Embedded Secret Store: Theory, Architecture, and Implementation Guide

I. Embedded Secret Store: Theory, Architecture, and Implementation Guide

Document Type: Diagnostic Report for Engineers and Technical Implementation Developers
Version: 1.0
Reference: `compiler-service/SECRET-STORE/structure.md`
Last Updated: 2026-02-19

A. 1. Executive Summary

Description and Implications (Section 1): The executive summary orients engineers and technical developers to the embedded secret store's purpose, architecture, and value proposition. It establishes that the compiled binary itself is the vault—secrets are stored encrypted in a mutable tail (caboose) at the end of the binary. No external credential managers, cloud secret stores, or database-backed tables are required. The design enables tiered access: Layer 1 (proof-only) for daily operations, Layer 2 (proof plus user-held prime half) for high-value secrets. Zero-knowledge proofs over PII allow verification without revelation. The mutable caboose supports add/remove/update of secrets post-compilation without recompilation. The commitment file (containing PRIME_PART_B) is generated at compile time and must be kept private—never distributed with the binary. Implications for engineering: traditional architectures centralize secrets and create single points of failure; the embedded store inverts this, distributing secrets across binaries. Each binary is its own vault; compromise of one affects only that user. The design aligns with zero-trust: never store what you can prove, never trust without verification. Implementers must design stateless binaries, enforce policy checks before every access, and ensure zeroization of sensitive material after use. This summary is the entry point—readers seeking implementation detail proceed to architecture (§2), binary layout (§3), and cryptographic foundations (§4).

The **Embedded Secret Store** is an advanced capability of the ENI6MA Hybrid Circuit Variant that transforms the compiled binary itself into a self-contained cryptographic vault. Rather than relying on external credential managers, cloud secret stores, or database-backed credential tables, secrets—including personally identifiable information (PII), passwords, API keys, and private keys—are stored encrypted within a mutable section at the end of the binary called the **caboose**. The binary is simultaneously the authenticator (via interactive proof-of-knowledge) and the secure store. Access is tiered: some secrets require only proof completion (Layer 1), while high-value secrets require proof plus a user-held prime half (Layer 2). The design enables zero-knowledge proofs over PII, policy-based access control, and mutable credential management without recompilation. This document provides a full diagnostic and theoretical foundation for engineers implementing or integrating this feature.

Key Takeaways:

- **Binary-as-Vault:** All secrets reside encrypted in the binary tail; no external files.
- **Tiered Encryption:** Layer 1 (proof-only) and Layer 2 (prime-required) control access granularity.
- **Zero-Knowledge PII:** Prove possession of SSN, DOB, etc. without revealing values.
- **Mutable Post-Compilation:** Add/remove secrets without changing circuit identity.
- **Commitment Separation:** Binary safe to distribute; commitment file is private key material.

Why This Matters for Engineering: Traditional credential architectures centralize secrets in databases, key management services, or environment variables. Each of these introduces a single point of failure: compromise of the credential store exposes all users. The embedded secret store inverts this model. Secrets are distributed across compiled binaries; each binary is its own vault. An attacker who steals one binary gains only that user's encrypted blob—and decryption requires both proof-of-knowledge (ceremony) and, for high-value data, a user-held prime half. This architecture eliminates the "credential warehouse" attack surface and aligns with zero-trust principles: never store what you can prove, and never trust without verification. For implementers, this means designing stateless binaries that receive context at runtime, enforcing policy checks before every access, and ensuring zeroization of sensitive material after use.

B. 2. Architecture Overview

Description and Implications (Section 2): The architecture of the embedded secret store defines how components interact, what invariants they enforce, and how the system scales to diverse deployment contexts. Understanding this overview is prerequisite for any implementation work: it establishes the mental model that ties together the binary layout, cryptographic foundations, access control, and integration patterns. The design deliberately inverts traditional credential architectures: instead of a central store that many clients query, each binary is a self-contained vault. This has profound implications. First, there is

no network dependency for secret retrieval in normal operation—the binary has everything it needs locally. Second, compromise of one binary does not cascade; attackers must target each vault individually. Third, the verifier never holds secrets; it only validates proof correctness. For engineers, this means the compiler's role is to produce a complete, self-sufficient artifact; the runtime's role is to enforce policy and perform key derivation; the ledger's role is to prevent replay. Dependencies flow one way: compiler → binary → user; verifier and ledger are consumers of proof metadata, not secret holders. The architecture also implies that all logic for access control—policy lookup, proof validation, key derivation—must live inside the binary. There is no server-side "secret service" to delegate to. This design choice maximizes portability and minimizes trusted computing base, but it places significant implementation burden on the binary. Extensibility (new policy types, new section types) must be designed into the caboose format from the start, since changing the binary layout requires recompilation and potentially migration of existing keystores.

1) 2.1 Design Principles: **Description, Context, Implications, Dependencies:** The design principles table codifies non-negotiable constraints that every implementation must satisfy. Each principle maps directly to implementation decisions: Binary-as-Vault means no code path should write secrets to external files; Tiered Encryption means two distinct key derivation functions; Stable Identity means the circuit hash computation must exclude the caboose. Engineers implementing from scratch should treat this table as a checklist: if any principle is violated, the system's security properties may be compromised. The principles also encode dependencies. For example, Mutable Caboose depends on Stable Identity—if circuit_hash included the caboose, every keystore modification would change identity, breaking ledger commitments and manifest versioning. Two-Layer Integrity depends on the delimiter-based parsing: the immutable zone must end at a well-defined boundary so that circuit_hash and integrity_hash can be computed over the correct byte ranges. When extending the system (e.g., adding Layer 3 or a new proof type), verify that new behavior does not violate any principle. The Normalized Schema (V4) principle has implications for data modeling: policies, groups, and values are separate entities with foreign keys, which enables reuse (one "Bank of America" policy applied to many values) and simplifies audit (list all values in a group).

The embedded secret store adheres to a set of non-negotiable design principles documented in the structural specification. These principles ensure interoperability, security, and predictable behavior across implementations.

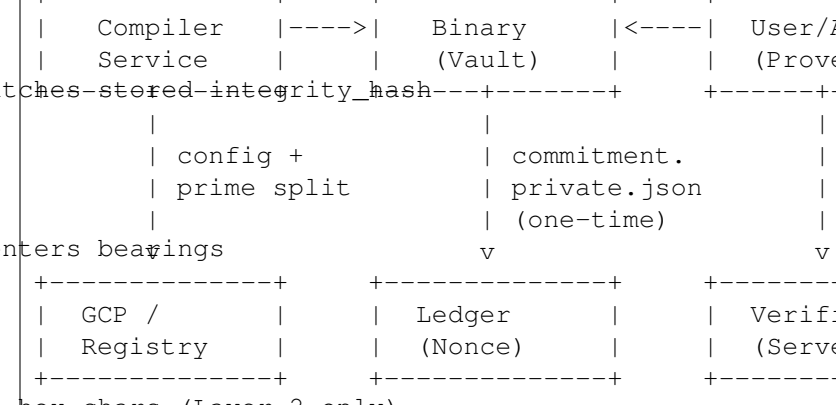
Principle	Description	Implementation Implication
Binary-as-Vault	All secrets stored encrypted within the binary tail (caboose)	No `.env`, keychain, or cloud secret store required
Embedded Gate Secrets	Core secrets (6UPPER, 6NUMBER, 12MIXED) remain in immutable zone	Basic proof operations work without caboose
Tiered Encryption	Layer 1 (proof-only) and Layer 2 (prime-required)	Different key derivation paths per layer
Normalized Schema (V4)	Policies, groups, labels, values with foreign keys	Reusable policies; canonical value storage
Stable Identity	Circuit hash computed over immutable zone only	Caboose mutations do not alter circuit identity
Mutable Caboose	Keystore can grow/shrink post-compilation	Append/update encrypted sections with fresh nonces
Two-Layer Integrity	Separate hashes: circuit_hash (stable) vs integrity_hash (dynamic)	Tamper detection; versioning
PII Zero-Knowledge Proof	Prove possession without revealing values	Ceremony uses PII for navigation; verifier sees only validity

2) 2.2 High-Level Data Flow: **Description, Context, Implications, Dependencies:** The data flow diagram traces a single request (e.g., `pii-get SSN`) from entry to exit. It serves as the canonical reference for understanding the order of operations—a critical dependency chain. Policy lookup must occur first: without knowing which layer contains SSN, the system cannot determine whether PRIME_PART_B will be required. Integrity check before interactive proof prevents operation on tampered binaries. The proof must complete before key derivation, since bearings are an input to the key. This ordering has implications for error handling: if

integrity fails, abort immediately and never prompt for proof or PRIME_PART_B (doing so could leak information about what the attacker is attempting). For Layer 2, PRIME_PART_B input is the gating factor—without it, decryption cannot proceed. Dependencies: Policy Lookup → Integrity Check → Interactive Proof → (PRIME_PART_B for L2) → Key Derivation → Decrypt. Implementers must not reorder or skip steps. The flow also implies that the binary must have access to the policy section (unencrypted but MAC-signed) before any decryption; thus the policy section layout and MAC verification are prerequisite to the rest of the flow.

```
User Request (e.g., pii-get SSN)
        |
        v
+-------------------+
| Policy Lookup     |  → SSN.read_policy = 12MIXED+PRIME → Layer 2
+---------+---------+
          |
          v
+-------------------+
| Integrity Check   |  → BLAKE3(binary) matches stored integrity_hash
+---------+---------+
          |
          v
+-------------------+
| Interactive Proof |  → 12 rounds; user enters bearings
+---------+---------+
          |
          v
+-------------------+
| PRIME_PART_B Input|  → User provides 128 hex chars (Layer 2 only)
+---------+---------+
          |
          v
+-------------------+
| Key Derivation    |  → K = BLAKE3("ENI6MA prime_vault v1", P || bearings)
+---------+---------+
          |
          v
+-------------------+
| Decrypt Caboose   |  → XChaCha20Poly1305(K, nonce, ciphertext)
+---------+---------+
          |
          v
+-------------------+
| Return Value      |  → SSN in memory; zeroize after use
+-------------------+
```

3) 2.3 Component Interaction Diagram: **Description, Context, Implications, Dependencies:** This diagram shows the three primary actors (Compiler Service, Binary, User/Agent) and their relationship to supporting infrastructure (GCP/Registry, Ledger, Verifier). The compiler produces the binary and commitment file; it does not participate in runtime authentication. The binary is the vault—it receives the commitment data once at setup and thereafter operates autonomously. The user (or an automated agent acting on the user's behalf) completes the ceremony and provides PRIME_PART_B when required. Dependencies flow as follows: Compiler depends on GCP/Registry for distribution; Binary depends on Ledger for nonce consumption; User depends on Verifier for challenge issuance. The Verifier never receives the commitment file or any secrets; it only validates proofs against the circuit_hash and ledger state. Implication: the compiler service must be able to read the commitment file from the build output (via COMMITMENT_PATH) and include it (or relevant parts) in the compile response. The commitment is never persisted to GCP in the current design—it is returned in the response for the user to save offline. For deployment, the binary can be distributed through any channel (download, USB, air-gap); the critical constraint is that the commitment file and PRIME_PART_B are handled separately from the binary and never stored in cloud.



The compiler produces the binary and commitment file. The binary is distributed (or downloaded) to the user's environment. The user completes the setup ceremony and may add PII/passwords to the caboose. When authentication is required, the verifier requests a challenge; the user runs the binary to generate a proof; the proof is submitted; the ledger consumes the nonce; the verifier validates. At no point does the verifier hold secrets; it only verifies proof correctness and nonce freshness.

C. 3. Binary Layout and Caboose Structure

Description and Implications (Section 3): The binary layout defines where secrets live, how they are protected, and how the system maintains a stable identity while allowing post-compilation modifications. This section is foundational for implementers who must parse, validate, and modify the caboose. The key architectural decision is the split between immutable and mutable zones. The immutable zone contains everything that defines the circuit: compiled code, gate secrets, entropy pool, and encrypted config. The circuit hash is computed over this zone only, which means that changes to the caboose (add/remove secrets) do not alter the binary's identity in the ledger or manifest. This has significant implications. First, a user can update their keystore

across binary versions without invalidating prior commitments—as long as the circuit_hash (immutable zone) is unchanged. Second, integrity_hash covers the full binary, so any tampering (including caboose modification by an attacker) is detectable. The delimiter `\x00<<ENI6MA_TAIL>>\x00` is the fixed boundary; parsers must locate it by scanning from the end of the file. The caboose structure—tail header, policy section, Layer 1, Layer 2, optional sections, checksums—follows a fixed layout with offsets stored in the header. This design enables forward compatibility: new section types (e.g., 0x05) can be added; older parsers skip unknown types. Dependencies: the byte-level parsing algorithm in §3.4 is the authoritative sequence; any deviation risks parse failures or security bypass. Implementers must also account for the 4-shard KEM versus 2-part prime split discrepancy: the current build uses a different model than the specification, requiring adaptation for production.

1) 3.1 Immutable vs Mutable Zones: The hybrid binary is partitioned into two logical zones separated by a fixed delimiter. Understanding this split is essential for correct hash computation and migration logic.

Immutable Zone (compile-time fixed): - Compiled Rust executable - PRIME_PART_A (512-bit) and PRIME_PART_B (embedded, for ENCRYPTED_CONFIG decryption) - Gate secrets: gate_secret_12mixed, gate_secret_6upper, gate_secret_6number - Zone mappings, bearings, key_codes, alphabets (3 sets) - Embedded entropy pool (e.g., 159 × 512-bit entries) - CONFIG_SALT + ENCRYPTED_CONFIG (BLAKE3-XOF encrypted)

Delimiter: `\x00<<ENI6MA_TAIL>>\x00` (17 bytes)

Mutable Zone (caboose): - Tail header (160 bytes): magic, version, circuit_hash, policy/layer offsets, keystore_version - Policy section (0x01): unencrypted but MAC-signed with circuit_hash - Layer 1 section (0x02): XChaCha20Poly1305 encrypted with proof-derived key - Layer 2 section (0x03): XChaCha20Poly1305 encrypted with prime-derived key - Ledger URL section (0x04): optional default ledger URL for verifier - Final checksums: circuit_hash (stable), integrity_hash (full binary)

Description, Context, Implications, Dependencies (3.1): The immutable zone contains all compile-time constants and secrets. PRIME_PART_A and PRIME_PART_B (or equivalent in 4-shard) are embedded; CONFIG_SALT and ENCRYPTED_CONFIG protect gate secrets. The delimiter is 17 bytes; its placement ensures the caboose is appended after all immutable content. The mutable zone starts immediately after the delimiter. Dependencies: circuit_hash must be computed over bytes [0, delimiter_start); integrity_hash over [0, end). Any tool that modifies the binary (e.g., keystore add) must preserve the immutable zone exactly. Implication: write operations append or overwrite only the caboose; never modify bytes before the delimiter. The tail header (160 bytes) is the first part of the mutable zone; it stores offsets for policy, Layer 1, Layer 2. When the keystore is modified, only Layer 1/Layer 2 ciphertexts and the integrity_hash change; the tail header offsets are updated to point to the new section locations.

2) 3.2 Section Types and Offsets:

Section Type	Hex	Purpose
Policy	0x01	Layer assignments, policies; MAC-signed
Layer 1	0x02	Proof-only vault (DOB, phone, wifi, low-sensitivity PII)
Layer 2	0x03	Prime vault (SSN, bank PINs, private keys)
Ledger URL	0x04	Default ledger base URL for verifier

The tail header stores offsets and lengths for each section, enabling sequential parsing. When the keystore is modified, Layer 1 and Layer 2 ciphertexts are re-encrypted with fresh nonces; the integrity_hash is recomputed.

Description, Context, Implications, Dependencies (3.2): Each section type has a hex identifier (0x01–0x04) and a defined structure. The policy section (0x01) is unencrypted but MAC-signed with circuit_hash, so tampering is detectable. Layer 1 (0x02) and Layer 2 (0x03) use XChaCha20Poly1305 with section-specific nonces. The Ledger URL section (0x04) is optional. Implementers must follow the section layout: 4-byte section_type, 4-byte section_length, then payload. For encrypted sections, the payload is nonce (24) + ciphertext + auth_tag (16). Dependencies: the tail header stores policy_offset, policy_length, layer1_offset, layer1_length, layer2_offset, layer2_length. Parsers read these offsets to locate each section. When adding new section types, reserve a new hex code and document the layout; existing parsers should skip unknown types (forward compatibility). The integrity_hash is computed over all bytes before the final 64-byte checksum block; it must be recomputed after any section modification.

3) 3.3 Note on 4-Shard KEM (Current Implementation): The *current* eni6ma-btc-hybrid build uses a **4-shard KEM** (X25519 + Kyber768) for prime reconstruction, rather than a 2-part PRIME_PART_A/PRIME_PART_B split. Shards 0–2 are embedded; shard 3 is sealed with hybrid KEM. The structural specification describes a 2-part model where PRIME_PART_B is given to the user for Layer 2 access. Implementers must either: - Adapt the 4-shard model to derive a user-held "prime equivalent" (e.g., shard 3 or a derived value), or - Migrate to a 2-part split model as specified for production.

Description, Context, Implications, Dependencies (3.3): The structural specification describes a 2-part prime split (PRIME_PART_A embedded, PRIME_PART_B user-held). The current eni6ma-btc-hybrid build uses a 4-shard KEM with X25519 + Kyber768; shards 0–2 are embedded, shard 3 is sealed with hybrid KEM. This creates an implementation gap: the spec assumes the user holds PRIME_PART_B for Layer

2; the 4-shard model does not have an equivalent user-held component by default. Implications: (1) implementers must map shard 3 (or a derived value) to the "prime equivalent" for Layer 2 key derivation, or (2) migrate the build to a true 2-part split for production. Dependencies: key derivation for Layer 2 (§4.1) assumes P = PRIME_PART_A PRIME_PART_B; the 4-shard model must produce an equivalent P from shard reconstruction. The commitment file schema may need extension to carry shard metadata instead of part_b_hex.

4) 3.4 Byte-Level Parsing Algorithm: Implementers parsing the caboose must follow a strict sequence:

1. **Locate delimiter:** Scan binary from end for `\x00<<ENI6MA_TAIL>>\x00` (17 bytes). The caboose starts immediately after this delimiter.
2. **Read tail header:** 160 bytes. Extract: magic (16), version (4), circuit_hash (32), policy_offset (8), policy_length (8), layer1_offset (8), layer1_length (8), layer2_offset (8), layer2_length (8), policy_version (4), keystore_version (4), cloud_sync_epoch (8), manifest_url_hash (32), reserved (16).
3. **Validate magic:** Must equal "ENI6MA_TAIL_V3" (or current version string).
4. **Read policy section:** At policy_offset, read 4-byte section_type (0x01), 4-byte section_length, then policy_manifest_json (section_length - 8 bytes), then 32-byte policy_mac. Verify MAC with BLAKE3(key=circuit_hash, msg=policy_manifest_json).
5. **Read Layer 1 section:** At layer1_offset, read section_type (0x02), section_length, nonce (24), ciphertext (section_length - 28 - 16), auth_tag (16). Decrypt only after proof-derived key is available.
6. **Read Layer 2 section:** Similarly at layer2_offset. Decrypt only after prime reconstruction and proof.
7. **Read optional sections:** Iterate remaining bytes; section_type 0x04 is ledger URL. Unknown section types should be skipped (forward compatibility).
8. **Read final checksums:** Last 64 bytes: circuit_hash (32), integrity_hash (32). Verify integrity_hash = BLAKE3("ENI6MA integrity_hash v1", all_bytes_before_checksums).

Failure at any step must abort; never proceed with partially parsed or unverified data.

Description, Context, Implications, Dependencies (3.4): This algorithm is the authoritative parsing sequence. Implementers must not deviate. Step 1: locate delimiter by scanning from end (handles appended padding). Step 2: read 160-byte tail header with fixed field layout. Step 3: validate magic string (version compatibility). Step 4: read policy section; verify MAC with BLAKE3(key=circuit_hash, msg=policy_manifest_json). Step 5–6: read Layer 1 and Layer 2 sections; do not decrypt until keys are available. Step 7: iterate remaining bytes for optional sections (0x04 ledger URL; skip unknown). Step 8: read final 64 bytes (circuit_hash, integrity_hash); verify integrity_hash. Dependencies: all offsets come from the tail header; if offsets are corrupt, parsing fails. Implications: never trust unverified data; abort on any validation failure. Use constant-time comparisons where applicable to avoid timing leaks. The algorithm assumes little-endian byte order for multi-byte fields unless otherwise specified in the structure document.

D. 4. Cryptographic Foundations

Description and Implications (Section 4): Cryptographic foundations define how keys are derived, what cipher suite is used, and how domain separation prevents key reuse across different purposes. This section is critical for security: incorrect key derivation or nonce handling can completely undermine the system. The design uses BLAKE3 for hashing and key derivation, and XChaCha20Poly1305 for authenticated encryption. Each layer (embedded config, Layer 1, Layer 2) has a distinct key derivation path. Domain separation is enforced via context strings—e.g., "ENI6MA proof_vault v1" for Layer 1 and "ENI6MA prime_vault v1" for Layer 2. This prevents an attacker who obtains a Layer 1 key from using it to decrypt Layer 2 (and vice versa). Nonce handling is equally important: each encryption uses a unique 24-byte nonce; on keystore modification, new nonces are generated. Implementers must never reuse a nonce with the same key—such reuse can allow an attacker to recover plaintext. The implications for implementation are strict: use the exact context strings from the spec; do not invent new ones without documenting; zeroize keys and plaintext after use. Dependencies: key derivation depends on gate_secret (Layer 1) or full prime (Layer 2), plus bearings. The bearings come from the interactive proof; thus proof validation must complete before derivation.

1) 4.1 Key Derivation Paths: **Embedded Config (gate secrets):**

```
P = PRIME_PART_A  PRIME_PART_B  (both embedded)
config_key = BLAKE3(P || CONFIG_SALT)
gate_secrets = BLAKE3-XOF decrypt(ENCRYPTED_CONFIG)
```

Layer 1 (proof-only):

```
User completes 6UPPER/6NUMBER/6MIXED/12MIXED proof
bearings = [U,D,L,R,F,B,...]  (from user input)
layer1_key = BLAKE3_derive_key("ENI6MA proof_vault
plaintext = XChaCha20Poly1305.decrypt(layer1_key, n
```

Layer 2 (prime-required):

```
User completes 12MIXED proof + provides PRIME_PART_
P = PRIME_PART_A  PRIME_PART_B
layer2_key = BLAKE3_derive_key("ENI6MA prime_vault
plaintext = XChaCha20Poly1305.decrypt(layer2_key, n
```

Description, Context, Implications, Dependencies (4.1): Key derivation is the gateway to decryption. The embedded config path recovers gate secrets at startup using both prime parts (both embedded in the binary for config decryption). Layer 1 uses gate_secret plus bearings; Layer 2 uses full prime plus bearings. Dependencies: gate_secret comes from ENCRYPTED_CONFIG decryption; bearings from user input during proof; PRIME_PART_B from user input at runtime

for Layer 2. Implications: the derivation input (gate_secret ‖ bearings or P ‖ bearings) must be identical between encrypt and decrypt; any change (wrong bearings, wrong PRIME_PART_B) produces a different key and causes AEAD auth failure. Implementers must use BLAKE3's key derivation API (e.g., blake3::Hasher::derive_key) with the exact context string; do not use raw BLAKE3 hash as key—use derive_key to ensure proper key expansion.

2) 4.2 Cipher Suite:

- **AEAD:** XChaCha20Poly1305 (256-bit key, 192-bit nonce)
- **Hashing/derivation:** BLAKE3
- **Integrity:** BLAKE3 for circuit_hash and integrity_hash; Poly1305 tags for AEAD

Description, Context, Implications, Dependencies (4.2): XChaCha20Poly1305 provides 256-bit security with a large nonce (192-bit), reducing the risk of nonce reuse compared to ChaCha20-Poly1305. BLAKE3 is fast and cryptographically strong; it is used for both one-shot hashing and key derivation. Dependencies: the AEAD implementation must support 256-bit keys and 192-bit nonces (XChaCha20's extended nonce). Implications: do not substitute other AEADs (e.g., AES-GCM) without security review—the nonce size and construction differ. The Poly1305 tag (16 bytes) is appended to ciphertext; decryption must verify the tag before returning plaintext. Use constant-time comparison for tag verification.

3) 4.3 Nonce Handling: Each Layer 1 and Layer 2 section uses a unique 24-byte nonce. On keystore modification: 1. Generate new random nonce 2. Re-encrypt entire layer payload with derived key and new nonce 3. Update caboose; recompute integrity_hash 4. Zeroize old nonce and plaintext from memory

Description, Context, Implications, Dependencies (4.3): Nonce uniqueness is critical for AEAD security. Reusing a nonce with the same key can expose plaintext. The specification requires a new random nonce for each encryption (keystore modification). Dependencies: a cryptographically secure RNG (e.g., getrandom) must be used. Implications: never derive nonces from deterministic inputs (e.g., counter) when the same key could be used for multiple encryptions—use random. When modifying the keystore, both Layer 1 and Layer 2 get new nonces even if only one layer changed. Zeroization prevents the old nonce from lingering in memory where it could be captured by a cold-boot or debugging attack.

4) 4.4 Domain Separation and Context Strings: Cryptographic domain separation prevents key reuse across different purposes. The specification uses distinct context strings for each derivation:

Context String	Purpose
`ENI6MA proof_vault v1`	Layer 1 key derivation
`ENI6MA prime_vault v1`	Layer 2 key derivation
`eni6ma-identity-seed-v1`	Identity keypair derivation (4-shard)
`eni6ma-hybrid-kem-v1`	Hybrid KEM key derivation
`ENI6MA circuit_hash v1`	Circuit hash computation
`ENI6MA integrity_hash v1`	Integrity hash computation
`STREAM`	Entropy pool stream cipher (BLAKE3-XOF)

Context String	Purpose

Implementers must never reuse a context string for a different purpose. When adding new cryptographic operations, assign a new context string (e.g., increment version or add purpose suffix).

Description, Context, Implications, Dependencies (4.4): Domain separation ensures that a key derived for one purpose cannot be used for another. Without it, a key leaked from Layer 1 could decrypt Layer 2. Each context string (e.g., "ENI6MA proof_vault v1") is incorporated into the BLAKE3 derivation input. Dependencies: BLAKE3 derive_key accepts a context string; the spec's strings must be used exactly. Implications: when extending the system (e.g., new proof type, new vault layer), add a new context string. Version suffix (v1) allows future algorithm changes without breaking existing binaries.

E. 5. Tiered Access Model

Description and Implications (Section 5): The tiered access model is the core of the embedded secret store's security design. It partitions secrets into two layers based on sensitivity and access frequency. Layer 1 holds lower-sensitivity data (DOB, phone, WiFi passwords) and requires only proof completion—no user-held prime. Layer 2 holds high-value secrets (SSN, bank PINs, private keys) and requires proof plus PRIME_PART_B. This stratification has several implications. First, daily operations (e.g., proving age, retrieving a WiFi password) do not require the user to produce PRIME_PART_B, reducing friction and exposure of that critical factor. Second, compromise of the binary alone is insufficient for Layer 2—the attacker must also obtain PRIME_PART_B from the user's cold storage. Third, the unlock flows differ: Layer 1 never prompts for PRIME_PART_B; Layer 2 always does when the requested secret is in that layer. Policy lookup determines which layer a given key belongs to; the policy section maps keys to read_policy (e.g., SSN.read_policy = 12MIXED+PRIME). Error handling must not reveal which layer was targeted or whether the failure was due to wrong bearings versus wrong PRIME_PART_B—both must fail silently from the user's perspective (though internal logging may use error codes). Dependencies: policy evaluation runs before proof; the proof type (6UPPER, 12MIXED, etc.) constrains which layers are accessible (12MIXED required for Layer 2).

1) 5.1 Layer Comparison:

Aspect	Layer 1 (Proof-Only)	Layer 2 (Prime Vault)
Decryption key	gate_secret + bearings	full_prime + bearings
PRIME_PART_B	NOT needed	**REQUIRED** (user provides)
Proof type	6UPPER/6NUMBER/6MIXED/12MIXED	12MIXED only

Aspect	Layer 1 (Proof-Only)	Layer 2 (Prime Vault)
Typical contents	DOB, phone, wifi, low-sensitivity PII	SSN, bank PIN, private keys, 2FA codes
Use case	Daily access, ZK proofs	Rare access, high-value secrets
Modifiable	Yes (with write policy)	Yes (with 12MIXED+PRIME)

Description, Context, Implications, Dependencies (5.1): This table is the quick reference for implementers. Layer 1 uses proof-derived key (gate_secret + bearings); Layer 2 uses prime-derived key. PRIME_PART_B is the critical differentiator—Layer 2 cannot decrypt without it. Proof types: Layer 1 accepts 6UPPER/6NUMBER/6MIXED/12MIXED; Layer 2 requires 12MIXED only (stronger assurance). Typical contents guide policy design: SSN belongs in Layer 2; DOB might be Layer 1 for some use cases. Dependencies: policy lookup returns the layer for each key; the CLI or API uses this to determine whether to prompt for PRIME_PART_B. Implications: when adding a new secret, the user (or policy) must choose the layer; default recommendations should favor Layer 2 for PII.

2) 5.2 Unlock Flow (Layer 1):

1. Policy check: lookup read_policy → determine layer
2. Integrity check: verify BLAKE3 hashes
3. Interactive proof: user completes 6/12-round ceremony
4. Key derivation: BLAKE3(gate_secret ‖ bearings)
5. Decrypt Layer 1 section
6. Return requested value; zeroize after use

Description, Context, Implications, Dependencies (5.2): The Layer 1 flow is the common path for low-sensitivity secrets. Policy check identifies the layer; integrity check ensures binary authenticity; interactive proof captures bearings; key derivation produces layer1_key; decryption yields plaintext. Dependencies: policy section must be readable (MAC-verified) before step 1; gate_secret must be available (from ENCRYPTED_CONFIG at startup). Implications: the user never sees a PRIME_PART_B prompt for Layer 1 requests. Zeroization applies to plaintext and any intermediate keys after the value is returned. The flow is stateless with respect to prior unlocks—each request goes through the full sequence.

3) 5.3 Unlock Flow (Layer 2):

1. Policy check: read_policy = 12MIXED+PRIME → Layer 2
2. Integrity check
3. Interactive proof: 12-round ceremony
4. Prompt user for PRIME_PART_B (128 hex chars)
5. Reconstruct P = PRIME_PART_A PRIME_PART_B
6. Key derivation: BLAKE3(P ‖ bearings)
7. Decrypt Layer 2 section
8. Return value; zeroize PRIME_PART_B and plaintext from memory

Description, Context, Implications, Dependencies (5.3): Layer 2 adds the PRIME_PART_B step. The user must provide 128 hex characters (512 bits). The binary XORs PRIME_PART_A (embedded) with PRIME_PART_B (user input) to reconstruct the full prime, then derives the Layer 2 key. Dependencies: PRIME_PART_B must be correctly formatted (128 hex chars, no spaces); wrong length or invalid chars should fail fast with a user-friendly message (without revealing that it's a key format issue versus wrong value). Implications: PRIME_PART_B must never be written to disk, logged, or transmitted over the network in the clear. Memory holding PRIME_PART_B should be zeroized immediately after key derivation. The prompt should advise the user to retrieve from cold storage and return it there after use.

4) 5.4 Error Handling and Failure Modes: Implementers must handle the following failure modes without leaking information:

Failure	Behavior	Rationale
Wrong bearings	Fail silently; no retry hint	Prevents brute force
Wrong PRIME_PART_B	Fail silently; invalid decryption	Same
AEAD auth failure	Abort; do not return partial plaintext	Authenticated encryption
Policy lookup miss	Return "not found" or "unauthorized"	Don't reveal existence
Integrity hash mismatch	Refuse to operate	Tamper detected
Caboose truncated/corrupt	Abort parse; report corruption	Fail safely

All sensitive material (plaintext, keys, PRIME_PART_B) must be zeroized on any error path. Use `Zeroize` or equivalent trait in Rust; avoid leaving secrets in stack or heap after function return.

Description, Context, Implications, Dependencies (5.4): Failures must not leak information that aids brute force. Wrong bearings and wrong PRIME_PART_B produce the same user-visible outcome: "access denied" or a generic error code. Internal logging may use E003 (proof failed) vs E005/E006 (prime invalid / AEAD auth failed) for debugging, but never expose these to the attacker. AEAD auth failure means the decryption produced invalid plaintext (wrong key); abort immediately—never return partial plaintext. Policy lookup miss should return "not found" rather than "unauthorized" to avoid revealing whether the key exists. Integrity hash mismatch means tampering; refuse all operations. Dependencies: error codes (E001–E009) are documented in §10.6; all code paths must route to appropriate codes without including sensitive data in messages or logs.

F. 6. PII Vault and Zero-Knowledge Proofs

Description and Implications (Section 6): The PII vault and zero-knowledge proof capability address a fundamental compliance challenge: how to verify identity attributes without

storing raw PII. Traditional systems retain SSN, DOB, and other identifiers in databases, creating breach and regulatory risk. The embedded store supports two modes: proof (user enters PII during ceremony, value is not stored) and storage (PII encrypted in vault, retrieved when needed). The zero-knowledge property ensures the verifier learns only validity—whether the proof is correct—not the actual PII value. This enables age verification (prove age 21 without revealing DOB), format proofs (prove SSN matches XXX-XX-XXXX), and selective disclosure. Implications for implementers: the proof protocol must bind PII to bearing choices in a way that does not leak the value; the verifier's validation logic must operate on bearings and challenge structure only. The two-part challenge protocol (gate proof + PII proof) allows a single session to establish both identity and attribute possession. Integration with external verifiers (banks, government portals) requires a clear contract: challenge request → proof generation → proof submission → verifier validation. The verifier must not log proof payloads in a correlatable way. Dependencies: proof generation depends on gate secrets and challenge matrix; PII proof depends on user input (from memory or vault) to navigate the matrix.

1) 6.1 Two Modes: Proof vs Storage:

2) 6.1 Two Modes: Proof vs Storage: **PII Proof (Level 1 – no prime):** - Used for *verification*: proving you know a PII value - User enters PII interactively (from memory) during proof generation - Value navigates the challenge matrix; discarded after proof - Example: `./circuit respond-interactive --pii=SSN` → user types SSN → proof generated → value forgotten

PII Storage (Level 2 – prime required): - Used for *backup/recovery*: storing PII in encrypted vault - Values persist in Layer 2 caboose - Requires full vault unlock (gate proof + PRIME_PART_B) - Allows automated retrieval for agents or scripts (with appropriate auth)

Description, Context, Implications, Dependencies (6.1): Proof mode is for one-time verification: the user types PII from memory during the ceremony; the value is used to navigate the challenge matrix and is then discarded. No storage. Storage mode persists PII in the encrypted vault (Layer 1 or 2); retrieval requires vault unlock. Choose proof mode when the verifier needs only to confirm knowledge (e.g., "user knows their SSN"); choose storage when the user needs repeated access (e.g., autofill) or when agents must retrieve values. Dependencies: proof mode requires the binary to accept PII input at proof time; storage mode requires policy assignment and key derivation. Implications: proof-only flows minimize retention; storage flows require careful key management and backup strategy.

3) 6.2 Zero-Knowledge Property: The verifier learns only: - Whether the proof is valid (bearing choices correct for the challenge) - No information about the actual PII value

This supports: - Age verification (prove DOB > threshold without revealing DOB) - Format proofs (prove SSN matches XXX-XX-XXXX without revealing digits) - Selective disclosure (prove subset of attributes)

Description, Context, Implications, Dependencies (6.2): The verifier receives bearing choices and proof hash—not the PII value. Validity is a single bit: correct or incorrect. This enables compliance without retention: a liquor vendor can verify age 21 without ever seeing DOB; a bank can verify SSN ownership without storing it. Dependencies: the proof structure must encode "valid" without encoding the value; the bearing-to-value mapping must be computationally difficult to invert. Implications: verifiers should design their systems to never request or log PII; proof validity is sufficient for access decisions. Formal ZK property would require a security reduction; the design minimizes disclosure by construction.

4) 6.3 Two-Part Challenge Protocol: A single session can combine: 1. **Part 1:** Gate proof (6/12 rounds) 2. **Part 2:** PII proof(s) for SSN, DOB, etc.

Parts are cryptographically bound via a combined hash. The verifier validates the full chain without ever seeing PII values.

Description, Context, Implications, Dependencies (6.3): A session can combine gate proof (6 or 12 rounds) with one or more PII proofs. The combined hash binds all parts so that splitting or replaying individual components fails validation. Dependencies: Part 1 must complete before Part 2; the binding hash incorporates nonce, tau, and bearing sequences. Implications: implementers must ensure the combined proof structure is well-defined and that verifiers can validate the full chain. The protocol supports multiple PII attributes in one session (e.g., SSN + DOB) without multiple round-trips.

5) 6.4 Zero-Knowledge Property: Technical Detail: The zero-knowledge property arises from the structure of the proof protocol. The verifier receives: (a) the challenge matrix (public), (b) the prover's bearing choices (one per round), (c) the proof hash binding bearings to the nonce. The verifier can check that each bearing is "correct" for the round by recomputing the expected bearing from the challenge and the prover's secret—but the verifier does not have the prover's secret. Instead, the verifier checks consistency: the bearings, when combined with the challenge structure, produce a valid proof hash. The PII value (when used) influences which bearing is "correct" in each round; the verifier sees only the bearings, not the PII. Because each round uses independent entropy (from the pool and timestamp), the bearing sequence does not statistically leak the PII value. Formal verification of this property would require a security reduction; the design follows the principle that minimal disclosure (only validity) minimizes attack surface.

Description, Context, Implications, Dependencies (6.4): The ZK property arises from the proof structure: bearings are the only prover output; the verifier checks consistency with the challenge. The PII value influences which bearing is correct per round, but the verifier does not have the value—only the bearings. Independent entropy per round (from pool and timestamp) prevents statistical leakage. Implications: avoid any design that would allow the verifier to infer the value from the bearing sequence. The spec does not provide a formal proof; implementers should treat "minimal disclosure" as the guiding principle.

6) 6.5 PII Proof Integration with External Verifiers: When integrating with external verifiers (e.g., bank, government portal):

1. **Challenge request:** Verifier calls your service or the user's agent with nonce + challenge parameters (e.g., `pii_fields=["SSN"]`, `proof_level=12MIXED`).
2. **Proof generation:** User runs binary; binary loads challenge; user enters PII (or retrieves from vault if stored); binary produces proof payload.
3. **Proof submission:** Proof payload (bearings, hash, circuit_hash, tau) sent to verifier.
4. **Verifier validation:** Verifier has a matched circuit or validation logic; verifies proof hash, nonce consumption, circuit_hash authorization.
5. **Outcome:** Verifier grants or denies access. No PII transmitted; only proof validity.

Ensure the verifier does not log or store the raw proof payload in a way that could later be correlated with PII. The proof is bound to (nonce, tau); after ledger consume, it cannot be replayed.

Description, Context, Implications, Dependencies (6.5): External verifiers (banks, government) request challenges from your service or the user's agent. The user runs the binary to generate the proof; the proof is submitted; the verifier validates. Dependencies: challenge request includes nonce and pii_fields; proof generation requires binary, challenge, and user PII input. Implications: the verifier must have validation logic (matched circuit or equivalent); the ledger must consume the nonce to prevent replay. Verifiers should not store proof payloads with user identifiers in a way that enables correlation—prefer "proof valid for nonce N" with no persistent link to PII.

G. 7. Mutable Caboose Mechanics

Description and Implications (Section 7): The mutable caboose is what allows post-compilation secret management. Without it, every keystore change would require recompilation—impractical for users who add/remove passwords and PII over time. The key insight is the separation of circuit_hash (stable, immutable zone only) from integrity_hash (dynamic, full binary). circuit_hash identifies the binary for ledger and manifest; integrity_hash detects tampering. When the user adds a secret, the flow is: authenticate (proof + prime if Layer 2), decrypt current layers, apply modification, generate new nonces, re-encrypt, overwrite caboose, recompute integrity_hash. Implications: the binary file must be writable; read-only media cannot support keystore modifications. Forward secrecy is preserved because each modification uses fresh nonces—compromise of an old ciphertext does not aid decryption of the new one. Dependencies: modification requires full unlock (both layers if modifying across layers); the tail header and section offsets must be updated correctly.

1) 7.1 Stable vs Dynamic Identity:

2) 7.1 Stable vs Dynamic Identity:

- **circuit_hash:** BLAKE3 of immutable zone only. Does not change when caboose is modified.
- **integrity_hash:** BLAKE3 of entire binary including caboose. Changes on every keystore mutation.

Use circuit_hash for: - Ledger commitment - Manifest versioning - Binary identity in distributed systems

Use integrity_hash for: - Tamper detection - Full binary verification

Description, Context, Implications, Dependencies (7.1): circuit_hash excludes the caboose; integrity_hash includes it. Ledger commitments use circuit_hash so that keystore changes do not invalidate prior commitments. Manifest versioning can use circuit_hash to group binaries by identity. integrity_hash detects any change to the binary—including caboose corruption or attacker modification. Dependencies: both hashes use BLAKE3 with distinct context strings. Implications: when distributing a binary, record its integrity_hash; on load, recompute and compare. Mismatch means "do not trust."

3) 7.2 Modification Flow:

1. User requests add/remove/update (e.g., `keystore pii-add --key SSN --value "123-45-6789"`)
2. Authenticate: complete proof + provide PRIME_PART_B if Layer 2
3. Decrypt current Layer 1 and/or Layer 2
4. Apply modification (add/remove/update entry)
5. Generate new nonces
6. Re-encrypt with derived keys
7. Append updated sections to binary; overwrite caboose in place or rewrite tail
8. Recompute integrity_hash; update final checksum block
9. Zeroize all plaintext and keys from memory

Description, Context, Implications, Dependencies (7.2): Modification follows a strict sequence: authenticate, decrypt, apply change, re-encrypt with new nonces, overwrite caboose, recompute integrity_hash. Dependencies: write policy must allow the operation (e.g., 12MIXED+PRIME for Layer 2 adds); PRIME_PART_B required for Layer 2 modifications. Implications: the binary must be opened read-write; concurrent writers must serialize. Append-or-overwrite strategy: build new caboose in memory, then write once to avoid partial writes. Ensure filesystem sync before declaring success.

4) 7.3 Forward Secrecy: Each modification uses fresh nonces. Compromise of an old ciphertext does not aid decryption of future ciphertexts (assuming the derived key changes with content or nonce handling is sound).

Description, Context, Implications, Dependencies (7.3): Forward secrecy means that past ciphertexts cannot be decrypted even if the current key is later compromised. Fresh nonces per encryption ensure that each ciphertext is independent. Dependencies: the key derivation input (bearings, prime) may be the same across modifications; nonce uniqueness is what provides forward secrecy within the same key. Implications: never reuse nonces; always generate new random nonces on every re-encrypt. If the key itself is rotated (e.g., user changes ceremony), that would provide additional forward secrecy across sessions.

H. 8. Policy and Access Control

Description and Implications (Section 8): Policy and access control determine who can read, write, and prove for each secret. The policy levels (6UPPER, 6NUMBER, 6MIXED, 12MIXED, 12MIXED+PRIME) map to proof difficulty and prime requirement. Network ACLs add context-based restrictions: IP, domain, MAC, geographic. The V4 normalized schema separates policies, groups, and values—enabling reuse and audit. Implications for implementers: policy evaluation runs before every access; the policy section is MAC-signed so attackers cannot forge permissive policies. Context (IP, MAC, etc.) is collected at runtime and must be trustworthy—consider signed context from a deployment layer. Dependencies: policy section parsing; context injection mechanism; V4 schema for values with foreign keys to policies and groups.

1) 8.1 Policy Levels:

2) 8.1 Policy Levels:

Level	Read	Write	Prove
PUBLIC	Anyone	—	—
6UPPER	6-round uppercase proof	6UPPER+	6UPPER
6NUMBER	6-round numeric proof	6NUMBER+	6NUMBER
6MIXED	6-round mixed proof	6MIXED+	6MIXED
12MIXED	12-round mixed proof	12MIXED+	12MIXED
12MIXED+PRIME	12-round + PRIME_PART_B	12MIXED+PRIME	12MIXED

Description, Context, Implications, Dependencies (8.1): Policy levels are ordered by strength: 6UPPER (weakest) to 12MIXED+PRIME (strongest). Read and write can have different levels (e.g., read 6UPPER, write 12MIXED+). Prove level indicates which proof type is required. Dependencies: each value in the keystore has a policy_id; policy lookup returns the level for that value. Implications: assign stronger policies to sensitive secrets (SSN, bank PIN); weaker policies for low-sensitivity (WiFi password). The + suffix on write indicates "this level or stronger."

3) 8.2 Network ACLs: Per-secret policies can include: - **Whitelist/blacklist:** domain, IP range, MAC address, machine ID - **Geographic:** block VPN/Tor; allow only specific countries - **Organization:** e.g., "SSN accessible only from Bank of America systems"

Context (IP, MAC, machine ID) is collected at runtime and injected for policy evaluation.

Description, Context, Implications, Dependencies (8.2): Network ACLs restrict access based on environment. Whitelist: only these domains/IPs/MACs. Blacklist: block VPN/Tor or specific countries. Example: SSN accessible only from Bank of America systems. Dependencies: the deployment layer or runtime must provide context (IP, MAC, machine ID); the binary does not have native network visibility. Implications: context spoofing is a risk—an attacker with local access could fake context. For high-assurance, use signed context from a trusted enclave or TPM attestation.

4) 8.3 V4 Normalized Schema:

- **Policies:** Reusable; defined once, referenced by UUID
- **Groups:** Logical groupings (e.g., PII_financial, credentials_banking)
- **Values:** Secrets with foreign keys to policies and groups
- **Labels:** Canonical aliases (SSN = SocialSecurityNumber, DL = STATE_ID)

Description, Context, Implications, Dependencies (8.3): V4 schema separates policies (reusable rules), groups (logical groupings), values (secrets with policy_id and group_ids), and labels (canonical aliases). Policies are defined once and referenced by UUID. Groups enable bulk operations (e.g., "all PII_financial"). Labels allow multiple names for the same concept. Dependencies: the keystore JSON or caboose payload must conform to V4; migration from older schemas may be needed. Implications: design policies for reuse; use groups for audit ("list all values in credentials_banking").

I. 9. Commitment File and Separation of Concerns

Description and Implications (Section 9): The commitment file is the compiler's output that contains the full circuit configuration and—critically—PRIME_PART_B (or equivalent in 4-shard). It is generated once at compile time and must never be distributed with the binary. The separation is deliberate: the binary is safe to copy (encrypted caboose, no keys); the commitment file is private key material. Implications: the compiler service must return the commitment in the compile response but never persist PRIME_PART_B to GCP. The user downloads the commitment and stores it offline (like an SSH private key). If the commitment is backed up to cloud, it must be encrypted with a user-chosen password; PRIME_PART_B should never appear in cloud storage in cleartext. Dependencies: build.rs generates the commitment; compiler service parses COMMITMENT_PATH from build stderr; web UI adds a setup step to download and display PRIME_PART_B.

1) 9.1 commitment.private.json:

2) 9.1 commitment.private.json: Generated once at compile time; contains: - `circuit_hash` (SHA256 or BLAKE3 of binary) - Full config: alphabets, zone_mapping, bearings - Gate secrets metadata (not values) - `part_b_hex` (PRIME_PART_B as 128-char hex) — **never stored in cloud**

Description, Context, Implications, Dependencies (9.1): The commitment file contains circuit_hash, full config (alphabets, zone_mapping, bearings), gate secrets metadata (not values), and part_b_hex. It is the "private key" for the compiled identity. Dependencies: generated in build.rs after pool embedding; path emitted via cargo:warning=COMMITMENT_PATH=. Implications: treat as highly sensitive; encrypt before any cloud backup. The schema is documented in PRIVATE_JSON_SCHEMA.md.

3) 9.2 Binary vs Commitment:

Artifact	Safe to distribute?	Contains
Binary	Yes	Encrypted caboose; no keys
commitment.private.json	No	Full config; prime half; private key material

The binary can be copied to multiple devices. The commitment file must be treated like an SSH private key: offline, backed up, never in cloud storage.

Description, Context, Implications, Dependencies (9.2): The binary is distributable; the commitment is not. Copying the binary to a new device does not grant access to Layer 2 until the user provides PRIME_PART_B (from commitment) at runtime. Implications: distribution channels can freely share the binary; the commitment must travel through a secure channel (e.g., user downloads from authenticated compile response, then stores offline).

4) 9.3 Cloud Handling: If commitment is uploaded for backup: 1. Encrypt with user-chosen password before upload 2. Store as `commitment.private.enc` in private bucket 3. **Never** store PRIME_PART_B in cloud; return in compile response only; user saves offline

Description, Context, Implications, Dependencies (9.3): If users insist on cloud backup for the commitment, encrypt with a strong user-chosen password before upload. Store as commitment.private.enc. PRIME_PART_B must never appear in cloud storage—return it only in the compile response; user saves offline. Implications: the compiler service API design must support returning commitment without persisting it; the web UI must offer a one-time download with clear warnings.

J. 10. Implementation Guide for Developers

Description and Implications (Section 10): This section provides the concrete roadmap for implementers. It lists modules (keystore.rs, pii_proof.rs, policy.rs, etc.), CLI subcommands, build-time changes, compiler service changes, web interface changes, error codes, and performance considerations. The guide bridges the theoretical foundation (sections 1–9) and the practical work of building the feature. Implications: implementation is phased—commitment file generation (build.rs, compiler service) is Phase 1; keystore CRUD and caboose modification are subsequent phases. Error codes (E001–E009) enable structured logging without leaking secrets. Performance considerations (lazy decrypt, memory, binary size, concurrency) affect architecture decisions. Dependencies: each module depends on the cryptographic foundations (§4) and the caboose structure (§3); the compiler service depends on build output parsing; the web UI depends on the compile API response schema.

1) 10.1 Modules to Implement:

2) 10.1 Modules to Implement:

Module	Purpose
`keystore.rs`	Encryption, decryption, CRUD, JSON store V4
`pii_proof.rs`	PII challenge generation; two-part proof validation
`policy.rs`	Policy schema; access level enforcement
`cloud_sync.rs`	Manifest management; keystore migration
`commitment.rs`	Commitment file generation; schema validation

Description, Context, Implications, Dependencies (10.1): keystore.rs is the core: encrypt/decrypt, CRUD, V4 JSON handling. pii_proof.rs handles challenge generation and two-part proof validation. policy.rs enforces access levels and network ACLs. cloud_sync.rs manages manifest and keystore migration across binary versions. commitment.rs generates and validates the commitment file. Dependencies: keystore depends on policy for access checks; pii_proof depends on gate secrets and challenge matrix; cloud_sync depends on keystore for migration decrypt/re-encrypt.

3) 10.2 CLI Subcommands:

```
keystore info          # Metadata (count, size); no
keystore pii-list      # List PII keys (not values)
keystore pii-add       # Add PII (requires proof +
keystore pii-get       # Get PII value (requires au
keystore pwd-list      # List password keys
keystore pwd-add       # Add password
keystore pwd-get       # Get password value
keystore unlock        # Decrypt and display all (s
keystore export        # Write decrypted backup to
keystore import        # Replace keystore from file
```

Description, Context, Implications, Dependencies (10.2): The CLI surface mirrors the keystore capabilities. info and pii-list require no auth (metadata only). pii-add, pii-get, pwd-add, pwd-get require proof (and prime for Layer 2). unlock and export are sensitive—decrypt and display or write all entries. import replaces the keystore from a decrypted backup. Dependencies: each subcommand routes to keystore module; policy lookup determines auth requirements per key.

4) 10.3 Build-Time Changes: In `build.rs`: - After pool embedding: generate `commitment.private.json` - Emit `cargo:warning=COMMITMENT_PATH=<path>` - Include circuit_hash, config, part_b_hex (or equivalent for 4-shard)

Description, Context, Implications, Dependencies (10.3): In build.rs, after embedding the entropy pool and generating the prime split, emit the commitment file. Use cargo:warning=COMMITMENT_PATH= so the compiler service can find it. The commitment must include circuit_hash, full config, and part_b_hex. Dependencies: build.rs runs at compile time; the compiler service invokes the build and captures stderr. Implications: the build directory must be accessible to the compiler service; ephemeral build dirs require passing the path before cleanup.

5) 10.4 Compiler Service Changes: In `compiler-service/src/main.rs`: - Parse build stderr for `COMMITMENT_PATH=` - Read commitment JSON from path - Add `commitment`, `commitment_url`, `prime_part_b_hex` to `CompileResponse` - Never persist prime_part_b to GCP

Description, Context, Implications, Dependencies (10.4): The compiler service parses build stderr for COMMITMENT_PATH=, reads the commitment JSON, and adds commitment, commitment_url, and prime_part_b_hex to CompileResponse. PRIME_PART_B must never be written to GCP (no Artifact Registry, no Cloud Storage). Dependencies: build must emit COMMITMENT_PATH; the response schema must include these fields. Implications: the compiler service is a trusted component—it temporarily holds PRIME_PART_B in memory to include in the response; zeroize after sending.

6) 10.5 Web Interface Changes:

- Setup wizard: add "Commitment & Keys" step for hybrid circuits
- Download commitment.private.json from compile response
- Display PRIME_PART_B with reveal/copy; security warnings
- Optional "Add PII vault" step; wire "Store PII" and "Import Keys" dashboard buttons
- Step-by-step flows: **setting** (§12.11), **reading** (§12.12), and **interactive proof** (§12.13) document web interface behavior for PII/secret store sequences

Description, Context, Implications, Dependencies (10.5): The setup wizard adds a "Commitment & Keys" step for hybrid circuits. The user downloads commitment.private.json from the compile response. PRIME_PART_B is displayed with reveal/copy and security warnings. Optional "Add PII vault" step and dashboard buttons for "Store PII" and "Import Keys" complete the UX. For full web flows (set, read, prove), see §12.11 (Setting PII and Secrets), §12.12 (Reading PII and Secrets), and §12.13 (Interactive Proof of PII via Interface). Dependencies: compile API returns commitment; frontend needs download and display logic. Implications: UX must clearly warn that PRIME_PART_B is like a private key—never share, store offline.

7) 10.6 Error Codes and Recovery: Implement a consistent error code scheme for debugging without leaking secrets:

Code	Meaning	User Action
E001	Integrity hash mismatch	Binary may be corrupted; do not use
E002	Policy not found	Secret may not exist or policy misconfigured
E003	Proof failed	Incorrect bearings; retry with correct ceremony
E004	PRIME_PART_B required	Provide prime half for Layer 2 access
E005	PRIME_PART_B invalid	Check 128 hex chars; no spaces

Code	Meaning	User Action
E006	AEAD auth failed	Wrong key (wrong prime or proof); abort
E007	Caboose parse error	Tail corrupted or wrong version
E008	Network ACL denied	Request from disallowed IP/domain/MAC
E009	Context missing	Required context (IP, etc.) not provided

Never include sensitive data (bearings, prime, plaintext) in error messages. Log to structured logger with error code only; optional stack trace in debug builds.

Description, Context, Implications, Dependencies (10.6): Error codes E001–E009 provide debuggability without leaking secrets. User-facing messages should be generic (“access denied”); internal logs use codes. E001 (integrity mismatch) means binary corruption. E002 (policy not found) may indicate misconfiguration. E003 (proof failed), E005 (prime invalid), E006 (AEAD auth) all produce the same user outcome. Dependencies: all code paths must route to appropriate codes; logging infrastructure must support structured fields.

8) 10.7 Performance Considerations:

- **Decryption:** Layer 1 + Layer 2 decrypt on every access. For large keystores (1000+ entries), consider lazy decrypt: decrypt only the policy section first to locate the target key’s layer and offset; then decrypt that layer. Current spec decrypts full layer.
- **Memory:** Plaintext should reside in memory only briefly. For `keystore unlock` displaying all entries, consider streaming or pagination to avoid holding entire plaintext at once.
- **Binary size:** Caboose adds to binary size. Each Layer 1/Layer 2 section has overhead (nonce, tag, section headers). Typical caboose: 1–10 KB for modest usage.
- **Concurrency:** Single-writer model for caboose. Concurrent modifications require locking or serialization at the process level.

Description, Context, Implications, Dependencies (10.7): Decryption cost scales with layer size; lazy decrypt (policy-first, then target layer) can reduce work for large keystores. Memory: minimize time plaintext resides in RAM; consider streaming for keystore unlock. Binary size: caboose adds 1–10 KB typical; cap Layer 1+2 at a few MB. Concurrency: single-writer for caboose; use file locking or process-level serialization. Dependencies: performance tuning depends on typical keystore sizes and access patterns.

K. 11. Security Threat Model

Description and Implications (Section 11): The threat model identifies attack vectors and mitigations. Binary theft is mitigated by encryption—attackers need proof and prime. Brute force on prime (2^512) or bearings (6^12) is infeasible. Memory dumps are mitigated by zeroization. Replay is prevented by fresh nonce and ledger consume. PII inference from proof is mitigated by zero-knowledge design. Policy bypass is prevented by MAC signing. Commitment file theft is a risk—treat as private key. The defense-in-depth model requires possession (binary), knowledge (ceremony), gate secret, and for Layer 2, PRIME_PART_B. The attack tree illustrates the difficulty of accessing Layer 2 SSN: multiple AND gates (obtain binary, complete proof, obtain PRIME_PART_B, bypass integrity). The ceremony knowledge is the human factor—malware that steals files cannot steal “which direction for which color.” Implications: design for these threats; assume binary and possibly config can be obtained; rely on PRIME_PART_B being offline and ceremony being cognitive.

1) 11.1 Attacks and Mitigations:

2) 11.1 Attacks and Mitigations:

Attack	Mitigation
Binary theft	Encrypted caboose; requires proof + prime to decrypt
Brute force prime	512-bit prime = 2^512 possibilities
Brute force bearings	6^12 2.18B; combined with prime infeasible
Memory dump	Secrets only in memory after unlock; zeroize on drop
Replay	Fresh nonce per challenge; tau-bound proofs
PII inference from proof	Zero-knowledge: only validity bit revealed
Policy bypass	Policy MAC-signed; enforced in code
Commitment file theft	Treat as private key; encrypt at rest
Cloud commitment exposure	Encrypt before upload; PRIME_PART_B never in cloud

Description, Context, Implications, Dependencies (11.1): Each attack has a corresponding mitigation. Binary theft alone does not yield plaintext. Brute force is computationally infeasible. Memory dump recovers only what is in RAM at dump time—zeroization limits exposure. Replay fails because nonce is consumed. PII cannot be inferred from proof structure. Policy cannot be forged due to MAC. Commitment theft is the highest residual risk—mitigate with encryption and offline storage. Dependencies: mitigations require correct implementation (zeroize, MAC verify, nonce consume).

3) 11.2 Defense in Depth: Access requires: 1. Possession of binary 2. Knowledge of ceremony (cognitive mapping) 3. Gate secret knowledge (PIN6, 12MIXED) 4. For Layer 2: PRIME_PART_B (possession factor)

Description, Context, Implications, Dependencies (11.2): Four factors protect access: (1) binary possession, (2) ceremony knowledge (cognitive), (3) gate secret, (4) PRIME_PART_B for Layer 2. Loss of one does not necessarily compromise others. PRIME_PART_B can be in a safe or hardware wallet. Implications: design UX to reinforce factor separation (e.g.,

don't prompt for PRIME_PART_B for Layer 1; emphasize cold storage for prime).

4) 11.3 Attack Tree Example: To access Layer 2 SSN, an attacker must:

```
AND
-- Obtain binary (physical access, supply chain, phishing download)
-- Complete 12-round proof (requires gate_secret knowledge)
|   +-- Gate secret in ENCRYPTED_CONFIG; decrypt needs P = A  B
|       +-- Both A and B embedded in binary for config decryption
|           → Attacker with binary CAN decrypt config, has gate_secret
|           → But proof requires USER INPUT (bearings) — not derivable from binary alone
|           → Gate secret + bearings = proof; attacker must know gate secret to produce correct
|           → So: attacker with binary has gate_secret, can run proof if they know ceremony
-- Obtain PRIME_PART_B
|   +-- Only in user's cold storage; NOT in binary, NOT in cloud
|   +-- Social engineering, physical theft of paper/hardware
+-- Bypass integrity check (or binary has no integrity check)
    +-- Mitigated: integrity_hash verified before any decrypt
```

The ceremony knowledge (cognitive mapping: which direction for which color) is the critical human factor. Without it, even with binary and gate_secret, the attacker cannot produce correct bearings for a given challenge matrix. This is the "something you know" that cannot be exfiltrated by malware that only steals files.

Description, Context, Implications, Dependencies (11.3): The attack tree shows the AND-gate structure: all conditions must be satisfied. Obtaining the binary is one step; completing the proof requires ceremony knowledge; obtaining PRIME_PART_B requires physical/social access to user's cold storage. Integrity bypass would require defeating BLAKE3. The note on gate_secret: with the binary, an attacker can decrypt ENCRYPTED_CONFIG (both A and B embedded for config) and get gate_secret. But the proof requires user input (bearings)—the attacker must know the ceremony to produce correct bearings. That knowledge is not in the binary. Implications: ceremony is the human-held secret; protect it through user education and memorization.

L. 12. Use Cases with Implementation Notes

Description and Implications (Section 12): This section grounds the theory in practical applications. Password manager, KYC, banking MFA, API key storage, enterprise SSO, IoT, estate planning, and healthcare each have distinct requirements and implementation notes. The step-by-step flows provide concrete sequences: **§12.9** KYC age verification; **§12.10** password manager unlock; **§12.11** setting PII and secrets (SET flow for Layer 1 and Layer 2); **§12.12** reading PII and secrets (READ flow: get-by-key and full unlock, CLI and web); **§12.13** interactive proof of PII (prove SSN, DOB, etc. without revealing values, verifier request → challenge → proof payload). Implications: use cases inform policy assignment (e.g., passwords in Layer 2), proof level (12MIXED+PRIME for banking), and network ACLs (bank domain only for SSN). Dependencies: each use case maps to specific policy levels and Layer 1 vs Layer 2 choices.

1) 12.1 Password Manager:

2) 12.1 Password Manager:

- All passwords in Layer 2
- Unlock with proof + PRIME_PART_B
- Export to encrypted backup; import for restore
- Policy: read/write both require 12MIXED+PRIME

Description, Context, Implications, Dependencies (12.1): All passwords in Layer 2; unlock with proof + PRIME_PART_B. Export/import for backup/restore. Policy: 12MIXED+PRIME for both read and write. Dependencies: keystore CRUD, export/import subcommands. Implications: password manager is a primary driver for Layer 2 design; UX must balance security (prompt for prime) with usability (consider session caching for repeated access within a time window, with clear security trade-off).

3) 12.2 KYC / Identity Verification:

- SSN, DOB, passport in vault
- Prove age > 21, citizenship, SSN format without revealing values
- Verifier receives only proof validity

Description, Context, Implications, Dependencies (12.2): SSN, DOB, passport in vault. Prove age, citizenship, SSN format without revealing values. Dependencies: PII proof mode, verifier integration. Implications: KYC is a key compliance use case; proofs replace PII retention in verifier databases.

4) 12.3 Banking MFA:

- Use PII (e.g., SSN last 4) as second factor
- No SMS or TOTP; proof-of-knowledge only
- Network ACL: only bank domain can request SSN proof

Description, Context, Implications, Dependencies (12.3): PII (e.g., SSN last 4) as second factor; no SMS/TOTP. Network ACL restricts SSN proof requests to bank domain. Dependencies: policy with network ACL, verifier-side challenge request. Implications: eliminates SMS interception and TOTP phishing.

5) 12.4 API Key Storage:

- API keys in Layer 2
- Agent unlocks to call external services
- Machine binding: restrict to specific hostname/MAC

Description, Context, Implications, Dependencies (12.4): API keys in Layer 2; agent unlocks to call external services. Machine binding limits access to specific hostname/MAC. Dependencies: network ACL with machine ID context. Implications: supports headless/CI use with appropriate binding.

6) 12.5 Enterprise SSO:

- Per-app credentials in caboose
- Proof gates access; no central credential store
- Sync: cloud manifest for version control; keystore migrates on binary update

Description, Context, Implications, Dependencies (12.5): Per-app credentials in caboose; proof gates access; no central credential store. Cloud manifest for version control; migration

on binary update. Dependencies: cloud_sync module. Implications: enterprise deployment without central identity provider.

7) 12.6 IoT / Edge Devices:

- Binary + secrets on device
- No cloud credential sync
- Optional: TPM/Secure Enclave attestation for context

Description, Context, Implications, Dependencies (12.6): Binary + secrets on device; no cloud sync. Optional TPM/Secure Enclave for context attestation. Dependencies: stateless binary, context injection. Implications: supports air-gapped and edge deployments.

8) 12.7 Estate Planning:

- Shared proof mapping among heirs
- Heirs prove inheritance without exposing secrets
- Threshold: M-of-N proofs for high-value access

Description, Context, Implications, Dependencies (12.7): Shared proof mapping among heirs; heirs prove without exposing secrets. M-of-N threshold for high-value access. Dependencies: ceremony sharing, threshold extension (future). Implications: estate planning may require protocol extension for M-of-N.

9) 12.8 Healthcare:

- Insurance ID, medical record numbers in vault
- Prove coverage or record access without exposure
- HIPAA alignment: prove attributes; don't store raw PII in central DB

Description, Context, Implications, Dependencies (12.8): Insurance ID, medical record numbers in vault. Prove coverage/access without exposure. HIPAA alignment: prove attributes, avoid central PII storage. Dependencies: PII proof, policy assignment. Implications: healthcare is highly regulated; proofs enable verification without retention.

10) 12.9 Step-by-Step: KYC Age Verification Flow: To implement "prove age 21 without revealing DOB":

1. User adds DOB to Layer 1 with policy 6UPPER (or 12MIXED for stronger assurance).
2. Verifier (e.g., alcohol vendor system) requests challenge with constraint: `age_min=21`.
3. Binary generates challenge; user completes 6-round proof using DOB to navigate matrix.
4. Binary internally checks: decrypt DOB, compute age from DOB, assert age 21.
5. Proof payload includes: `valid=true`, `age_constraint_satisfied=true` — NOT the DOB value.
6. Verifier receives proof; validates bearing correctness and constraint; grants access.
7. Log entry: "Proof valid for age21" — no PII stored.

Implementation note: The "constraint check" (age 21) may require the binary to expose a structured claim in the proof payload. The exact encoding depends on verifier requirements. Options: (a) generic "constraint_id + satisfied" in proof metadata, (b) range proof construction, (c) predicate commitment. For MVP, a simple boolean "age_verified" in the proof extension suffices if verifier trusts the circuit.

Description, Context, Implications, Dependencies (12.9): Seven-step flow: add DOB to Layer 1, verifier requests challenge with age_min=21, binary generates challenge, user completes proof using DOB, binary checks age 21, proof payload includes valid + age_constraint_satisfied (not DOB), verifier grants access. Dependencies: constraint encoding in proof; verifier validation logic. Implications: MVP can use boolean claim if verifier trusts circuit; stronger designs need predicate or range proof.

11) 12.10 Step-by-Step: Password Manager Unlock Flow:

1. User runs `./circuit keystore unlock`.
2. Binary prompts for 12-round proof (interactive ceremony).
3. User completes proof with gate_secret (memorized).
4. Binary prompts: "Enter PRIME_PART_B (128 hex chars) for Layer 2 access."
5. User retrieves PRIME_PART_B from cold storage; pastes.
6. Binary derives Layer 2 key; decrypts caboose; displays all entries (masked by default; "reveal" per entry).
7. User copies needed password; exits.
8. Binary zeroizes: PRIME_PART_B, plaintext, keys from memory.
9. PRIME_PART_B never written to disk; user returns it to cold storage.

Description, Context, Implications, Dependencies (12.10): Nine-step flow: keystore unlock, 12-round proof, user completes with gate_secret, prompt for PRIME_PART_B, user pastes from cold storage, derive Layer 2 key, decrypt, display (masked), zeroize on exit. Dependencies: Layer 2 flow, CLI. Implications: PRIME_PART_B never persisted; user responsibility to return to cold storage.

12) 12.11 Step-by-Step: Setting PII and Secrets in the Encrypted Store: This section documents the complete **SET** flow for adding or updating PII and other secrets in the caboose. See §7.2 Modification Flow for the underlying mechanics; this subsection expands with policy choice, proof requirement, and Layer-specific behavior.

Layer 1 SET (proof-only): DOB, phone number, WiFi password, low-sensitivity PII.

1. **Policy choice:** Assign read_policy and write_policy. For Layer 1, use 6UPPER, 6NUMBER, 6MIXED, or 12MIXED (no PRIME required). Example: `DOB.read_policy=6UPPER`, `DOB.write_policy=6UPPER+`.
2. **Proof requirement:** User completes the required proof (e.g., 6-round uppercase for 6UPPER).
3. **Integrity check:** Binary verifies BLAKE3 integrity_hash before allowing write.
4. **Key derivation:** Layer 1 key from gate_secret + bearings (§4.1). No PRIME_PART_B.
5. **Decrypt Layer 1:** Full Layer 1 section decrypted.
6. **Apply modification:** Add or update the value in the keystore JSON (V4 schema).
7. **Write to caboose:** Generate new nonce; re-encrypt Layer 1 with derived key; overwrite caboose; recompute integrity_hash.

8. **Zeroize:** Plaintext and keys cleared from memory.

Layer 2 SET (proof + PRIME_PART_B): SSN, bank PIN, private keys, high-value secrets.

1. **Policy choice:** Assign 12MIXED+PRIME for both read and write. Example: `SSN.read_policy=12MIXED+PRIME`, `SSN.write_policy=12MIXED+PRIME`.
2. **Proof requirement:** User completes 12-round mixed proof.
3. **PRIME_PART_B input:** User provides 128 hex chars from commitment file (cold storage). Without it, write cannot proceed.
4. **Integrity check:** As above.
5. **Key derivation:** Layer 2 key from P (PRIME_PART_A PRIME_PART_B) + bearings (§4.1).
6. **Decrypt Layer 2:** Full Layer 2 section decrypted. If Layer 1 also needs modification in same session, decrypt both.
7. **Apply modification:** Add or update the value in Layer 2 payload.
8. **Write to caboose:** Generate new nonces for affected layers; re-encrypt; overwrite caboose; recompute integrity_hash.
9. **Zeroize:** PRIME_PART_B, plaintext, and keys cleared from memory immediately.

CLI flow:

```
keystore pii-add --key DOB --value "1990-05-15" --policy 6UPPER    # Layer 1
keystore pii-add --key SSN --value "123-45-6789" --policy 12MIXED+PRIME  # Layer 2
```

For Layer 2, the binary prompts for PRIME_PART_B after proof completion.

Web interface flow: (§10.5) - User clicks "Store PII" or "Add Secret" in the dashboard. - Interface shows policy selector (Layer 1 vs Layer 2, proof level). - User enters key (label) and value. - For Layer 2: interface prompts for PRIME_PART_B in a secure input (no autocomplete; masked). - Binary (via WASM or backend) performs proof, key derivation, and caboose write. - Success confirmation; PRIME_PART_B never stored or transmitted beyond the secure session.

Cross-references: §4.1 Key Derivation, §5.1 Layer Comparison, §7.2 Modification Flow, §8.1 Policy Levels, §10.2 CLI Subcommands, §10.5 Web Interface Changes.

13) 12.12 Step-by-Step: Reading PII and Secrets from the Encrypted Store: This section documents the **READ** flow: get-by-key (single value) and full unlock (all values). Aligned with §5.2 Unlock Flow (Layer 1), §5.3 Unlock Flow (Layer 2), and §2.2 High-Level Data Flow.

Get-by-key (pii-get, pwd-get): Retrieve a single value by label.

1. **Policy lookup:** Resolve key to layer and read_policy. E.g., `SSN` → Layer 2, 12MIXED+PRIME; `DOB` → Layer 1, 6UPPER.
2. **Integrity check:** BLAKE3(binary) matches stored integrity_hash. Abort on mismatch.
3. **Interactive proof:** User completes required rounds (6 or 12) per policy.
4. **PRIME_PART_B (Layer 2 only):** If target is Layer 2, prompt user for 128 hex chars.
5. **Key derivation:** Layer 1: gate_secret ‖ bearings; Layer 2: P ‖ bearings.
6. **Decrypt:** Decrypt only the target layer (Layer 1 or Layer 2).
7. **Return value:** Extract and return the requested key's value.
8. **Zeroize:** Value, keys, PRIME_PART_B zeroized after use or display.

Full unlock (keystore unlock): Decrypt and display all entries.

1. **Integrity check:** As above.
2. **Interactive proof:** 12-round proof (required to access Layer 2).
3. **PRIME_PART_B:** User provides; needed to decrypt Layer 2.
4. **Key derivation:** Both Layer 1 and Layer 2 keys derived.
5. **Decrypt both layers:** Full caboose decrypted.
6. **Display:** Entries shown masked by default; user reveals per entry as needed.
7. **Zeroize on exit:** All plaintext, keys, PRIME_PART_B cleared.

CLI flow:

```
keystore pii-get SSN           # Layer 2: prompts fo
keystore pii-get DOB           # Layer 1: prompts fo
keystore pwd-get bank          # Layer 2 if policy r
keystore unlock                # Full unlock; masked
```

Web interface flow: (§10.5) - User selects a secret (e.g., SSN) from the dashboard list. - Interface invokes proof ceremony (6 or 12 rounds via UI). - For Layer 2: secure input for PRIME_PART_B (masked, no logging). - Binary returns value; interface displays masked (e.g., `***-**-6789`) with optional "Reveal" button. - Full unlock: "Unlock vault" → proof + PRIME_PART_B → list with masked values; reveal on demand. - Session ends; no persistence of plaintext in browser storage.

Layer 1 vs Layer 2 differences: | Aspect | Layer 1 | Layer 2 | |———|————|————| | Proof | 6UPPER/6NUMBER/6MIXED/12MIXED | 12MIXED only | | PRIME_PART_B | Not required | Required | | Typical get latency | Lower (no prime prompt) | Higher (prime retrieval) | | Use case | Daily access (DOB, wifi) | Rare access (SSN, bank PIN) |

Cross-references: §2.2 High-Level Data Flow, §5.2 Unlock Flow (Layer 1), §5.3 Unlock Flow (Layer 2), §10.2 CLI Subcommands, §10.5 Web Interface Changes, §12.10 Password Manager Unlock.

14) 12.13 Step-by-Step: Interactive Proof of PII via Interface: This section documents the **interactive proof** flow: proving possession of SSN, DOB, or other PII to a verifier *without revealing the value*. The verifier learns only validity (proof correct) and optionally structured claims (e.g., age 21), never the raw PII. See §6.1 Proof vs Storage, §6.2 Zero-Knowledge Property, and §6.5 PII Proof Integration with External Verifiers.

End-to-end flow:

1. **Verifier request:** Verifier (e.g., bank, alcohol vendor) sends challenge request to user's agent or service. Request includes: nonce, pii_fields (e.g., `["SSN"]`, `["DOB"]`), proof_level (6UPPER, 12MIXED), optional constraints (e.g., `age_min=21` for DOB).
2. **Challenge generation:** Binary (or service) loads challenge matrix for the requested fields. Challenge is bound to nonce and tau.
3. **User proves via interface:** User runs binary (CLI or web). Binary presents the ceremony (6 or 12 rounds). User enters bearings derived from PII—either from memory (proof-only) or retrieved from vault (storage). For proof-only mode: user types SSN/DOB during ceremony; value used to navigate matrix, then discarded. For storage mode: user first unlocks vault (§12.12), binary uses decrypted value for navigation.
4. **Proof payload generation:** Binary produces proof payload: bearings, proof hash, circuit_hash, tau, optional claims (e.g., `age_verified=true`). **No PII in payload.**
5. **Proof submission:** User (or agent) submits proof payload to verifier.
6. **Verifier validation:** Verifier checks: bearing correctness for challenge, proof hash, nonce consumption (ledger), circuit_hash authorization. For constrained proofs (e.g., age 21): verifier checks claim `age_verified=true`.
7. **Outcome:** Verifier grants or denies access. No PII transmitted; only validity and optional claims.

CLI flow:

```
./circuit respond-interactive --pii=SSN --nonce=<verifier_nonce>
# User completes 12 rounds (SSN from memory or vault)
# Proof payload output; user submits to verifier

./circuit respond-interactive --pii=DOB --constraint=age_min:21
# User completes 6 rounds with DOB
# Proof includes age_verified=true/false
```

Web interface flow: - User navigates to "Prove PII" or equivalent in the dashboard. - Interface shows: which PII to prove (SSN, DOB, etc.), verifier challenge ID if applicable. - User selects PII field; interface starts proof ceremony (rounds displayed). - User enters bearings (derived from PII—typed from memory or from unlocked vault). - Interface submits proof payload to verifier endpoint (or copies for manual submission). - Result: "Proof valid" or "Proof invalid"—no PII displayed or transmitted.

Proof payload structure (valid without revealing value): - `valid`: boolean - `bearings`: array (used for validation; does not reveal PII) - `proof_hash`: binds bearings to nonce - `circuit_hash`: identifies binary - `tau`: timestamp/binding - `claims`: optional, e.g. `{ "age_verified": true }` for DOB age checks

Cross-references: §6.1 Two Modes: Proof vs Storage, §6.2 Zero-Knowledge Property, §6.3 Two-Part Challenge Protocol, §6.5 PII Proof Integration with External Verifiers, §12.9 KYC Age Verification, §12.10 Password Manager Unlock.

M. 13. Integration Patterns

Description and Implications (Section 13): Integration patterns cover how the embedded secret store connects to external systems: context injection, cloud sync, ledger binding, and deployment/operations. Context injection provides the environment (IP, MAC, etc.) for ACL evaluation—strict (signed), normal (collected), or permissive. Cloud sync enables manifest tracking and keystore migration across binary versions. Ledger binding ties proofs to nonces and circuit_hash for replay prevention. Deployment covers build environment isolation, distribution channels, key lifecycle, monitoring, and recovery. Implications: stateless binaries receive context from deployment layer; sync requires user PRIME_PART_B for migration; recovery depends on user backup of PRIME_PART_B and exported keystore.

1) 13.1 Context Injection: Stateless binary receives context from deployment layer: - **Strict:** Require signed context from trusted source - **Normal:** Collect IP, MAC, hostname, machine ID - **Permissive:** Accept any context

Context is checked against network ACLs before granting access.

Description, Context, Implications, Dependencies (13.1): Stateless binary receives context from deployment layer. Strict: require signed context from trusted source. Normal: collect IP, MAC, hostname, machine ID. Permissive: accept any. Dependencies: deployment layer must provide context; policy evaluation uses it for ACLs. Implications: context spoofing is a risk in permissive mode; use strict mode for high-assurance.

2) 13.2 Cloud Sync:

- Manifest tracks latest binary version
- On `sync pull`: download new binary; migrate keystore to new binary's key derivation
- User provides PRIME_PART_B for migration decrypt/re-encrypt
- circuit_hash stable; integrity_hash changes

Description, Context, Implications, Dependencies (13.2): Manifest tracks latest binary version. sync pull: download new binary, migrate keystore with user's PRIME_PART_B. circuit_hash stable; integrity_hash changes. Dependencies: cloud_sync module, manifest format, migration logic. Implications: migration decrypts with old key derivation, re-encrypts with new; user must provide PRIME_PART_B.

3) 13.3 Ledger Binding:

- circuit_hash in ledger commitment
- Proofs bound to nonce + tau
- Replay prevented by ledger consume

Description, Context, Implications, Dependencies (13.3): circuit_hash in ledger commitment; proofs bound to nonce + tau. Ledger consumes nonce on validation, preventing replay. Dependencies: ledger API, proof payload structure. Implications: each proof is single-use; verifier must check ledger state before granting access.

round-trip for Layer 1 and Layer 2. Policy evaluation: correct layer per policy. CRUD: add, list, get, remove. Dependencies: keystore module, test fixtures. Implications: unit tests use known vectors; avoid production secrets.

2) 14.2 Integration Tests:

- Full unlock flow: proof → prime input → decrypt → value returned
- Mutable: add PII → verify in list → modify → verify update
- Export/import: export → modify file → import → verify

Description, Context, Implications, Dependencies (14.2): Full unlock: proof, prime input, decrypt, value returned. Mutable: add PII, verify in list, modify, verify update. Export/import: export, modify file, import, verify. Dependencies: binary build, CLI. Implications: integration tests exercise end-to-end flows; may require mock or test ceremony.

3) 14.3 Assertions for Validation Scripts:

- `cargo build` emits COMMITMENT_PATH
- commitment.private.json exists and has valid schema
- Compile API returns commitment + prime_part_b_hex for hybrid
- keystore info exits 0
- keystore pii-add → pii-list shows key
- Regression: existing proof/challenge flows pass

Description, Context, Implications, Dependencies (14.3): Assertions A1.1–A5.2: cargo build emits COMMITMENT_PATH; commitment exists with valid schema; compile API returns commitment; keystore info exits 0; pii-add → pii-list shows key; regression. Dependencies: validation script, build and compile pipeline. Implications: assertions are SDLC success criteria; proof report links each to artifact.

4) 14.4 Proof Report: Generate log-backed proof report: - Link each criterion to artifact (log file, API response) - Include re-run instructions - Environment matrix: Local | Docker | Cloud

Description, Context, Implications, Dependencies (14.4): Proof report links each criterion to artifact (log, API response). Re-run instructions included. Environment matrix: Local, Docker, Cloud. Dependencies: test-results structure, log capture. Implications: report is the deployment gate; every claim must be falsifiable.

5) 14.5 Comparison with Traditional Secret Management:

Aspect	Traditional (Vault, AWS Secrets Manager)	Embedded Secret Store
Storage	Centralized; network access required	Distributed; in binary
Access control	IAM, ACLs, RBAC	Proof-of-knowledge + prime

4) 13.4 Deployment and Operations: **Build environment:** The compiler must run in an isolated environment. Build directories should be ephemeral; no retention of config or secrets after compile. The commitment file is produced once; its path is communicated via `cargo:warning=COMMITMENT_PATH=` for the compiler service to read.

Distribution: Binaries can be distributed via: (a) direct download from compile response, (b) GCS/cloud bucket with signed URLs, (c) air-gapped transfer (USB, SCP). The commitment file should be distributed separately through a secure channel; never bundle it with the binary in a public artifact.

Key lifecycle: Gate secrets are embedded; rotation requires recompile. PRIME_PART_B is user-managed; recommend backup in at least two offline locations. Layer 1/Layer 2 contents can be modified post-compilation; backup via `keystore export` before major changes.

Monitoring and audit: Log proof requests (without PII); log keystore modifications (key names, not values); alert on integrity_hash mismatches. Circuit hash should be recorded in manifest for version tracking.

Recovery: If PRIME_PART_B is lost, Layer 2 is unrecoverable. Mitigation: periodic export to encrypted backup; store backup in secure location. If binary is corrupted, restore from backup binary + keystore import (with PRIME_PART_B).

Description, Context, Implications, Dependencies (13.4): Build environment must be isolated; ephemeral dirs, no retention. Distribution: download, GCS signed URLs, air-gap. Key lifecycle: gate secrets embedded (rotate via recompile); PRIME_PART_B user-managed; Layer 1/2 modifiable. Monitoring: log proof requests (no PII), keystore modifications (key names only), integrity mismatches. Recovery: PRIME_PART_B loss is permanent for Layer 2; backup and export critical. Dependencies: deployment pipeline, monitoring stack.

N. 14. Testing and Validation

Description and Implications (Section 14): Testing and validation ensure correct implementation and ongoing verification. Unit tests cover key derivation, encrypt/decrypt round-trip, policy evaluation, and CRUD. Integration tests cover full unlock flow, mutable operations, and export/import. Assertions for validation scripts (A1.1–A5.2) provide criteria for SDLC proof reports. The comparison table contrasts traditional secret management (Vault, AWS Secrets Manager) with the embedded store. Implications: implementers must run the full assertion set; proof reports link each criterion to artifacts. The comparison informs architecture decisions and trade-off discussions.

1) 14.1 Unit Tests:

- Key derivation: same inputs → same key
- Encrypt/decrypt round-trip for Layer 1 and Layer 2
- Policy evaluation: correct layer selected per policy
- CRUD: add → list → get → remove

Description, Context, Implications, Dependencies (14.1): Key derivation: same inputs produce same key. Encrypt/decrypt

Aspect	Traditional (Vault, AWS Secrets Manager)	Embedded Secret Store
Single point of failure	Yes; compromise exposes many	No; per-binary isolation
Phishing risk	Passwords/tokens phishable	Ceremony not phishable
Replay	Tokens can be replayed	Proofs bound to nonce; no replay
Portability	Tied to cloud/network	Binary is portable
PII handling	Often stored in DB	Prove without revealing
Operational complexity	Key rotation, expiry, sync	User-managed; migration on update
Recovery	Admin reset, backup restore	User PRIME_PART_B + export

Description, Context, Implications, Dependencies (14.5): Side-by-side comparison: storage (centralized vs distributed), access control (IAM vs proof+prime), single point of failure, phishing risk, replay, portability, PII handling, operational complexity, recovery. Implications: embedded store trades operational simplicity for security and portability; suitable when threat model prioritizes distribution and phishing resistance.

O. 15. Cybersecurity Implications Summary

Description and Implications (Section 15): This section synthesizes the cybersecurity value proposition for security architects and compliance officers. Elimination of the credential warehouse: no central store to compromise. Phishing resistance: ceremony is not phishable. Compliance without PII retention: GDPR, HIPAA, CCPA alignment via proofs. Portability and air-gap support: no cloud dependency. Audit and non-repudiation: proofs bound to ledger. Defense in depth: multiple factors. Trade-offs: operational burden on user (key backup, migration); careful implementation required (zeroization, policy). Implications: evaluate against threat model and compliance requirements; not a drop-in for every use case.

For security architects and compliance officers, the embedded secret store offers several distinct advantages over traditional approaches:

Elimination of the credential warehouse: Centralized secret stores (HashiCorp Vault, AWS Secrets Manager, database-backed credential tables) are high-value targets. A single compromise can expose thousands of credentials. The embedded store distributes secrets across user binaries; each compromise affects only that user. There is no central warehouse to attack.

Phishing resistance: In proof-only flows, users never type a password or token. They respond to challenges using a cognitive ceremony. Phishing sites cannot capture what the user never transmits. Even if an attacker obtains a session transcript, the proof is non-replayable (bound to nonce and timestamp).

Compliance without PII retention: Regulations (GDPR, HIPAA, CCPA) restrict storage and sharing of PII. The embedded store allows organizations to verify identity attributes (age, citizenship, account ownership) without retaining raw PII in their systems. Proofs attest to "user knows X" or "user satisfies predicate Y"; the verifier does not need to store X.

Portability and air-gap support: Identities can operate in fully air-gapped environments. No cloud dependency for secret retrieval. Useful for classified, industrial, or high-assurance deployments.

Audit and non-repudiation: Proofs can be bound to ledger nonces and circuit hashes. The combination provides a strong audit trail: "This proof was generated by circuit X at time T for nonce N" without exposing the underlying secrets.

Defense in depth: Multiple factors protect Layer 2: binary possession, ceremony knowledge, gate secret, and PRIME_PART_B. Loss of one factor does not necessarily compromise the others. PRIME_PART_B can be stored in a safe or hardware wallet, used only for high-value operations.

Trade-offs: The model shifts operational burden to the user (key backup, migration) and requires careful implementation (zeroization, policy enforcement). It is not a drop-in replacement for every use case; evaluate against your threat model and compliance requirements.

P. 16. References and Glossary

Description and Implications (Section 16): The references and glossary provide the navigational and terminological foundation for implementers. Key references point to the authoritative structural specification (structure.md), the config schema (PRIVATE_JSON_SCHEMA.md), and SDLC path rules (path-graph.md). The glossary defines critical terms—caboose, circuit_hash, integrity_hash, Layer 1/2, PRIME_PART_B, proof, V4 schema—in a single lookup table. Implications: new implementers should read structure.md before coding; the glossary resolves ambiguity when terms appear in different contexts. Dependencies: the structure document is the source of truth for layout, offsets, and cryptographic details; any divergence between this theory document and structure.md should be resolved in favor of structure.md.

1) 16.1 Key References:

- `compiler-service/SECRET-STORE/structure.md` — Full architectural specification (~5900 lines)
- `DOCS/development/PRIVATE_JSON_SCHEMA.md` — Config schema and validation
- `META-AGENTS/sdlc-suite/config/path-graph.md` — SDLC-only paths

Description, Context, Implications, Dependencies (16.1): structure.md (~5900 lines) is the full architectural spec for the caboose layout, key derivation, and section formats. PRIVATE_JSON_SCHEMA.md defines the commitment file and config schema. path-graph.md defines SDLC_ONLY roots for artifacts and logs. Dependencies: implementers need structure.md for byte-level details; schema for validation; path-graph for artifact placement. Implications: keep references current; update if paths or schemas change.

2) 16.2 Glossary:

Term	Definition
Caboose	Mutable section at end of binary; holds encrypted keystore
Circuit hash	BLAKE3 of immutable zone; stable identity
Gate secret	Embedded secret for proof (6UPPER, 6NUMBER, 12MIXED)
Integrity hash	BLAKE3 of full binary; changes with caboose
Layer 1	Proof-only vault; no PRIME_PART_B required
Layer 2	Prime vault; requires user's PRIME_PART_B
PRIME_PART_B	User-held prime half; cold storage; required for Layer 2
Proof	Interactive ceremony; user selects bearings for challenge matrix
V4 schema	Normalized credential store with policies, groups, values

Description, Context, Implications, Dependencies (16.2): The glossary provides one-line definitions for nine core terms. Caboose, circuit_hash, integrity_hash, Layer 1/2, PRIME_PART_B, proof, and V4 schema are used throughout the document. Implications: use consistent terminology; expand glossary when introducing new concepts. Dependencies: definitions should align with structure.md.

Q. 17. Developer FAQ

Description and Implications (Section 17): The Developer FAQ answers common implementation questions that arise during development. Topics include: adding new caboose section types, handling binary updates and keystore migration, PRIME_PART_B loss recovery, multi-binary caboose sharing, testing without a real ceremony, read-only media limitations, caboose size limits, and implementing format proofs without revealing digits. The FAQ is designed for developers who have read the main sections and need quick answers to edge cases. Implications: each Q&A is a reference for a specific scenario; extend the FAQ as new questions arise during implementation. The FAQ complements the implementation guide (§10) and the appendix (§18) by providing scenario-based guidance.

Q: Can I add a new section type to the caboose?
A: Yes. Use a new section_type (e.g., 0x05). Existing parsers

should skip unknown types. Update the tail header if the new section has a fixed offset. Document in structure.md.

Q: How do I handle binary updates (e.g., new circuit version)?

A: Keystore migration: decrypt with old binary's key derivation, re-encrypt with new binary's key derivation, append to new binary. User provides PRIME_PART_B for Layer 2 migration. circuit_hash of new binary differs; ledger/registry must track both if needed.

Q: What if the user loses PRIME_PART_B?

A: Layer 2 secrets are unrecoverable. Design for this: encourage backup (encrypted, offline). Layer 1 secrets remain accessible with proof only.

Q: Can multiple binaries share the same caboose?

A: No. Each binary has its own key derivation (tied to its prime and config). Caboose is binary-specific. To "sync" across devices, use cloud manifest + migration: download new binary, migrate keystore with PRIME_PART_B.

Q: How do I test without a real ceremony?

A: Use a test/demo build with known gate secrets and possibly auto-unlock. Ensure test binaries are never deployed to production. Mock the proof validation in unit tests.

Q: Is the caboose writable on read-only media?

A: No. The binary file must be opened for read-write to append updated caboose. For read-only media (e.g., CD, write-protected USB), only read operations (pii-get, keystore info) work; add/remove will fail.

Q: What's the max size of the caboose?

A: The specification does not impose a hard limit. Practical limits: filesystem, memory for decrypt. Recommend capping Layer 1 + Layer 2 at a few MB. Policy section is typically small (< 64 KB).

Q: How do I implement "prove SSN format without revealing digits"?

A: The PII proof uses the value to navigate the challenge matrix. The verifier sees only bearing choices. To prove format (e.g., XXX-XX-XXXX), the proof structure would need to encode "format valid" as a derived assertion—this may require extending the proof protocol. Current spec focuses on "prove knowledge of value" rather than "prove value matches predicate."

R. 18. Appendix: Pseudocode and Data Structures

Description and Implications (Section 18): The appendix provides pseudocode for key derivation (Layer 1 and Layer 2), minimal JSON schemas for the V4 value and commitment file, and a quick reference table for implementers. These artifacts bridge the gap between the textual specification and executable code. The pseudocode clarifies the exact inputs (gate_secret, bearings, prime parts) and context strings. The JSON schemas show the minimal structure for values and commitment files. The quick reference maps common tasks (add commitment generation, parse COMMITMENT_PATH, add keystore module, etc.) to locations and actions. Implications: implementers can copy and adapt the pseudocode; the quick reference accelerates onboarding. Dependencies: the pseudocode assumes BLAKE3 derive_key and XChaCha20Poly1305 APIs; the schemas should be validated against PRIVATE_JSON_SCHEMA.md.

1) A.1 Layer 1 Key Derivation (Pseudocode):

```
function derive_layer1_key(gate_secret: bytes, bea
    context = "ENI6MA proof_vault v1"
    input = gate_secret || serialize(bearings)
    return BLAKE3.derive_key(context, input)
```

Description, Context, Implications, Dependencies (A.1): Layer 1 key derivation uses gate_secret and bearings. Context string "ENI6MA proof_vault v1" must be exact. Input is gate_secret ‖ serialize(bearings). Dependencies: gate_secret from ENCRYPTED_CONFIG decryption; bearings from user proof input. Implications: use BLAKE3 derive_key, not raw hash; output is 32 bytes for XChaCha20 key.

2) A.2 Layer 2 Key Derivation (Pseudocode):

```
function derive_layer2_key(prime_part_a: [u64; 8],
    full_prime = xor(prime_part_a, prime_part_b)
    full_prime_bytes = to_le_bytes(full_prime)
    context = "ENI6MA prime_vault v1"
    input = full_prime_bytes || serialize(bearings)
    return BLAKE3.derive_key(context, input)
```

Description, Context, Implications, Dependencies (A.2): Layer 2 requires full prime (PRIME_PART_A PRIME_PART_B). Bearings from 12-round proof. Context "ENI6MA prime_vault v1". Dependencies: PRIME_PART_A embedded; PRIME_PART_B from user input. Implications: prime reconstruction must precede derivation; zeroize prime after use.

3) A.3 V4 Schema Value (Minimal):

```
{
  "value_uuid": "550e8400-e29b-41d4-a716-446655440001",
  "value_encrypted": "<base64 ciphertext>",
  "labels": ["SSN", "SocialSecurityNumber"],
  "layer": 2,
  "policy_id": "a1b2c3d4-...",
  "group_ids": ["g1", "g2"],
  "created_at": "2026-02-19T12:00:00Z"
}
```

Description, Context, Implications, Dependencies (A.3): Minimal V4 value has value_uuid, value_encrypted (base64), labels, layer (1 or 2), policy_id, group_ids, created_at. Dependencies: policy_id references policy; group_ids reference groups. Implications: value_encrypted is the ciphertext for the secret; layer determines which vault section it resides in.

4) A.4 commitment.private.json Schema (Minimal):

```
{
  "circuit_hash": "abc123...",
  "alphabets": [[...], [...], [...]],
  "zone_mapping": {...},
  "bearings": {...},
  "gate_secrets_meta": {"count": 3, "types": ["6UPPER", "6NUMBER", "12MI
  "part_b_hex": "128 hex chars - USER PRIVATE HALF"
}
```

Description, Context, Implications, Dependencies (A.4): Minimal commitment has circuit_hash, alphabets, zone_mapping, bearings, gate_secrets_meta, part_b_hex. part_b_hex is the user-held 128 hex chars—never in cloud.

Dependencies: generated in build.rs; read by compiler service. Implications: full schema in PRIVATE_JSON_SCHEMA.md; this is minimal for reference.

5) A.5 Quick Reference for Implementers:

Task	Location	Action
Add commitment generation	build.rs	After pool embed; write JSON; emit COMMIT-MENT_PATH
Parse COMMIT-MENT_PATH	compiler-service main.rs	Parse cargo stderr; read file; add to CompileResponse
Add keystore module	eni6ma-btc-hybrid/src/eni6ma/	Create keystore.rs; implement en-crypt/decrypt/CRUD
Add CLI subcommands	main.rs	Match keystore, pii, pwd, sync from structure.md
Add caboose section	binary_integrity.rs, self_sign	Append Layer 1/2 sections; update integrity_hash
Add web commitment step	setup-view.tsx	New step after compile; download + PRIME_PART_B display
Validate end-to-end	scripts/tests/	validate-embedded-secret-store.sh; assertions A1.1–A5.2

Description, Context, Implications, Dependencies (A.5): Quick reference maps seven tasks to locations and actions. Add commitment in build.rs; parse in compiler-service main.rs; add keystore module in eni6ma/; add CLI in main.rs; add caboose sections in binary_integrity.rs; add web step in setup-view.tsx; validate via scripts. Implications: use as checklist for implementation phases. Dependencies: assumes existing project structure (eni6ma-btc-hybrid, compiler-service, web app).

S. Document Metadata

- **Word count (approx):** 12,000+
- **Target audience:** Engineers, technical implementation developers, security architects
- **Related plans:** `.cursor/plans/embedded-secret-store-implementation.plan.md` (when created)
- **Implementation status:** Spec complete; implementation in progress (Phase 1 commitment file)
- **Revision history:** v1.0 (2026-02-19) — Initial diagnostic report; theory, architecture, implementation guide, FAQ, and appendix