BTC Medusa

Complete Cryptographic Specification

Privacy-Preserving KYC Query Engine for Bitcoin Wallets

This document provides a hyper-detailed specification of every cryptographic primitive, protocol flow, data structure, and security property in the BTC Medusa system, as implemented. It covers both the Rust server/crypto layer and the Java wallet (Sparrow fork) integration. Features that are designed but not yet built are clearly marked with amber PLANNED callouts. Designed for LLM ingestion and deep technical review.

June 2026
Groth16 Spend Circuit (6 Constraints, ~21,000 R1CS), Depth-16 Revocation Tree, Single-Point Spend, Payment-Gated Issuance, Encrypted Two-Column Block Filters with Codebook Indices

Table of Contents

1. System Overview and Architecture

BTC Medusa is a token-gated, privacy-preserving query protocol that lets Bitcoin wallet users check the KYC provenance of their UTXOs without revealing which UTXOs they own or which transactions they are interested in. The system combines Verifiable Oblivious Pseudorandom Functions (VOPRF), Groth16 zero-knowledge proofs on BN254, in-circuit DLEQ signature verification, and encrypted block filters — all coordinated through a single-point spend protocol where the ZK proof itself serves as authentication.

1.1 High-Level Data Flow

Issuance: Client pays (BIP-352 silent payment, Lightning, or Stripe) → payment produces a one-shot redemption credential → client generates blinded token commitments → server verifies the credential, consumes it, and blind-signs with per-token DLEQ proofs (Poseidon Fiat-Shamir) → client unblinds, verifies, stores tokens. The server never signs tokens without a consumed payment

credential.

Spend: Client fetches the current revocation root and a non-inclusion proof for its pack_id → generates a Groth16 proof (6 constraints, ~21,000 R1CS) → sends proof + a single OPRF blinded query point to POST /oprf/evaluate → server verifies the proof (including revocation non-inclusion and activation), burns the nullifier, evaluates the OPRF, and returns the blinded evaluation → client unblinds γ and decrypts the per-transaction codebook index from the encrypted block filter.

Transport: All wallet API calls route through the privacy transport layer. Default mode is AUTO (Tor first, with cascading fallback). Tor connects through an external SOCKS5 daemon to the server's native .onion service. Direct HTTPS exists for development. OHTTP is planned (Section 23).

1.2 Circuit Architecture and Design Choices

The spend circuit has 6 constraints (~21,000 R1CS). Three deliberate design choices define its shape:

- PK_server is a hardcoded R1CS constant, not a public input. Baking the key into the circuit at setup time means the prover cannot even attempt to substitute a key it controls, and it keeps the DLEQ gadget on cheap fixed-base scalar multiplication — a public-input key would force variable-base arithmetic whose deep linear-combination chains consume tens of GB of memory during trusted setup. The trade-off: rotating the server key requires a new ceremony; FROST threshold key management is the planned remedy (Section 11.8).
- Depth-16 Sparse Merkle revocation tree. 65,536 leaf slots cover decades of chargeback revocations while keeping the in-circuit non-inclusion walk cheap (one Poseidon hash per level). Issuance-time slot uniqueness keeps the 16-bit keyspace collision-free (Section 12).
- Activation epoch (start_month). A witness field and the Activation Check constraint enforce that a token pack cannot be spent before its designated activation month, enabling yearly prepaid subscriptions issued as a bridge pack plus 12 staggered monthly packs. The constraint mirrors the expiration check: it proves current_month − start_month is non-negative via 18-bit range decomposition.

Equally deliberate is what the circuit does not contain — in-circuit DLEQ signature verification makes several superficially attractive constraints unnecessary:

- No Pedersen quota commitment: the DLEQ proves the token was server-signed, which already proves legitimate paid issuance.
- No board-membership proof: the bulletin board is not needed for user security; it serves as an attribution layer (Sections 8, 14).
- No quota range check: there is no quota value in the circuit — the DLEQ limits spending to tokens the server actually signed.
- No query-point binding: the ZK proof is the authentication; the OPRF query point rides separately (Section 13).

1.3 Components

- perseverus-core — Rust crate: types, Poseidon, H2C, DLEQ, OPRF, token packs, SMT, filters, Golomb-Rice, Laplace decoy sampling
- circuits — Groth16 R1CS circuit on BN254 with BabyJubJub in-circuit arithmetic (6 constraints, including depth-16 SMT revocation and activation epoch)
- server — Axum server: payment-gated issuance, attribution board, spend verification, OPRF evaluation, filter and codebook serving, revocation tree, Stripe/Lightning/BTC payment routes

• client-native — Rust JNI cdylib bridging the Java wallet to the Rust crypto, including the transport layer (Direct / Tor / OHTTP-stub / Auto)

• Sparrow fork ("BTC Medusa" wallet) — Java wallet UI: Privacy tab, sign-up wizard, trial flow, settings, status-bar shield indicator

• sp-scanner — Python BIP-352 silent-payment scanner exposing /subscribe for on-chain payment claims

2. Elliptic Curve Parameters

2.1 BN254 (Proof System Curve)

The Groth16 proof system operates over the BN254 pairing-friendly curve. The scalar field Fr(BN254) is approximately 254 bits. This is the 'outer' curve that hosts the R1CS constraint system.

2.2 BabyJubJub (In-Circuit Curve)

All OPRF operations, token base points, DLEQ signatures, and nullifier derivation happen on the BabyJubJub twisted Edwards curve (ark_ed_on_bn254) embedded in Fr(BN254). This enables efficient elliptic-curve arithmetic inside the Groth16 circuit without cross-curve gadgets or pairing verification. Curve equation: $ax^2 + y^2 = 1 + dx^2y^2$, $a = -1$. Subgroup order $r_{jj} \approx 2^{251}$.

2.3 Generator Points

Generator	Purpose	Derivation
G	BabyJubJub base generator	Curve standard generator
H_TOKEN	Token base-point generator	NUMS via H2C: DST "Perseverus-V3-HTOKEN-BabyJubJub_SHA256_TAI_", label "perseverus-h-token-nums-generator-v3"
PK_server	Server public key V = v·G	Server key generation (hardcoded R1CS constant)

3. Poseidon Hash Configuration

Poseidon is the algebraic hash used throughout the circuit for nullifier derivation, Fiat-Shamir challenges, base-point derivation, and board/SMT hashing. It operates natively over Fr(BN254), making it vastly more efficient inside R1CS than SHA-256 or BLAKE.

Parameter	Value
Field	Fr(BN254), ~254 bits
Width (t)	3 (rate 2 + capacity 1)
Full rounds (R_f)	8
Partial rounds (R_p)	57
S-box	x^5 (quintic, ALPHA = 5)
Round constants	Grain LFSR-derived per Poseidon paper
MDS matrix	3×3 Cauchy matrix
Output	first squeezed rate element after the final permutation

Security level: 128-bit collision resistance, ~256-bit preimage resistance. The Poseidon configuration is deterministic — the same constants are used everywhere (circuit, server, client).

4. Hash-to-Curve (H2C)

Arbitrary byte strings are mapped to BabyJubJub points using SHA-256-based try-and-increment with cofactor clearing. Used for OPRF input hashing and NUMS generator derivation.

```
hash_to_curve_with_dst(data, dst):
  for counter in 0..:                       // unbounded; ~50% success per attempt
    x_candidate = SHA256(dst || data || counter.to_le_bytes())  (as field element)
    if point_on_curve(x_candidate):
      P = decompress(x_candidate)
      return cofactor_clear(P)              // mul_by_cofactor_to_group()
```

The DST (domain separation tag) ensures different contexts produce independent random-oracle instances. The OPRF input DST is "Perseverus-V3-VOPRF-BabyJubJub_SHA256_TAI_"; the H_TOKEN NUMS DST is "Perseverus-V3-HTOKEN-BabyJubJub_SHA256_TAI_". Cofactor clearing maps into the prime-order subgroup.

5. VOPRF Protocol (Verifiable Oblivious PRF)

The Verifiable Oblivious PRF is the core cryptographic mechanism. The server holds a secret scalar v and public key V = v·G. Clients blind their inputs so the server evaluates f_v(x) = v·H2C(x) without learning x. The 'V' in VOPRF comes entirely from the DLEQ proof — it proves the server used its real secret key without revealing it. The same key v is both the filter-encryption key and the token-signing key (the OPRF-gated per-tx key is a core design invariant).

5.1 Key Generation

```
v <- Fr::rand(rng)       // server secret
V = v * G                // public key
```

5.2 Blind Evaluation

```
CLIENT:
  P = H2C(input)
  r <- Fr::rand(rng)
  alpha = r * P                  // blinded point
  send alpha to server

SERVER:
  beta = v * alpha               // blinded evaluation
  (c, s) = dleq_prove(v, G, V, alpha, beta)
  send (beta, c, s)

CLIENT:
  verify dleq_verify(G, V, alpha, beta, c, s)
  gamma = r^(-1) * beta          // unblind: gamma = v * P = f_v(input)
```

6. DLEQ Proofs (Chaum-Pedersen)

Discrete Log Equality proofs are used in three contexts: (1) single OPRF evaluation (SHA-256 Fiat-Shamir), (2) batched token issuance, and (3) in-circuit token verification (Poseidon Fiat-Shamir).

6.1 Single DLEQ (SHA-256 Fiat-Shamir)

```
prove(v, G, V, alpha, beta):
  k <- Fr::rand(rng)
  R1 = k * G
  R2 = k * alpha
  c = SHA256(V || beta || R1 || R2) mod r
  s = k + c * v                          // response (addition)
  return (c, s)

verify(G, V, alpha, beta, c, s):
  R1' = s*G - c*V                        // subtraction form
  R2' = s*alpha - c*beta
  c' = SHA256(V || beta || R1' || R2') mod r
  assert c == c'
```

Correctness of the subtraction form: s·G − c·V = (k + cv)·G − c·(vG) = k·G.

6.2 In-Circuit DLEQ (Poseidon Fiat-Shamir)

Inside the ZK circuit, SHA-256 would cost ~25,000 constraints. Instead, the Fiat-Shamir challenge uses Poseidon over the affine coordinates of the transcript points: c = Poseidon(PK.x, PK.y, σ.x, σ.y, R1.x, R1.y, R2.x, R2.y). The server must issue per-token DLEQ proofs using this Poseidon-based transcript (not SHA-256) so the circuit can verify them. PK_server is a hardcoded R1CS constant. Cost: ~6,000 R1CS.

6.3 Batch DLEQ (Issuance)

During issuance the server signs N tokens. A batch DLEQ proof (batched_dleq_prove) proves all N evaluations used the same secret v. Batch proofs cannot be decomposed into per-token proofs, so the server also generates N individual Poseidon-Fiat-Shamir DLEQ proofs for circuit consumption. The issuance response carries both: the aggregate (challenge, response) pair and the per-token proof array.

7. Token Pack Structure

7.1 Token Definition

```
Token {
  base_point:  EdwardsAffine   // P_i = Poseidon(seed, pack_id, i) * H_TOKEN
  blinded_sig: EdwardsAffine   // sigma_i = v * P_i (server signature)
}
// per-token Poseidon DLEQ proofs (c_i, s_i) are stored alongside in the pack
```

7.2 Pack Constants

Constant	Value	Purpose
MAX_PACK_SIZE / MAX_TOKENS_PER_PACK	1,024 (1 << 10)	Maximum tokens per pack
Standard pack size (wallet)	100	All paid plans issue 100-token packs
Trial pack size	1–99	User-selected scan count on the $0.25 trial

7.3 Pack Blob Format (Persisted)

The issued pack is serialized to a compact binary blob persisted in the wallet's Config. It contains: pack_id, token_seed, the token array with per-token Poseidon DLEQ proofs (c_i, s_i), expiration_month, and start_month. Spent flags are tracked by the wallet alongside the blob (a

boolean array passed at persist time), not inside the core pack structure. The pack_id is bound into the token base-point derivation (P_i = Poseidon(seed, pack_id, i) · H_TOKEN), which lets the revocation constraint verify pack validity without revealing the pack_id to the server.

8. Bulletin Board and Attribution Layer

The bulletin board is not a public input to the ZK spend circuit and is never referenced by it — revocation uses the separate SMT (Section 12). The board is an attribution/audit ledger that records issued packs.

8.1 Board Structure (As Implemented)

Parameter	Value
Tree type	Append-only Sparse Merkle tree, depth 20 (~1M leaf capacity)
Hash function	Poseidon (same configuration as the circuit)
Leaf hash	Poseidon(pack_id, tag, expiration_month)
Internal hash	Poseidon(left, right)
Entry fields	pack_id (u64), tag (Poseidon field element), expiration_month (YYYYMM u32)

Entries are published via POST /board/publish at issuance time and served back via GET /board/root, /board/leaves, /board/entries, and an interactive HTML viewer at /board/viewer. The append-only property is a server-side invariant: publishing appends a leaf and rebuilds the tree; existing leaves are never modified.

PLANNED — NOT YET IMPLEMENTED: A dual-layer ECDH capsule design — leaf = Poseidon(R.x, R.y, tag, H(enc_payload)) with tag = H('wallet_attr' ‖ ss), an ephemeral key R, and an encrypted payload carrying pack_id, expiration, start_month, revenue_amount, and a wallet-attested inner blob under the wallet maker's key P_dev — is specified in Section 14 as planned work. None of it is implemented yet: today the board entry is the plaintext triple (pack_id, tag, expiration_month).

8.2 Why the Board Is Not a Circuit Input

The DLEQ signature already proves a token was server-signed, so a board-membership proof would add nothing to user security. The board therefore exists purely for attribution and audit, entirely separate from the ZK spend flow — board_root is never a public input to the circuit.

9. Token Issuance Protocol (Payment-Gated)

Invariant: the server never signs tokens without a consumed payment credential. Issuance is reachable only through the payment-gated endpoints /stripe/redeem (one-shot redemption code from a confirmed Stripe Checkout), /btc/redeem (HMAC-signed code from a confirmed on-chain silent payment), and /ln/claim (single-use nonce from a settled Lightning invoice). Each credential is atomically consumed before signing, so replay mints nothing. The bare POST /issue endpoint returns 403 FORBIDDEN unless the server is started with the development flag PERSEVERUS_ALLOW_UNGATED_ISSUE — it must never be set in production. Regression tests (Rust and Python) pin this behavior.

9.1 Phase 1: Client Preparation

```
prepare_pack_request(cfg, n, rng, pack_id) -> (PackRequest, PackSecrets):
  token_seed = Fr::rand(rng)
  for i in 0..n:
    P_i = Poseidon(token_seed, pack_id, i) * H_TOKEN
    r_i = Fr::rand(rng)
    alpha_i = r_i * P_i
  return (PackRequest { alphas, base_points }, PackSecrets { seed, pack_id, blinds })
```

9.2 Phase 2: Server Signing

```
// reached only via /stripe/redeem, /btc/redeem, /ln/claim
redeem(credential, alphas, base_points):
  consume_one_shot(credential) or 403/409   // atomic; replay-proof
  assert alphas.len() == base_points.len() <= MAX_TOKENS_PER_PACK
  for i in 0..n:
    beta_i = v * alpha_i
    (c_i, s_i) = dleq_prove_poseidon(v, G, PK, alpha_i, beta_i)   // per-token
  (C, S) = batched_dleq_prove(v, alphas, betas)                   // aggregate
  return { betas, proof_challenge: C, proof_response: S,
           poseidon_proofs: [(c_i, s_i); n] }
```

The request carries only {alphas, base_points}; the response carries only the blinded evaluations and proofs. There are no wallet-maker fields, amounts, or board proofs in the issuance API. The client publishes the board attribution entry separately, and chooses expiration_month / start_month locally when building the pack (the yearly path derives the schedule from the server's /epoch month).

9.3 Phase 3: Client Finalization

```
finalize_pack(secrets, response) -> IssuedPack:
  verify batch DLEQ over (alphas, betas)
  for i in 0..n:
    sigma_i = r_i^(-1) * beta_i           // unblind
    verify dleq_poseidon(G, PK, P_i, sigma_i, c_i, s_i)
  return IssuedPack { pack_id, seed, tokens, poseidon_proofs,
                      expiration_month, start_month }
```

9.4 Yearly Issuance (13 Packs)

The yearly plan redeems one payment credential for 13 packs in a single call sequence (issuanceRedeemYearly): a bridge pack with start = M and expiration = M+1 (covers the purchase month), then 12 monthly packs with start = expiration = M+2 … M+13 — each valid in exactly its designated month. M is the server's current epoch month (GET /epoch). The activation constraint (Section 11) makes future packs unspendable until their month arrives.

10. Nullifier Derivation

Nullifiers are one-shot spend tags. Each token produces a unique, deterministic nullifier that the server tracks (in a persistent sled-backed ledger) to prevent double-spending.

```
nullifier = Poseidon(token_seed, pack_id, token_index)
```

- Deterministic: the same token always produces the same nullifier — double-spend detection is trivial.
- Unlinkable: different tokens from the same pack produce independent nullifiers.
- Private: the nullifier reveals nothing about token_seed, pack_id, or token_index (Poseidon preimage resistance).

pack_id is included because it is a private witness in the revocation constraint; the nullifier must bind to the same pack_id to prevent substituting a different pack_id in the revocation proof. The binding chain is: pack_id → P_i → sigma_i → DLEQ → nullifier.

11. ZK Circuit Specification (6 Constraints)

The Spend Circuit is a Groth16 R1CS circuit on BN254 with BabyJubJub in-circuit arithmetic — approximately 21,000 R1CS constraints (the circuits crate's constraint_count test prints the exact figure). It proves the prover owns a valid, activated, unexpired, non-revoked, server-signed token and knows the secret seed that generated it.

11.1 Public Inputs (3 Fr elements)

Input	Type	Purpose
nullifier	Fr	One-shot spend tag (double-spend prevention)
current_month	Fr	YYYYMM — global expiration/activation parameter
revocation_root	Fr	Current root of the depth-16 Sparse Merkle revocation tree

11.2 Private Witnesses

Witness	Type	Purpose
token_seed	Fr	Deterministic token derivation seed
token_index	u32 as Fr	Which token in the pack
pack_id	Fr	Binds token to issuance batch (for revocation)
sigma_i (x, y)	2 × Fr	Server's blind signature on P_i
dleq_c, dleq_s	2 × Fr	Per-token Poseidon DLEQ challenge / response
expiration_month	Fr	YYYYMM — when the pack expires
start_month	Fr	YYYYMM — when the pack activates
smt_siblings[0..15]	16 × Fr	Sibling hashes along the SMT path for pack_id
smt_path_bits[0..15]	16 × bit	Left/right direction bits for SMT traversal

11.3 Constants (Hardcoded in R1CS)

H_TOKEN (NUMS token base-point generator), G (BabyJubJub generator), and PK_server (the server's VOPRF public key). PK_server as a constant is strictly stronger than as a public input — the prover cannot even attempt to substitute a different key, because the key is baked into the R1CS at setup time.

11.4 Derived In-Circuit

```
P_i(x,y) = Poseidon(token_seed, pack_id, token_index) * H_TOKEN
```

P_i is derived inside the circuit — never an input. Including pack_id binds each token to its issuance batch and prevents substituting a non-revoked pack_id while spending tokens from a revoked pack.

11.5 Constraint Details

Constraint 1: Base-Point Derivation (~4.500 R1CS)

```
scalar = Poseidon(token_seed, pack_id, token_index)
P_i = scalar * H_TOKEN
```

Prevents Arbitrary Point Injection (choosing $P_i = k \cdot G$ with known k and forging $sigma_i = k \cdot V$) and pack_id substitution: lying about pack_id changes P_i and invalidates the DLEQ signature.

Constraint 2: Nullifier Binding (~1.200 R1CS)

```
nullifier_derived = Poseidon(token_seed, pack_id, token_index)
enforce: nullifier_derived == nullifier
```

Zero degrees of freedom: the same token always produces the same nullifier, preventing nullifier grinding and double-spends.

Constraint 3: DLEQ Signature Verification (~6.000 R1CS)

```
// PK_server is a hardcoded R1CS constant
R1' = s * G - c * PK_server
R2' = s * P_i - c * sigma_i
c' = Poseidon(PK.x, PK.y, sigma_i.x, sigma_i.y, R1'.x, R1'.y, R2'.x, R2'.y)
enforce: c == c'
```

The most critical constraint: verifies the server's blind signature on the specific token base point. Forging (c, s) requires breaking discrete log on BabyJubJub (~125-bit) or finding a Poseidon preimage/collision (128-bit).

Constraint 4: Expiration Check (~254 R1CS)

```
diff = expiration_month - current_month
bits = bit_decompose(diff, 18)
enforce: reconstruction fits in 18 bits
```

If the token is expired, the subtraction wraps in Fr to a $\sim 2^{254}$ element that cannot fit in 18 bits. expiration_month is a private witness — the server learns only that the token is still valid.

Constraint 5: Activation Check (~254 R1CS)

```
diff = current_month - start_month
bits = bit_decompose(diff, 18)
enforce: reconstruction fits in 18 bits
```

Mirror of the expiration check, opposite direction: a pack with a future start_month cannot be spent yet. This enforces yearly-subscription pacing (Section 9.4). start_month is a private witness — the server cannot determine subscription tier, duration, or pacing schedule.

Constraint 6: Revocation Non-Inclusion (~8.600 R1CS, depth 16)

```
leaf_key = Poseidon(pack_id)            // truncated to 16 bits for the path
current = EMPTY_LEAF                     // zero — must be empty (not revoked)
for level in 0..16:
  sibling = smt_siblings[level]
  if smt_path_bits[level] == 0: current = Poseidon(current, sibling)
  else:                         current = Poseidon(sibling, current)
enforce: current == revocation_root
```

Proves the token's pack_id is not in the Sparse Merkle revocation tree (16 levels at ~540 R1CS per level: one Poseidon hash plus conditional selects and a zero-subtree shortcut). Policy: revocations apply only to credit-card (Stripe) payments; Bitcoin and Lightning payments are final and irrevocable. The pack_id is woven into base-point derivation, the nullifier, and the SMT proof, so substitution attacks fail. The server checks root freshness at spend time to prevent stale-root replay.

11.6 Why These 6 Constraints Are Sufficient

- Without Constraint 1 (Base-Point): inject known-DL points, forge DLEQ signatures, substitute pack_ids.
- Without Constraint 2 (Nullifier): random nullifiers — unlimited double-spending.
- Without Constraint 3 (DLEQ): fabricate tokens without server interaction — no payment needed.
- Without Constraint 4 (Expiration): one payment, lifetime access.
- Without Constraint 5 (Activation): burn all yearly packs in month one — defeats subscription pacing.
- Without Constraint 6 (Revocation): chargebacked packs remain usable indefinitely.

11.7 What the Circuit Deliberately Omits

- Pedersen quota commitment: the DLEQ already proves the token was server-signed during paid issuance.
- Board membership: same reasoning; the board is an attribution layer, not a security mechanism.
- Quota range check: there is no quota value in the circuit.
- Query-point (alpha) binding: single-point protocol — the proof is authentication, the query is separate.

11.8 PK_server as Constant — Key Rotation

PK_server is a hardcoded R1CS constant rather than a public input, which keeps the public-input surface at the minimal 3 field elements and the DLEQ gadget on cheap fixed-base arithmetic. As implemented, rotating the server key requires a new trusted setup ceremony, since the key is baked into the proving and verifying keys.

PLANNED — NOT YET IMPLEMENTED: FROST threshold key management is the planned rotation mechanism: distribute the server secret v across multiple key holders so the aggregate public key V = v·G stays fixed while individual shares are refreshed — no ceremony for share rotation, and a new ceremony only on catastrophic aggregate-key compromise. No FROST code exists in the codebase today; references to FROST in code comments are forward-looking.

12. Revocation Tree (Sparse Merkle Tree)

12.1 Purpose

The revocation tree lets the server invalidate issued token packs. Revocations apply only to credit-card (Stripe) payments — Bitcoin and Lightning payments are final and irrevocable by design. When a chargeback occurs, the associated pack_id is inserted into the tree. The tree is separate from the attribution board (which remains append-only).

12.2 Tree Structure

Parameter	Value
Tree type	Sparse Merkle Tree, depth 16 (mutable)
Hash function	Poseidon (same config as circuit)

Parameter	Value
Key derivation	leaf_position = Poseidon(pack_id) truncated to 16 bits
Valid leaf	0 (empty — pack is not revoked)
Revoked leaf	Non-zero value
Default nodes	0 at leaves, precomputed Poseidon(0,0) chain at each level
Max capacity	65,536 concurrent revocations

The SMT is sparse — only revoked pack_ids have non-zero entries, and only non-default nodes are stored. With N revocations, storage is O(N × 16) hashes.

12.3 Revocation Flow (Server-Side)

Revocation is performed through POST /revocation/revoke, an admin-authenticated endpoint (bearer auth token) that inserts the pack_id and republishes the root. The server exposes GET /revocation/root (current root + tree size) and GET /revocation/proof/{pack_id} (a non-inclusion proof, or an inclusion flag if the pack is revoked).

PLANNED — NOT YET IMPLEMENTED: Automatic revocation from Stripe chargeback webhooks (charge.dispute events driving /revocation/revoke) is not wired up yet — today an operator revokes chargebacked packs manually via the admin endpoint.

12.4 Client-Side Proof Generation

Before each spend the client fetches GET /revocation/root and GET /revocation/proof/{pack_id}, obtaining the 16 siblings and path bits it feeds the circuit as private witnesses. All non-revoked packs share the same root, so the request pattern does not distinguish packs; the pack_id itself never leaves the client.

12.5 Separation from Attribution Board

Property	Attribution Board	Revocation Tree
Structure	Append-only SMT, depth 20	Sparse Merkle tree, depth 16 (mutable)
Mutability	Never modified — invariant	Updated on each revocation
Purpose	Issuance attribution / audit	Token pack invalidation for chargebacks
Verified by	Off-chain consumers	ZK circuit (every spend)
Contains	(pack_id, tag, expiration) leaves	Revoked pack_id entries only
In circuit?	No	Yes — non-inclusion constraint

13. Single-Point Spend Protocol

The spend protocol is single-point: the ZK proof itself serves as authentication — no OPRF point is bound to the proof. The client sends the proof and a single OPRF query point to POST /oprf/evaluate (batch variant: /oprf/evaluate/batch, one spend proof per entry).

13.1 Client-Side Steps

```
Step 1: alpha_query = rho * H2C(outpoint)        // OPRF blinded query
Step 2: revocation_root = GET /revocation/root
        (siblings, bits) = GET /revocation/proof/{pack_id}
Step 3: proof = groth16_prove({ token_seed, token_index, pack_id, sigma_i,
          dleq_c, dleq_s, expiration_month, start_month,
          smt_siblings[0..15], smt_path_bits[0..15] },
          { nullifier, current_month, revocation_root })
Step 4: POST /oprf/evaluate { blinded_points: [alpha_query],
          spend: { nullifier, current_month, revocation_root, proof_hex } }
```

13.2 Server-Side Steps

```
Step 1: verify groth16_verify(vk, proof, [nullifier, current_month, revocation_root])
Step 2: check revocation_root == current published root   (403 on mismatch)
Step 3: check_and_burn(nullifier)                          (atomic; persistent ledger)
Step 4: beta_query = v * alpha_query
Step 5: return { beta_query, dleq proof }
```

The nullifier ledger is persistent (sled-backed in production, in-memory for tests), so restarts do not resurrect spent tokens. The root-freshness check prevents replaying a root from before the client's pack was revoked.

13.3 Client Decryption

```
gamma_query = rho^(-1) * beta_query
key = derive_tag_key(gamma_query)          // HMAC-SHA256(key = gamma, msg = "kyc-tag")
code = ctr_decrypt(key, nonce(txid, block_hash), columnB_entry)   // codebook index
tag24 = codebook[code]                     // 24-bit heuristic tag
```

13.4 Why Single-Point Is Sufficient

The ZK proof already proves possession of a valid, activated, unexpired, non-revoked, server-signed token, and is non-transferable (the prover must know token_seed). The OPRF query is a 'free rider' evaluated after authentication. Binding the query point to the proof would add constraints without adding security.

14. Attribution Board and Revenue Verification

As implemented, the attribution layer is the plaintext board of Section 8: one (pack_id, tag, expiration_month) entry per issued pack on an append-only depth-20 Merkle tree, published at issuance and queryable by anyone (root, leaves, entries, HTML viewer). This provides a tamper-evident record of how many packs were issued and when they expire — sufficient for transparency and audit, but it does not yet carry revenue amounts or wallet-maker attribution.

> **PLANNED — NOT YET IMPLEMENTED:** Everything below in this section is the planned revenue-verification design (dual-layer ECDH capsules). None of it is implemented: there is no wallet-maker registration, no P_dev key, no encrypted payloads, no revenue amounts on the board, and the wallet performs no board inclusion check before accepting tokens.

14.1 Planned: Wallet Maker Registration

```
x_dev <- Fr::rand(rng)   // wallet maker's secret key
P_dev = x_dev * G        // wallet maker's public attribution key
Register P_dev with the BTC Medusa server
```

14.2 Planned: Dual-Layer Attestation (Inner + Outer ECDH)

```
Inner layer (wallet-side):          Outer layer (server-side):
  r_w <- Fr::rand(rng)                 r <- Fr::rand(rng)
  R_w = r_w * G                        R = r * G
  ss_w = r_w * P_dev                   ss = r * P_dev
  inner_blob = AES(ss_w,               tag = H('wallet_attr' || ss)
               {amount_paid})          outer = {pack_id, expiration, start_month,
                                                revenue_amount, R_w, inner_blob}
                                       enc_payload = AES(ss, outer)
                                       leaf = Poseidon(R.x, R.y, tag, H(enc_payload))
```

14.3 Planned: Why Dual-Layer Prevents Server Fraud

- The server cannot forge the inner blob: it is encrypted under P_dev, which requires x_dev.
- Discrepancy detection: if the wallet-attested amount differs from the server-claimed revenue_amount, the wallet maker has cryptographic evidence (assert revenue_amount == amount_paid × split_rate).
- Trust assumption: relies on users running unmodified wallet software.

14.4 Planned: Anti-Cheating Enforcement

The wallet would require a board inclusion proof before accepting a token pack, so a server that skips the board entry (to avoid paying the wallet maker) gets its tokens refused. Today the wallet accepts packs after verifying the DLEQ proofs only.

15. Server API and Route Handlers

The complete implemented route table (axum):

Method	Route	Purpose	Gate
GET	/health	Liveness check	None
GET	/server/pubkey	Compressed OPRF public key (hex)	None
POST	/oprf/evaluate	Single-point spend: Groth16 proof + OPRF eval	ZK proof
POST	/oprf/evaluate/batch	Batch spends, one proof per entry	ZK proof
POST	/issue	Bare blind-signing (dev/test only)	403 unless dev flag
POST	/board/publish	Append attribution entry	None
GET	/board/root	Current board Merkle root	None
GET	/board/leaves	All leaf hashes	None
GET	/board/entries	Full board entries with metadata	None
GET	/board/viewer	Interactive HTML board viewer	None
GET	/revocation/root	Revocation SMT root + size	None
GET	/revocation/proof/{pack_id}	Non-inclusion (or inclusion) proof	None
POST	/revocation/revoke	Insert pack_id into revocation SMT	Admin auth token
GET	/circuit/pk	Groth16 proving key download	None
GET	/circuit/vk	Groth16 verifying key download	None
GET	/filters/{height}	Single block filter by height	None

Method	Route	Purpose	Gate
GET	/filters/{height}/exists	Existence check (no build)	None
POST	/filters/batch	Batch filter download	None
GET	/codebook	Global tag codebook	None
GET	/epoch	Current epoch month (YYYYMM)	None
POST	/clock/advance/{n}, /clock/set/{m}	Test clock control	Test builds
POST	/stripe/checkout	Create hosted Stripe Checkout Session	None
POST	/stripe/webhook	checkout.session.completed handler	Stripe HMAC sig
GET	/stripe/status/{nonce}	Poll for redemption code	None
POST	/stripe/redeem	Blind-sign one pack	One-shot redemption code
POST	/stripe/redeem-batch	Blind-sign yearly pack set	One-shot redemption code
POST	/btc/redeem	Blind-sign after on-chain SP payment	HMAC-signed one-shot code
POST	/ln/invoice	Create BOLT11 invoice (amount, tokens 1–100)	None
GET	/ln/status/{nonce}	Poll Lightning payment status	None
POST	/ln/claim	Blind-sign after settled invoice	Single-use LN nonce

15.1 Spend Endpoint

POST /oprf/evaluate accepts {blinded_points, spend} where spend carries the Groth16 proof and its three bound public inputs. The server verifies the proof, cross-checks revocation_root against the current SMT root, atomically burns the nullifier, then evaluates the OPRF and returns the blinded evaluation with a DLEQ proof. PK_server is baked into the verifying key as a constant.

16. Tag Encryption Pipeline (AES-GCM / AES-CTR)

Each transaction's KYC information is encrypted under a key derived from the OPRF evaluation γ. Only a client who successfully evaluates the OPRF (by spending a token) can derive the decryption key.

16.1 Key and Nonce Derivation

```
derive_tag_key(gamma) -> [u8; 32]:
  return HMAC-SHA256(key = gamma.serialize_compressed(), msg = "kyc-tag")

derive_nonce(txid, block_hash) -> [u8; 12]:
  deterministic per-(tx, block) derivation   // NOT random
```

The nonce is derived deterministically from txid and block hash so the server and client agree without transmitting it; γ itself is the HMAC key, with the fixed context string "kyc-tag" as the message.

16.2 Encryption Modes

Two symmetric modes are used. AES-256-GCM (12-byte derived nonce; truncated 8-byte auth tags where space matters) encrypts authenticated tag payloads. The filter's per-transaction codebook index (code_bits wide, currently 11 bits) is instead encrypted with a short AES-256-CTR keystream: encrypt_code_from_gamma draws 16 bits of keystream via the zero-block trick (ctr_keystream_u16), XORs it with the code, and masks the result to code_bits — (code ^ ks) & ((1 << code_bits) − 1). CTR without an auth tag is intentional here: Column B carries codebook indices, not authenticated KYC payloads, and integrity is provided by the filter pipeline itself.

17. Block Filter Structure (Two-Column)

The filter format is per-transaction and two-column: a BIP-158-style GCS membership column over txids, and a separate bit-packed column of per-tx OPRF-encrypted codebook indices.

17.1 Wire Format

```
Header (55 bytes):
  magic        : 4 bytes  "PSVR"
  version      : 1 byte   (5)
  height       : 4 bytes  LE u32
  block_hash   : 32 bytes
  num_entries  : 4 bytes  LE u32
  golomb_p     : 1 byte   (adaptive, chosen per block)
  code_bits    : 1 byte   (width of each Column-B code; currently 11)
  codebook_id  : 4 bytes  (hash of the global codebook)
  stream_len   : 4 bytes  LE u32

Payload:
  Column A : stream_len bytes — Golomb-Rice delta-encoded, sorted SipHash(txid) values
  Column B : bit-packed encrypted codebook indices, num_entries x code_bits bits
```

Column A supports O(1)-ish membership rank lookup; the rank indexes directly into Column B, so the client reads exactly one encrypted code per matched transaction. Decryption requires γ for that specific txid (Section 16), which requires spending a token — the OPRF-gated per-tx key.

17.2 Codebook

Column B does not store the 24-bit heuristic tag itself; it stores a compact index into a global codebook (GET /codebook) mapping each distinct 24-bit tag to a code. The current production codebook uses code_bits = 11 (1,030 distinct tags observed); the builder scales code_bits up (e.g. 13 bits for ~6,000 tags) as the tag population grows. The codebook_id in the header pins the codebook version a filter was built against.

18. Golomb-Rice Encoding (Adaptive P)

SipHash values of txids are sorted, delta-encoded, and compressed with Golomb-Rice coding, as in BIP-158. Unlike BIP-158's fixed P = 19/20, the filter builder chooses P adaptively per block: starting from a minimum, it increments P until the reduced hash space is collision-free for that block's transaction set (bounded by MAX_GOLOMB_P = 32). The chosen value is recorded in the header's golomb_p byte. Collision-freedom matters because Column A ranks must map 1:1 onto Column B entries.

```
Encoding each delta (parameter P, m = 2^P):
  q = delta / m ; r = delta % m
  write q ones + one zero        // unary quotient
  write r in P bits              // binary remainder
```

19. Decoy Block Fetching (Discrete Laplace)

Filter downloads themselves could leak which blocks (and hence which transactions) a wallet cares about. The client therefore fetches each real block's filter alongside decoy blocks sampled from a discrete Laplace (two-sided geometric) distribution centered on the real height: magnitude geometric with q = exp(−1/scale), uniform sign, bounded to the valid height range, deduplicated and sorted so real and decoy heights are indistinguishable in the request.

Parameter	Default	Meaning
numDecoys	10	Decoy blocks fetched per real block
scale	1,335	Laplace scale; spread = scale × ln(20) ≈ ±4,000 blocks
range	configurable	Wallet setting (e.g. month-scale spreads)

Implemented in core (select_decoy_blocks) and exposed through prefetchFiltersWithDecoys and queryUtxoAuthenticated (Section 22). Defaults: 10 decoys, ±4,000-block spread.

20. Silent Payment Integration (BIP-352)

BTC Medusa on-chain payments are received via BIP-352 silent payments. The service publishes a single static silent-payment address; each payment derives a unique on-chain output that only the service can detect, eliminating address reuse and preventing on-chain observers from linking subscribers. A Python scanner (sp-scanner) watches the chain for SP outputs. After confirmation, the payer proves ownership of the paying input by signing the claim message "perseverus-sp-claim-v1" ‖ pubkey(33) ‖ nonce(16) with the input key (ECDSA, single SHA-256 digest); the scanner's /subscribe endpoint verifies the signature against the observed a_sum input key, marks the payment redeemed (single-use), and returns an HMAC-signed redemption code ("psbtc1…") that /btc/redeem on the token server verifies and consumes for blind-signing.

21. Wallet Payment Flows

21.1 Path 1: Hot Wallet (Has Seed) — On-Chain Silent Payment

Single on-chain transaction. The wallet constructs a BIP-352 silent-payment output directly, signs with its private keys, and broadcasts. Coin selection is automatic or manual (the user can select UTXOs on the Privacy tab). The broadcast step is a pure payment — no claim signature is created or stored at send time. After the payment confirms, the wallet computes the claim proof at claim time from the confirmed transaction (recovering the spent input's key, requiring a fresh unlock for encrypted wallets), opens a fresh Tor circuit, and runs /subscribe → /btc/redeem (Section 20) to obtain the token pack.

21.2 Path 2: Watch-Only Wallet (No Seed) — Staging + Auto-Forward

Two-step process. (1) The wallet derives a deterministic hot P2TR staging child wallet (hardened index 2147483640) and shows a staging address; the user funds it from their hardware-signed wallet via the normal Send screen. (2) The wallet watches for the staging UTXO and automatically forwards it to the BTC Medusa silent-payment address (the staging child is hot, so it can sign the forward). The claim proof is then computed from the forward transaction as in Path 1.

21.3 Path 3: Stripe (Credit Card) — Hosted Checkout

The wallet generates a UUID nonce and calls POST /stripe/checkout over clearnet (Tor is deliberately skipped here: the user opens a clearnet Stripe page seconds later, so routing the bookkeeping calls over Tor adds latency with no privacy benefit). The server creates a hosted Stripe Checkout Session (mode=payment for one-time/yearly, mode=subscription for monthly) with the nonce in its metadata and returns the checkout.stripe.com URL, which the wallet opens in the system default browser. Card details are entered on Stripe's hosted page and never touch the BTC Medusa server. On checkout.session.completed, the webhook mints a 32-byte one-shot redemption code keyed by the nonce; the wallet polls GET /stripe/status/{nonce} (clearnet), then redeems over Tor to the .onion via /stripe/redeem with the code plus blinded points (pack size 100; the yearly plan uses issuanceRedeemYearly to obtain 13 packs — Section 9.4).

PLANNED — NOT YET IMPLEMENTED: A minimal embedded card form (collecting only card number, expiry, and CVC — no email, name, or address), and routing the checkout/status calls over OHTTP, are planned hardening. Today the payment page is Stripe's standard hosted Checkout form in the system browser.

21.4 Path 4: Lightning (phoenixd) — Including the $0.25 Trial

The wallet calls POST /ln/invoice {amount_sat, tokens} and receives a BOLT11 invoice plus a 16-byte single-use nonce. The invoice renders as a QR with a copy button (the trial screen also links to Cash App or any Lightning wallet). The wallet polls GET /ln/status/{nonce}; once the phoenixd invoice settles, it calls issuanceClaimLn → POST /ln/claim, which consumes the nonce and blind-signs the pack. The trial flow prices 1–99 scans at $0.25-per-scan equivalent (USD converted to sats via the mempool.space price API; the server clamps the authorized token count to 1–100). Lightning payments are final — never revoked.

22. JNI Bridge and Native Interface

The Java wallet communicates with the Rust crypto library via JNI. The native library perseverus_client_native exposes the following entry points (complete list):

Method	Parameters	Returns
`version`	—	String
`testG1GeneratorHex`	—	String (self-test)
`configureTransport`	mode (DIRECT/TOR/OHTTP/AUTO), ohttpRelayUrl	void
`httpGet / httpPost`	url [, jsonBody]	String (routed via transport)
`httpPostIsolated`	url, jsonBody	String (fresh Tor circuit)
`issuanceCreate`	baseUrl, serverPubkeyHex	long (handle)
`issuanceDestroy`	handle	void
`issuanceIssuePack`	handle, packSize, expirationMonth	byte[] (dev/test; gated server-side)
`issuanceRedeemPack`	handle, packSize, expirationMonth, redemptionCode	byte[] (Stripe)
`issuanceRedeemBtc`	handle, packSize, expirationMonth, code	byte[] (on-chain)
`issuanceClaimLn`	handle, packSize, expirationMonth, nonce	byte[] (Lightning)
`issuanceRedeemYearly`	handle, packSize, baseMonth, redemptionCode	byte[][] (13 packs)

Method	Parameters	Returns
`issuanceRepublish`	handle, issuedPackBytes	republish board entry
`spendBootstrap`	baseUrl, serverPubkeyHex, cachePath	long (handle)
`spendDestroy`	handle	void
`spendExecute`	handle, issuedPack, spendIdx, input, currentMonth	byte[] (gamma)
`spendBatchExecute`	handle, issuedPacks, spendIndices, inputs, currentMonth	byte[][] (gammas)
`spendBatchProofTimingsMs`	—	long[] (per-proof ms)
`queryUtxo`	serverUrl, txid, vout, blockHeight, numDecoys, chainTip	String (unauthenticated)
`queryUtxoAuthenticated`	serverUrl, txid, vout, blockHeight, numDecoys, chainTip, gamma	String (KYC tag)
`prefetchFilters`	serverUrl, blockHeights[]	void
`prefetchFiltersWithDecoys`	serverUrl, blockHeights[], numDecoys, chainTip, scale	void
`prefetchProgress`	—	int

spendExecute internally: derives P_i, fetches /revocation/root and /revocation/proof/{pack_id}, generates the Groth16 proof, POSTs to /oprf/evaluate, DLEQ-verifies, and unblinds γ. Groth16 proving/verifying keys are fetched from /circuit/pk and /circuit/vk and cached at cachePath.

23. Privacy Transport Layer (Tor / OHTTP / Auto)

23.1 Threat Model

Without transport privacy, the server sees the client's IP on every API call. Even though the cryptographic layer prevents the server from learning which UTXO is queried, a persistent IP lets it correlate issuance with spends, fingerprint request patterns, and build a per-IP activity log.

23.2 Transport Modes (As Implemented)

Mode	Status	How It Works	IP Hidden From
TOR	Implemented	SOCKS5 to a local Tor daemon (default socks5h://127.0.0.1:9050) → native .onion service; per-call circuit isolation available (httpPostIsolated)	Server + network observers
OHTTP	Planned (stub)	RFC 9458 relay; transport-layer code is a placeholder and the Settings option is disabled ("coming soon")	— (not functional)
DIRECT	Implemented (dev)	Plain HTTP(S); requires an explicit warning opt-in in Settings	Nobody
AUTO (default)	Implemented	Cascading fallback: Tor → OHTTP → Direct, in order of privacy strength	Depends on winner

PLANNED — NOT YET IMPLEMENTED: Tor is reached through an external SOCKS5 daemon — the wallet does not bundle its own Tor client; embedding one is a candidate future improvement. OHTTP is a non-functional placeholder pending relay infrastructure (the default relay URL is reserved at relay.btcmedusa.com/ohttp).

23.3 Tor Transport — Native .onion Hidden Service

The BTC Medusa server runs a native Tor hidden service; client traffic never exits the Tor network. No exit node is involved, the server's real IP is also hidden behind the .onion address, and neither party learns the other's network identity. Sensitive single-shot calls (e.g. token redemption) use an isolated fresh circuit.

23.4 Configuration and UI

The status bar shows a shield icon: green for Tor (protected), yellow for OHTTP (fallback), orange for Direct (IP exposed). The transport mode is configurable in Settings (DIRECT / TOR / OHTTP / AUTO); OHTTP is shown but disabled, and selecting DIRECT requires acknowledging an explicit warning. Exception: Stripe checkout-creation and status-poll calls are intentionally clearnet (Section 21.3); token redemption still rides Tor.

24. Token Lifecycle (End-to-End)

- Phase 1 — Purchase: user pays via Bitcoin silent payment, Lightning, or Stripe. Payment confirmation produces a one-shot credential (HMAC code / LN nonce / redemption code).
- Phase 2 — Issuance (payment-gated): client generates blinded commitments; server consumes the credential and blind-signs with per-token Poseidon DLEQ proofs plus an aggregate batch DLEQ; client unblinds, verifies, persists the pack (with expiration_month and start_month) in Config; an attribution entry (pack_id, tag, expiration_month) is published to the board.
- Phase 3 — Spend: client fetches the revocation root and non-inclusion proof, generates the Groth16 proof (6 constraints), sends proof + single blinded OPRF point to /oprf/evaluate; server verifies, burns the nullifier, evaluates.
- Phase 4 — Query: client unblinds γ, derives the HMAC tag key, decrypts the Column-B codebook index from the block filter (fetched with Laplace decoys), resolves the 24-bit heuristic tag via the codebook, and marks the token spent locally.
- Phase 5 — Activation: while current_month < start_month the circuit rejects the token; the wallet selects tokens from packs whose start_month has arrived.
- Phase 6 — Expiration: when current_month > expiration_month the circuit rejects the token.
- Phase 7 — Revocation (Stripe only): on a chargeback, the operator revokes the pack_id via the admin endpoint (webhook automation planned). Bitcoin and Lightning packs are never revoked.
- Phase 8 — Settlement (planned): wallet-maker revenue verification via dual-layer ECDH capsules (Section 14).

25. Exchange Registry and Heuristic Tags

The tag layer has two pieces: a registry of exchange identities, and the per-transaction heuristic tag the filters carry.

25.1 Exchange Registry

KYC exchange identifiers are encoded as compact 2-byte IDs. The implemented registry:

ID	Exchange	ID	Exchange
1	Coinbase	14	Poloniex
2	Binance	15	FTX
3	Kraken	16	BlockFi
4	Bitfinex	17	Celsius
5	Gemini	18	Nexo
6	Bitstamp	19	SwanBitcoin
7	OKX	20	River
8	Bybit	21	Cash App
9	KuCoin	22	PayPal
10	Huobi	23	Robinhood
11	Gate.io	24	eToro
12	Crypto.com	25	Strike
13	Bittrex		

25.2 Heuristic Tags

Filters do not ship raw exchange IDs. The ingest pipeline computes a 24-bit (3-byte) heuristic tag per transaction — encoding a risk grade (A–F), alert flags, and an entity class — and the filter stores an OPRF-encrypted codebook index for that tag (Sections 16–17). The wallet renders the decoded grade and alerts on the Privacy tab.

26. Security Properties and Invariants

Payment-Gated Issuance (NEW)

Tokens are never issued without a fresh, consumed payment credential. All three signing paths (/stripe/redeem, /btc/redeem, /ln/claim) atomically consume a one-shot credential before signing; the bare /issue endpoint is 403 by default. Regression tests in the server (Rust) and scanner (Python) continuously enforce: ungated issue rejected; malformed/forged/replayed credentials rejected.

DLEQ Token Integrity

Clients cannot fabricate tokens without a valid DLEQ proof under PK_server, and the in-circuit Poseidon-Fiat-Shamir DLEQ verification is the foundation of the spend circuit — it replaces the board membership check as the primary integrity mechanism.

OPRF Obliviousness

The server evaluates f_v(x) without learning x; the blinding factor ρ makes α a uniformly random group element independent of the input.

ZK Soundness

Groth16 soundness guarantees no polynomial-time adversary can prove a false statement except with negligible probability. The 6 constraints are individually necessary and jointly sufficient.

Double-Spend Prevention

Persistent nullifier ledger (sled-backed); nullifier = Poseidon(token_seed, pack_id, token_index) is deterministic and unique per token; burn is atomic with verification.

Expiration and Activation Enforcement

The circuit enforces start_month ≤ current_month ≤ expiration_month via 18-bit decompositions, enabling prepaid yearly pacing without revealing tier or schedule.

Revocation Enforcement

Depth-16 SMT non-inclusion (Constraint 6) ensures revoked packs cannot be spent; the server rejects stale roots. Revocations apply exclusively to credit-card payments.

Server Key Management

PK_server is a hardcoded R1CS constant; rotation currently requires a new trusted setup ceremony. FROST threshold management (stable aggregate key, rotating shares) is planned.

Transport Privacy

Default AUTO transport prefers Tor (.onion, external SOCKS5 daemon) so the server never learns the client's IP for issuance redemption, spends, or filter fetches. OHTTP fallback is planned. Stripe checkout bookkeeping is intentionally clearnet (Section 21.3).

Revenue Attribution Integrity (Planned)

The append-only board prevents removing or altering entries after publication. The dual-layer ECDH capsule scheme that would let wallet makers independently verify revenue is planned (Section 14).

27. Cryptographic Primitives Summary

Primitive	Algorithm	Field/Curve	Purpose
Hash-to-Curve	SHA-256 try-and-increment + cofactor clear	BabyJubJub	OPRF input mapping, NUMS
OPRF	Blind scalar mult + DLEQ	BabyJubJub	Private query evaluation
DLEQ (single)	Chaum-Pedersen (SHA-256 FS)	BabyJubJub	OPRF correctness
DLEQ (circuit)	Chaum-Pedersen (Poseidon FS)	BabyJubJub	In-circuit signature verify
Nullifier	Poseidon(seed, pack_id, idx)	Fr(BN254)	Double-spend prevention
ZK Proof	Groth16, ~21,000 R1CS	BN254 + BabyJubJub	6-constraint spend proof
Revocation	Depth-16 SMT, Poseidon	Fr(BN254)	Pack revocation (chargebacks)
Attribution	Depth-20 append-only SMT, Poseidon	Fr(BN254)	Issuance audit (capsules planned)
Tag key	HMAC-SHA256(γ, "kyc-tag")	—	Per-tx decryption key
Tag encryption	AES-256-CTR (16-bit keystream)	Symmetric	Codebook-index encryption
Block filter	Two-column, adaptive Golomb-Rice	SipHash	Compact membership + codes

Primitive	Algorithm	Field/Curve	Purpose
Decoys	Discrete Laplace (geometric)	—	Filter-fetch privacy
Payment	BIP-352 Silent Payment / BOLT11 / Stripe	secp256k1	Subscription payment
Transport	Tor (.onion via SOCKS5); OHTTP planned	—	Client IP privacy

— End of Specification —