aphanite

TECHNICAL WHITEPAPER · V1.0 · AS IMPLEMENTED

Aphanite: A Wallet-Native Encrypted Messaging Protocol

A specification of the protocol as built: wallet-rooted identity, X3DH + Double Ratchet direct messages (dr-v1) with X25519 sealed boxes for groups and the API (box-v1), an append-only Merkle transparency log, consent-gated first contact with per-sender rate limits, and a blind store-and-forward relay.Ciphersuite v1: secp256k1/ECDSA (SIWE) · X25519 · Ed25519 (prekey sigs) · X3DH + Double Ratchet · HKDF-SHA-256 · AES-256-GCM · SHA-256 Merkle · Argon2id · JWT-HS256

DOCUMENT
Protocol Spec & Security Notes

REFLECTS
Shipped v1 codebase

DATE
July 2026

STATUS
Draft / Technical Preview

TECHNICAL WHITEPAPER · V1.0 · AS IMPLEMENTED

Aphanite: A Wallet-Native Encrypted Messaging Protocol

A specification of the protocol as built: wallet-rooted identity, X3DH + Double Ratchet direct messages (dr-v1) with X25519 sealed boxes for groups and the API (box-v1), an append-only Merkle transparency log, consent-gated first contact with per-sender rate limits, and a blind store-and-forward relay.

Ciphersuite v1: secp256k1/ECDSA (SIWE) · X25519 · Ed25519 (prekey sigs) · X3DH + Double Ratchet · HKDF-SHA-256 · AES-256-GCM · SHA-256 Merkle · Argon2id · JWT-HS256

DOCUMENT	REFLECTS	DATE	STATUS
Protocol Spec & Security Notes	Shipped v1 codebase	July 2026	Draft / Technical Preview

CONTENTS

SECTION 1

1 · Introduction & Scope

This document specifies the Aphanite Protocol as implemented in the current codebase, so that a reader can reason about exactly what the system guarantees — and what it does not yet guarantee.

Aphanite is a wallet-native encrypted messenger. A self-custodied wallet address is the sole identity; message bodies are end-to-end encrypted on the client; and the handle-to-key directory is written to an auditable append-only log. The design maintains one invariant throughout — **content–revenue separation**: the infrastructure processes only ciphertext plus routing data, and every monetized capability is realizable without decryption.

Shipped v1 uses two message ciphersuites: **dr-v1** (X3DH + Double Ratchet) for in-app direct messages, and **box-v1** (per-message X25519 sealed boxes) for groups and the developer API. Handle-to-key bindings still use a plain Merkle log — not a VRF-blinded CONIKS directory. MLS/TreeKEM for groups remains roadmap. This specification documents both paths and is explicit, in §14 and §16, about what each achieves.

IMPLEMENTATION NOTES

All cryptography uses audited primitives from the `@noble` family (X25519, SHA-256, HKDF, AES-GCM, Argon2id), with `siwe` for EIP-4361 verification and `jose` for session tokens. The v1 server is a reference implementation over a single append-structured store; it is the directory, consent ledger, and relay queue, and is never a decryption oracle. Keywords MUST / SHOULD / MAY follow RFC 2119.

SECTION 2

2 · Notation & Primitives

We fix notation and the primitives actually used. ‖ denotes octet concatenation; ←$ uniform sampling; H is SHA-256; DH(sk, pk) is X25519.

ROLE	PRIMITIVE (V1)	LIBRARY / NOTE
Wallet identity & login	ECDSA / secp256k1	siwe, viem/wagmi · via EIP-4361
Session token	JWT (HS256)	jose; HttpOnly cookie, 14-day expiry
Device messaging key	X25519 keypair	@noble/curves
Prekey signatures (dr-v1)	Ed25519	over the signed prekey · @noble/curves
DM session (dr-v1)	X3DH → Double Ratchet	own impl, lib/crypto/signal/
Key agreement	X25519 ECDH	dr-v1 ratchet · box-v1 sealed box
KDF	HKDF-SHA-256	labels sealed-v1, sealed-dr-v1-{x3dh,root,chain} · @noble/hashes
AEAD	AES-256-GCM	96-bit IV, 128-bit tag · @noble/ciphers
Hash / Merkle log	SHA-256	domain-separated leaves/nodes, append-only tree
Backup KDF	Argon2id	t=3, m=64 MiB, p=1 — memory-hard
Fingerprints	SHA-256 of public keys	safety numbers

Table 1 — Aphanite v1 primitives, all standardized with audited implementations. Aphanite contributes composition and key management, not new primitives.

DEFINITION 2.1 — SEALED BOX

A sealed box encrypts a plaintext to a recipient public key pk_R using a fresh ephemeral keypair (e, E): the sender computes k = HKDF(DH(e, pk_R)) and outputs (E, AEAD_k(m)). The recipient recovers k = HKDF(DH(sk_R, E)). It provides recipient-only confidentiality and ciphertext integrity, but — using only an ephemeral–static agreement — carries no cryptographic sender authentication; sender identity is asserted out-of-band (§4.4, §5).

DEFINITION 2.2 — APPEND-ONLY MERKLE LOG

Leaves l_i = H("leaf:" ‖ payload_i); internal nodes H("node:" ‖ L ‖ R); an odd node is promoted by hashing it with itself. The root over t leaves is published with its size and timestamp on every append. An inclusion proof for index i is the authentication path recomputing the root in O(log t) hashes.

SECTION 3

3 · System & Threat Model

We enumerate parties, the adversary, and the goals each section discharges — and we scope the goals to what the implementation actually achieves.

Parties. Users own a wallet keypair and one or more devices, each holding an X25519 messaging key. The server plays three roles — registry (handle/device directory + Merkle log), relay (store-and-forward of sealed envelopes), and consent ledger — and additionally holds opaque backup blobs. All server roles are untrusted for confidentiality.

DEFINITION 3.1 — ADVERSARY $\mathcal{A}$

$\mathcal{A}$ is an active network attacker that also controls the server: it observes and manipulates all ciphertext and routing metadata and holds all backup blobs. $\mathcal{A}$ may compromise a device's stored X25519 private key, and may attempt to serve an incorrect key for a handle. $\mathcal{A}$ is computationally bounded and cannot break the primitives of Table 1. We do not claim detection of a server that equivocates on log roots in v1 (§7.4).

GOAL	STATUS IN V1	WHERE
G1 — Content confidentiality	Achieved (AEAD sealed box)	§6, §14.1
G2 — Ciphertext integrity	Achieved (GCM tag)	§6, §14.1
G3 — Forward secrecy	DMs: full (ratchet) · groups: sender-side	§6, §14.2
G4 — Directory inclusion auditability	Achieved (Merkle proof)	§7, §14.4
G5 — Non-equivocation of directory	Partial — roadmap	§7.4, §16
G6 — Sender privacy vs. relay	Established DMs: achieved (mailbox tags) · first contact/groups: not in v1	§10.3, §16
G7 — Post-compromise security	Achieved for DMs (dr-v1 ratchet); not for groups/API (box-v1)	§14.3, §16

Table 2 — Security goals and their honest status in the shipped v1.

SECTION 4

4 · Identity & Key Hierarchy

A wallet address is the root identity; a per-device X25519 key does all encryption, bound to the address by a wallet signature.

Key material (4.1). A user owns a wallet keypair over secp256k1 with address `addr`. Each device `d` generates an X25519 keypair (`dsk_d` , `DPK_d`) locally and stores it in browser storage. The wallet key performs no Diffie–Hellman and holds no message state; its only protocol role is to sign the §4.2 authorization.

Device authorization (4.2). The wallet authorizes a device key by signing a fixed, human-readable statement (a `personal_sign` message, not typed EIP-712 data; legacy string retained verbatim for signature continuity):

AUTHORIZATIONMESSAGE(ADDR, DPK) → STRING TO SIGN

```
sealed Device Authorization
Address: {addr}
Device messaging key: {DPK}
This key may encrypt and decrypt your sealed messages on this device.
It does not authorize spending.
```

The resulting signature σ_auth is stored with the device record and a `device` leaf is appended to the transparency log (§7). Verification recovers the signer from σ_auth and checks it equals `addr`. This binds DPK_d to the wallet without ever exposing the wallet key to the message path.

Multi-device & revocation (4.3). An address may authorize several device keys; `resolveDeviceKeys(addr)` returns all non-revoked keys, and a sender seals one copy of each message per key (§6.2). Revocation sets the device's `revokedAt` and appends a `device-revoke` leaf, making the change publicly visible.

Fingerprints & live verification (4.4). The **fingerprint** of a key is H(DPK) rendered as grouped hex. The pairwise **safety number**is H(a ‖ ":" ‖ b) over the two device keys in sorted order, truncated to 96 bits and grouped — a stable per-conversation value. For live proof of the address behind a chat, one party sends an in-band challenge and the other signs `"sealed — identity verification\nchallenge: <r>"` with their wallet; the verifier recovers the signer and checks it matches.

INVARIANT

The wallet key MUST NOT be used as an encryption or DH key. Encryption is performed exclusively by device X25519 keys; the wallet only ever signs identity assertions and never spends.

SECTION 5

5 · Authenticated Sessions (SIWE)

Sessions are bootstrapped by proving control of addr under EIP-4361, with a single-use nonce and a signed session cookie.

SESSION BOOTSTRAP (CHALLENGE-RESPONSE)

```
Client  ──  POST /api/auth/nonce { addr }  ──►
Server  ◄── r ←$ {0,1}^128; store (r, addr, exp = now+10m, used=false);
send r
Client  ──  M ← SIWE(domain, addr, r, …); σ ← walletSign(M);
            POST /api/auth/verify { M, σ }  ──►
Server  ◄── verify & issue session (Alg 1)
```

ALG 1 · VERIFYSIWE + ISSUESESSION — M, Σ → COOKIE | 401

```
1  { addr, nonce } ← siwe.verify(M, σ)   // EIP-4361 signature check
2  require rec = nonces.find(nonce, addr) ∧ !rec.used ∧ rec.exp > now
3  rec.used ← true                       // single-use: burn the nonce
4  primary ← linkedWallets.primaryOf(addr) ?? addr
5  sid ← newSession(primary, userAgent, loginAddress=addr)
6  jwt ← JWT{ sub: primary, sid }.sign(HS256, SESSION_SECRET), exp=14d
7  set-cookie sealed_session = jwt        // HttpOnly, SameSite=Lax
```

PROPOSITION 5.1 — REPLAY & CROSS-ORIGIN RESISTANCE

With 128-bit nonces marked used atomically and expiring after 10 minutes, a captured transcript cannot re-authenticate: the nonce is burned on first verify, and the SIWE message binds the site domain, so a signature solicited on another origin fails verification. Session validity additionally requires a non-revoked server-side session record, so logout or session revocation invalidates the token even before expiry.

A **linked wallet** signs `"sealed:link-wallet:<primary>:<linked>:<issuedAt>"`; thereafter a SIWE login from the linked address resolves to a session whose sub is the primary identity, with the login address recorded for transparency.

SECTION 6

6 · Message Encryption — Two Suites

Aphanite ships two interoperable encryption suites. Direct messages use dr-v1 — X3DH key agreement plus the Double Ratchet, for forward secrecy and post-compromise security. Groups and the developer API use box-v1 — a per-message sealed box. Both feed AES-256-GCM and are opaque to the relay.

SUITE	USED FOR	CONSTRUCTION	GUARANTEES
`dr-v1`	1:1 direct messages	X3DH → Double Ratchet, AES-256-GCM	FS + post-compromise security
`box-v1`	Groups, developer API	Ephemeral X25519 sealed box, AES-256-GCM	Sender-side FS (no ratchet)

Table 3 — The two message suites as implemented in lib/crypto/signal/ (dr-v1) and lib/crypto/device.ts (box-v1).

box-v1 — the sealed box (6.1).

ALG 2 · ENCRYPTFOR(M, DPK_R) → { CT, E, IV }

```
e ←$ X25519.secretKey(); E ← X25519.public(e)
s ← X25519(e, DPK_R)            // ephemeral–static ECDH
k ← HKDF-SHA256(ikm=s, salt=∅, info="sealed-v1", len=32)
iv ←$ {0,1}^96
ct ← AES-256-GCM(k, iv).encrypt(m)
return { ct: b64(ct), E: hex(E), iv: b64(iv) }
```

Opening reverses it: s = X25519(dsk_R, E), k = HKDF(...), then AES-GCM decryption. Each message uses a fresh ephemeral key that is discarded, giving sender-side forward secrecy; the recipient device key is long-lived, so box-v1 alone has no post-compromise security (§14). `sealForAllKeys(m, keys)` deduplicates the recipient's device keys and emits one envelope per key; an empty key set is a hard error. Groups reuse this suite by fanning a message out pairwise to every member device (§11).

dr-v1 — prekeys & identity (6.2). Each device publishes a **prekey bundle**so a peer can start a session while it is offline. The bundle contains the device's X25519 identity key IK, an Ed25519 signing key, a **signed prekey** SPK (X25519, signed by the Ed25519 key over `"sealed-dr-v1-spk"‖SPK`), and a set of **one-time prekeys** {OPK} consumed one per new session. The initiator verifies the SPK signature before use; private prekeys are vaulted locally on the responder.

dr-v1 — X3DH initial agreement (6.3). The initiator samples an ephemeral EK_A and mixes the four (or three) Diffie–Hellman terms:

X3DH · SEALED-DR-V1

```
F   = DH(IK_A, SPK_B) ‖ DH(EK_A, IK_B) ‖ DH(EK_A, SPK_B) [‖ DH(EK_A,
OPK_B)]
RK₀ = HKDF(F, info = "sealed-dr-v1-x3dh")
AD  = IK_A^pub ‖ IK_B^pub
```

Binding the IK terms authenticates both parties; the ephemeral and one-time terms give the first message forward secrecy. The handshake fields (ek, spkId, opkId) ride the first envelope only; the responder recomputes the symmetric secret from its stored prekey privates.

dr-v1 — the Double Ratchet (6.4). From RK_0 the session runs a symmetric-key ratchet nested inside a Diffie–Hellman ratchet, exactly as in Signal:

ALG 3 · KDF_RK / KDF_CK — 64-BYTE HKDF, SPLIT 32 | 32

```
1  (RK', CK) ← HKDF(RK ‖ DH_out, info="sealed-dr-v1-root", 64)   // DH
ratchet
2  (CK', MK) ← HKDF(CK, info="sealed-dr-v1-chain", 64)           //
symmetric ratchet
3  hdr = { dhPub, pn, n }
4  aad = AD ‖ dhPub(32B) ‖ pn(u32-BE) ‖ n(u32-BE)
5  ct ← AES-256-GCM(MK, nonce₁₂₈, aad).encrypt(m)
```

Each message key MK is used once and deleted, giving **forward secrecy**; every new ratchet public key folds fresh DH entropy into the root, giving **post-compromise security** — a session self-heals once an uncompromised message is exchanged. Out-of-order delivery is handled by a skipped-message-key cache (capped at 256 keys, oldest-evicted), and a single decrypt will compute at most 10,000 skipped steps before rejecting an implausible header. Header counters are validated as u32. Sessions persist per local address in IndexedDB (`sealed-dr-v1:<addr>`).

Envelope & proof-of-sealing (6.5). A dr-v1 envelope is `{ suite:"dr-v1", toAddress, recipientDeviceKey, header, ciphertext, nonce }` , the header carrying dhPub / pn / n plus the one-time handshake fields. A box-v1 envelope carries the fields below, and the relay applies a structural proof-of-sealing check (Alg 4) — AES-GCM output is indistinguishable from random, so a decoded ciphertext that is entirely printable ASCII is base64-wrapped plaintext and is rejected. The server enforces that clients sealed the message without ever reading it.

FIELD	ENCODING	CONSTRAINT (SERVER-ENFORCED)
`ephemeralPublicKey`	hex	64 hex chars (32-byte X25519 point)
`nonce`	base64	decodes to exactly 12 bytes
`ciphertext`	base64	≥ 17 bytes; MUST NOT be all-printable ASCII
`recipientDeviceKey`	hex	which device key this copy is sealed to

Envelope wire fields (box-v1), checked structurally by the relay before acceptance.

ALG 4 · SEALEDENVELOPEERROR(ENV) → ERROR | OK (BOX-V1)

```
1  if !/^[0-9a-f]{64}$/i.test(env.E) return "bad ephemeral key"
2  if len(b64decode(env.iv)) ≠ 12 return "bad nonce"
3  ct ← b64decode(env.ct); if len(ct) < 17 return "too short"
4  if allPrintableAscii(ct) return "ciphertext decodes to plaintext"
5  return ok
```

Structured payloads (6.6). Delete/retract control messages, disappearing (burn-on-timer) messages, group fan-out wrappers, and live-verification challenges all travel inside the encrypted body, marked with invisible prefixes. The relay therefore sees only ciphertext even for these control actions.

SECTION 7

7 · Transparency Log

Handle and device bindings are written to a single append-only SHA-256 Merkle tree that serves inclusion proofs against a published root.

Leaves (7.1). Three leaf kinds are appended. A `handle` leaf commits handle ‖ addr ‖ DPK ‖ timestamp; a `device` and a `device-revoke` leaf commit kind ‖ addr ‖ DPK ‖ timestamp. Each leaf hash is H("leaf:" ‖ payload), and the tree uses H("node:" ‖ L ‖ R) internally (Def. 2.2).

Roots & proofs (7.2). On every append the root is recomputed and a snapshot (root, at, treeSize) is stored. The public endpoint returns the latest root and recent entries, or — given a handle or address — an inclusion proof and its verification result.

ALG 4 · INCLUSIONPROOF / VERIFYINCLUSION

```
proof:  walk levels; at each, record sibling (hash, left|right); dup self
if odd
verify: h ← leafHash
        for sib in siblings: h ← sib.left ? H("node:"‖sib‖h) :
H("node:"‖h‖sib)
        return h == root
```

Client use (7.3). Before trusting a resolved key, a client MAY fetch and verify the inclusion proof for the handle or address, confirming the served binding is present under the published root. Because a rotation or revocation is a new leaf, a contact's key change is observable — the signal a client surfaces as "safety number changed."

7.4 — WHAT THE LOG DOES NOT YET PROVE

The v1 log proves inclusion and exposes key history, but it does not: (i) sign its roots, so a client cannot attribute a root to the operator cryptographically; (ii) publish consistency proofs between successive roots, so append-only-ness is not externally verifiable; (iii) gossip roots between clients or run an auditor/monitor protocol, so an operator serving divergent histories to different users is not detectable in-band; (iv) blind handle indices, so the log is a public directory with no lookup privacy. Consequently goal G5 (non-equivocation) is only partially met in v1. Closing this gap — signed roots, consistency proofs, gossip, and optional on-chain anchoring — is the primary directory item on the roadmap (§16).

SECTION 8

8 · Handle Registry

Handles are short, unique, human-readable names bound to an address and its device key.

A handle matches `^[a-z0-9_]{3,20}$`, is globally unique, and is limited to one per address. Claiming a handle appends both a `handle` leaf and (for a new key) a `device` leaf, upserts the device record with its authorization signature, and returns the new log root. The registry supports lookup by handle, reverse lookup by address, prefix search, and a public directory listing; verified accounts (§13) are flagged in each result. The device-key registry is independent of handles, so users who never claim a handle still have an authorized, logged key others can encrypt to.

SECTION 9

9 · Consent & Anti-Spam Controls

First contact is gated by recipient consent, backed by recipient-controlled spam marks, hard blocks, and per-sender rate limits. All of this gates delivery, never content.

Requests (9.1). A first message from A to B creates a pending request carrying one sealed intro per recipient device key, rather than landing in B's inbox. On accept, a mutual allowlist entry is written in both directions and subsequent messages route normally; on reject or contact removal, the request and any queued envelopes for the pair are cleared.

Spam marks, blocks & rate limits (9.2). A recipient may mark a sender as **spam**: delivery from that sender stops silently, the sender is not notified, and only the recipient can lift the mark — there is no payment or protocol path around it. A **hard block** stops delivery entirely. Independently, the relay enforces per-sender rate limits on request origination, throttling mass outreach before it reaches any Requests queue.

ALG 5 · ROUTEFOR(RECIPIENT, SENDER) → INBOX | REQUEST | BLOCKED

```
1  if isBlocked(recipient, sender) v isSpamMarked(recipient, sender) return
"blocked"
2  if !isAllowlisted(recipient, sender) return "request"  // subject to
per-sender rate limits
3  return "inbox"
```

COST MODEL

Anti-spam here is consent plus throttling, not payment: per-sender rate limits bound the request volume any one sender can originate, and a spam mark is a recipient-held veto that cannot be bought out. The messaging path carries no payment gates in either direction.

SECTION 10

10 · Relay & Delivery

The relay is a blind store-and-forward buffer with client-confirmed delivery and scheduled pruning.

Routing (10.1). On submit, `sendEnvelope` resolves one of three outcomes: `blocked` (hard block or spam mark — silent stop); `request` (queued as a consent intro, not yet allowlisted); or `inbox` (stored for pull). A spam mark is lifted only by the recipient.

Pull, acknowledge, prune (10.2). `pullInbox` returns undelivered envelopes without consuming them; the relay marks an envelope delivered only when the client acknowledges the specific id it decrypted. This ensures a session with a stale or wrong device key cannot destroy mail meant for another device, and nothing is lost if decryption fails. Delivered envelopes older than seven days, and any envelope older than thirty days, are pruned — the relay is a short-lived buffer, not an archive.

Unlinkable mailbox tags — sealed sender (10.3). Once a direct conversation is established, the two clients exchange a per-conversation secret via an encrypted control message inside the ratcheted channel and switch delivery to tag addressing. Each side derives a rotating mailbox tag `tag = HMAC-SHA256(per-conversation key, epoch day index)`, rotated daily. Envelopes on this path carry `{tag, ciphertext}` only — no sender address, no recipient address, no device identity — and are submitted, pulled, and acknowledged by naming the tag over unauthenticated, sessionless calls. Per-device routing uses HMAC-derived device hints in place of device keys. Plaintext lengths are hidden by padding to fixed-size buckets, and clients poll on a jittered schedule to blunt timing correlation. For these conversations the relay stores ciphertext under an opaque label and cannot see who is talking to whom.

10.4 — RELAY-VISIBLE METADATA (RESIDUAL)

First contact (consent requests) and group envelopes remain identity-addressed in v1: those envelopes carry plaintext senderAddress, recipientAddress, and recipientDeviceKey. Across all paths the relay still observes client IP addresses and coarse timing; reaching the relay over Tor or a VPN is the user-side mitigation for IP exposure. Goal G6 is therefore met for established direct conversations and unmet for first contact and groups — extending tag addressing to groups is roadmap work (§16).

SECTION 11

11 · Groups

Groups reuse the sealed-box channel: a group message is fanned out as pairwise envelopes to every member device. Bodies never touch the group record.

A group stores only membership and metadata (name, visibility, creator, optional emoji/image avatar, and an optional SHA-256 join-password hash). Sending seals the

message — wrapped with the group id and display name — once to each member device, so the content guarantee is identical to one-to-one chat. Free accounts cap groups at 8 members; the `largerGroups` entitlement raises the creator's cap to 64. Public groups are joinable (optionally behind a password, with a signed join statement), private groups are invite-only, and official verified system groups float to the top of discovery. Creators may mint group-scoped bot tokens (`gtk_live_…`, stored only as SHA-256 hashes) so services can post through the API (§13).

TRADE-OFF

Pairwise fan-out keeps groups consistent with the 1:1 model with no new cryptography, at a send cost linear in member devices — hence the member caps. A tree-based scheme (e.g. MLS/TreeKEM) is a natural future optimization but is out of scope for v1.

SECTION 12

12 · Encrypted Backup

Backups are encrypted client-side under a passphrase; the server stores only an opaque blob.

ALG 6 · ENCRYPTBACKUP(STATE, PASSPHRASE) → BLOB

```
1  salt ←$ {0,1}^128
2  k ← Argon2id(passphrase, salt; t=3, m=65536 KiB, p=1, dkLen=32)
3  iv ←$ {0,1}^96
4  ct ← AES-256-GCM(k, iv).encrypt(JSON(state))  // all sealed.* + device
key
5  return { v:1, salt: hex(salt), nonce: b64(iv), ct: b64(ct), count,
createdAt }
```

The archived state includes every `sealed.*` client store — messages, settings, contacts, and the device keypair itself — excluding only the demo key and the locally cached passphrase. The blob is uploaded to the backup store, which cannot decrypt it; restore re-derives k and recovers the exact messaging identity, able to decrypt history immediately. Backups may run automatically (hourly, if a passphrase is remembered) or be exported as a file. Without the passphrase the backup is unrecoverable — the intended cost of end-to-end encryption.

PROPOSITION 12.1 — BACKUP CONFIDENTIALITY

Given AES-256-GCM (IND-CCA2) and Argon2id as a memory-hard one-way function, an adversary holding (salt, nonce, ct) learns nothing about the state beyond its length, except with advantage bounded by the passphrase-guessing probability times the per-guess Argon2id cost. The store operator, lacking the passphrase, cannot decrypt.

SECTION 13

13 · Developer API & Entitlements

Paid features are entitlements unlocked by an on-chain payment; the developer API routes sealed envelopes without ever seeing plaintext.

An entitlement is keyed by `address:featureId` and recorded from a payment transaction hash. Features include verified accounts, larger groups, encrypted backup, and developer API access. API keys (`sk_live_…`) are minted under the `apiKeys` entitlement, stored only as SHA-256 hashes with a short display prefix, shown once at creation, rate-limited, and revocable. Requests to `/api/v1/*` authenticate with an Authorization: Bearer header.

ENDPOINT	PURPOSE
`GET /api/v1/me`	Introspect the calling key's identity
`GET /api/v1/handles/{handle}`	Resolve a handle to its record
`GET /api/v1/keys/{address}`	Fetch an address's device public keys
`GET /api/v1/directory`	Search the public directory
`GET · POST /api/v1/envelopes`	Pull inbox · submit sealed envelopes
`POST /api/v1/envelopes/ack`	Acknowledge decrypted envelopes
`GET /api/v1/groups · /{id}`	List groups · group + member device keys
`POST /api/v1/groups/{id}/envelopes`	Send sealed envelopes into a group

Table 4 — The versioned developer surface (OpenAPI 1.0.0, "Aphanite API"). Callers submit only opaque ciphertext plus recipient info; the API forwards it into the same relay used by the app and never handles plaintext.

SECTION 14

14 · Security Analysis

We revisit each goal and state the property achieved and its precise limit — reductions to the primitives of Table 1, not new assumptions.

Confidentiality & integrity — G1, G2 (14.1). Every message key is HKDF of an X25519 shared secret that is pseudorandom under Gap-CDH; under IND-CCA2 / INT-CTXT AEAD, AES-256-GCM hides the plaintext and detects tampering — in dr-v1 the message header is additionally bound as associated data. The server sees only ciphertext, and the proof-of-sealing check (Alg 4) prevents accidental plaintext submission on box-v1.

Forward secrecy — G3 — split by suite (14.2). In dr-v1 direct messages the symmetric ratchet derives a fresh message key per message and deletes it, so a later compromise cannot recover earlier keys — full forward secrecy, as in Signal. In box-v1 (groups, API) each message uses a fresh ephemeral sender key that is discarded, giving sender-side forward secrecy; but the recipient's long-lived device key can decrypt any box-v1 message ever sealed to it, so box-v1 has no forward secrecy against recipient-key compromise.

Post-compromise security — G7 — split by ciphersuite (14.3). For in-app direct messages (`dr-v1`), the Double Ratchet provides post-compromise security: after a device-key compromise, new DH ratchet steps heal the session so later messages are protected under fresh keys. Groups and the developer API still use stateless sealed boxes (`box-v1`) with no ratchet; compromise of a recipient device key there exposes all messages sealed to it until the key is rotated (a new authorized key + log leaf).

Authentication & directory — G4, G5 (14.4). A device key accepted for an address is wallet-authorized (recover-and-compare on σ_auth), and its binding is inclusion-provable in the Merkle log (G4). In dr-v1 the X3DH agreement mixes both parties' identity keys into the root, so a decrypting session mutually authenticates the peer's device identity; in box-v1 the sealed box carries no sender authentication, which rests on the authenticated session and, on demand, live wallet-signature verification (§4.4). Directory non-equivocation (G5) is only partial in v1 (§7.4).

Sender privacy — G6 — split by path (14.5). For established direct conversations, sealed sender via unlinkable mailbox tags (§10.3) removes sender, recipient, and device identity from the envelope: unlinkability reduces to the pseudorandomness of HMAC-SHA-256 under the per-conversation key, message lengths are bucket-padded, and fetches are sessionless. First contact and group envelopes remain identity-addressed, and the relay observes client IPs and coarse timing on every path (§10.4).

SCOPE OF CLAIMS

These are per-component arguments under standard assumptions, not a single composed proof. The construction is intentionally small and auditable; an independent review and, ideally, machine-checked models of §5–§7 SHOULD precede high-risk deployment.

SECTION 15

15 · Parameters & Ciphersuite

A single fixed ciphersuite for v1, versioned by a wire byte rather than in-band negotiation.

COMPONENT	CHOICE	PARAMETER
Wallet identity / login	secp256k1 · EIP-4361 (SIWE)	nonce 128-bit, 10-min TTL, single-use
Session	JWT HS256 (jose)	HttpOnly cookie; 14-day; server-revocable
Device key	X25519	generated & stored client-side
Key agreement	X25519 ECDH (ephemeral–static)	fresh ephemeral per message
KDF	HKDF-SHA-256	salt = ∅, info = `"sealed-v1"`, 32 B
AEAD	AES-256-GCM	96-bit IV, 128-bit tag
Transparency log	SHA-256 append-only Merkle	domain-separated `leaf:` / `node:`; roots unsigned (v1)
Backup KDF	Argon2id	t = 3, m = 64 MiB, p = 1, dkLen = 32
Safety number	SHA-256 of sorted device keys	96-bit, grouped display
Anti-spam	consent + spam marks	per-sender rate limits at the relay
Mailbox tags	HMAC-SHA-256(per-conversation key, epoch day index)	rotates daily; established DMs only; bucket-padded
Group join password	SHA-256(`"sealed-group:"` ‖ pw)	optional
API / bot keys	`sk_live_` · `gtk_live_`	24 random bytes; stored as SHA-256 hash

Table 5 — Aphanite v1 ciphersuite as implemented, built on @noble, siwe, and jose.

SECTION 16

16 · Limitations & Out-of-Scope

Stated plainly. Each item is a scoped property of v1, not a defect hidden from the reader.

- **Relay metadata.** Established direct conversations are tag-addressed (sealed sender, §10.3), but first contact and groups remain identity-addressed, and the relay observes client IPs and coarse timing on every path (§10.4). Tor or a VPN is the user-side mitigation for IP exposure.
- **Directory equivocation.** The log proves inclusion but roots are unsigned and there is no consistency-proof, gossip, or VRF layer, so a divergent-history attack is not yet in-band detectable (§7.4).
- **Post-compromise security (groups/API only).** Group and API traffic still uses `box-v1` sealed boxes with no ratchet; recipient device-key compromise exposes all messages sealed to it. In-app DMs use `dr-v1` and self-heal after new ratchet steps (§14.3). Key rotation remains the mitigation for long-lived compromise.
- **Sender authentication.** The sealed box does not authenticate the sender; identity rests on the session and optional live verification (§4.4).
- **Wallet-key compromise.** An attacker with the wallet key can authorize a rogue device; the event is logged and visible, but the root-key assumption is the user's to uphold.
- **Key/recovery loss.** Losing all devices and the backup passphrase renders history unrecoverable.
- **Reference server.** The v1 server needs production hardening (secret management, durable storage) before scale.
- **Deferred.** Signed/gossiped log roots, on-chain anchoring, tag addressing for groups and first contact, a ratcheted (MLS-style) group suite, and an independent audit are post-v1.

CLOSING POSITION

Aphanite v1 pairs `dr-v1` DMs with `box-v1` groups/API and a Merkle log — and this specification documents exactly what each path provides. The private core is real and the paid perimeter never touches it; the honest gaps (metadata, non-equivocation, PCS on sealed-box paths) are named and scheduled. Build the hardening in the open, keep the perimeter off the content path, and let every claim in this document remain checkable against the code.

SECTION A

A · References & Standards

Aphanite composes standardized primitives and libraries. The references below define the building blocks named throughout.

1. W. George et al. **EIP-4361 — Sign-In with Ethereum.** Ethereum, 2021. `eips.ethereum.org/EIPS/eip-4361`
2. D. J. Bernstein et al. **RFC 7748 — Elliptic Curves for Security (X25519).** IETF, 2016.
3. H. Krawczyk, P. Eronen. **RFC 5869 — HMAC-based Key Derivation Function (HKDF).** IETF, 2010.
4. NIST SP 800-38D — **Galois/Counter Mode (GCM) for AES.** 2007.
5. A. Biryukov et al. **RFC 9106 — Argon2 Memory-Hard Function for Password Hashing.** IRTF, 2021.
6. B. Laurie, A. Langley, E. Kasper. **RFC 6962 — Certificate Transparency.** IETF, 2013. Model for append-only Merkle logs.
7. Paul Miller. **noble-curves / noble-ciphers / noble-hashes.** Audited JS cryptography. `paulmillr.com/noble`
8. Spruce et al. **siwe — Sign-In with Ethereum reference library.** `login.xyz`
9. panva. **jose — JavaScript JSON Web Token / JWS / JWE.**
10. M. Marlinspike, T. Perrin. **The X3DH Key Agreement Protocol / The Double Ratchet.** Signal, 2016. Used for in-app DMs (`dr-v1`); groups/API remain sealed boxes.

About this document. Aphanite — Protocol Specification, Version 1.0 (Technical Preview), July 2026. This edition documents the protocol as implemented in the current codebase, including its honest limitations. It is a specification with per-component security arguments, not a completed proof or audit; the §5–§7 constructions SHOULD be independently reviewed before high-risk use. Keywords per RFC 2119.