Privacy-Preserving Query Engine

A Zero-Knowledge Oblivious Pseudorandom Function Framework for Anonymous Data Lookups in Bitcoin Wallets

Valentino Zertuche (z.valentino@gmail.com)

Perseverus Privacy Solutions
May 2026

1 Abstract

The principles of privacy and self-custody face continuing and increasing erosion in the Bitcoin ecosystem due to the widespread adoption of Know-Your-Customer (KYC) exchanges. There is now a need for tools that not only highlight this erosion but also empower users to reclaim control over their financial privacy. We propose a novel wallet-integrated scoring protocol that analyzes wallet UTXOs and provides users with a dynamic privacy score. As users adopt better acquisition practices, they can see their privacy scores improve in real-time, promoting a stronger self-custody ethos.

Historically, enabling wallets to query sensitive compliance information, such as whether an address has been flagged for KYC association, without sacrificing privacy, was seen as impractical. Addressing this challenge has revealed an innovative solution. We are introducing a token-gated, privacy-preserving query protocol that combines Verifiable Oblivious Pseudorandom Functions (VOPRFs) with Groth16-based Zero-Knowledge Succinct Non-Interactive Arguments of Knowledge (zk-SNARKs). Our system enforces usage limits via publicly verifiable nullifiers, distributes encrypted block indexes that are small enough to be downloaded on mobile, and publishes a tamper-evident attribution board so wallet makers can independently verify the revenue they are owed. No identifiable information is ever exposed to the privacy oracle server.

In this white paper, we detail the cryptographic constructions, payment flows (on-chain, Lightning, and fiat), threat model, and reference implementation, demonstrating that secure, anonymous, and monetizable query services are not only achievable but practical for integration into open-source Bitcoin wallets.

2 Open Development

Bitcoin's censorship-resistant design inherently clashes with the growing regulatory demands for Know-Your-Customer (KYC) enforcement. Today's compliance APIs require plaintext submission of wallet addresses or transaction IDs (TXIDs), thereby exposing user intent and fundamentally violating the principles of privacy and self-custody.

We propose a new system architecture wherein wallets can securely download encrypted block indexes and locally verify TXID statuses without ever revealing which transactions they are interested in. No user identifiers, TXIDs, addresses, or behavioral patterns are exposed to the server at any point.

At BTC Medusa, we are committed to developing simple, robust, and secure privacy-enhancing technologies that are accessible to all. We believe transparency and open community collaboration are critical to achieving this vision. Accordingly, this paper is presented as an open call for peer review, critique, and contribution, as we work to make meaningful privacy protections available to the broader Bitcoin self-custody community.

2.2 Outline

In the following sections, we delve into the technical details of the proposed architecture, demonstrating how it addresses the inherent challenges of bridging self-custodial private Bitcoin wallets with a for-profit oracle and contributes to advancing the industry toward secure, private, and user-friendly solutions.

- **Section 3. Secure Setup:** How a watch-only wallet derives a hot wallet for paying fees and registration, and how Oblivious Pseudorandom Functions blind the issuance handshake so the server cannot link a token pack to its eventual user.
- **Section 4. Secure Bitcoin Payment:** Paying our silent payment address through a deterministic staging hop, and how a single signature binds the entire blinded token pack to the on-chain payment.
- **Section 5. Private Fiat Payments:** Receiving credit-card payments via Stripe Checkout and issuing the token pack at settlement, with a chargeback-revocation path that does not weaken privacy.
- **Section 6. Lightning Payments:** A one-invoice user experience (BOLT 11 inbound) paired with automatic BOLT 12 settlement to wallet makers, all carried over independent Tor circuits.
- **Section 7. Block Indexes:** A compact block index with all TXID's and associated encrypted data that fits comfortably on mobile clients.
- **Section 8. Secure Queries:** Single-point oblivious lookups against a block index so that only the queried transaction can ever be decrypted.
- **Section 9. Zero-Knowledge Proofs:** A broad overview of how a zk-SNARK works, and a walk-through of the six constraints in our spend circuit that together gate every spend.
- **Section 10. Anonymizing Block Index Selection:** Discrete-Laplace decoy selection so that an observer cannot tell which block the wallet truly cares about.
- **Section 11. Attribution Board & Revenue Verification:** A public, append-only Merkle ledger of dual-layer ECDH capsules that lets wallet makers verify their revenue share without trusting the server, and lets the server publish revocations for chargebacked packs.
- **Section 12. Privacy Transport Layer:** A native Tor hidden service as the primary endpoint, Oblivious HTTP as a fallback, and explicit Tor-circuit boundaries at every step where linkability matters.

3 Secure Setup

While BTC Medusa offers free services, it is a for-profit company seeking to maintain verifiable security and privacy for its users. Our protocol ensures that no payment information, made via Bitcoin, Lightning, or fiat, can be linked to any wallet. Balances and outpoint queries are always cryptographically blinded to us.

We will have paid tiers of service that can be charged monthly or yearly. The most basic plan allows a wallet to run a full KYC check on roughly a hundred UTXOs at a time. To avoid misuse of our system and to enhance privacy, we use a token-based verification scheme. Each query on a (TXID, vout) pair costs one token and is exchanged via a zk-SNARK. The zero-knowledge proof guarantees that each token can be spent at most once and provably ties the spend to a paid subscription without revealing who that subscriber is.

Our scheme is double-blinded: a wallet can send our database a blinded query, and we can verify both that the token has never been used and that the holder is an authorized subscriber. BTC Medusa then blindly responds, granting the wallet access to the information it is seeking in a cryptographically provable manner. The open-source wallet software implementing all of this can be audited and verified. Throughout the rest of this paper we assume the basic plan ships **100 tokens** per pack, enough to cover the average self-custodial user's full UTXO set with comfortable headroom.

3.1 Deriving the Hot Wallet

We will be using Silent Payments to receive payments due to the robust privacy they maintain. However, due to the limited support of paying to a Silent Payment address from a cold storage wallet, we needed to develop an easy path for cold storage users to temp use a hot wallet as a staging wallet for payment.

We support two derivation paths. The first is the default and works with any watch-only setup. The second, will be made available for users who want the strongest possible isolation between their cold storage keys and the staging account.

Method 1: From the extended public key (default). Almost every modern wallet exposes its account-level XPub along with the four-byte master fingerprint. The wallet plugin uses the existing XPub and fingerprint as entropy to derive an unhardened child branch dedicated to payments. The derived branch produces a fresh, deterministic stream of receive addresses that will be immediately forwarded to our payment address. This path makes it easy for the user to recover the staging wallet from nothing more than the wallet descriptor they already back up.

$$\text{staging\_branch} = \text{XPub} / \text{“BTCMedusa”} / i \ \ (\text{master\_fingerprint})$$

The first staging address stg_0 becomes the public destination shown for on-chain subscription payments. The child transaction making payment to our silent payment address is immediately created after the staging transaction is signed. Both transactions are Broadcast simultaneously.

Method 2: From a BIP-85 child mnemonic (hardware wallet). Users who want stronger compartmentalisation can have their hardware wallet emit a dedicated child mnemonic via BIP-85 and import it as a hot wallet. Most modern signing devices support this directly:

Menu → Advanced / Tools → Derive Seeds (BIP-85) → 12 words → index 42

returning a child mnemonic $W = (w_0, \ldots, w_{11})$ at derivation path

$$m / 83696968' / 39' / 0' / 12' / 42'$$

The child mnemonic expands to a binary seed and is then folded down to a single 32-byte root key for the BTC Medusa engine:

$$S = \text{PBKDF2-HMAC-SHA512}(W, \text{“mnemonic”}, 2048) \in \{0,1\}^{512}$$

$$priv_{root} = \text{HMAC-SHA256}(S, \text{“BTCMedusa-v1”}, \text{“root”})$$

Either method produces a deterministic root from which we can recover the user's account state at any time. The remainder of this section assumes a root key called $priv_{\text{root}}$ without caring which method produced it.

3.2 Blinding the Token Pack (VOPRF Setup)

The wallet pings the BTC Medusa server, selects a tier, and initiates the issuance handshake. The plan counter i increments with each new pack, deriving a fresh token seed:

$$token\_seed_i = \text{HMAC-SHA256}(priv_{root}, \text{“BTCMedusa-v1”}, \text{“}token\_seed_i\text{”})$$

For each of the 100 tokens in the pack, the wallet hashes the seed and index to a curve point on the zk-SNARK curve, then multiplies by a public NUMS point on the same curve H_{token} to produce the blinded input the server will see:

$$P_i = H2C(token\_seed_i \parallel i) \cdot H_{token}$$

$$\alpha_i = r_i \cdot P_i, i \in [0, 99]$$

The blinding scalars r_i are randomly generated and are used just for the initial VOPRF setup. The wallet sends the chosen $plan\_id$ together with $100\ \alpha_i\ values$. The server applies its secret v to each blinded point and returns the evaluations.

$$\beta_i = v \cdot \alpha_i$$

3.3 Per-Token DLEQ Proofs

For each β_i the server also generates a Discrete Log Equality (DLEQ) proof under its long-term public key $V = v \cdot G$. Without this proof, a malicious or mis-configured server could return a β_i that fails verification at spend time, silently bricking the pack. Worse, a server could slip in one β_i computed under a different key, fingerprinting that token so the eventual spend would identify the user.

The proof itself is a standard Chaum-Pedersen DLEQ, made non-interactive with the Fiat-Shamir transform. For each token, the server picks a fresh nonce k and computes:

$$R_1 = k \cdot G, R_2 = k \cdot \alpha_i$$

$$c = H(V \parallel \beta_i \parallel R_1 \parallel R_2), s = k - c \cdot v$$

The wallet recomputes:

$R_1' = s \cdot G + c \cdot V$ and $R_2' = s \cdot \alpha_i + c \cdot \beta_i$, and accepts if $H(V \parallel \beta_i \parallel R_1' \parallel R_2') = c$.

At spend time the wallet does not just want the server to evaluate the VOPRF correctly; it wants to *prove inside its zk-SNARK* that the token it is about to burn was signed under the server's public key. Constraint 3 of the spend circuit (Section 9) re-runs the DLEQ check using Poseidon as the Fiat-Shamir hash so that it is cheap to verify inside the R1CS. Without a per-token DLEQ proof, that constraint has nothing to consume, and the entire token-economy guarantee collapses into “the server says so.” With it, every spend cryptographically demonstrates that a real, paid-for token is being burned.

After verification the wallet unblinds each evaluation and stores it for later.

$$\sigma_i = r_i^{-1} \cdot \beta_i = v \cdot P_i$$

The unblinded signatures σ_i are kept private and never revealed in cleartext. They are only ever consumed inside the zk-SNARK as a private witness. This completes the secure setup.

4 Secure Bitcoin Payment

When paying via Bitcoin, the user sends to our **Silent Payment Address** (BIP-352). Silent payments give us the highest degree of privacy with no address reuse and let any wallet that implements our protocol pay into the same static endpoint without leaking that fact on chain.

Most hardware wallets cannot yet construct a silent payment output directly, so we use a small staging hop on the way to our SP address. The staging hop is the hot wallet derived in Section 3.1, either from the XPub or the BIP-85 child mnemonic. Either way, the user sees a single deterministic address that they treat as the payment destination.

Treating the two transactions as fully independent keeps the flow robust (a stuck parent never blocks settlement) and lets the user pick any reasonable fee rate for the cold-side transaction. The hot wallet only ever signs the forwarding hop, so the user's main keys never have to be hot.

The split output to the wallet developer's silent payment address turns BTCMedusa into a source of residual income for any wallet that ships the integration.

4.1 Binding the Payment to the Token Pack

How does the server know that *this* incoming silent payment corresponds to *that* set of 100 blinded α_i values, without the user revealing any link between their wallet and their account?

After broadcasting the transaction to our silent payment address, it does nothing until that transaction confirms in a block. Confirmation is the trust anchor, it removes any race conditions around replacement and any need for the server to track unconfirmed mempool state.

Once the payment has at least one block confirmation, the wallet opens a **fresh Tor circuit** to our servers onion address and posts a single request containing three things:

- the wallet-chosen $pack\_id$
- a signature over $pack\_id$ produced with the private key that signed the input spending to our silent payment address
- the 100 blinded tokens $\{\alpha_0, \ldots, \alpha_{99}\}$.

$$\sigma_{\text{pack}} = \text{Sign}_{sk_{\text{pay}}}(pack\_id)$$

The server can see the silent payment output on chain, recover its sender pubkey, and verify the signature directly against it. A valid signature is unforgeable proof that the request is coming from the party who actually paid for the subscription. The server ties the $pack\_id$ to the confirmed payment, runs the VOPRF on each α_i deriving the β_i, attaches the per-token DLEQ proofs from Section 3.3, and returns the signed pack.

Because the binding step happens over a Tor circuit after confirmation, the server learns the $pack\_id$ and the payment-key signature, but it does not learn the user's IP or identity. The blinded queries can be tied to a token pack, but it still cannot be tied to any particular wallet.

5 Private Fiat Payments

For users who cannot or do not want to pay on chain, we accept credit-card payments through Stripe. The flow is a two-step exchange that pairs a Stripe Checkout Session with a one-shot redemption code, and at no point does the wallet register any long-lived identifier with us. The card details, the

token-issuing step, and the user's identity are kept in three separate places: the card goes only to Stripe, the tokens are issued only over Tor, and no wallet public key is ever bound to the purchase.

5.1 Flow

The wallet generates a random UUID nonce locally and sends it to our server in a POST /stripe/checkout call, together with the chosen plan. This call is made over clearnet to the token server. Tor is deliberately not used here: the very next step sends the user to a clearnet Stripe page in their own browser, so routing the session-creation call over Tor would add latency with no privacy benefit. The server creates a Stripe Checkout Session — mode=payment for the one-time and annual plans, mode=subscription for the monthly plan — selects the matching Stripe price ID for the plan, embeds the nonce in the session's metadata, and returns the hosted Checkout URL (https://checkout.stripe.com/...). No wallet public key, no blinded tokens, and no identifying material change hands at this step; the only thing the wallet sends is the plan name and the random nonce.

The wallet opens that URL in the user's system default browser. The hosted Checkout page is served by Stripe directly over clearnet, and the card details are entered there. This is the only point in the flow where the user's card identity is disclosed, and it is disclosed to Stripe alone — our server never sees the card number, CVV, expiry, or billing details. Stripe is about to learn the user's full card identity anyway, so the IP that loads the Checkout page adds nothing on top.

When the charge succeeds, Stripe fires a checkout.session.completed webhook to our server. The server's webhook handler reads the nonce out of the session's metadata, generates a fresh 32-byte random redemption code, and parks that code in memory keyed by the nonce. The webhook signature is verified (v1 = HMAC-SHA256(timestamp.payload)) before anything is stored.

Meanwhile the wallet has been polling GET /stripe/status/:nonce over clearnet. The endpoint returns pending until the webhook fires, then ready together with the redemption code.

The wallet now performs the second and final exchange, and this one runs over Tor. It opens a Tor circuit to the server's onion address and calls POST /stripe/redeem, attaching the redemption code together with the 100 blinded points $\{\alpha_i\}$ (and their base points). The redemption code is the only authentication for this call — there is no wallet public key and no signature, because none is needed: the code was minted against a confirmed Stripe payment and it is one-shot. The server locates the nonce whose stored code matches, deletes it so it can never be replayed, checks that the number of blinded points matches and is within the per-pack maximum, runs the VOPRF to compute each $\beta_i = \nu \cdot \alpha_i$, attaches both the aggregate DLEQ proof and the per-token Poseidon DLEQ proofs from Section 3.3, and returns the signed pack. (The annual plan issues twelve monthly packs in one redemption; the one-time and monthly plans issue a single pack.)

5.2 What the Server Learns (and What It Does Not)

The current implementation makes a deliberate trade: the card and the issued tokens are unlinkable, but the session-creation and status-poll calls run over clearnet, so the server does see the user's IP for those two calls. The tokens themselves — the thing that actually gets spent later — are issued over Tor against an anonymous one-shot code, so they cannot be tied back to the card or to the wallet.

Concretely, across the whole flow the server can correlate four things: the nonce sent on the create-checkout call, the same nonce echoed inside Stripe's webhook payload, the redemption code it mints, and the blinded points presented at redeem time. That is the entire log it can keep on a card subscription. None of it contains the card, and none of it binds a wallet public key to the purchase.

The card itself never reaches our server. Stripe handles the entire PCI side — card number, CVV, address verification, everything — on its own hosted Checkout page.

Our server does see the user's IP on the create-checkout and status-poll calls, because those run over clearnet. It does not see the IP on the token redemption, which runs over Tor to the onion address. Stripe sees the IP that loads the Checkout page, but Stripe already holds the user's full card identity through the same transaction, so that IP is not a marginal disclosure to them.

The token issuance is unlinkable to the payment session. The redeem call carries only the anonymous redemption code and the blinded points, over a fresh Tor circuit, with no wallet public key and no signature.

The blinded points $\{\alpha_i\}$ are statistically independent of the user's identity. The VOPRF blinding factor r_i is a uniformly random scalar the server never sees, so α_i leaks nothing about P_i or about who the holder is.

5.3 Revocation for Chargebacks

Credit Card payments introduce one risk that on-chain and Lightning payments do not. A user can pay, receive a full pack, and then issue a chargeback. To prevent the chargebacked pack from continuing to burn queries forever, we maintain a **Indexed Merkle revocation tree** keyed by $\text{Poseidon}(pack\_id)$. A valid leaf is the zero element (the pack is good). A revoked leaf holds $\text{Poseidon}(pack\_id, t)$ for the revocation timestamp t.

When Stripe fires a charge.dispute.created webhook, the server looks up the associated $pack\_id$ and inserts it into the revocation tree. It then publishes the new tree root at GET /revocation/root. The wallet fetches this root before every spend, and Constraint 6 of the spend circuit (Section 9) proves *in zero knowledge* that the wallet's $pack\_id$ is **not** in the tree. As soon as a $pack\_id$ is inserted, any remaining unspent tokens in that pack become un-spendable. The wallet cannot construct a valid non-inclusion proof against the new root.

Crucially, **revocation applies only to credit-card payments**. Bitcoin and Lightning payments are both final and irreversible, there is no chargeback path, so packs paid that way are never inserted into the revocation tree under any circumstances. This is a real, user-visible advantage of paying in cryptocurrency. Once settled, the pack is permanently valid.

We discuss revocation again in Section 9 (the constraint), Section 11 (the published root), and the privacy properties in Section 12.

5.4 Attribution and Wallet-Maker Accountability

Even though the server has full knowledge of the dollar amount on a CC charge, the wallet maker who shipped the integration cannot just take our word for what their revenue share should be. To solve this without weakening user privacy, credit card and Lightning payment issuances produces an entry on a public **attribution board** that anyone can mirror and audit.

The entry is built in two encryption layers. First, on the wallet side, the wallet picks an ephemeral scalar r_w, derives $R_w = r_w \cdot G$ and the shared secret $ss_w = r_w \cdot P_{\text{dev}}$ with the wallet maker's public key P_{dev}, and produces an **inner blob** that AES-encrypts the actual amount the user paid:

$$\text{inner\_blob} = \text{AES}(ss_w, \{amount\_paid\})$$

The wallet hands $(R_w, \text{inner\_blob})$ to the server along with the rest of the redeem payload. The server cannot decrypt or modify the inner blob because it does not hold x_{dev}. Next, on the server side, the server picks its own ephemeral scalar r, derives $R = r \cdot G$ and a separate shared secret $ss = r \cdot P_{\text{dev}}$, and wraps the wallet's inner blob inside an **outer blob** together with the metadata only the server

knows:

$$\text{enc\_payload} = \text{AES}\left(ss, \{pack\_id, expiration\_month, revenue\_amount, exchange\_rate, R_w, \text{inner\_blob}\right.$$

For credit cards, the outer payload also carries the BTC/USD exchange rate used at issuance so the wallet maker can verify the fiat-to-sat conversion.

What actually gets posted to the bulletin board is a single Poseidon hash over four fields:

$$\text{leaf} = \text{Poseidon}\left(R_x, R_y, \text{tag}, H\left(\text{enc\_payload}\right)\right)$$

where $\text{tag} = H\left(\text{“wallet\_attr”} \parallel ss\right)$ is a deterministic identifier the wallet maker uses to spot their own leaves without trial-decrypting every entry on the board. No cleartext metadata (no pack ID, no expiration, no revenue amount, no card identity) is exposed in the leaf itself. An external observer sees only an opaque curve point and a hash. The wallet maker, holding x_{dev}, recovers ss, matches tag, decrypts the outer blob, then decrypts the inner blob and cross-checks the wallet-attested amount against the server-claimed revenue share. Any discrepancy is cryptographic evidence of dishonesty, and the wallet maker can prove it without revealing which user it came from.

To make explicit why the wallet maker can open both layers: the wallet maker's public key $P_{\text{dev}} = x_{\text{dev}} \cdot G$ ships inside the wallet integration code, so the wallet and the server are each encrypting to the same long-term key. The standard Diffie–Hellman identity makes those secrets recoverable from either half of their key pairs: because scalar multiplication commutes, $r \cdot P_{\text{dev}} = r \cdot \left(x_{\text{dev}} \cdot G\right) = x_{\text{dev}} \cdot \left(r \cdot G\right) = x_{\text{dev}} \cdot R$, and likewise $r_{\text{w}} \cdot P_{\text{dev}} = x_{\text{dev}} \cdot R_{\text{w}}$.

In other words, ss and ss_{w} are not separate secrets that must somehow be handed to the wallet maker; each is one shared secret known to exactly two parties, its encryptor and the holder of x_{dev}. Note that the leaf itself is never decrypted. It is a one-way Poseidon hash that serves purely as tamper-evidence; the board publishes the full capsule $\left(R, \text{enc\_payload}\right)$ alongside every leaf. Recovering a payment record therefore requires nothing beyond x_{dev} and a mirror of the board: compute $ss = x_{\text{dev}} \cdot R$ and match tag, decrypt the outer blob with ss to recover R_{w} and inner_blob, then compute $ss_{\text{w}} = x_{\text{dev}} \cdot R_{\text{w}}$ and decrypt the inner blob to reveal the wallet-attested amount_paid.

The board is an **append-only** Merkle tree, hashed with the same Poseidon configuration as the zk-SNARK so that any audit logic can be ported into a circuit later if we ever need to. We chose this structure for two reasons. First, *append-only* is exactly the security property wallet makers need. A server that could remove or rewrite past leaves could quietly hide subscriptions and underpay them. An append-only tree makes that impossible without forking the published root. Second, *it* gives us cheap inclusion proofs (logarithmic in the size of the board), so a wallet maker who has been offline for months can fetch only their own leaves with Merkle paths against the current root and verify completeness without re-downloading everything. The wallet itself uses the same proofs as an anti-cheating gate. It refuses to accept an issued pack until it has verified an inclusion proof, which forces the server to publish the attribution entry before it can release the tokens.

Note that the attribution board is a different structure than the Indexed Merkle revocation tree introduced in Section 5.3. The two coexist on purpose. The attribution board is strictly append-only because the server must never be able to take entries down, while the revocation tree is mutable by design because the server must be able to insert new revocations. The full walkthrough of the board lives in Section 11.

6 Lightning Payments

Lightning payments are arguably the best fit for our usage pattern, small, frequent, and instantly

settled, but they introduce a subtle UX problem if approached naively. Paying both us and the wallet maker would require two separate Lightning invoices. Users tolerate one QR code. Two breaks the flow and would likely be confusing to the user. We solve this with a **nonce-based claim flow** that lets the user pay exactly one BOLT 11 invoice, with our server automatically forwarding the wallet maker's share over BOLT 12 within seconds.

The wallet opens a Tor circuit, generates a wallet-side attestation blob $= \mathrm{AES}(ss_w, \{amount\_paid\})$ encrypted under the wallet maker's public key and posts it to our invoice endpoint together with the chosen tier and wallet-maker ID. The server creates a BOLT 11 invoice, mints a 16-byte claim nonce, and returns both. The nonce never appears in the Lightning invoice. It lives only in the API response over this Tor circuit.

The user pays the BOLT 11 invoice from any Lightning wallet, Cash App, Wallet of Satoshi, Phoenix, Strike, an embedded LDK channel, anything that speaks BOLT 11. There are no custom TLVs and no HTLC interceptors, so custodial wallets work just as well as self-custodial ones.

Once the wallet has paid, it opens a *new* Tor circuit, posts the nonce plus its 100 blinded α_i points to our server and receives the signed β_i values, the per-token DLEQ proofs, the pack's expiration, and a board-inclusion proof.

Splitting the invoice request and the claim across two independent Tor circuits is critical. If both calls came out of the same circuit, the server could trivially link "the person who asked for the invoice" to "the person who claimed the tokens", which is exactly the linkability we are paying Tor to prevent. The wallet explicitly closes Circuit A before opening Circuit B. The only state that crosses the boundary is the opaque 128-bit nonce.

Each Lightning subscription produces three independent pieces of evidence that wallet makers can compare:

1. **Inner blob.** The wallet's encrypted attestation of how much the user actually paid.
2. **Outer blob.** The server's encrypted attestation of pack metadata and revenue share.
3. **Lightning receipt.** A real settlement on the wallet maker's Lightning node.

A wallet maker scanning the board (Section 11) computes the shared ECDH secret with both the server-side and wallet-side ephemeral keys, decrypts both layers, and verifies that the inner amount, the outer revenue share, and the Lightning receipt all line up. Any discrepancy is a cryptographic indictment.

7 Block Indexes

Per-block lookups are served as compact block indexes containing all the block information that can be queried. A typical block of 4,500 outputs produces an index of roughly 5 KB.

For every spendable transaction in a block, we form a 36-byte tuple $t = \mathrm{TXID}$ and assign it a data blob that the wallet will be interested in querying. The key for each entry is derived from the server's OPRF evaluation on the outpoint, then stretched with HKDF:

$$t = TXID \qquad \gamma_t = v \cdot H2C(t), \kappa_t = \mathrm{HKDF}(\gamma_t \,\|\, \text{“data-tag”})$$

The plaintext is a single little-endian bit tag ID. We encrypt it with **AES-256-CTR** under a per-(block, txid) deterministic 96-bit nonce derived from SipHash over the block hash K_h:

$$\text{nonce} = \text{Trunc}_{96}\left(\text{SipHash-2-4}_{K_h}(t)\right)$$

$$C_t = \text{AES-CTR}_{\kappa_t,\,\text{nonce}}(tag_t)$$

The block index for each block contains the ID of every transaction in that block together with that transaction's encrypted data. Each entry is encrypted under AES with a unique 256-bit key derived from the OPRF evaluation γ = v H2C(txid), so decrypting any single entry requires spending a token on that specific transaction.

8 Secure Queries

The wallet downloads block indexes the same way a BIP-158 light client does. Each encrypted blob of data attached to a TXID has a unique encryption key. The only remaining question is how to obtain the per-entry decryption key without leaking the transaction your interested in.

The answer is the same VOPRF that powered the issuance handshake, used now for a single online evaluation. The wallet picks a fresh blinding scalar ρ_t and sends a blinded query point:

$$X_t = \rho_t \cdot H2C(\text{TXID} \parallel \text{vout})$$

The server verifies the wallet's zk-SNARK proof (the same proof described in Section 9), records the public nullifier so this token can never be spent again, then evaluates the OPRF on the query point:

$$Y_t = v \cdot X_t$$

The wallet unblinds:

$$\gamma_t = \rho_t^{-1} \cdot Y_t = v \cdot H2C(\text{TXID} \parallel \text{vout})$$

and derives $\kappa_t = \text{HKDF}(\gamma_t \parallel \text{“tag”})$, exactly the key used by the server in Section 7.2, to decrypt C_t. If the resulting 16-bit value is not in the wallet's local copy of the tag registry, the entry was a collision false positive and the wallet keeps scanning.

There is only one OPRF point on the wire (X_t). The zk-SNARK itself is the authentication, the OPRF is the lookup. The server never learns which TXID is being checked, even if it collates every request, because the blinded query points are computationally unlinkable to their inputs under the discrete-log assumption. Even when the same outpoint is queried twice across spends, a fresh ρ_t makes the two blinded points statistically independent.

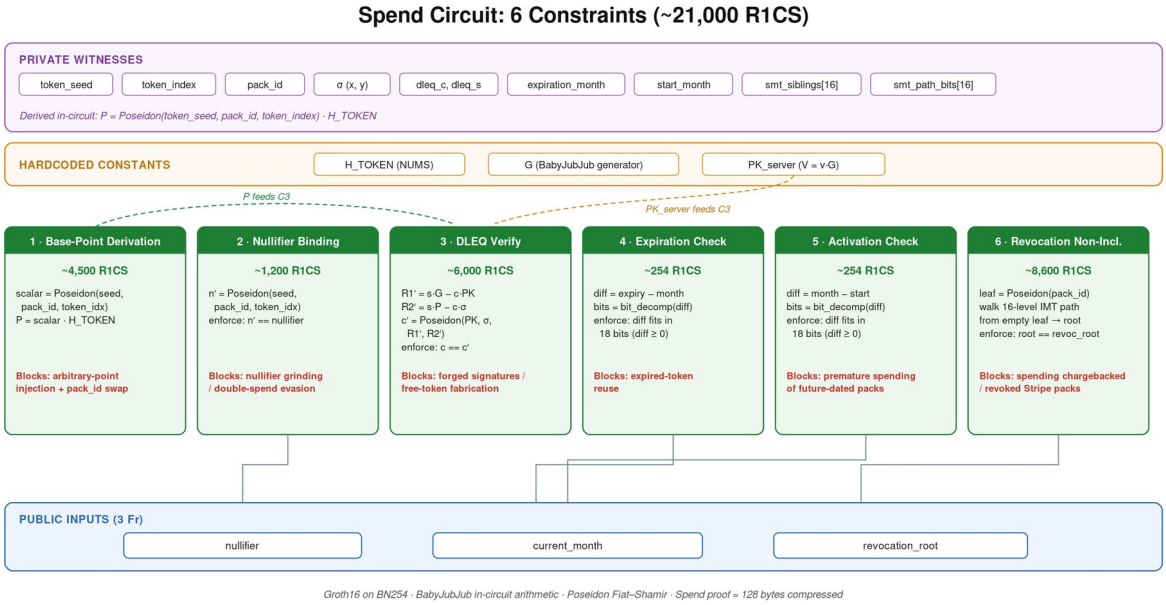
9 Zero-Knowledge Proofs

A zk-SNARK is a cryptographic proof system that lets one party (the prover) demonstrate to another (the verifier) that a computation was carried out correctly *without* revealing any of the secret inputs or intermediate values. In our case the prover is the user's wallet, and the statement being proved is, in plain English: “I hold a valid, unexpired, non-revoked token that this very server signed at some point in the past, and I am about to burn it.” Crucially, the server learns *only* the fact that this statement is true, plus the public nullifier that prevents the same token from being spent twice.

Under the hood the program is compiled into a logical circuit composed of tens of thousands of AND/OR/XOR/NOR gates, with a private input wire per secret and a public input wire per shared value. Each gate is then re-expressed as a quadratic equation over the scalar field of the curve, producing a **Rank-1 Constraint System** of equations. The prover's job is to find an assignment of values to

every wire that satisfies every equation simultaneously. The SNARK then compresses that assignment into a 96-byte proof π consisting of just three curve points. The verifier accepts π if and only if a small pairing check holds, they never have to re-run the circuit. Further details of how SNARKS build and verify π are outside the scope of this whitepaper.

Each constraint is load-bearing: removing any one opens a concrete, exploitable attack. The diagram below shows how the private witnesses on top flow through the six constraint boxes and what they ultimately commit to in the four public inputs along the bottom.



To understand why the SNARK is so important let's consider first constraint 3. This is the heart of the circuit and the reason the wallet is able to privately validate itself. The circuit recomputes the verifier's side of the Chaum-Pedersen check using Poseidon as the Fiat-Shamir hash (SHA-256 inside R1CS would cost ~25,000 constraints all on its own):

$$R_1' = s \cdot G - c \cdot PK_{\text{server}}, R_2' = s \cdot P_i - c \cdot \sigma_i$$

$$c' = \text{Poseidon}\left(PK_{\text{server}}, \sigma_i, R_1', R_2'\right), \text{enforce: } c = c'$$

Without this constraint σ_i is just a private witness, the prover could set it to any point and fabricate unlimited tokens with no server interaction at all. With it, every valid spend cryptographically demonstrates that the burned token was signed under the server's key v, while still not revealing anything about which token it was.

However, the circuit still lacks a method to meter the users queries since there is nothing preventing the wallet from using a single signed token repeatedly. To enforce a limit, we introduce the use of a nullifier.

$$\text{nullifier}' = \text{Poseidon}\left(token\_seed, pack\_id, token\_index\right), \text{enforce: nullifier}' = \text{nullifier}$$

The public nullifier must equal a Poseidon hash of the private $\left(token\_seed, pack\_id, token\_index\right)$ triple. Without this constraint, a prover could submit a fresh random nullifier on every spend of the same token and double-spend indefinitely. With it, every token produces exactly one possible nullifier, and Poseidon's preimage-resistance means the server still learns nothing about the witnesses behind that nullifier.

The circuit derives the token's base point in zero knowledge, never accepting it as a free input:

$$P_i = \text{Poseidon}(token\_seed, pack\_id, token\_index) \cdot H_{\text{TOKEN}}$$

H_{TOKEN} is a "Nothing Up My Sleeve" generator whose discrete log relative to G is unknown to everyone. If P_i were a free witness, a malicious prover could pick a scalar k, set $P_i = k \cdot G$, and synthesise a fake $\sigma_i = k \cdot PK_{\text{server}}$ that satisfies the DLEQ algebra without the server ever having signed the token. The hash-to-curve derivation closes that door and folding $pack\_id$ into the hash means the prover cannot quietly substitute a different (non-revoked) pack_id later in the proof.

The pack_id is needed to deal with the edge case of a credit card chargeback. If a user pays for a year subscription upfront, they are given the future token packs upfront as well. If we receive a chargeback we retain the ability to cancel retroactively the already issued future token packs.

The Indexed Merkle revocation tree from Section 5.3 is consumed here. The circuit hashes the wallet's $pack\_id$ to a leaf position and walks the 16-level Merkle path:

$$\text{leaf\_key} = \text{Poseidon}(pack\_id), \text{current} = 0$$

$$\text{for } l = 0..11 : \text{current} = \begin{cases} \text{Poseidon}(\text{current}, \text{sibling}_l) & \text{if bit}_l = 0 \\ \text{Poseidon}(\text{sibling}_l, \text{current}) & \text{otherwise} \end{cases}$$

$$\text{enforce: current} = \text{revocation\_root}$$

Starting from the *empty* leaf value (zero) means the prover can only complete the walk if their $pack\_id$'s leaf is indeed empty, i.e., the pack has not been revoked. If the server has inserted a non-zero value at that leaf, the path does not reach the published root, and no valid proof exists. The server checks that the revocation_root in the proof matches its currently published root, blocking replay of stale roots.

Leaf positions are derived from the low 16 bits of Poseidon(pack_id), so two packs can in principle map to the same slot. A slot collision never weakens soundness. A revoked slot stays revoked but it could strand an innocent pack behind another pack's revocation. Slot uniqueness is therefore enforced at issuance. If a freshly drawn pack_id maps to an already-occupied slot, the wallet draws a new pack_id before the pack is blinded and signed. With 65,536 slots this re-draw is rare and costs one random number.

To avoid the stockpiling of unused tokens and possibly the re-selling of them from other wallets, the SNARK also includes current month and expiration_date in the circuit so that tokens eventually expire.

$$\text{diff} = \text{expiration\_month} - \text{current\_month}, \text{bits} = \text{bit\_decompose}(\text{diff}, 254)$$

We enforce $\text{expiration\_month} \geq \text{current\_month}$ by proving the difference is non-negative, i.e., fits inside 254 bits without wrapping around in the scalar field. An expired token forces diff to wrap to a value near 2^{254} and the bit decomposition fails. The expiration month itself stays private, so the server cannot bucket users by age of their subscription; only the *fact* of validity is revealed.

The activation check is the mirror image of the expiration check. Because an annual subscriber receives all twelve monthly packs upfront, nothing would otherwise stop them from spending a December pack in June. Each pack therefore carries a private start_month, and the circuit enforces the same non-negativity trick in the other direction:

$$\text{diff} = \text{current\_month} - \text{start\_month}, \text{bits} = \text{bit\_decompose}(\text{diff}, 254)$$

A future-dated pack forces diff to wrap around the scalar field and the bit decomposition fails, so a pack becomes spendable only once its start month arrives. As with expiration, the start month itself

stays private: the server learns only that the pack is active right now, never how much of the subscription lies ahead.

The proof itself is exactly three curve points (96 bytes). The server's verification is a single pairing check that runs in milliseconds. Everything the user cares about, their seed, their pack ID, their expiration date, the specific token they burned, stays private.

10 Anonymizing Block Index Selection

Block indexes are excellent for light-client privacy because *every* block produces one. In contrast, a BTC Medusa wallet only downloads indexes when the user wants their privacy score, which on its own would leak a focal point: the server (or any passive observer) could see that the wallet requested block 845,123 and infer that something interesting lives in that block. Since our indexes are small (Section 7), it is more than affordable for even a mobile client to download a handful of decoys alongside the real target.

We use a **discrete-Laplace** (symmetric geometric) distribution to pick decoy block heights around the real target. Discrete-Laplace puts most decoys near the target, where they are most plausible, while still sprinkling a few farther away. The probability mass function is

$$P(\text{offset}=k) \propto q^{|k|} \text{ with } q=e^{-1/s}$$

where s is a scale parameter under the wallet's control.

Let

- h_0 be the block height the wallet genuinely needs,
- m the number of camouflage blocks the wallet also downloads,
- $s>0$ the scale parameter,
- $H_{\min}$ the lowest height the node is willing to serve.

We want the set

$$H=\{h_0\} \cup \{h_0+\Delta_j\}_{j=1}^{m}$$

to be indistinguishable from a uniformly random sample as seen by the server.

1. **Discrete-Laplace offsets.** Define $q=e^{-1/s}$ with $0<q<1$ and the PMF

$$\Pr[\Delta=k]=\frac{1-q}{1+q}\cdot q^{|k|}, k \in \mathbb{Z}$$

   The distribution is symmetric (mean 0), has variance $2q/(1-q)^2$, and 95% of samples lie in $[-\lceil s\ln 20\rceil, \lceil s\ln 20\rceil]$.

2. **Sampling procedure.** For each $j=1,\ldots,m$ draw a magnitude $k_j \geq 0$ with $\Pr[k_j=k]=(1-q)q^k$, a sign $\sigma_j \in \{-1,+1\}$ uniformly, and set $\Delta_j=\sigma_j k_j$. Reject and resample if $h_0+\Delta_j<H_{\min}$ or duplicates an existing entry. Stop when $|H|=m+1$ and sort to canonical order before transmission.

3. **Parameters and intuition.** Scale $s = 2$ keeps 95% of decoys within ±6 blocks; $s = 5$ widens that to ±15. Choosing $m = 8$ multiplies an exhaustive de-anonymisation by 9 (entropy $\log_2(m+1)$). Larger values cost more bandwidth but buy more anonymity.

4. **Download phase.** The wallet requests every $h \in H$ in a single batch (or in parallel) over its current Tor circuit, decrypts locally, and discards everything except the index for h_0. Because every request in the batch is indistinguishable on the wire, the server obtains only the distribution of decoys and no information about which member of H was the real target.

11 Attribution Board & Revenue Verification

Wallet makers who ship the BTC Medusa integration deserve to be paid their fair share of revenue, and they should not have to take the servers word for the totals. The **attribution board** is a public, append-only Merkle tree that records one leaf per issued token pack and lets any wallet maker independently verify the revenue they are owed without learning anything about the user behind each entry. The board is purely a transparency and revenue layer. Integrity of the tokens themselves is enforced by the in-circuit DLEQ check described in Section 9.

11.1 Board Structure

Each leaf is a Poseidon hash over four logical fields:

$$\text{leaf} = \text{Poseidon}\left(R_x, R_y, \text{tag}, H(\text{encrypted\_payload})\right)$$

where

- $R = r \cdot G$ is the server's ECDH ephemeral point for this entry,
- $\text{tag} = H(\text{“wallet\_attr”} \,\|\, ss)$ is a deterministic identifier the wallet maker uses to spot their leaves without trial-decrypting everything, with $ss = r \cdot P_{\text{dev}}$ the shared secret with the wallet maker's public key, and
- encrypted_payload is the dual-layer ECDH capsule described below.

No cleartext metadata, pack ID, expiration month, or revenue amount, is exposed in the leaf. An external observer sees only opaque curve points and ciphertext.

11.2 Dual-Layer ECDH Attestation

The attribution payload uses two nested ECDH layers so that the server cannot lie about how much the user paid. The **inner layer** is created by the wallet software:

$$r_w \xleftarrow{\$} F_q, R_w = r_w \cdot G, ss_w = r_w \cdot P_{\text{dev}}$$

$$\text{inner\_blob} = \text{AES}\left(ss_w, \{amount\_paid\}\right)$$

The wallet encrypts the *actual* amount paid (in satoshis) under the wallet maker's public key P_{dev}. The server cannot decrypt or modify this blob because it does not know x_{dev}. The wallet submits $(R_w, \text{inner\_blob})$ as part of the issuance request.

The server then creates the **outer layer** with its own ephemeral key r, and wraps the wallet's inner blob along with server-side data:

$$\text{outer\_payload} = \{\, pack\_id, \text{exp}, revenue\_amount, R_w, \text{inner\_blob} \,\}$$

$$\text{enc\_payload} = \text{AES}(ss, \text{outer\_payload})$$

The leaf is computed and appended; the wallet receives a Merkle inclusion proof and refuses to accept the issued pack unless that proof is valid. This refusal is the **anti-cheating gate**: the server literally cannot issue tokens without writing an attributable entry.

11.3 Verification

A wallet maker scans the board at any cadence they like and walks each leaf:

1. Compute $ss = x_{\text{dev}} \cdot R$ and re-derive $\text{tag}' = H(\text{“wallet\_attr”} \,\|\, ss)$. If $\text{tag}' \neq \text{tag}$, skip, this leaf belongs to a different wallet maker.

2. Otherwise decrypt the outer layer with ss to recover $(pack\_id, \text{exp}, revenue\_amount, R_w, \text{inner\_blob})$.

3. Decrypt the inner layer with $s s_w = x_{\text{dev}} \cdot R_w$ to recover the wallet-attested $amount\_paid$.

4. Cross-check: $revenue\_amount$ must equal $amount\_paid \cdot \text{split\_rate}$ given the agreed deal terms. For Lightning subscriptions, also confirm that a matching incoming Lightning payment arrived within roughly ten seconds of the board timestamp. For Stripe, confirm against the published exchange rate.

If any of those checks fail the wallet maker holds cryptographic evidence of dishonesty.

11.4 The Revocation Tree

The Indexed Merkle revocation tree from Section 5.3 is published alongside the attribution board so that any client (and any independent auditor) can fetch the current root and verify chargeback enforcement. The two structures sit side by side but never get conflated: the attribution board is **strictly append-only** because removing entries would let the server hide subscriptions from wallet makers; the revocation tree is mutable by design because the server *must* be able to insert new revocations. Keeping them separate preserves both invariants.

The wallet fetches revocation/root before every spend, and Constraint 6 of the spend circuit (Section 9) refuses to verify against any other root.

12 Privacy Transport Layer

Even a perfect cryptographic stack leaks one piece of information by default: the client's IP address. A server that sees every request's IP can correlate issuance with spends, geolocate users, build behavioural profiles, and become a single subpoena away from de-anonymising everything. We therefore treat the transport layer as a first-class part of the protocol rather than an implementation detail.

12.1 Tor (Default), Native Hidden Service

Our server runs as a **native Tor hidden service** with no clearnet endpoint. The client uses an

embedded Arti circuit to reach the .onion address, which means traffic never leaves the Tor network, there is no exit node, and the most common Tor vulnerability simply does not apply. Both ends of the conversation are behind onion routing, so mutual anonymity is the default.

12.2 OHTTP (Fallback)

Some networks block Tor entirely. As a fallback we support **Oblivious HTTP** (RFC 9458): the client encrypts its request to a relay, the relay strips the outer encryption and forwards the inner request to us. The relay sees the client's IP but not the request content; the server sees the relay's IP but not the client's. The trust assumption, that the relay does not collude with the server, is explicit and the relay URL is user-configurable, so a privacy-conscious user can pick a third-party relay they trust.

12.3 Direct (Development Only)

Standard HTTPS exists only for local development. The wallet UI shows an orange-shield status indicator whenever this mode is active so that users cannot accidentally ship a build that talks to us directly. Green shield means Tor or OHTTP; orange means the IP is exposed.

12.4 Tor Circuit Lifecycle

Several steps in the protocol are critical for unlinkability, and we always close one Tor circuit and open another between them. Reusing a single circuit across those boundaries would defeat much of the privacy work done elsewhere in the stack. The full lifecycle for a Lightning subscription, the most circuit-intensive flow we support, looks like this:

These boundaries are encoded directly in the wallet plugin, the user does not have to remember to rotate circuits. Combined with the cryptographic guarantees of the earlier sections, they yield a system in which the server's view of any single user is reduced to: "some Tor circuit submitted a valid proof for a token it never identified, and queried a UTXO whose identity it cannot recover."

Conclusion

Privacy is not merely an optional feature in Bitcoin, it is essential to preserving the network's integrity and the financial sovereignty of its users. As regulatory pressures mount, solutions that reconcile privacy with usability and compliance will be critical to sustain Bitcoin's foundational principles.

Our token-gated, zero-knowledge-enhanced protocol offers a clear path forward. By empowering wallets to perform private, verifiable queries against sensitive KYC-related data, without ever exposing transactions, payment identities, or query patterns, we provide a blueprint for how Bitcoin applications can serve both their users and the broader ethos of the network. The combination of a six-constraint Groth16 spend circuit, a compact mobile-friendly block-index format, an unbatched per-token DLEQ chain, dual-layer ECDH attribution that holds the service provider accountable, and a Tor-first transport gives users a complete, auditable privacy stack.

We welcome feedback, collaboration, and contributions as we move toward productionizing these designs. Together, we can build tools that allow self-custody, privacy, and open innovation to thrive in the next era of Bitcoin's development.