional	No

Cryptographic Construction

CBSS implements the threshold IBE construction from drand’s `tlock` paper, specialised to Cowboy’s per-secret-version identities and bound to chain-anchored authorization. This section restates the math in terms of CBSS objects; cryptographers should treat it as a presentation of standard threshold-BLS / Boneh-Franklin IBE rather than a novel scheme.

Curve and Primitives

The curve is **BLS12-381**. We use:

- **G1**: 48-byte compressed points, used for identities I and partial / aggregate signatures σ_i , σ .

- **G2**: 96-byte compressed points, used for the generator **G2_gen**, the master public key **MPK**, and per-proxy VSS commitments **S_i**.
- **Gt**: target group of the pairing $e: G1 \times G2 \rightarrow Gt$.
- **Scalar**: **Fr** (the prime field of order **r**), used for shares **s_i** and the (notional) master secret **MSK**.

Hash-to-curve uses RFC 9380 with the suite tag **BLS12381G1_XMD:SHA-256_SSWU_RO_** and the CBSS domain **cbss/ibe/v1**. Symmetric encryption is **AES-256-GCM**; key derivation is **HKDF-SHA-256**.

Identity Construction

For a secret version (**account**, **key_hash**, **version**, **wrap_epoch**):

```
canonical(SecretId) = account || key_hash           -- 52 bytes
base_aad             = canonical(SecretId)
                    || u64_le(version)
                    || u64_le(wrap_epoch)           -- 68 bytes total
I                    = hash_to_G1(base_aad, domain = "cbss/ibe/v1")
```

The identity derives from the 68-byte **base_aad** only. The AEAD AAD extends it with the per-wrap encryption ephemeral: **aad** = **base_aad** || **compress(U)** (68 + 96 = 164 bytes; see Encryption below).

Notation: **U** denotes the **G2** group element and **ephemeral_u** = **compress(U)** its stored 96-byte canonical compressed encoding; formulas over the stored field use **ephemeral_u** directly. In **base_aad**, **u64_le(version)** is the **secret version number** — not the envelope’s leading 1-byte format-version tag, which is never part of the AAD or the key derivation.

wrap_epoch is the **committee_epoch** at the moment the secret version was wrapped. It is **immutable** once set on **SecretVersion** and is the load-bearing field that ties the identity to the historical share polynomial under which it was encrypted.

Encryption (Owner Side)

Given **MPK**, identity **I**, plaintext DEK **dek** (256 bits, fresh CSPRNG), and **base_aad**:

```
r      ←$ Fr, r != 0           -- fresh per-wrap encryption ephemeral
U      = r · G2_gen            -- 96-byte compressed G2, published
```

```

aad          = base_aad || compress(U)                -- 164 bytes
gt           = e(I, MPK)^r                            -- pairing in Gt
ikm          = serialize(gt)                          -- canonical compressed Gt encoding
K            = HKDF-SHA-256(salt = "cbss/ibe/v1",
                        ikm,
                        info = "cbss/ibe/v1" || aad,
                        L = 32)
nonce        ← CSPRNG(12)
WrappedDek   = AES-256-GCM(key = K,
                        nonce = nonce,
                        aad = aad,
                        plaintext = dek)

```

The ephemeral exponent is load-bearing: `MPK`, `I`, and `aad` are all public, so without $\hat{r}$ anyone reading the chain could recompute `K` and decrypt the stored envelope. `r` MUST be sampled fresh per wrap and never reused.

The `WrappedDek` (version, ciphertext, nonce, ephemeral_u, aad — serialized format version 2, carried as a leading 1-byte tag on every serialization and rejected on unknown values; the earlier three-field envelope is not valid and there is no migration path) is stored on chain. The tag is wire framing only — never part of the AEAD `aad` or the key derivation. The plaintext payload is separately AES-256-GCM-encrypted under `dek` and the same `aad` and uploaded to a CBFS volume; the resulting `cbfs_pointer` is also stored on chain.

DKG (Once Per Account / Per-Secret Override)

The production daemon uses Commonware’s BLS12-381 Feldman–Desmedt DKG state machine with per-proxy Ed25519 DKG keys, signed dealer logs, player acknowledgements, and `N3f1` quorum selection. Each finalized player output contains its scalar share, the `MPK`, and the public-sharing commitments used to verify later partial signatures.

The chain records a requested `threshold`, but that value is not passed into Commonware’s DKG API. Commonware derives the sharing quorum from committee size through `N3f1`. The default (`n=5`, `t=4`) is consistent. For overrides, callers MUST use the `N3f1`-derived threshold; other ratios can be accepted into `DkgPending` but do not produce a polynomial with the requested reconstruction threshold.

The DKG is bootstrapped by a `RequestAccountDkg` (default-path, owner-paid bond) or by `SetSecret` with `committee_override` (override-path, owner-paid bond). It commits via `RotateCommittee` (signed by $\geq t$ of the new committee) when the ceremony completes; if it

does not complete within `DKG_FINALIZE_BLOCKS`, anyone may call `ExpireDkgPending` to refund the requester (minus a small slash) and clear the pending record.

Partial Sign and Threshold Combine (Runner Side)

A proxy holding share `s_i` for committee_epoch `e`, on receiving a runner-signed `PartialSignRequest`, validates the request (§5.3), reads `wrap_epoch` from the on-chain `SecretVersion`, derives `I = hash_to_G1(base_aad)` (the 68-byte identity input — never the extended AEAD `aad`), and computes:

```
 $\sigma_i = s_i \cdot I$            -- a G1 point, 48 bytes compressed
```

The proxy signs `(request_hash, proxy_id, sigma_i, committee_epoch_at_serve, served_at_block)` under its operator key (domain `cbss/partial-sign-response/v1`) and returns the response.

The runner verifies each σ_i via the pairing equation:

```
 $e(\sigma_i, G2_{gen}) == e(I, S_i)$ 
```

where `S_i` is proxy `i`'s aggregate VSS commitment looked up on chain. With `t` valid partials, the runner computes Lagrange coefficients $\lambda_i(0)$ over the chosen index set and combines:

```
 $\sigma = \sum (\lambda_i(0) \cdot \sigma_i)$  over the t selected indices
```

By bilinearity, $\sigma = MSK \cdot I$.

Decryption (Runner Side)

With the threshold signature σ , the runner derives the same `K` the owner used:

```
U          = decode(WrappedDek.ephemeral_u)           -- canonical, non-
identity, prime-order; rejected otherwise
aad        = base_aad || WrappedDek.ephemeral_u       -- must equal WrappedDek.aad EXACTLY
gt         = e( $\sigma$ , U)
ikm        = serialize(gt)
K          = HKDF-SHA-256(salt = "cbss/ibe/v1",
                        ikm,
```

```

        info = "cbss/ibe/v1" || aad,
        L = 32)
dek          = AES-256-GCM-decrypt(K, nonce, aad, WrappedDek.ciphertext)

```

The bilinearity of the pairing guarantees the equality $e(\sigma, U) = e(\text{MSK} \cdot I, r \cdot G2_{\text{gen}}) = e(I, G2_{\text{gen}})^{\text{MSK} \cdot r} = e(I, \text{MPK})^r$, so the runner's K matches the owner's K exactly. Recovering K from the public (I, MPK, U) without σ reduces to the co-BDH assumption on BLS12-381. The runner then fetches the CBFS object at `cbfs_pointer`, AES-256-GCM-decrypts under (dek, aad) , and wipes the DEK and plaintext buffers that expose mutable storage as described in §12.6.

Reshare via Proactive Secret Sharing

A reshare replaces all proxy shares with fresh shares for the same MSK. Each old-committee participant j (with old share s_j) selected as a dealer constructs a polynomial:

```

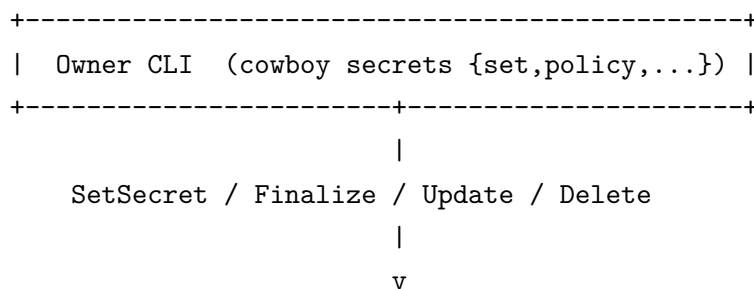
g_j(0)    = s_j · L_j          -- L_j is the Lagrange coefficient at zero for the chosen
g_j(x)    = g_j(0) + r_j(x)   -- r_j is a fresh degree-(t-1) polynomial with r_j(0) = 0

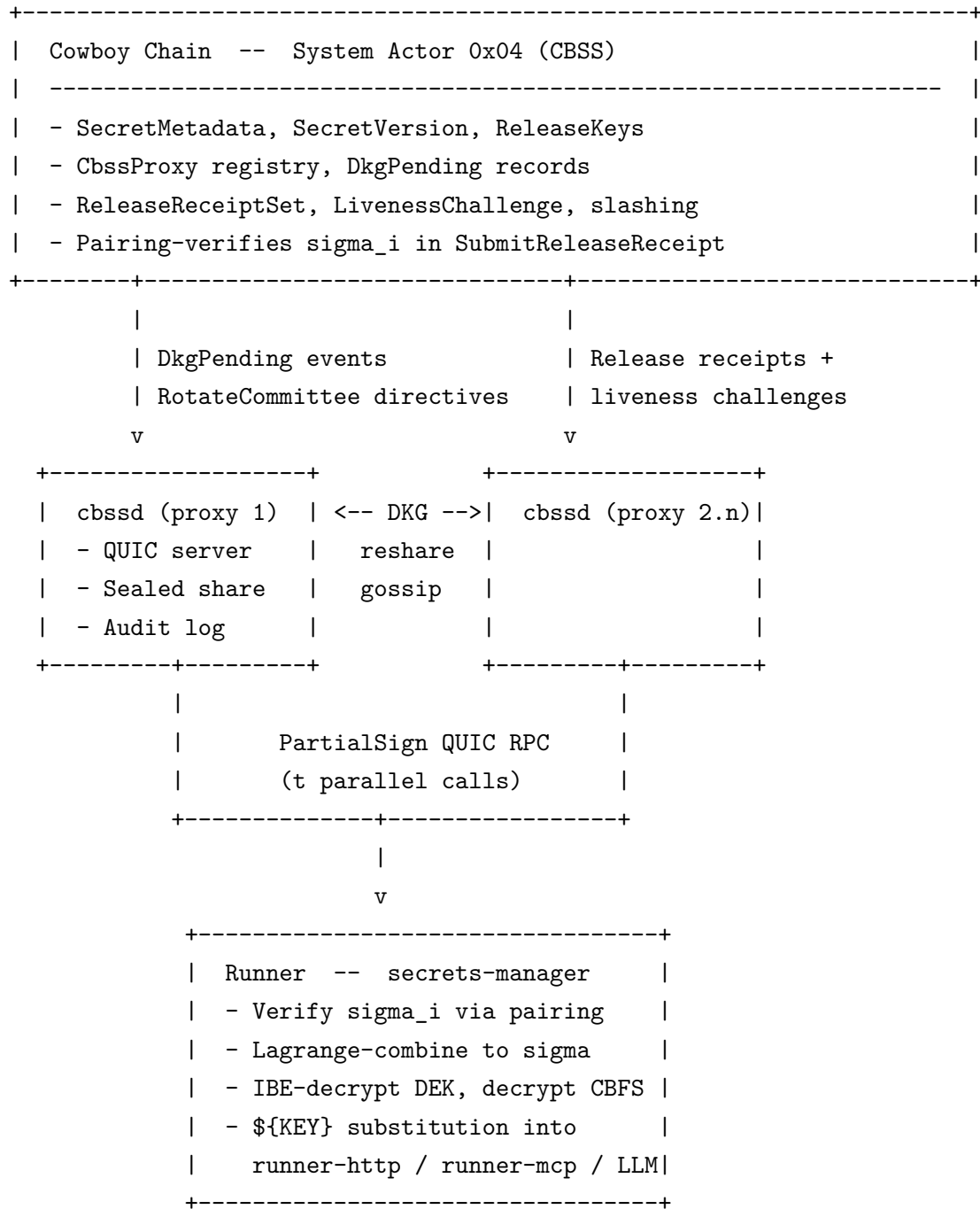
```

Each new-committee recipient k receives $g_j(k)$ from every dealer (HPKE-encrypted), verifies it against published commitments, and sets its new share to $s'_k = \sum_j g_j(k)$. Because $\sum_j g_j(0) = \sum_j (s_j \cdot L_j) = \text{MSK}$, the new shares lie on a polynomial whose constant term is MSK; MPK is preserved.

The chain enforces `args.new_mpk == old.mpk` and `args.new_committee_epoch == old.committee_epoch + 1` on `RotateCommittee`. It also retains the prior committee's commitments in `prior_committee_epochs` for `RESHARE_GRACE_BLOCKS`, so receipts under the prior epoch remain acceptable during the grace window.

System Architecture





On-Chain Component (System Actor 0x04)

The CBSS system actor occupies address 0x04 (per Technical Whitepaper §9). It owns the storage shapes in §2, dispatches the 20 CBSS instructions in §3, and emits the events in §4. It performs all pairing verifications for `SubmitReleaseReceipt` and validates committee co-signatures for

RotateCommittee. It does **not** hold any plaintext, any DEK, or any share scalar. TEE verifier handler semantics are logically allocated to 0x05, but their state is currently co-located under 0x04.

Off-Chain Component (cbssd Daemon)

Each CBSS proxy operator runs **cbssd**, a Tokio-based binary that:

- Maintains a **secp256k1** operator identity, an X25519 HPKE key, an Ed25519 Commonware DKG key, and a pinned TLS SPKI, all registered on chain.
- Listens on a QUIC endpoint multiplexing four logical streams (**PartialSign**, **Health**, **DkgRound**, **ReshareRound**) on a single port (default 9601).
- Watches the chain for **DkgPending** records and **RotateCommittee** directives that name this proxy and orchestrates the corresponding ceremonies.
- Stores per-scope sealed shares at rest, sealed via AES-256-GCM under a key derived from the operator key.
- On **PartialSign**, validates the request against current head, computes $\sigma_i = s_i \cdot I$, signs the response, and returns it before asynchronously attempting a **SubmitReleaseReceipt** transaction after 0–10 blocks of jitter.
- Maintains a process-local FIFO set of the most recent 65,536 successful request hashes. It is neither time-based nor persistent across restart.
- Bounds transport and receipt worker concurrency. Partial-sign scalar and pairing work currently executes inline in the async request task; there is no per-actor partial-sign rate limiter.
- Submits **LivenessChallengeResponse** for any chain-recorded challenge against this proxy.

cbssd is a separate process from **cbfsd** and **runnerd**. Operators MAY co-locate these processes, but SHOULD isolate their keys, logs, and configuration. CBSS proxy stake is registered independently from other service roles.

Runner-Side Client (secrets-manager Crate)

The runner-side client is a Rust crate consumed by **runner-node**, **runner-http**, **runner-mcp**, and **runner-llm**. It exposes a single trait:

```
#[async_trait]
pub trait SecretsClient: Send + Sync {
    async fn fetch_for_op(
        &self,
        ctx: &JobContext,
```

```

    actor: ActorAddress,
    keys: &[SecretRef],
) -> Result<SecretBundle, SecretsError>;
}

```

Production implementations fan out partial-sign QUIC RPCs to t of n committee proxies, with a 5-second per-proxy timeout, verify each σ_i via the pairing equation, Lagrange-combine into σ , IBE-decrypt the DEK, fetch and AES-decrypt the CBFS payload, and return a `SecretBundle` whose entries are `Zeroizing<Vec<u8>>`. The bundle is consumed by $\${KEY}$ substitution helpers wired into runner-http/mcp/llm at request-construction time. Mutable rendered string fields are wiped after execution. Checked `HeaderMap` and `CanonicalJsonBytes` values lack mutable accessors, so their cleared replacements do not actively overwrite the old freed allocations.

1. Identifiers and Canonical Forms

1.1 Account. A `secp256k1`-derived 20-byte Cowboy address (Technical Whitepaper §1.1).

1.2 Key name. Non-empty and ≤ 128 bytes. Ordinary names MUST begin with A–Z; remaining bytes MUST be uppercase ASCII letters, digits, or `_`. The reserved Trading Post form is `tradingpost/{32-byte declaration id hex}/{u32 version}/{u32 artifact id}`. The hash on chain is `key_hash = keccak256(account || key_name)`. Plaintext key names MUST NOT appear in storage paths or events — only `key_hash`.

1.3 SecretId. (`account`, `key_hash`). Canonical encoding is the 52-byte concatenation `account || key_hash`.

1.4 Versioning. Versions are monotonic `u64` integers per `SecretId`. Quotas: `MAX_SECRETS_PER_ACCOUNT = 256`, `MAX_VERSIONS_PER_SECRET = 32`, `MAX_ACL_ACTORS = 64`.

1.5 Release nonce and ID. Each `fetch_for_op` samples a 32-byte CSPRNG nonce. The consensus release ID is `keccak256(canonical(SecretId) || u64_le(version) || actor || runner_id || job_id || release_nonce || u64_le(serve_epoch))`. Binding `serve_epoch` prevents a pre-reshare receipt set from colliding with a request under the new committee.

1.6 Wrap epoch. The `committee_epoch` of the release key at the moment a secret version is wrapped. Stored on `SecretVersion`. **IMMUTABLE** once set.

1.7 Serve epoch. The `committee_epoch` the runner pinned in the request body when fanning out partial-sign RPCs. Equal to the `committee_epoch` of the release key at that block height. May differ from `wrap_epoch` after one or more reshares.

1.8 Domain separators. The implementation uses the following strings as message prefixes, hash-to-curve domain-separation tags, HKDF salt/info prefixes, or protocol contexts as indicated

by the construction:

- `cbss/ibe/v1` — identity I and IBE key derivation.
- `cbss/release-request/v1` — runner signature over `ReleaseRequestBody`.
- `cbss/partial-sign-response/v1` — proxy signature over `(request_hash, proxy_id,  $\sigma_i$ , committee_epoch_at_serve, served_at_block)`.
- `cbss/rotate-committee/v1` — committee co-signature over `RotateCommitteeArgs`.
- `cbss/liveness-challenge/v1` — runner signature over `SubmitLivenessChallengeArgs.runner_request`.
- `cbss/liveness-response/v1` — proxy signature over a liveness response.
- `cbss/share-storage/v1` — at-rest seal of operator share material.
- `cbss/operator-storage-key/v1` — derivation of the operator-local share-storage key.
- `cbss/dkg-share-hpke/v1` — HPKE info for inter-proxy DKG share envelopes.
- `cbss/dkg-round1/v1` and `cbss/dkg-round2/v1` — signed DKG dealer material.
- `cbss/dkg-round-msg/v1` and `cbss/reshare-round-msg/v1` — authenticated ceremony transport messages.
- `cowboy/cbss/commonware-dkg/v1` — Commonware DKG protocol context.
- `cbss/proxy-select/v1` — deterministic committee-selection sampling.
- `cbss/tee-attestation/v1` — runner TEE-attestation signature.
- `cbss/tlock/v1` — time-lock identity and IBE key derivation.

2. Storage Shapes

0x04 storage uses path-prefixed keys; values are JSON unless noted. The principal logical records are shown below in human-readable notation; the implementation concatenates raw fixed-width bytes and big-endian versions rather than ASCII hex fields. Plaintext secret key names are replaced by `key_hash`. Internal scan, sweep, and garbage-collection indexes are omitted unless they affect authorization or accountability.

<code>secret:{account_hex}:{key_hash_hex}</code>	→ <code>SecretMetadata</code>
<code>secret_version:{account_hex}:{key_hash_hex}:{version_be8}</code>	→ <code>SecretVersion</code>
<code>secret_version_tombstone:{...}:{version_be8}</code>	→ <code>SecretVersionTombstone</code>
<code>account_release_key:{account_hex}</code>	→ <code>AccountReleaseKey</code>
<code>secret_release_key:{account_hex}:{key_hash_hex}:{version_be8}</code>	→ <code>SecretReleaseKey</code>
<code>cbss_proxy:{proxy_id_hex}</code>	→ <code>CbssProxy</code>
<code>cbss_proxy_release_scopes:{proxy_id_hex}</code>	→ <code>Vec<ReleaseKeyScope></code>
<code>cbss_proxy_governance_review:{proxy_id_hex}</code>	→ <code>ProxyGovernanceReviewFlag</code>
<code>cbss_operator_proxy:{operator_hex}</code>	→ <code>ProxyId</code>
<code>cbss_dkg_pubkey:{dkg_pubkey_hex}</code>	→ <code>ProxyId</code>

cbss_slash_tombstone:{proxy_id_hex}:{fault}:{evidence_hash}	→ bool
release_receipt:{release_id_hex}	→ ReleaseReceiptSet
liveness_challenge:{challenge_id_hex}	→ LivenessChallenge
liveness_challenge_index:{release_id_hex}:{proxy_id_hex}	→ ChallengeId
dkg_pending:{scope_serialized}	→ DkgPending
dkg_ceremony_record:{scope_serialized}:{epoch_be8}	→ DkgCeremonyRecord
actor_release_debt:{actor_hex}	→ ActorReleaseDebt
tlock_req:{tlock_id_hex}	→ TlockRequest
tlock_parts:{tlock_id_hex}	→ TlockPartialSet

The principal value shapes:

```

struct SecretMetadata {
    id: SecretId,
    latest_version: u64,
    active_versions: Vec<u64>,
    created_at: BlockHeight,
    last_updated_at: BlockHeight,
}

struct SecretVersion {
    version: u64,
    pending: bool, // true between override SetSecret and
    ↪ FinalizeSecretVersion
    recipient: ReleaseKeyRef, // Account(_) or SecretSpecific[...]
    policy: SecretPolicy, // actors allowlist + tee_required
    wrap_epoch: Option<u64>, // immutable once set; None iff pending
    cbfs_pointer: Option<CbfsPointer>, // None iff pending
    wrapped_dek: Option<WrappedDek>, // None iff pending
    created_at: BlockHeight,
}

struct AccountReleaseKey {
    account: Address,
    mpk: G2Bytes,
    committee: Vec<ProxyId>,
    committee_operators: Vec<Address>, // operator snapshot parallel to committee
    threshold: u8,
    committee_epoch: u64,
    last_reshare_block: BlockHeight,
    vss_commitments: Vec<G2Bytes>,
    prior_committee_epochs: Vec<PriorCommitteeEpoch>, // capped at
    ↪ MAX_PRIOR_COMMITTEE_EPOCHS

```

```
}

struct SecretReleaseKey {
    secret_id: SecretId,
    version: u64,
    mpk: G2Bytes,
    committee: Vec<ProxyId>,
    committee_operators: Vec<Address>,
    threshold: u8,
    committee_epoch: u64,
    last_reshare_block: BlockHeight,
    vss_commitments: Vec<G2Bytes>,
    prior_committee_epochs: Vec<PriorCommitteeEpoch>,
}

struct PriorCommitteeEpoch {
    epoch: u64,
    committee: Vec<ProxyId>,
    committee_operators: Vec<Address>, // operator snapshot for the prior epoch
    vss_commitments: Vec<G2Bytes>,
    expires_at: BlockHeight,           // last_reshare_block + RESHARE_GRACE_BLOCKS
}

struct ReleaseReceiptSet {
    quorum_epoch: u64,                 // pinned to first accepted receipt's
    ↪ serve_epoch
    threshold: u8,
    entries: Vec<ReleaseReceipt>,     // capped at threshold
}

struct DkgCeremonyRecord {
    scope: ReleaseKeyScope,
    committee: Vec<ProxyId>,
    threshold: u8,
    committee_epoch: u64,
}

struct ActorReleaseDebt {
    actor: Address,
    outstanding: u64,
    delinquent_since: BlockHeight,
    last_updated_block: BlockHeight,
}
```

The invariant `pending == true ⇔ wrap_epoch.is_none() ⇔ cbfs_pointer.is_none() ⇔ wrapped_dek.is_none()` MUST hold for every `SecretVersion`. Once `wrap_epoch` is set on a non-pending version, it MUST NOT change.

3. System Instructions (Opcodes 68–84, 152–153, 158)

The protocol codec implements 20 CBSS instructions, encoded as `SystemInstruction` variants per Technical Whitepaper §3 dispatch.

Op	Name	Caller	Purpose
68	<code>SetSecret</code>	Owner	Set or override-intent a secret version
69	<code>UpdateSecretPolicy</code>	Owner	Mutate <code>policy</code> only; never re-wraps
70	<code>DeleteSecretVersion</code>	Owner	Tombstone one version
71	<code>DeleteSecret</code>	Owner	Tombstone all versions; drop release key if scoped
72	<code>RegisterCbssProxy</code>	Operator	Stake and register operator, HPKE, DKG, TLS, and network identity
73	<code>DeregisterCbssProxy</code>	Operator	Mark unbonded at current height and enqueue reshare
74	<code>RotateCommittee</code>	Committee co-sign	Commit a DKG or reshare result
75	<code>SlashCbssProxy</code>	Anyone (with evidence)	Apply per-class slash + suspend
76	<code>SubmitReleaseReceipt</code>	Proxy	Anchor a partial signature on chain, earn fee
77	<code>RequestAccountDkg</code>	Owner	Bond a default-path account DKG
78	<code>FinalizeSecretVersion</code>	Owner	Attach <code>cbfs_pointer</code> + <code>wrapped_dek</code> after override DKG commits
79	<code>SubmitLivenessChallenge</code>	Runner	Bond a non-response challenge against a proxy
80	<code>LivenessChallengeResponse</code>	Proxy operator	Respond to a chain-recorded challenge
81	<code>ExpireDkgPending</code>	Anyone	Permissionless cleanup of expired DKG bonds
82	<code>ExpireLivenessChallenge</code>	Anyone	Permissionless transition of an expired challenge to Unanswered
83	<code>RequestReshare</code>	Owner / scheduler	Trigger a PSS reshare of an existing release key

Op	Name	Caller	Purpose
84	<code>ForcedDeregisterCbssProxy</code>	Governance	Forced deregister via approved governance proposal
152	<code>RegisterTlockRelease</code>	Anyone	Register a height-gated threshold release identity
153	<code>SubmitTlockRelease</code>	Committee proxy	Post a verified partial after the target height
158	<code>GovRequestSystemDkg</code>	Governance	Bootstrap or rekey a keyless system-actor scope

3.1 SetSecret (default path). Owner has an existing `AccountReleaseKey`. Args carry `(key_hash, recipient = Account(self), policy, wrap_epoch, cbfs_pointer, wrapped_dek)`. Chain checks `wrap_epoch == release_key.committee_epoch`, that `wrapped_dek.ephemeral_u` is a canonical non-identity G2 point, and `wrapped_dek.aad == aad_for(secret_id, version, wrap_epoch) || wrapped_dek.ephemeral_u` (full equality); writes a non-pending `SecretVersion`.

3.2 SetSecret (override path). Args carry `(key_hash, recipient = SecretSpecific{...}, policy, committee_override = {t, n, preferred_proxies?})` and NO ciphertext. Chain validates $0 < t \leq n$, validates or selects the committee, locks `DKG_BOND` from the owner, enqueues a `DkgPending` record, and writes a *pending* `SecretVersion` with `wrap_epoch = None`. The chain does not enforce that `t` equals Commonwealth's N3f1 quorum; callers must observe the restriction in the DKG section.

3.3 FinalizeSecretVersion. Once the per-secret DKG commits via `RotateCommittee`, the owner calls `FinalizeSecretVersion(key_hash, version, wrap_epoch, cbfs_pointer, wrapped_dek)`. Chain checks `wrap_epoch == new_release_key.committee_epoch`, that `wrapped_dek.ephemeral_u` is a canonical non-identity G2 point, `wrapped_dek.aad == aad_for(secret_id, version, wrap_epoch) || wrapped_dek.ephemeral_u` (full equality), clears `pending`, and attaches the ciphertext.

3.4 UpdateSecretPolicy. Pure metadata. Args name an explicit `(key_hash, version)` and replace that version's `policy`, including while it is pending. No `WrappedDek` is touched. Proxies query the chain authorizer on each request; successfully submitted receipts are revalidated against submission-time state.

3.5 DeleteSecretVersion / DeleteSecret. Writes a `SecretVersionTombstone` so subsequent reads return `SecretVersionDeleted` rather than `SecretNotFound`. `DeleteSecret` on an override-scoped secret also drops the `SecretReleaseKey`.

3.6 RegisterCbssProxy. Args: `(pubkey, hpke_pubkey, dkg_pubkey, tls_spki,`

`network_addr`, `stake_amount`). Chain requires `stake_amount` $\geq$ `MIN_PROXY_STAKE`, snapshots all key material, sets `eligible_after` = `block_height` + `PROXY_SOAK_PERIOD_BLOCKS`, and inserts the proxy plus operator and DKG-key reverse indexes. The proxy is not selection-eligible until `eligible_after` is reached.

3.7 DeregisterCbssProxy. Sets `unbond_at` = `block_height`. The proxy is removed from new committee selections immediately and the handler enqueues reshare events for scopes that contain it. There is no delayed unbond period in the current implementation.

3.8 RotateCommittee. Args: (`scope`, `new_committee`, `new_committee_epoch`, `new_mpk`, `vss_commitments`, `new_committee_sigs`, `prior_committee_sigs`). Chain computes `rotate_committee_hash` and verifies:

- $\geq t$ signatures from the new committee under domain `cbss/rotate-committee/v1`.
- If updating an existing release key: $\geq t$ signatures from the *prior* committee, and `new_mpk` == `old.mpk`, and `new_committee_epoch` == `old.committee_epoch` + 1.
- Threshold reused from `old.threshold` (or, on first DKG, from `DkgPending.threshold`).
- `vss_commitments` valid: `vss_commitments[0]` == `new_mpk`, length is at least threshold, and every point decodes; the MPK itself must be non-identity.

On success, the prior committee's epoch is appended to `prior_committee_epochs`, capped at `MAX_PRIOR_COMMITTEE_EPOCHS` and pruned of expired entries.

3.9 SlashCbssProxy. Args: (`proxy_id`, `fault_class` $\in$ {`InvalidPartialReencrypt`, `DkgSabotage`, `Equivocation`}, `evidence`). The chain re-derives the offence from `evidence`, charges the per-class slash percentage (5 % / 5 % / 50 %), and suspends the proxy. A permanent evidence tombstone prevents replay of the same (`proxy_id`, `fault_class`, `evidence_hash`).

3.10 SubmitReleaseReceipt. Args: (`runner_request`: `SignedReleaseRequest`, `proxy_id`, `sigma_i`, `committee_epoch_at_serve`, `served_at_block`, `proxy_sig`). Chain validates:

- `body.request_block` $\in$ [`head` - `REQUEST_FRESHNESS_BLOCKS`, `head`].
- `args.committee_epoch_at_serve` == `body.serve_epoch`; both match a current or grace-window release key view.
- `runner_sig` recovers to `body.runner_id` under `cbss/release-request/v1`.
- `proxy_sig` recovers to `proxy.operator` under `cbss/partial-sign-response/v1`.
- `SecretVersion` exists, is non-pending, has `recipient` == `body.recipient`, and yields the `wrap_epoch` for identity derivation.
- `policy.actors` (if set) contains `body.actor`; `policy.tee_required` validates against the runner's TEE attestation at submission-time current head.
- The pairing equation $e(\sigma_i, G2_gen) == e(I, S_i)$ holds, with $S_i = \sum_k vss_commitments[k] \cdot proxy_index^k$ evaluated from the release-key view.

- The `(release_id, proxy_id)` pair has not already been recorded; the `ReleaseReceiptSet` has not yet reached `threshold` entries.

On success, the receipt is appended to the set, the actor is charged `RELEASE_FEE_PER_RECEIPT`, and the proxy is credited the same. If the charge uses the permitted overdraft, the chain records `ActorReleaseDebt` and may mark the actor delinquent. The chain emits `cbss.release.receipt_recorded` per accepted receipt and `cbss.secret.released` when the accepted set reaches threshold.

3.11 SubmitLivenessChallenge. Args: `(runner_request, proxy_id, challenge_sig)`. Chain enforces `block_height ≥ body.request_block + MIN_CHALLENGE_DELAY_BLOCKS` and `block_height ≤ body.request_block + LIVENESS_CHALLENGE_MAX_AGE_BLOCKS`. `challenge_sig` MUST recover to `body.runner_id` under `cbss/liveness-challenge/v1`. The chain checks for an existing receipt: if present, the challenge admits as `Resolved` immediately and the bond transfers to the proxy. Otherwise it admits as `Pending` with `deadline = block_height + LIVENESS_RESPONSE_BLOCKS`.

3.12 LivenessChallengeResponse. Proxy operator submits with reason $\in \{\text{InvalidRequest}, \text{RateLimited}, \text{NeverReceived}\}$. Chain transitions the challenge to `Unverifiable` (terminal), refunds the bond to the runner, and leaves the proxy's health neutral.

3.13 Liveness deadline. A challenge that reaches `block_height > deadline` without a response transitions to `Unanswered`, refunds the bond to the runner, and decrements the proxy's `health.score_bps`. The transition is performed permissionlessly via an `ExpireLivenessChallenge` instruction (mirroring `ExpireDkgPending`); a per-block tick is NOT used.

3.14 ExpiredDkgPending. After the deadline, anyone may clean up an expired DKG bond; the requester may also self-cancel before the deadline. Both paths refund `DKG_BOND - DKG_TIMEOUT_SLASH` and clear the `DkgPending` record. A third-party timeout caller receives one token unit from the slashed amount; a requester self-cancel receives no caller tip. For override-path cleanup, the corresponding pending `SecretVersion` is erased without a tombstone.

4. Events

The CBSS system actor emits the following string topics with byte payloads. Payloads marked JSON are `serde_json` encodings; raw IDs are 32 bytes, actor addresses are 20 bytes, and integer fields in raw composite payloads are big-endian unless marked `_le`.

Topic	Body	When
cbss.secret.created	account(20) key_hash(32) version(8)	After every SetSecret, including creation of a pending override version
cbss.secret.finalized	account(20) key_hash(32) version(8)	After FinalizeSecretVersion makes a pending override version usable
cbss.secret.policy_updated	account(20) key_hash(32) version(8)	After UpdateSecretPolicy
cbss.secret.version_deleted	account(20) key_hash(32) version(8)	After DeleteSecretVersion
cbss.secret.deleted	account(20) key_hash(32) latest_version(8)	After DeleteSecret
cbss.proxy.registered	proxy_id	After RegisterCbssProxy
cbss.proxy.deregistered	proxy_id	After DeregisterCbssProxy
cbss.proxy.forced_deregistered	proxy_id	After governance-forced deregistration
cbss.proxy.slashed	proxy_id	After SlashCbssProxy
cbss.proxy.governance_review_required	JSON {proxy_id, reason, score_bps, flagged_at_block}	After sustained low health crosses the review window
cbss.committee.rotated	JSON {scope, epoch}	After RotateCommittee succeeds
cbss.dkg.requested	JSON {scope, epoch, force_rekey, threshold, committee_metadata}	After account, override, or governance system DKG request
cbss.dkg.expired	canonical binary ReleaseKeyScope	After ExpiredDkgPending
cbss.reshare.requested	JSON ceremony metadata, committees, signers, and old commitments	When cadence or committee churn enqueues a reshare
cbss.release.receipt_recorded	release_id	Per accepted SubmitReleaseReceipt
cbss.secret.released	release_id	Exactly once when the t-th receipt lands
cbss.release_receipts.gc	upto_block(8) removed_count(8)	After each bounded permissionless receipt-GC pass
cbss.actor.delinquent	raw actor address	When receipt charging first exceeds the paid balance
cbss.actor.debt_repaid	actor(20) repaid_le(8) remaining_le(8)	When FundActor repays release-fee debt
cbss.liveness.challenge	JSON {challenge_id, proxy_id, release_id, deadline}	After SubmitLivenessChallenge admits
cbss.liveness.response	challenge_id	After LivenessChallengeResponse
cbss.liveness.expired	challenge_id	After ExpireLivenessChallenge
cbss.tlock.registered	tlock_id	After RegisterTlockRelease
cbss.tlock.partial_recorded	tlock_id	Per accepted SubmitTlockRelease
cbss.tlock.released	tlock_id	When the tlock partial set reaches threshold
cbss.tlock.gc	upto_block(8) removed_count(8)	After each bounded permissionless tlock-GC pass

GC event bodies encode `upto_block` followed by the removed-record count as two big-endian `u64` values.

5. The Release Path (Normative)

5.1 Construction. The runner's `secrets-manager` client constructs:

```
release_nonce    ← CSPRNG(32)
request_block    = chain_head_height
serve_epoch      = release_key.committee_epoch (at chain_head_height)
body             = ReleaseRequestBody { secret_id, version, actor, runner_id,
                                     recipient, job_id, release_nonce,
                                     request_block, serve_epoch }
runner_sig       = ECDSA-sign(runner_sk,
                              keccak256("cbss/release-request/v1" || encode(body)))
request          = SignedReleaseRequest { body, runner_sig }
```

The canonical chain body has no `hire_id` or other Trading Post context field. Hired-job authorization is derived from the on-chain `JobSpec.hire_binding`. A client that adds `hire_id` bytes to the signed preimage creates a signature the node cannot recover and **MUST** be rejected.

5.2 Fanout. The runner orders committee proxies by health and observed latency and sends requests with bounded concurrent fanout via QUIC until it obtains `t` valid partials or exhausts the committee. The current per-proxy timeout is 5 seconds.

5.3 Proxy validation. On receipt of a `PartialSignRequest`, the daemon enforces:

1. The protocol version is supported and the canonical request hash is absent from the process-local 65,536-entry served-request FIFO.
2. `runner_sig` recovers to `body.runner_id` under `cbss/release-request/v1`.
3. The `secret/version/recipient` resolution matches the request and `body.request_block` $\in$ `[head - REQUEST_FRESHNESS_BLOCKS, head]`.
4. The runner is currently assigned to `body.job_id`; the actor manifest declares the requested key; Trading Post labels additionally pass lease authorization.
5. `policy.actors` authorizes the actor and `policy.tee_required` passes the current TEE check when set.
6. The proxy is a member of the resolved committee and `body.serve_epoch == release_key.committee_epoch`.

7. The local share belongs to that proxy and epoch, is active at the served block, and matches the VSS commitments.

5.4 Signing. The proxy decrypts its share `s_i`, reads `wrap_epoch` from the on-chain `SecretVersion`, verifies the local share against the stored VSS commitments, computes `I = hash_to_G1(base_aad)` (the 68-byte identity input — never the extended AEAD `aad`), computes $\sigma_i = s_i \cdot I$, signs the response payload, records the request hash after success, and returns. These cryptographic steps currently execute inline in the async handler.

5.5 Combine. The runner verifies each σ_i against the resolved release-key snapshot, Lagrange-combines `t` valid partials into σ , and IBE-decrypts the DEK. The DEK is wrapped in `Zeroizing<u8; 32>` for the duration of the operation.

5.6 Payload. The runner fetches the CBFS object at `cbfs_pointer` from a configured CBFS-node HTTP endpoint, mounted path, or generic HTTP fallback, AES-256-GCM-decrypts under `(dek, aad)`, inserts the resulting `Zeroizing<Vec<u8>>` into the `SecretBundle` keyed on the original `KeyName`, and zeroes the DEK.

5.7 Substitution. Runner-host modules consume the bundle:

- **runner-http:** substitutes `${KEY}` in URL, headers (including `Authorization`), and request body bytes.
- **runner-mcp:** substitutes `${KEY}` in MCP tool arguments (JSON-aware) and, for stdio MCP servers, spawns the child via `tokio::process::Command::env_clear().envs(env)` so the parent process environment is never mutated.
- **runner-llm:** only when server-side tool plumbing is active; otherwise rejects with `SecretsNotApplicableHere`.

5.8 Drop. When the request completes (or panics), the bundle drops; entries are zeroized via `Zeroizing<Vec<u8>>::drop`. Rendered URL, prompt, and system-prompt strings are actively wiped. Header and canonical-JSON containers are replaced with cleared values, but their old allocations are freed without an overwrite because the checked codec types expose no mutable or consuming accessor. The PVM does not install a per-secret-key environment deny list. In deterministic mode it sets RustPython's `ignore_environment = true`, but that flag ignores Python configuration variables; RustPython's POSIX `os.environ` module still copies the host process environment.

5.9 Receipts. Asynchronously and independently, each participating proxy attempts `SubmitReleaseReceipt` with 0–10 blocks of jitter. The runner does not submit. The partial-sign response is returned before this work completes. A saturated receipt worker pool drops the attempt; a publication error is logged without retry; an accepted transaction hash is not followed

through execution by the receipt path. Consequently, accepted receipt events form an auditable subset of releases, not a complete release ledger.

5.10 Current release-ID interoperability defect. The node computes the consensus `release_id` defined in §1.5, including `serve_epoch`. The current `cbss-client` helper omits `serve_epoch`; daemon audit records and liveness lookup therefore use an incompatible ID and can report `NeverReceived` for a release whose node-side receipt uses the consensus ID. The consensus formula is normative.

6. The DKG and Reshare Path (Normative)

6.1 DKG trigger. A `DkgPending` record exists for (`scope`, `threshold`, `deadline`, `requester`). The committee is either selected by the chain (`select_eligible_proxies`) or named explicitly via `committee_override.preferred_proxies`. Each member discovers the others' `network_addr` and `hpke_pubkey` from the chain.

6.2 DKG ceremony. The committee establishes a full-mesh QUIC topology and runs the Commonware Feldman–Desmedt protocol described in the **Cryptographic Construction** section. Dealer messages, player acknowledgements, and signed dealer logs are routed using the registered Ed25519 DKG keys. Each recipient validates the transcript before finalizing its share; malformed signed dealer material can yield `DkgSabotage` evidence eligible for `SlashCbssProxy`.

6.3 DKG commit. Once the Commonware canonical commit quorum finalizes and the daemon collects at least the chain threshold of rotate signatures, it queues `RotateCommittee`. The chain refunds the requester's `DKG_BOND` on successful execution. If the ceremony remains pending until its deadline, a third party may submit `ExpiredDkgPending`; the requester is refunded `DKG_BOND` — `DKG_TIMEOUT_SLASH` and the cleanup caller earns one token unit from the slashed amount.

6.4 Reshare trigger. Block-height scheduler (`now` $\geq$ `last_reshare_block` + `RESHARE_INTERVAL_BLOCKS`), churn detection (`committee size` near `t` + `RESHARE_SAFETY_MARGIN`), or owner-paid `RotateCommittee` request.

6.5 Reshare ceremony. A subset of $\geq t$ old-committee members run the PSS protocol of the **Cryptographic Construction** section. New-committee members verify each dealer's contribution against the on-chain old commitments. On success, the new committee submits `RotateCommittee` co-signed by $\geq \text{threshold}$ of itself AND $\geq \text{threshold}$ of the prior committee.

6.6 Old-share deletion. The intended safe boundary is confirmed execution of `RotateCommittee`, after which the old sealed share is deleted and an audit record is appended. The current daemon also deletes the old share earlier, when a signed rotation is merely

queued for submission. If that transaction is rejected or never executes, the member may lose its old share before the chain changes epochs. Operators **MUST** treat queued-but-unconfirmed rotations as a recovery risk; deletion is not currently atomic with observing the on-chain rotation.

6.7 Grace window. For `RESHARE_GRACE_BLOCKS` after a rotation, receipts under the prior epoch remain acceptable: the chain looks up the receipt's `committee_epoch_at_serve` first against the current release key, then against `prior_committee_epochs`. The grace window is capped at `MAX_PRIOR_COMMITTEE_EPOCHS` simultaneously retained epochs.

7. Slashing

CBSS slashes for three on-chain-provable fault classes only. All other operational issues (non-response, slow proxies, miscalibrated stake) flow through the liveness-challenge audit trail and on-chain governance, not through automatic slashing.

Fault	Penalty	Evidence
<code>InvalidPartialReencrypt</code>	5 % of stake + suspend	A <code>SubmittedReleaseReceipt</code> whose σ_i fails the pairing check $e(\sigma_i, G2_gen) == e(I, S_i)$ for the proxy's index. The chain re-derives I from the body, looks up S_i from the release key, and re-runs the pairing.
<code>DkgSabotage</code>	5 % of stake + suspend	A round-2 message whose recipient share is inconsistent with the dealer's published round-1 commitments. The chain verifies $g_2 \cdot \text{share_scalar} \neq \sum \text{commitments}[k] \cdot \text{recipient_index}^k$.

Fault	Penalty	Evidence
Equivocation	50 % of stake + suspend	Two distinct PartialSignResponse payloads carrying identical request_hash but different σ_i . Both signatures must verify under the same proxy operator pubkey.

The handler is **SlashCbssProxy**. The chain writes a permanent tombstone for (**proxy_id**, **fault_class**, **evidence_hash**) so the same evidence cannot slash twice. Plaintext leaks (off-chain disclosure) and persistent low health are not auto-slash; they route through governance (Technical Whitepaper §11) via **ForcedDeregisterCbssProxy**.

8. Gas Pricing

CBSS instructions consume gas under both meters (Cycles + Cells; Technical Whitepaper §4 / §17.1). Pricing:

Instruction	Cycles	Cells
SetSecret (default)	12,000	500
SetSecret (override)	38,000	500
FinalizeSecretVersion	8,000	500
UpdateSecretPolicy	6,000	500
DeleteSecretVersion	5,000	500
DeleteSecret	5,000 × active-version count	500
RegisterCbssProxy	15,000	500
DeregisterCbssProxy / forced variant	8,000	500
RequestAccountDkg / GovRequestSystemDkg	30,000	500
RequestReshare	30,000	500
RotateCommittee	25,000 + 1,000 × new_committee.len()	500

Instruction	Cycles	Cells
<code>ExpireDkgPending</code>	5,000	500
<code>SubmitReleaseReceipt</code>	8,500	500
<code>SubmitLivenessChallenge</code>	5,000	500
<code>LivenessChallengeResponse</code>	4,000	500
<code>ExpireLivenessChallenge</code>	5,000	500
<code>SlashCbssProxy</code>	20,000–22,000 by fault class	500
<code>RegisterTlockRelease</code>	6,000	500
<code>SubmitTlockRelease</code>	8,500	500

Every listed handler charges one flat `GasCosts::base_cells` value, currently 500. Cells are not dynamically proportional to encoded payload length in the current implementation.

9. Liveness, Health, and Slashing Fairness

9.1 Liveness challenge bond. `LIVENESS_CHALLENGE_BOND` (default 100 token units). Locked on `SubmitLivenessChallenge`, refunded on `Unverifiable` or `Unanswered`, transferred to the proxy on `Resolved`.

9.2 Health score. Each `CbssProxy` starts with `health.score_bps = 10,000`. An unanswered challenge subtracts 100 bps, capped at 1,000 bps per 7,200-block attrition window. A successful slash subtracts the fault penalty itself (500, 500, or 5,000 bps). Committee-selection weight floors the effective score at 500 bps. If actual score remains below 1,000 bps for 2,592,000 blocks, the chain records a governance-review flag and emits `cbss.proxy.governance_review_required`; it does not itself enact forced deregistration. There is no per-block health recovery in the current implementation.

9.3 Bond economics. Spam against responsive proxies is expensive: every challenge costs `LIVENESS_CHALLENGE_BOND` and resolves immediately to the proxy if the proxy already produced a receipt. The constraint `MIN_CHALLENGE_DELAY_BLOCKS > REQUEST_FRESHNESS_BLOCKS` ensures honest receipts have already landed (or can no longer land) before a challenge is admissible.

10. Owner-Side Flows (CLI)

10.1 Default-path set:

```
cowboy secrets set OPENAI_API_KEY --value-file -
```

`--value-file -` reads the value from standard input and avoids placing it in shell history or the process argument list; the CLI refuses an inline plaintext value when attached to a TTY. The CLI looks up the owner's `AccountReleaseKey`. If absent, it submits `RequestAccountDkg` and waits for `cbss.committee.rotated`. It generates a fresh DEK, samples a fresh $r \neq 0$ with $U = r \cdot G2_{\text{gen}}$, computes `aad = base_aad || compress(U)`, derives $K = \text{HKDF}(\text{serialize}(e(I, \text{MPK})^r), \text{"cbss/ibe/v1"} || \text{aad})$, AES-encrypts the value to a CBFS object, AES-encrypts the DEK under K into a `WrappedDek` (which carries `ephemeral_u = compress(U)`), and submits `SetSecret`.

10.2 Override-path set:

```
cowboy secrets set STRIPE_LIVE_KEY --value-file - \  
  --committee-n 5 --committee-t 4
```

The CLI submits `SetSecret(committee_override = {t=4, n=5})` with no ciphertext and a recipient of `SecretSpecific{...}`. It waits for the per-secret DKG to commit, then submits `FinalizeSecretVersion` with the wrapped DEK and CBFS pointer. For any other committee size, `t` must equal Commonware's `N3f1` quorum; arbitrary ratios such as 5/9 do not match the production ceremony.

10.3 Policy update:

```
cowboy secrets policy --key-hash 0x... --version 3 --actor 0x...
```

This replaces the selected version's complete actor allowlist and `tee_required` flag; repeat `--actor` for multiple actors and pass `--tee-required` when required. It is a pure metadata update: no DEK is touched. Proxies query current chain state when authorizing a release.

10.4 Versioning and deletion:

```
cowboy secrets list --actor 0xOWNER  
cowboy secrets delete-version --key-hash 0x... --version 3 --yes  
cowboy secrets delete --key-hash 0x... --yes
```

10.5 Operator flows:

```
cowboy proxy register --pubkey 0x... --hpke-pubkey 0x... --dkg-pubkey 0x... \  
  --tls-spki 0x... --network-addr quic://1.2.3.4:9601 --stake 1000  
cowboy proxy list  
cowboy proxy deregister --proxy-id 0x...
```

11. SDK Integration

11.1 **Manifest.** Actors declare in their `ActorManifest`:

```
{
  "entitlements": [
    {
      "id": "secrets.read",
      "params": {
        "keys": [
          "OPENAI_API_KEY",
          "0x0123456789abcdef0123456789abcdef01234567:SLACK_API_KEY"
        ]
      }
    }
  ]
}
```

An unqualified key name refers to a secret owned by the actor's account; a cross-account reference uses `0x<40-hex-address>:KEY_NAME`. Manifest upgrades MAY shrink `secrets.read.keys` but MUST NOT expand them; the entitlement diff check is enforced when the actor's manifest is updated via the upgrade flow (Technical Whitepaper §3).

11.2 **Python SDK.** Runner ops accept a `secrets=[KEYS]` kwarg:

```
result = runner.http(
    method="POST",
    url="https://api.openai.com/v1/chat/completions",
    headers={"Authorization": "Bearer ${OPENAI_API_KEY}"},
    json=payload,
    secrets=["OPENAI_API_KEY"],
)
```

`secrets=[...]` MUST be a subset of the manifest's `secrets.read.keys`; an undeclared request is rejected by manifest and chain authorization and surfaces as secret-access denial. `runner.llm(..., secrets=[...])` is accepted only when server-side tool plumbing is active and raises `SecretsNotApplicableHere` otherwise.

A parallel `secrets.verify` entitlement (same `keys` shape and subset rule) grants **verify-only** use of a key as an inbound-MAC key via `runner.http(..., verify=VerifyInput(...))`: the runner MAC-checks an inbound webhook before any executor work and never substitutes the key into a request, so a key held only under `secrets.verify` cannot be exfiltrated through `secrets=[...]`.

11.3 PVM environment boundary. The PVM has no declared-key-specific environment filter. In deterministic execution it sets RustPython’s `ignore_environment = true`, which suppresses environment-driven interpreter configuration such as `PYTHONPATH`; it does **not** clear `os.environ`. RustPython’s POSIX module constructs `os.environ` from the host process environment, so actors can observe any variables inherited by the PVM host process. Deployments **MUST** therefore launch that process with an empty or explicitly allowlisted environment. `SecretBundle` debug formatting redacts values, and runner HTTP logging records a redacted URL origin rather than path, query, or user information; the implementation does not claim a general trace-wide scanner that removes every occurrence of an active secret value.

12. Security Considerations

12.1 Threshold IBE assumption. Security reduces to two assumptions on BLS12-381 plus the random-oracle model for `hash_to_G1`: the Computational co-Diffie-Hellman assumption (unforgeability of BLS partial/threshold signatures) and the co-Bilinear-Diffie-Hellman (co-BDH) assumption (confidentiality of the IBE wrap key — recovering K from the public (I, MPK, U) without σ ; see Decryption). Standard threshold-BLS analysis (Boneh, Lynn, Shacham 2001; Boldyreva 2003) covers the partial signature combine. The IBE construction matches Boneh-Franklin’s BasicIdent transformed to threshold form via a vetted t-of-n DKG; we follow the drand `tlock` paper’s specialization with our chain-anchored identity tuple.

12.2 DKG implementation. Production `cbssd` uses Commonware’s BLS12-381 `feldman_desmedt` dealer/player/log state machines with `N3f1` quorum selection. The older local DKG coordinator remains test-only. The requested chain threshold and Commonware’s derived quorum are separate inputs today, which is why override callers must select the `N3f1` threshold described above. An expired ceremony’s record remains available for `DkgSabotage` verification only until a same-scope retry starts: expiry does not advance the release-key epoch, so the retry reuses the storage key and overwrites the prior record. Evidence from the first ceremony can then become unverifiable.

Proxy registration rejects all-zero HPKE and DKG public keys, but it does not perform stronger point or small-subgroup validation. A nonzero but unusable key can therefore register and fail only when a ceremony first uses it.

12.3 Reshare insider rotation. PSS preserves MSK; this means a post-reshare committee can re-sign any historical-`wrap_epoch` identity. Reshare provides *insider rotation* (a previously-compromised share is replaced) but **NOT** forward secrecy. This is a deliberate design choice — forward secrecy would require re-encrypting every chain-anchored ciphertext on every reshare, which is operationally infeasible. Owner accounts that want a fresh release key can create a new

secret version with `--committee-n`, `--committee-t`, and a fresh committee selection.

12.4 Collusion of `t` proxies. `t` colluding proxies can reconstruct `MSK · I` for any identity they choose, decrypting any ciphertext under that release key. CBSS does not prevent or detect off-line collusion. Mitigation is limited to operator diversity through block-hash-seeded, health-weighted committee selection plus the soak period, and governance action based on evidence obtained outside the protocol. The `InvalidPartialReencrypt` slash proves an invalid partial; it does not prove or punish a valid off-line collusive decryption.

12.5 Owner key compromise. A compromised owner key can submit `UpdateSecretPolicy` to add an attacker actor and pull the secret. CBSS does not prevent this; it is a documented security boundary. Mitigations are wallet-level (hardware keys, multisig) and event-driven (chain emits `cbss.secret.policy_updated` for any change).

12.6 Memory hygiene. DEKs and secret-bundle values use `Zeroizing`; mutable rendered `String` fields are wiped after use. `HeaderMap` and `CanonicalJsonBytes` expose no in-place mutable buffer, so the runner clears them by replacement and the old plaintext allocation is freed without active overwrite. Network, allocator, and checked-codec copies therefore remain outside a strict every-allocation zeroization guarantee. AES-GCM AAD binds (`account`, `key_hash`, `version`, `wrap_epoch`, `ephemeral_u`) end to end so any field-level tamper fails authentication.

12.7 Replay protection. Three layers:

- Process-local 65,536-entry FIFO keyed on the canonical signed request hash; restart clears it and FIFO eviction is not time-based.
- Chain-side `release_receipt:{release_id}` map dedupes accepted receipts by (`release_id`, `proxy_id`) until receipt-set garbage collection.
- `request_block` freshness window `REQUEST_FRESHNESS_BLOCKS` constrains how far back a release request can target.

12.8 Revocation latency. Each proxy's HTTP authorizer fetches current secret, job, manifest, and TEE state for the request rather than consulting a documented TTL cache. A receipt that arrives later is revalidated at receipt-submission current head. Releases already served cannot be recalled, and failed receipt submission leaves no chain record.

12.9 TEE composition. When `policy.tee_required = true`, release authorization consults TEE verifier state, whose handler role is allocated to `0x05` but whose records are currently persisted under `0x04`. AWS Nitro's full path verifies its COSE document and X.509 chain against active collateral. For SGX, TDX, and SEV-SNP, chip-root proof and complete TCB enforcement remain soft-fail: a failed or unavailable proof can create a self-asserted active measurement binding, and the release path accepts any active, unexpired binding. Consequently

`tee_required` is not yet a hardware-rooted guarantee for non-Nitro runners.

12.10 Side channels and external review. Pairing operations and scalar arithmetic in `cbss-crypto` use `blstrs`’s constant-time primitives. Share scalars use `subtle::CtOption` for branch-free deserialization. HPKE and AES-GCM rely on RustCrypto’s constant-time implementations. The repository contains an external-review packet and report template but no accepted external cryptographic review report; that review remains a pre-mainnet gate.

12.11 Runner trust boundary. Threshold IBE protects a secret before authorized release; it does not protect plaintext from the selected runner, its host administrator, a compromised runner process, or downstream libraries after substitution. A malicious authorized runner can exfiltrate any secret it is permitted to request. TEE policy is an additional gate with the enforcement limits in §12.9, not a substitute for runner accountability.

13. Parameters

Parameter	Default	Tunable by
<code>MIN_PROXY_STAKE</code>	1,000 token units	Consensus code constant
<code>PROXY_SOAK_PERIOD_BLOCKS</code>	100	Consensus code constant
Deregistration delay	0 (<code>unbond_at = current block</code>)	Consensus code behavior
<code>DEFAULT_N</code>	5	Consensus code constant
<code>DEFAULT_T</code>	4 (<code>N3f1</code> for <code>n=5</code>)	Consensus code constant
Override <code>n</code> wire range	1–255 (<code>u8</code>)	Protocol codec
<code>MAX_PRIOR_COMMITTEE_EPOCHS</code>	4	Consensus code constant
<code>RESHARE_INTERVAL_BLOCKS</code>	1,296,000 ($\approx$ 15 days at 1 s/block)	Consensus code constant
<code>RESHARE_GRACE_BLOCKS</code>	100	Consensus code constant
<code>RESHARE_SAFETY_MARGIN</code>	1	Consensus code constant
<code>DKG_BOND</code>	1,000 token units	Consensus code constant
<code>DKG_FINALIZE_BLOCKS</code>	7,200 ($\approx$ 2 hours at 1 s/block)	Consensus code constant
<code>DKG_TIMEOUT_SLASH</code>	50 token units	Consensus code constant
Third-party DKG-expiry reward	1 token unit	Consensus code constant
<code>REQUEST_FRESHNESS_BLOCKS</code>	384	Consensus code constant
<code>RELEASE_FEE_PER_RECEIPT</code>	1 token unit	Consensus code constant
<code>MAX_RELEASE_OVERDRAFT</code>	16 token units	Consensus code constant
<code>LIVENESS_CHALLENGE_BOND</code>	100 token units	Consensus code constant
<code>MIN_CHALLENGE_DELAY_BLOCKS</code>	416	Consensus code constant
<code>LIVENESS_RESPONSE_BLOCKS</code>	64	Consensus code constant
<code>LIVENESS_CHALLENGE_MAX_AGE_BLOCKS</code>	480	Consensus code constant
<code>LIVENESS_UNANSWERED_DECREMENT_BPS</code>	100	Consensus code constant
<code>HEALTH_ATTRITION_EPOCH_BLOCKS</code>	7,200	Consensus code constant

Parameter	Default	Tunable by
<code>MAX_HEALTH_LOSS_PER_EPOCH_BPS</code>	1,000	Consensus code constant
Committee weight floor	500 bps	Consensus code constant
Governance review threshold	1,000 bps	Consensus code constant
<code>GOVERNANCE_REVIEW_BLOCKS</code>	2,592,000 ($\approx$ 30 days)	Consensus code constant
<code>MAX_SECRETS_PER_ACCOUNT</code>	256	Consensus code constant
<code>MAX_VERSIONS_PER_SECRET</code>	32	Consensus code constant
<code>MAX_ACL_ACTORS</code>	64	Consensus code constant
Proxy request timeout	5,000 ms	Local runner configuration
Served-request replay cache	65,536 entries	Daemon code constant
<code>MIN_TLOCK_LEAD_BLOCKS</code>	16	Consensus code constant
<code>MAX_TLOCK_REGISTRATIONS_PER_BLOCK</code>	256	Consensus code constant
<code>TLOCK_RETENTION</code>	256 blocks	Consensus code constant

Except for local client/daemon settings, these values are compile-time constants in the current implementation. Changing one requires a consensus-compatible software upgrade; there is no generic on-chain parameter store that tunes this table at runtime.

14. Implementation Status and Mainnet Gate

The implementation is tracked by `cbss-review.sh`, `cbss-review.md`, and `cbss-coverage-manifest.tsv`; those artifacts, not this summary, are the release-readiness source of truth. The latest checked-in full review record reports **NOT COMPLETE - TESTS FAILING**, a 67% completion estimate, and 9 of 10 cargo targets passing. It leaves DOD-01 and final DOD-13 sign-off outstanding and records the DKG, release-path, CLI/liveness, full-devnet, TEE, stress, and adversarial gates as partial.

Implemented surfaces include the threshold-IBE primitives, Commonware-backed production DKG path, PSS reshare machinery, chain storage and handlers, pairing-verified receipt admission, runner threshold fanout/decryption, QUIC daemon plumbing, CLI transaction builders, liveness lifecycle, and slashing evidence handlers. These surfaces do not imply mainnet readiness.

The current implementation also has material cross-component limits described normatively above: best-effort receipts are not a complete audit trail; `cbss-client` and the node disagree on `release_id`; optional client-side `hire_id` signing is incompatible with the chain request codec; non-N3f1 threshold overrides do not match the production DKG polynomial; old shares can be deleted before rotation confirmation; non-Nitro `tee_required` accepts the soft-fail binding path; a same-scope DKG retry can overwrite expired-ceremony sabotage evidence; proxy registration only

rejects all-zero HPKE/DKG keys; and the PVM exposes its host process environment through `os.environ` despite the interpreter configuration flag; checked-codec plaintext allocations cannot all be actively overwritten. No accepted external cryptographic review report is present in the repository.

14.1 Out of Scope

- Forward-secret reshare (would require re-encryption of all anchored ciphertexts on every rotation).
- Plaintext leak slashing (handled by on-chain governance, not on-chain evidence).
- Zero-knowledge proofs of correct partial-sign computation (the pairing check on chain is sufficient for the threat model).
- Hierarchical secrets (sub-keys, scoped namespaces beyond per-account).
- Push-based re-fetch on policy change (runners resolve authorization at operation time).

15. Relationship to the Technical Whitepaper

This document layers on, but never supersedes, the Technical Whitepaper. Specifically:

- Actors, messages, timers, reentrancy, and per-handler resource limits follow Technical Whitepaper §3. The current PVM does not clear `os.environ`; its host process must be launched with a scrubbed environment as described in §11.3.
- Fee markets (Cycles, Cells) and per-block basefee adjustment follow Technical Whitepaper §4 and §17. CBSS instructions consume both meters per §8 of this document.
- State rent for CBSS chain storage (metadata, release keys, receipts, challenges) follows the standard system-actor accounting at 0x04. Off-chain plaintext incurs no rent (it does not exist on chain); CBFS objects holding wrapped payloads bill under CBFS storage rent.
- The Runner Marketplace (Technical Whitepaper §5) is the consumer of release requests. Job dispatch integrates with CBSS via the actor's `secrets.read` manifest entitlement; the assigned runner is the only eligible signer of release requests for the job's duration.
- CBFS (Storage Whitepaper) is the data plane for the encrypted payload. The current owner path sends an owner-signed ciphertext envelope to the `cbssd` ingest service, which fans it out to CBFS nodes with short-lived operator-signed blob capabilities. The runner resolves the content-addressed pointer through configured CBFS-node HTTP endpoints, a mounted path, or an HTTP fallback. The CBFS pointer on `SecretVersion` is the only chain-resident reference.

- Consensus, randomness, and networking follow Technical Whitepaper §6. Current committee selection derives a deterministic seed from the block hash, height, requested count, and scope, then performs health-weighted sampling over eligible proxies.
- Governance, upgrades, and system actor hot-code replacement follow Technical Whitepaper §11. CBSS's mainnet flag-flip is a governance proposal contingent on external cryptographic review of `cbss-crypto`.
- Entitlements (Technical Whitepaper §15) gate `secrets.read` at the manifest level; CBSS is the second, fine-grained layer (per-key, per-version, per-actor).
- TEE state is consulted when `policy.tee_required = true`, subject to the Nitro versus non-Nitro enforcement boundary in §12.9; its records are currently persisted under `0x04` despite the logical `0x05` allocation.

Where this document and the Technical Whitepaper disagree on shared platform values or procedures, the Technical Whitepaper is the normative reference. For CBSS wire formats and consensus behavior, the protocol codec and deployed node handler are the executable authority.