Shielded Bitcoin

Private Transfers on the Bitcoin L1

Clara Shikhelman
[[alloc] init]
clara@allocinit.xyz

Mikhail Komarov
[[alloc] init]
nemo@allocinit.xyz

Aleksei Moskvin
[[alloc] init]
zerg@allocinit.xyz

Abstract

This paper introduces Shielded Bitcoin, a protocol for private transfer of Bitcoin directly on the Bitcoin L1, requiring no changes to Bitcoin consensus. Its design borrows from the architecture of Zcash and earlier shielded-note systems: value is held as encrypted notes, spends are marked by public nullifiers, and validity is established by zero-knowledge proofs. Unlike Zcash, Shielded Bitcoin introduces no dedicated consensus chain: all protocol data is published on Bitcoin, transfer validity is enforced through PIPEs v2 [ACG+26], and shielded state is reconstructed by deterministic replay of accepted transfers in Bitcoin order. At the metaprotocol transfer level, Shielded Bitcoin is non-custodial: spending authority remains with user-controlled wallet keys encrypted in a ciphertext, and neither indexers nor the replay layer receive authority to spend user notes.

A shielded transfer does not publicly reveal its amount, sender, receiver, and place in the transfer graph, while keeping the validity of every encrypted state transition publicly verifiable. Shielded state, comprising the note tree, nullifier set, and block-level root history, is maintained entirely off-chain: any two correct implementations replaying the same Bitcoin history derive identical protocol state without coordination. The paper defines the transfer model, note structure, ciphertext-derived note leaves, nullifier rules, transaction binding, replay semantics, and publication assumptions. Peg-in and peg-out, the mechanisms by which value enters and exits the shielded transfer system, belong to the broader system and fall outside the scope of this paper.

Contents

I Introduction 3

1 Introduction 3

2 Related Work 4

II Model 6

3 Model Overview 6

4 System Architecture 8

5 Note Model 9

6 Key and Address Model 10

7 Disclosure and Audit Model 11

III Protocol 13

8 Replayed Shielded State 13

9 Note Leaves and Nullifiers 15

10 Transfer Pipeline 16

11 Publication Layer 17

12 Output Encryption 18

13 Transfer Construction and Binding 20

14 Transfer Statement 22

15 Replay and Acceptance 23

16 Recovery and Transfer Lifecycle 23

IV Analysis 25

17 Assumptions 26

18 Security Goals 27

19 Security 28

20 Privacy Model 31

V Closing Remarks 37

21 Summary 38

22 Future Work 38

A Implementation Profile and Recommended Defaults 42

B Transfer Constraint Decomposition 46

C Appendix: Alice-to-Bob Transfer Walkthrough 48

D Opt-in KYC Evidence: Certified-Recipient Lineage 53

References 55

Part I
Introduction

1 Introduction

Bitcoin's transaction graph is permanently public. Although addresses are pseudonymous, transaction graph analysis, address clustering, and chain metadata can often be combined to infer ownership in practice [MPJ+16]. Existing systems address parts of this problem. Privacy-focused cryptocurrencies provide confidential transfers, but run on networks separate from Bitcoin. On Bitcoin, CoinJoin [Max13, MO15] and PayJoin [D+19] make common transaction-graph heuristics less reliable, while Silent Payments [SB24] avoid address reuse. Amounts and transaction structure nevertheless remain public. More expressive constructions may be possible with proposed opcodes such as OP_CAT and OP_CSFS, but those constructions depend on changes to Bitcoin's consensus rules.

This paper introduces Shielded Bitcoin, a shielded-note transfer metaprotocol whose envelope data is designed for publication on Bitcoin L1 today, without any consensus changes. Its design builds on the shielded-note architecture used in Zcash and earlier systems [BSCG+14, HBHW25], with encrypted notes, public nullifiers, and zero-knowledge transfer proofs. It departs from that lineage in one structural way - it introduces no blockchain of its own. Transfer-envelope data is published on Bitcoin L1 and a deployment-supplied shielded state is advanced by deterministically replaying accepted transfers, so Bitcoin serves only as a neutral publication and ordering layer.

Each shielded note encodes a satoshi amount and recipient addressing material, never appearing on-chain in plaintext. To transfer, the sender publishes a transfer envelope to Bitcoin carrying recipient ciphertexts, public spend markers called nullifiers, and a succinct zero-knowledge proof attesting to note membership, spend authorization, and value conservation. Shielded state, comprising a global note tree and nullifier set, is not maintained by Bitcoin consensus but is reconstructed by any implementation that deterministically replays accepted envelopes in Bitcoin order. Peg-in and peg-out, the mechanisms by which value enters and exits the shielded transfer system, are enabled by PIPEs v2 [ACG+26] and are outside the scope of this paper. Two correct implementations using the same completed deployment profile, well-formed initial state and declared note inventory, and the same active-chain prefix for which replay is defined derive the same state. Under those assumptions, the design keeps confidential note values and does not directly identify which earlier leaves were spent. Transfer timing, arity, output grouping, fees, and carrier-transaction metadata remain public.

1.1 Contribution

This paper presents the design, specification, and analysis of Shielded Bitcoin. The contributions are as follows.

- **Transfer model.** We define a note-based private transfer model whose envelopes are designed for publication on Bitcoin L1 requiring no consensus changes. A note encodes a satoshi amount and recipient addressing material and never appears on-chain in plaintext. We specify the note format, the ciphertext-derived note leaf derivation, nullifier construction binding the spend marker to the replay-assigned tree position, and the sliding block-level anchor window that decouples proof construction from the live chain tip.
- **Protocol specification.** We give a logical protocol specification covering transfer envelope construction and canonical serialization, envelope binding via a commitment over the full transfer body, replay semantics and indexer acceptance rules, output encryption with sender-side recovery, and the transfer proof statement. The concrete choices still needed for interoperable implementations are listed explicitly.
- **Replay-based shielded state.** We introduce an architecture in which the global note tree, nullifier set, and block-level root history are advanced from the profile's initial state through deterministic replay of accepted Bitcoin envelopes above the base chain. Bitcoin provides only publication and ordering. Any two correct implementations using the same completed profile, activation state and declared inventory, and the same active Bitcoin prefix in the replay domain derive the same protocol state without coordination.

- **Key hierarchy and wallet model.** We specify a key-derivation chain separating spend authority, incoming viewing, outgoing recovery, and nullifier derivation from a single wallet seed, together with diversified payment addresses and a wallet recovery model covering post-activation outputs and initial notes supplied with deployment-defined bootstrap recovery data.
- **Security analysis.** For a well-formed deployment-supplied initial state and a fixed Bitcoin prefix in the replay domain we analyze double-spend resistance, value conservation, spend authorization, recovery integrity, replay determinism, malleability resistance, and reorganization handling under the stated trust model.
- **Privacy analysis.** We give an explicit leakage model for an honestly generated accepted transfer and identify the primary residual leakage surfaces: transfer arity, anchor age, envelope grouping, and fee wallet linkage.
- **Disclosure model.** We define a scoped disclosure model through user-held capabilities (incoming viewing, outgoing recovery, selective note disclosure, and derived reports) with a formal boundary establishing that no disclosure path grants spend authority. These read capabilities are not, by themselves, proofs of ownership, completeness, or balance.

1.2 Scope

This specification covers only the transfer component of Shielded Bitcoin: private transfer inside a shielded Bitcoin-denominated protocol. It covers one Bitcoin-denominated asset, transfers after entry into the metaprotocol, one global note tree, one canonical binary encoding, and one transfer-proof statement. Transfer relies on notes, ciphertext-derived note leaves, nullifiers, encrypted outputs, proofs, and replayed state derived from published envelopes. Each transfer references replayed note-tree state compactly with h_{anchor}, a Bitcoin block height whose block-level root is reconstructed by indexers. The transfer protocol is non-custodial by design. Users retain spending authority through wallet-controlled keys; protocol viewing and indexing functions are read-only with respect to user funds. Peg-in and peg-out are separate boundary components and are not covered by this transfer-layer non-custodial claim.

`Peg-in` and `peg-out` are outside the scope of this paper. At a high level, a peg-in is the process by which Bitcoin liquidity is committed on Bitcoin L1 and represented as spendable notes inside the metaprotocol, while a peg-out is the corresponding exit from shielded notes back to ordinary Bitcoin ownership on L1. Those flows may be built with PIPE v2 [ACG+26], but they are separate components with their own trust, operational, and protocol constraints and belong in a dedicated paper. We therefore treat them as future work and discuss the required bridge and unlock mechanisms separately in Section 22.

1.3 Paper structure

The paper is organized into five parts. Part I closes with Section 2, which surveys related privacy techniques across Bitcoin-native overlays, dedicated-consensus privacy chains, client-side validation systems, and bridged execution environments, and positions Shielded Bitcoin within that landscape. Part II (Sections 3-6) establishes the model: notation and the protocol object vocabulary, system architecture and the role of deterministic replay, the note structure and lifecycle, and the key derivation and address model. Part III (Sections 8–16) gives the logical protocol specification, covering replayed shielded state and the sliding anchor window, note leaves and nullifiers, the publication layer, output encryption and wallet recovery, transfer construction and envelope binding, the proof system role, replay semantics, and the recovery and transfer lifecycle. Part IV (Sections 17–20) presents the analysis. Sections 17 and 18 state the assumptions, security goals, and trust model. Section 19 analyzes the security properties of the transfer protocol. Section 20 analyzes the privacy model. Part V (Sections 21–22) summarizes the work and identifies open questions and future directions.

2 Related Work

Privacy protocols for public ledgers must allow transfers to be verified independently without revealing their participants or amounts. Bitcoin demonstrated that decentralized consensus can maintain a shared ledger without trusted intermediaries, but it also made transaction history publicly observable. Although addresses are pseudonymous, transaction graph analysis, address clustering, and auxiliary metadata often

allow ownership relationships to be inferred in practice [MPJ+16]. This motivated several lines of work that trade differently between confidentiality, changes to Bitcoin, and how protocol state is made available.

Privacy within the Bitcoin UTXO model. Bitcoin-native privacy techniques such as CoinJoin [Max13, MO15, SSH+22], PayJoin [D+19, G+24], address rotation, and Silent Payments [SB24] improve privacy within the ordinary UTXO model, but they do not change the fact that Bitcoin transactions are transparent. Amounts, timing, fees, input and output structure, and much of the transaction graph remain visible to chain observers. These techniques can make ownership clustering harder, but they do not provide a shielded transfer model in which sender, recipient, amount, and note plaintext are kept confidential inside a common encrypted state. Shielded Bitcoin therefore addresses a different layer of the problem: it moves transfer activity from ordinary Bitcoin spends into encrypted note-state updates, while still acknowledging that Bitcoin-visible metadata such as publication time, carrier transaction structure, fees, and boundary events remain public.

Shielded cryptocurrencies with dedicated consensus. Zerocoin pioneered privacy-preserving mint and spend operations, allowing users to break the public transaction graph while preserving public verification of coin validity [MGGR13]. This proposal later was generalized by Zerocash into a fully shielded payment system based on zero-knowledge proofs, keeping confidential sender, recipient, and amount while still allowing public verification of transaction correctness [BSCG+14]. Zcash later instantiated shielded transfers in its Sapling and Orchard protocol families [HBHW25, Ele20], with Sapling adopting the Groth16 proof system [Gro16].

In Zcash, shielded validity is part of the native consensus rules of the chain.

Another branch of privacy-focused cryptocurrency design is represented by Monero chain. Instead of shielded notes and succinct zero-knowledge proofs, it combines ring signatures, stealth addresses, and confidential transaction techniques to protect the confidentiality of transaction participants and values [Noe15]. This design is important background for private cryptocurrency systems, but it follows a validation model enforced by the native consensus rules of its own chain, whereas Shielded Bitcoin keeps Bitcoin consensus unchanged and derives shielded state from Bitcoin-published transfer envelopes.

Related, but broader direction is Aztec [Azt26, Azt23]: programmable privacy, where zero-knowledge proofs are used not only for private payments but also for private smart-contract execution and private application state. General private execution environment is very crucial direction, meanwhile it's very complex compared with private transfers.

Bitcoin-anchored overlays with client-side data. A closer family keeps Bitcoin as the base chain and interprets richer semantics above it. Shielded CSV is the closest recent work in the Bitcoin-specific design space [NEL25]. It also aims to provide private Bitcoin-denominated transfers without requiring a Bitcoin soft-fork, and it builds on the client-side-validation tradition using proof-carrying data. At present, Shielded CSV should be understood as a theoretical protocol proposal rather than a deployed Bitcoin system. Its use for Bitcoin-denominated value also depends on bridging mechanism to the base chain, with BitVM-style constructions [Lin23]. A key trade-off in Shielded CSV is its client-side data-availability model. The protocol does not publish all information needed to reconstruct the private system state from Bitcoin history alone. Instead, the coin owner must retain the client-side state and proof data required to validate and later spend the coin. If this local state is lost, the private transaction history is not generally recoverable from the public chain alone.

Client-side validation systems such as RGB and Taproot Assets demonstrate a method in which Bitcoin is used primarily as a publication, ordering, or commitment layer, while asset-specific semantics are interpreted outside Bitcoin consensus [LNP, Lig]. In these systems, ownership and transfer validity are established by presenting the relevant off-chain history and proofs to the recipient. This separation shifts validation and data-availability responsibilities to the asset protocol and its users.

Bridged execution environments. A separate class of systems moves execution into a bridged environment, including federated sidechains and EVM-compatible networks, where the asset can be represented and transferred under a different execution model. BitVM-style mechanisms explore how stronger verification or dispute systems can be built around Bitcoin without changing its consensus rules [Lin23, LAA+25, WAA+26]. These systems can support substantially richer semantics than Bitcoin

L1, but their connection to Bitcoin is mediated by the bridge and therefore inherits the bridge's trust, custody, and liveness assumptions. In this sense, their relationship to Bitcoin is primarily at a separate bridge boundary, rather than through continuous derivation of higher-layer state from Bitcoin's own publication and ordering.

Positioning. Shielded Bitcoin builds on the shielded-note architecture used in Zcash and earlier systems, combining notes, encrypted outputs, public nullifiers, zero-knowledge transfer proofs, and a shared note tree reconstructed by deterministic replay of Bitcoin-published transfers.

However, unlike Zcash, Shielded Bitcoin does not introduce its own blockchain or consensus protocol. Instead, transfer-envelope data is published on Bitcoin, and shielded state is reconstructed by deterministic replay of accepted transfers in Bitcoin order, with transfer validity established by the zero-knowledge proofs verified during that replay.

Deterministic replay over canonical Bitcoin history is what makes this possible: every correct implementation reconstructs the same shielded state from the same published transfers, so divergent private states cannot arise and the transfer layer needs no challenge, dispute, or optimistic-rollback mechanism. PIPEs [ACG+26] enter only at the peg-in/peg-out boundary, where value crosses between ordinary Bitcoin and the Shielded Bitcoin.

Shielded Bitcoin explores a different point in the design space from both dedicated-consensus privacy chains and client-side validation systems. It maintains a shared replayable shielded state derived from Bitcoin history.

Part II
Model

3 Model Overview

Shielded Bitcoin represents private value as notes. A note encodes a satoshi amount and recipient addressing material and never appears on-chain in plaintext. To transfer value, a sender publishes a transfer envelope to Bitcoin carrying recipient ciphertexts, nullifiers for the consumed notes, and a zero-knowledge proof attesting to note membership, spend authorization, and value conservation. Bitcoin records and orders these envelopes but performs no shielded validation. Shielded state, the global note tree, nullifier set, and block-level root history, is not maintained by Bitcoin consensus. Instead, any correct implementation that deterministically replays accepted envelopes in Bitcoin block order reconstructs the same state. Sender, recipient, and amount remain confidential inside the metaprotocol, while Bitcoin-visible metadata, publication time, transfer arity, fee behavior, and boundary events, remains public.

Notes and per-note values.

A note is the private object that represents shielded ownership of value. The objects in this group describe what is encrypted for the recipient, what is derived from the note randomness, and what public leaf is later inserted into the note tree.

$\mathsf{ct}^{\mathsf{note}}$ Recipient ciphertext for one output note, carrying the encrypted $\mathsf{pt}^{\mathsf{note}}$. Proof-level ciphertext/-plaintext consistency and wallet-side decryption checks establish receipt.

$\mathsf{ct}^{\mathsf{out}}$ Batched sender-recovery ciphertext under $\mathsf{vk}_{\mathsf{out}}$; plaintext is the ordered per-output list $(\mathsf{pk}_d,\ \mathsf{sk}_{\mathsf{eph}})_0, \ldots, (\mathsf{pk}_d,\ \mathsf{sk}_{\mathsf{eph}})_{m-1}$.

h_{aux} Optional hash commitment embedded in a note leaf, binding auxiliary protocol data to the note. This can be used, for example, to bind an atomic-swap hash-lock at note creation time without placing the auxiliary data itself in the note plaintext.

ℓ The replay-derived public leaf inserted into the global note tree for each created output note. It is derived from the published ciphertext data, the transfer binding hash, the output index, and the note's public h_{aux}.

v A canonical unsigned satoshi amount. Transfer circuits range-check input and output values and enforce value conservation without field wraparound.

pos Canonical note-tree index assigned to a ℓ when replay appends it to the note tree.

$\mathsf{pt}^{\mathsf{note}}$ Private note payload $(v,\ d,\ r_{\mathsf{seed}})$ encrypted into $\mathsf{ct}^{\mathsf{note}}$.

ρ, $\mathsf{sk}_{\mathsf{eph}}$, $\mathsf{pk}_{\mathsf{eph}}$ Per-note values derived from r_{seed}. ρ feeds nullifier derivation, $\mathsf{sk}_{\mathsf{eph}}$ is the note's ephemeral secret key, and $\mathsf{pk}_{\mathsf{eph}} = [\mathsf{sk}_{\mathsf{eph}}]G_d$ is the published ephemeral public key.

Replay state.

Bitcoin orders the published envelopes, but the shielded state is reconstructed outside Bitcoin consensus. Replay is the deterministic process that turns accepted envelopes into a note tree, a nullifier set, and a history of roots. The objects below describe the state maintained by an indexer and the roots used as spend anchors.

r_{anchor} The note-tree root locally derived by replay as $\mathcal{R}[h_{\mathsf{anchor}}]$. It is used as the Merkle root public input for proof verification but is not serialized in the transfer envelope.

h_{anchor} The Bitcoin block height, encoded as a `uint32` little-endian field in the transfer envelope, that selects the block-level root used by the spend proof.

`note tree` The global append-only Merkle tree of ℓ values maintained by deterministic replay.

`replay` The deterministic process by which indexers validate accepted envelopes in Bitcoin block order and update the note tree, nullifier set, positions, and block-level root history.

$\mathcal{R}[h]$ The note-tree root after replaying all accepted Shielded Bitcoin envelopes in Bitcoin block height `h`.

`valid-root window` The sliding range of recent Bitcoin block heights whose $\mathcal{R}$ values may be used as spend-proof anchors.

Spend objects.

Spending a note does not reveal the note itself. Instead, the transfer publishes a nullifier that marks the consumed note as spent, together with a proof that the nullifier was derived correctly.

nf Nullifier: the public spend marker derived when an existing note is consumed.

`nullifier set` The replay-maintained set of accepted nullifiers. A valid transfer has no duplicate nullifier internally and no nullifier already present in this set.

Transfer envelope and public data.

A transfer envelope is the Bitcoin-published object that carries the public data needed for verification and replay. Some fields are checked directly by the circuit, while others are used by replay to derive leaves, update state, and bind the proof to the exact serialized transfer.

`header` The fixed envelope discriminator `("btc", 1, 0x01)`, where `1` is the protocol version byte and `0x01` is the transfer type.

`public statement` The ordered public inputs verified by the transfer circuit: derived r_{anchor}, counts, nullifiers, public $\mathsf{pk}_{\mathsf{eph}}$ values, public $\mathsf{ct}^{\mathsf{note}}$ values, and h_{body}.

e Canonical binary object published to Bitcoin, carrying the selected h_{anchor}, counts, public nullifiers, per-output $\mathsf{pk}_{\mathsf{eph}}$ values, recipient ciphertexts, auxiliary hashes, sender-recovery ciphertext, and proof.

h_{body} The derived binding hash of the canonical transfer body that ties the proof to the exact published envelope.

w The private inputs used by the prover to reconstruct spent notes, authorize spends, derive nullifiers, construct output ciphertexts and note leaves, and prove Merkle membership.

Wallet keys and addresses.

Wallet keys are separated by capability. The address exposes only the receive-side public material needed by a sender to create an encrypted note.

$\mathsf{vk}_{\mathsf{in}}$ Incoming viewing capability used for note detection and recipient-side decryption attempts.

$\mathsf{sk}_{\mathsf{nf}}$ Nullifier secret derived from $\mathsf{sk}_{\mathsf{spend}}$ and used in nullifier derivation.

$\mathsf{vk}_{\mathsf{out}}$ Outgoing viewing capability used for sender-side recovery of sent notes.

addr_d The pair $(d,\ \mathsf{pk}_d)$. The short diversifier `d` identifies a receive path, and pk_d is the diversified transmission key used for note encryption.

G_d The diversified curve base `DiversifyHash(d)` for receive path `d`.

$\mathsf{sk}_{\mathsf{master}}$ Master secret derived from $\mathsf{seed}_{\mathsf{wallet}}$ and used as parent material for downstream wallet capabilities.

$\mathsf{sk}_{\mathsf{spend}}$ Spending secret that authorizes note spends in the transfer circuit.

$\mathsf{sk}_{\mathsf{view}}$ Scalar derived from $\mathsf{vk}_{\mathsf{in}}$ for diversified receive-key construction and incoming Diffie-Hellman.

$\mathsf{seed}_{\mathsf{wallet}}$ Root wallet secret from which the master secret and all wallet capabilities are derived.

4 System Architecture

Bitcoin records the bytes of each transfer envelope and its position in the chain. Shielded state transitions are validated entirely outside Bitcoin's consensus. **Indexers** watch the Bitcoin chain for Shielded Bitcoin transfers, replay accepted envelopes in block order, and maintain the note tree, nullifier set, and block-level root history. **Wallets** consume that replayed state to detect incoming notes, obtain Merkle witnesses, and construct spends.

Indexers are not custodians and do not hold spending authority. They reconstruct and serve replay state; a wallet may verify that state independently from Bitcoin-published data. Correct implementations that process the same accepted chain history must derive identical state. A dishonest indexer can censor envelopes or report an arbitrary root, but it cannot make that root consistent with an independent replay of the same Bitcoin history. Because replay state is derived from the active chain, a Bitcoin reorganization may invalidate previously computed roots for affected block heights. Any spend proof anchored to a root on the replaced chain must be rebuilt against the new history.

Figure 1 separates the three responsibilities. Bitcoin provides the ordered publication log. The shielded replay layer reads that log, invokes verifier logic, and maintains replayed state. Wallets publish envelopes and consume replayed state.

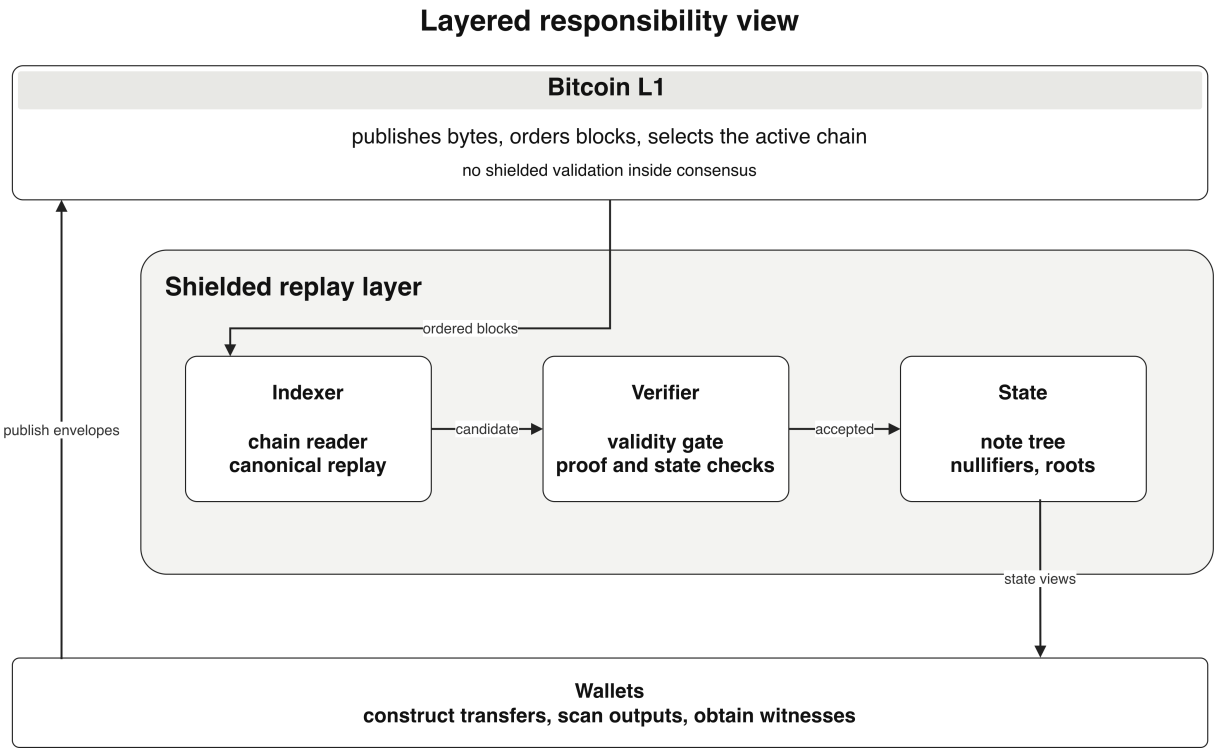

Figure 1: *Layered responsibility view for Shielded Bitcoin. Bitcoin provides publication and ordering only. The replay layer validates shielded transitions and maintains state above consensus.*

5 Note Model

The basic object of shielded ownership is a note. A note plays a role similar to a Bitcoin UTXO, in the sense that it represents a discrete piece of value that can later be spent by its owner. The difference is that the note is represented on the public ledger by ciphertext and associated public protocol data, rather than by its plaintext contents. Its plaintext contains the information needed to recognize and later spend it, such as the value, recipient-related key material, and note-specific randomness, but this information is not published directly. The note never appears on-chain in plaintext, it is represented by one recipient ciphertext, one published $\mathsf{pk}_{\mathsf{eph}}$, and the transfer's batched sender-recovery ciphertext. The note-tree leaf is derived by replay from those public output fields and the transfer's h_{body}.

A note is spent by proving three things at once: it exists in the global tree, the spender is authorized to spend it, and the public spend marker derived from it is correct. The note plaintext is very small. It contains only the fields that must be recovered from the encrypted output: the value, the receive-path diversifier, and one note seed. In this design, a note plaintext is the tuple

$$\mathsf{pt}^{\mathsf{note}} := (v,\ d,\ r_{\mathsf{seed}}).$$

Here v is a canonical unsigned satoshi amount, d identifies the diversified receive path, and r_{seed} is fresh note-specific randomness.

The seed r_{seed} is the source of the additional per-note values used in the protocol. Instead of storing those values as plaintext fields, the wallet and the circuit derive them deterministically from r_{seed}:

$$\rho := H_\rho(r_{\mathsf{seed}}), \qquad \mathsf{sk}_{\mathsf{eph}} := \mathsf{ScalarFromBytes}(H_{\mathsf{eph}}(r_{\mathsf{seed}})),$$

where H_ρ and H_{eph} are distinct domain-separated derivations fixed by the protocol. The value ρ is the note-specific input to nullifier derivation. The value $\mathsf{sk}_{\mathsf{eph}}$ is the note's ephemeral secret key for output encryption. Given the diversifier d, the corresponding public ephemeral key is

$$\mathsf{pk}_{\mathsf{eph}} := [\mathsf{sk}_{\mathsf{eph}}]G_d.$$

The recipient only needs to recover $(v, d, r_{\mathsf{seed}})$ from the recipient ciphertext. After decryption, the wallet re-derives ρ, $\mathsf{sk}_{\mathsf{eph}}$, and $\mathsf{pk}_{\mathsf{eph}}$, checks that the derived $\mathsf{pk}_{\mathsf{eph}}$ matches the public $\mathsf{pk}_{\mathsf{eph}}$ in the accepted

envelope, and then checks that the corresponding ℓ is the leaf assigned by replay. The transfer circuit performs the same derivations for the notes it proves about, so precomputed ρ, $\mathsf{sk}_{\mathsf{eph}}$, or note leaves are not independent witness values.

The note leaf also carries an h_{aux} field that is not part of the encrypted note plaintext and is not transmitted inside $\mathsf{ct}^{\mathsf{note}}$. It is provided by the sender at note creation time and bound into the public leaf derivation. A note with no auxiliary commitment uses the canonical null value for h_{aux}.

A note exists in one of three states: created, once its leaf has been appended to the note tree by replay; spent, once its nullifier has been published and accepted; or unconfirmed, if it appears only in an unaccepted or not-yet-replayed envelope. A note is not spendable until it has a confirmed position in the note tree.

The note plaintext carries only the fields the recipient cannot recover by other means: the value, the receive path identifier, and the random seed from which ρ, $\mathsf{sk}_{\mathsf{eph}}$, and $\mathsf{pk}_{\mathsf{eph}}$ are deterministically derived. All other per-note material is re-derivable from these three fields, so nothing else needs to be transmitted. Since the note is published to Bitcoin as an encrypted ciphertext, a smaller plaintext directly reduces the per-output on-chain cost.

6 Key and Address Model

The key hierarchy separates spend authority, incoming viewing, and outgoing recovery into distinct wallet capabilities. These capabilities are derived from a single seed using fixed, domain-separated `HKDF-SHA256` labels, but their disclosure meanings are different. Figure 2 shows the dependency graph. Table 1 fixes the corresponding derivation rules.

Let $\mathsf{seed}_{\mathsf{wallet}}$ be the wallet's seed entropy. The following objects are part of the protocol definition.

Object	Derivation rule	Capability role
$\mathsf{sk}_{\mathsf{master}}$	$\mathsf{HKDF}_{\mathsf{master}}(\mathsf{seed}_{\mathsf{wallet}})$	Root wallet secret.
$\mathsf{sk}_{\mathsf{spend}}$	$\mathsf{ScalarFromBytes}(\mathsf{HKDF}_{\mathsf{spend}}(\mathsf{sk}_{\mathsf{master}}))$	Spend authorization scalar.
$\mathsf{sk}^{\mathsf{ser}}_{\mathsf{spend}}$	$\mathsf{BytesFromScalar}(\mathsf{sk}_{\mathsf{spend}})$	Canonical bytes for downstream derivations.
$\mathsf{sk}_{\mathsf{nf}}$	$\mathsf{HKDF}_{\mathsf{nf}}(\mathsf{sk}^{\mathsf{ser}}_{\mathsf{spend}})$	Nullifier secret bound to spend authority.
$\mathsf{vk}_{\mathsf{in}}$	$\mathsf{HKDF}_{\mathsf{in}}(\mathsf{sk}^{\mathsf{ser}}_{\mathsf{spend}})$	Incoming note detection and decryption.
$\mathsf{vk}_{\mathsf{out}}$	$\mathsf{HKDF}_{\mathsf{out}}(\mathsf{sk}_{\mathsf{master}})$	Sender-side recovery of sent-output records.

Table 1: *Formal wallet capability derivation rules. Each* $\mathsf{HKDF}_\bullet$ *label is distinct and fixed by the protocol.*

The structural point is the parentage of each capability. The nullifier secret $\mathsf{sk}_{\mathsf{nf}}$ is a child of $\mathsf{sk}^{\mathsf{ser}}_{\mathsf{spend}}$, so nullifier derivation remains tied to spending authority. The incoming viewing key $\mathsf{vk}_{\mathsf{in}}$ is also a child of $\mathsf{sk}^{\mathsf{ser}}_{\mathsf{spend}}$, but it is read-only: disclosure of $\mathsf{vk}_{\mathsf{in}}$ can reveal incoming notes, not authorize spends. The outgoing viewing key $\mathsf{vk}_{\mathsf{out}}$ is separated at $\mathsf{sk}_{\mathsf{master}}$ and can reveal outgoing history, not $\mathsf{sk}_{\mathsf{spend}}$ or $\mathsf{sk}_{\mathsf{nf}}$.

For a receive path, the wallet chooses a diversifier d. The diversified payment address is defined by

$$\begin{aligned} \mathsf{sk}_{\mathsf{view}} := \mathsf{ScalarFromBytes}(\mathsf{HKDF}_{\mathsf{view}}(\mathsf{vk}_{\mathsf{in}})),\ G_d := \mathsf{DiversifyHash}(d), \\ \mathsf{pk}_d := [\mathsf{sk}_{\mathsf{view}}]\, G_d,\ \mathsf{addr}_d := (d,\ \mathsf{pk}_d). \end{aligned} \tag{1}$$

A payment address is public and consists only of the objects in Equation 1. The sender uses pk_d in the recipient Diffie-Hellman computation and places d in the encrypted note plaintext. After decryption, the recipient uses d to identify the local receive path and re-derives pk_d from $\mathsf{vk}_{\mathsf{in}}$. The diversifier d is therefore part of the address namespace, not an independent source of spend authority.

Output creation then uses fresh randomness r_{seed}. The encrypted plaintext contains only the value, d, and r_{seed}. The sender derives the ephemeral secret $\mathsf{sk}_{\mathsf{eph}}$ from r_{seed}, publishes the corresponding ephemeral public key $\mathsf{pk}_{\mathsf{eph}}$, and derives the note-encryption key from the recipient Diffie-Hellman result and $\mathsf{pk}_{\mathsf{eph}}$. Thus the public output fields bind the ciphertext to the selected address and output randomness, but they do not disclose recipient spend authority.

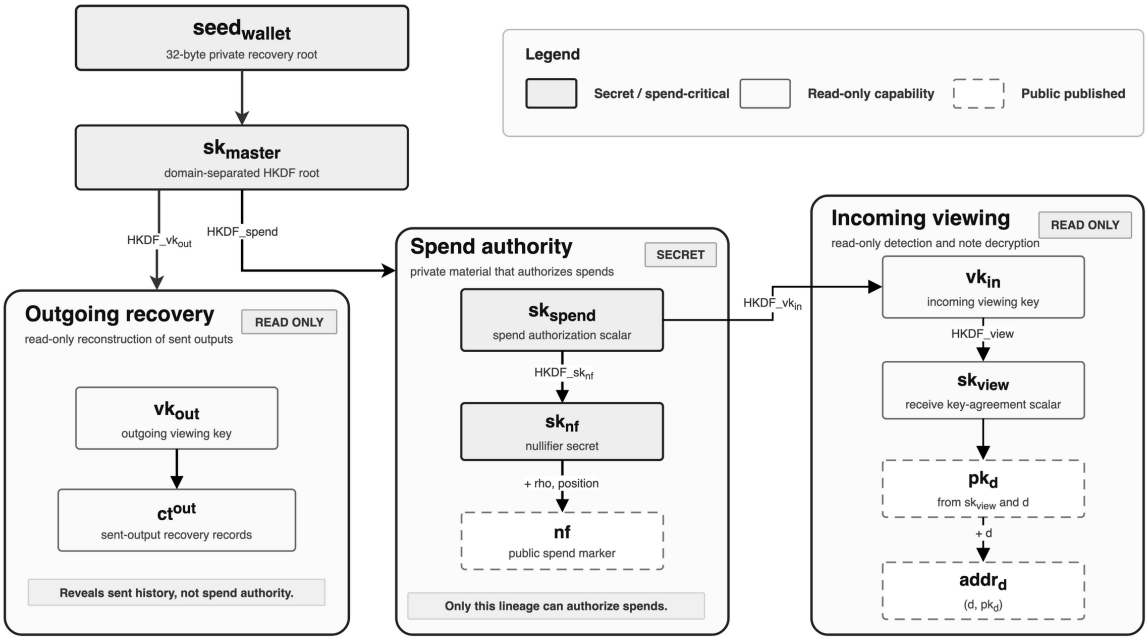

Figure 2: *Wallet key separation and address publication. Only the* $\mathsf{sk}_{\mathsf{spend}}$ *block carries spend authority. The* $\mathsf{vk}_{\mathsf{in}}$ *and* $\mathsf{vk}_{\mathsf{out}}$ *branches are read-only capabilities for incoming detection and outgoing recovery, respectively. The payment address is public and does not disclosure protocol privacy.*

7 Disclosure and Audit Model

Disclosure is a wallet-controlled read-only operation. It is not a consensus rule, a protocol backdoor, or an alternative spend path. There is no global disclosure key. Disclosure capabilities are read-only and do not transfer custody or spending authority to the auditor or any protocol service provider.

An audit asks the wallet to justify a bounded statement about replayed wallet activity. The statement may concern a particular note, a particular accepted output, a period summary, or a balance at a replay height. The wallet may disclose a viewing key for broad history, a note opening for a local check, or a report with supporting openings. These materials support only claims that the auditor can independently recompute, a wallet-produced report remains an assertion unless a separate proof supports it.

Figure 3 is a selection guide for disclosure material. The wallet should disclose the smallest object sufficient for the requested audit.

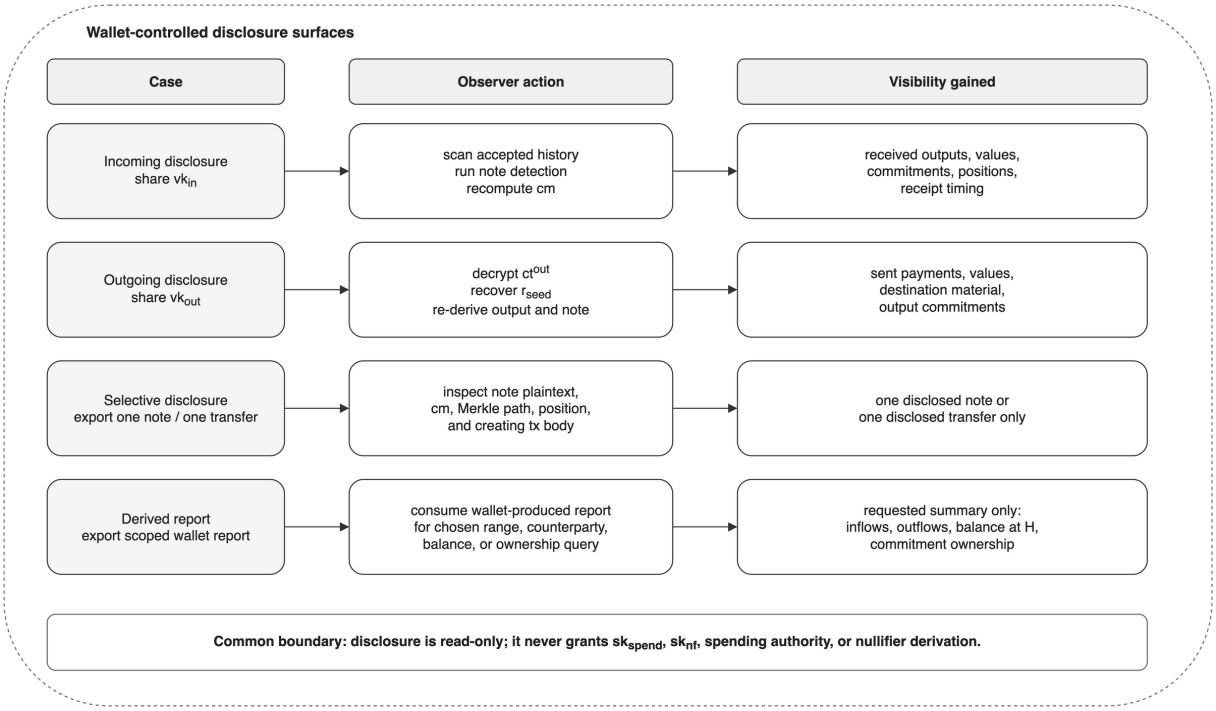

Figure 3: *Wallet-controlled disclosure modes. Each mode gives the auditor a specific read-only view of replayed activity. None of the modes grants spend authority or a nullifier-derivation capability.*

Each audit is evaluated at a replay height H and over a scope Q. The height H fixes the active Bitcoin prefix and the replayed state used for checking. The scope Q describes the intended boundary of the check, such as a block interval, report period, set of transfer hashes, set of output indices, or set of note leaves. The auditor recomputes the requested statement from deterministic replay at height H and the material the wallet disclosed. The audit supports only what can be recomputed from those inputs.

Disclosure cases.

IncomingView. Share $\mathsf{vk}_{\mathsf{in}}$ when the auditor must independently find wallet-received outputs in accepted history. This is useful for receipt audits, inbound revenue checks, and recovery of received note plaintexts over the history the auditor scans. The incoming viewing capability is read-only. It does not, by itself, provide access to outgoing recovery records, identify which notes were selected as inputs to a transfer, or confer authority to spend notes.

OutgoingView. Share $\mathsf{vk}_{\mathsf{out}}$ when the auditor must inspect wallet-produced sender records. This is useful for expense review, counterparty reconciliation, and proof of sent-payment history derived from $\mathsf{ct}^{\mathsf{out}}$. It does not identify the input notes consumed by those transfers.

NoteOpening. Open a specific note, output, transfer, or report entry when the audit is local. The auditor checks the opened plaintext against the accepted envelope, replayed note leaf, Merkle path, and claimed position. This exposes the opened object and the public replay data needed to verify it.

DerivedReport. Export a report when the auditor only needs a bounded statement, such as period inflows, period outflows, a counterparty total, note ownership, or balance at height H. The report must include enough supporting openings or view-key-derived evidence for the auditor to recompute the requested statement.

Raw $\mathsf{vk}_{\mathsf{in}}$ and $\mathsf{vk}_{\mathsf{out}}$ are broad read capabilities. The scope Q limits the audit statement, but it does not cryptographically restrict a raw viewing key. An observer holding $\mathsf{vk}_{\mathsf{in}}$ or $\mathsf{vk}_{\mathsf{out}}$ can apply that key to any replay history it can access. When the auditor should not learn unrelated matching history, the wallet should use note openings or a derived report.

No disclosure material contains $\mathsf{seed}_{\mathsf{wallet}}$, $\mathsf{sk}_{\mathsf{spend}}$, or $\mathsf{sk}_{\mathsf{nf}}$. Disclosure can give read access or evidence for an audit, but it cannot authorize a spend. Excluding $\mathsf{sk}_{\mathsf{nf}}$ also avoids turning disclosure into a general nullifier-linking tool.

Part III

Protocol

Indexer implementations reconstruct and maintain shielded state off-chain by deterministically replaying accepted transfer envelopes from the Bitcoin chain. Bitcoin provides only ordered publication of transfer envelopes. No shielded validity checks are performed inside Bitcoin consensus, and no changes to Bitcoin's protocol are required. Any correct implementation that processes the same accepted chain history derives the same protocol state, without any coordination or consensus of its own.

The basic building block of the protocol is a note, encoding a private satoshi amount together with recipient addressing material and a random seed from which note-specific values are derived. Notes never appear in plaintext on-chain. Bitcoin carries only recipient ciphertexts, ephemeral public keys, and the spend markers of consumed notes. To transfer value, the sender consumes one or more input notes and creates output notes, producing a zero-knowledge proof that binds private note plaintexts, Merkle membership witnesses, and spend secrets to the public transfer envelope. The proof enforces note membership under a recently accepted tree root, correct derivation of a nullifier for each consumed note, output ciphertext consistency, and value conservation. Once a nullifier is published and replayed, the corresponding note is unspendable while that spend remains in the active replay history. The following sections specify each component of the protocol in detail.

8 Replayed Shielded State

All protocol data is published to Bitcoin, but shielded state exists at the metaprotocol layer. This state is defined by replay, not by a separate consensus protocol. An implementation scans the active Bitcoin chain, extracts candidate transfer envelopes, validates them under the protocol rules, and applies accepted transfers in canonical Bitcoin order.

Definition 1 (Replayed shielded state)**.** *At any replay boundary, the shielded metaprotocol state is the tuple* $\mathcal{S} := (\mathcal{T},\ \mathcal{N},\ \mathcal{R},\ \mathcal{H})$*, where* $\mathcal{T}$ *is the append-only note-leaf Merkle tree,* $\mathcal{N}$ *is the set of accepted nullifiers,* $\mathcal{R}$ *is the retained block-height map* $\mathcal{R}[h]$*, and* $\mathcal{H}$ *is the accepted transfer history in replay order.*

Two correct implementations processing the same active chain and applying the same replay rules derive the same $\mathcal{S}$, without communicating with each other and without running a separate consensus protocol.

As fixed in Definition 1, replay maintains the note tree $\mathcal{T}$, nullifier set $\mathcal{N}$, and block-level root history $\mathcal{R}$. For each accepted transfer, replay derives ℓ values from the public output data, appends those leaves to $\mathcal{T}$, and inserts the published nullifiers into $\mathcal{N}$. A transfer is invalid if it contains the same nullifier twice or if any of its nullifiers already appears in $\mathcal{N}$. Invalid envelopes do not mutate replayed state.

Roots used as anchors are not recorded after each accepted transfer. For each Bitcoin block H, the indexer starts from the root after block $H-1$, processes Bitcoin transactions in canonical order, mutates the working note tree and nullifier set for each accepted transfer, and stores the resulting root only after the whole block has been replayed: $\mathcal{R}[H] := \texttt{root}(\mathcal{T} \text{ after block } H)$.

If block `H` contains no accepted transfers, replay performs no mutations during that block, so $\mathcal{R}[H] = \mathcal{R}[H-1]$. Every retained height, therefore, has a well-defined root, including heights with no shielded activity.

The transfer envelope does not publish the full anchor root. Instead, it publishes h_{anchor}, a Bitcoin block height. During replay, the indexer reconstructs the corresponding root locally as $r_{\mathsf{anchor}} := \mathcal{R}[h_{\mathsf{anchor}}]$. Thus, h_{anchor} is a compact reference to a block-level replay checkpoint. It is not a shielded-transfer sequence number and not a counter of accepted protocol events. It is a chain-local reference: if two indexers replay the same active Bitcoin chain, they compute the same $\mathcal{R}[h]$ for every retained height H.

The transfer proof is generated relative to one such replayed root. The selected anchor is used for the input-note membership checks inside the transfer circuit: it determines the Merkle root against which the path for each consumed note is verified.

The protocol permits a transfer to prove membership against an older retained root, provided the anchor lies inside the valid-root window. An accepted transfer is still applied to the current replay state, but the membership claims for its input notes are proven relative to the selected r_{anchor}. This is safe because the note tree is append-only. If a note exists in the selected root, it also exists in later roots until it is spent. The current nullifier set $\mathcal{N}$, not the age of the anchor, determines whether the note has already been consumed. Because any retained root inside the window is a valid anchor, a single root can serve many transfers, and a wallet ordinarily selects the most recent root for which it already holds the membership witnesses it needs.

The protocol-level valid-anchor window is defined by a maximum and a minimum block depth:

$$H - W \leq h_{\mathsf{anchor}} \leq H - K_{\min},$$

where H is the Bitcoin block currently being replayed, W is the maximum anchor age, and $K_{\min}$ is the minimum anchor depth required by the protocol. Both W and $K_{\min}$ are measured in Bitcoin blocks.

The value of W is a replay rule, not deployment configuration. Different accepted values of W give indexers different validity predicates for already-built transfers. For any chain containing a transfer accepted by one predicate and rejected by the other, replay derives divergent shielded state from that same chain.

This bound determines how long an already-built transfer can remain acceptable before it must be rebuilt against a newer anchor. If a transfer is not included before its chosen h_{anchor} falls out of this window, the note is not lost. Only that serialized transfer has expired. The wallet must choose a newer valid anchor, recompute the Merkle path against that root, and generate a fresh transfer proof.

$K_{\min}$ is the minimum depth needed for the anchor to be a well-defined replayed root before the current block is processed. The smallest natural choice is $K_{\min} = 1$. With it a transfer in block H may reference $\mathcal{R}[H-1]$, but not $\mathcal{R}[H]$, because the latter is defined only after block H has been replayed.

Reorganization safety is a separate wallet-policy question. A wallet may choose a stricter effective depth $K_{\mathsf{wallet}} \geq K_{\min}$ when selecting anchors: $h_{\mathsf{anchor}} \leq H - K_{\mathsf{wallet}}$. This bound is not needed to be set inside the protocol. It expresses how much confidence the wallet wants that the root used by the transfer proof will remain on the active Bitcoin chain.

Each transfer proof is generated relative to a particular state root. If a wallet builds the proof against $\mathcal{R}[H-1]$, then a one-block reorganization can replace the block at height $H-1$, change the root at that height, and make the transfer invalid on the new active chain. The proof is not malformed. It was generated relative to a state that is no longer the state obtained by replaying the active chain.

Thus, $K_{\min}$ defines what replay is allowed to accept, while K_{wallet} defines what a wallet considers safe to build. A deployment may set $K_{\min} = 1$ at the protocol level and recommend a larger default K_{wallet} for wallets. Deployment may hard-code a larger $K_{\min}$ if it wants all accepted transfers to use anchors that are at least that deep.

Bitcoin reorganizations change the active-chain input to replay and can therefore change $\mathcal{R}[h]$ for the first replaced height and every height after it. The resulting acceptance and rebuild rules are stated in Section 15.

The maximum age W serves a different purpose. It determines how long a chosen block-level anchor remains acceptable after a transfer proof has been generated. If W is too small, a wallet has little time to publish a transfer before the chosen h_{anchor} ages out of the valid window. This makes transfers more sensitive to fee delays, offline signing, slow coordination, and relayer workflows. Making W larger does not create any additional direct risks. Replay still checks every published nullifier against the current nullifier set $\mathcal{N}$, regardless of how old the selected anchor is. The main costs are operational. Indexers must retain a longer block-level root history, protocol upgrades that change the accepted statement or replay rules may need to keep old anchors valid for longer, and wallets expose a wider public range. Unusual anchor ages can fingerprint wallet behavior. For example, if most wallets use anchors between 6 and 20 blocks deep, but one transfer uses a 700-block-old anchor, that may reveal offline signing, relayer delay, or a nonstandard wallet policy.

9 Note Leaves and Nullifiers

Once notes exist, most of the protocol's security burden sits in note-tree leaves and nullifiers. Replay derives one canonical note-tree leaf for each created output from public envelope data:

$$\ell_j := H_{\mathsf{leaf}}\big(\texttt{const_salt},\ h_{\mathsf{body}},\ j,\ \mathsf{pk}_{\mathsf{eph},j},\ \mathsf{ct}_j^{\mathsf{note}},\ h_{\mathsf{aux},j}\big).$$

j is the zero-based output index inside the current e: the first created output has $j = 0$, the second has $j = 1$, and so on. It is not the global Merkle-tree position. Replay assigns the global pos only later, when accepted note leaves are appended to the note tree in Bitcoin order. For later spending, the wallet stores both identifiers: the creation output index j, which re-derives the note leaf, and the assigned tree pos, which is used in the Merkle path and nullifier relation.

H_{leaf} is a fixed domain-separated leaf hash for the transfer protocol. Including h_{body} and the output index separates otherwise identical ciphertexts appearing in different transfers or at different output positions. The field $h_{\mathsf{aux},j}$ is a reserved extension slot (Section 22.5), currently we fix it to the canonical constant `aux_null` and is not serialized in this version.

The transfer circuit must prove that every public $(\mathsf{pk}_{\mathsf{eph}},\ \mathsf{ct}^{\mathsf{note}})_j$ pair encrypts the same private output note used for value conservation. Output leaf derivation itself is a public replay computation; the proof checks the public fields that feed that computation. For spent input notes, the circuit proves the analogous relation against the creation h_{body}, output index, $\mathsf{pk}_{\mathsf{eph}}$, and $\mathsf{ct}^{\mathsf{note}}$ stored by the wallet for that note. This makes the ciphertext-derived leaf the public bridge between private note plaintext, replayed state, and later spending.

For spent notes, the public spend-marker relation is:

$$\mathsf{nf} := H_{\mathsf{nf}}\big(\mathsf{BytesToField}(\mathsf{sk}_{\mathsf{nf}}), \mathsf{BytesToField}(\rho),\ \mathsf{pos}\big),$$

where H_{nf} is the domain-separated nullifier derivation function fixed by the protocol.

$\mathsf{sk}_{\mathsf{nf}}$ ties nullifier derivation to spender-controlled secret material. ρ makes the nullifier note-specific. pos ties the spend to the assigned position of the note leaf in the global tree. In a replayed metaprotocol, that position is part of spend identity.

A spent input note is spendable only by someone who knows the $\mathsf{sk}_{\mathsf{spend}}$ corresponding to the same wallet secret lineage that generated the pk_d used in the ciphertext relation for that note. The short d changes how the wallet identifies the receive path, not how spend authority is enforced. Spend authority is not exported as a separate public key inside the payment address. $\mathsf{sk}_{\mathsf{nf}}$ is not an independent witness secret. The circuit derives $\mathsf{sk}_{\mathsf{nf}}$ deterministically from $\mathsf{sk}_{\mathsf{spend}}$ and rejects any construction that would let the prover choose $\mathsf{sk}_{\mathsf{nf}}$ separately. That binding is required so one spend authority implies one nullifier derivation path for a given note and position. Without it, the same note could be spent more than once under different nullifiers.

The circuit-level ownership rule is:

$$\begin{aligned}
\mathsf{vk}_{\mathsf{in}}^{\mathsf{drv}} &:= \mathsf{HKDF}_{\mathsf{in}}(\mathsf{BytesFromScalar}(\mathsf{sk}_{\mathsf{spend}})),\\
\mathsf{sk}_{\mathsf{view}}^{\mathsf{drv}} &:= \mathsf{ScalarFromBytes}(\mathsf{HKDF}_{\mathsf{view}}(\mathsf{vk}_{\mathsf{in}}^{\mathsf{drv}})),\\
\mathsf{pk}_d^{\mathsf{drv}} &:= [\mathsf{sk}_{\mathsf{view}}^{\mathsf{drv}}]\,\mathsf{DiversifyHash}(d),\\
\mathsf{pk}_d^{\mathsf{drv}} &= \mathsf{pk}_d
\end{aligned}$$

for every spent input note. This ownership rule applies only to inputs. Created outputs instead encrypt to the recipient-side pk_d chosen from the destination payment address and do not derive that transmission key from the sender's $\mathsf{sk}_{\mathsf{spend}}$.

pk_d determines which diversified receive path the sender targets for a given d, while the corresponding $\mathsf{sk}_{\mathsf{view}}$ lets the recipient derive the same shared secret from the published $\mathsf{pk}_{\mathsf{eph}}$. $\mathsf{sk}_{\mathsf{spend}}$ remains the secret that determines who can spend it.

10 Transfer Pipeline

A transfer pipeline instance begins with a wallet construction view $\mathcal{S}_{\mathsf{build}} = (\mathcal{T}_{\mathsf{build}}, \mathcal{N}_{\mathsf{build}}, \mathcal{R}_{\mathsf{build}}, \mathcal{H}_{\mathsf{build}})$, described in 1, obtained by replaying a Bitcoin prefix. The wallet uses that view to construct a candidate envelope e. The envelope becomes canonical protocol state only if deterministic replay later accepts e from the active Bitcoin chain. Figure 4 summarizes this flow across sender construction, proof generation, Bitcoin publication, deterministic replay, and wallet-side note recovery. It also marks the boundary between local wallet activity and canonical replay state mutation.

Roles. The pipeline uses the architecture of Section 4: wallets construct envelopes, Bitcoin publishes and orders them, and indexers replay them. If the prover is separated from the wallet, it is inside the sender's trusted computing base, because the witness contains note plaintexts and spend authority. A wallet that does not run its own Bitcoin node and replay indexer treats the selected node and indexer as local trusted sources for the claimed chain prefix and replay state.

Pipeline stages. A correct sender wallet constructs and submits one transfer through the following stages.

1. **Address and output intent.** The sender obtains recipient payment addresses (d_j, pk_{d_j}) and chooses the logical output set, including any change output. Address exchange is outside replay, but it fixes the recipient-side diversified receive paths and transmission keys used by the created notes.

2. **Synchronized construction view.** The wallet constructs against a replayed state $\mathcal{S}_{\mathsf{build}}$, obtained by local replay, by a verified indexer response, or by an explicitly trusted indexer. Mempool data and unaccepted envelopes are not part of this state. The construction view must provide the current nullifier set or an equivalent query interface, retained block-level roots, note positions, and Merkle witnesses for the candidate input notes.

3. **Input and anchor selection.** The sender chooses input notes whose leaves have replay-assigned positions in $\mathcal{T}_{\mathsf{build}}$ and whose nullifiers are absent from $\mathcal{N}_{\mathsf{build}}$. It chooses an admissible h_{anchor} from the retained block-level root window and retrieves Merkle paths against $r_{\mathsf{anchor}} = \mathcal{R}[h_{\mathsf{anchor}}]$. The selected inputs must already exist under that root. These are construction checks; replay repeats the public checks when the envelope appears on-chain.

4. **Output and recovery construction.** For each output j, the sender creates a note plaintext $(v, d, r_{\mathsf{seed}})_j$ with fresh note randomness, derives the output encryption material, and writes the public $(\mathsf{pk}_{\mathsf{eph}},\ \mathsf{ct}^{\mathsf{note}})_j$ pair into the transfer body. The sender also constructs $\mathsf{ct}^{\mathsf{out}}$ under $\mathsf{vk}_{\mathsf{out}}$ so the outgoing transfer can be recovered later. Recipient ciphertext consistency is part of the transfer relation.

5. **Body binding and proof generation.** The wallet finalizes the canonical transfer body containing the public data that replay and wallet recovery will later consume. At this point the body is no longer a mutable construction object. The wallet computes h_{body} from the canonical encoding and gives the corresponding public statement and private witness to the prover. The proof attests that the private inputs and created outputs satisfy the transfer relation, and it binds that relation to the exact public body through h_{body}. Changing any body component after proof generation changes the statement and requires a new proof.

6. **Bitcoin publication.** The final envelope is placed in a Bitcoin-valid carrier transaction. Inclusion in a block publishes and orders the bytes. It does not by itself spend any shielded note or create any spendable shielded output. Until replay accepts the envelope, the sender keeps the input notes locally locked. They become canonically spent only after replay accepts their nullifiers. Recipients treat any visible outputs as unconfirmed candidates without tree positions.

7. **Replay acceptance.** For a candidate envelope e appearing in Bitcoin block H, replay evaluates the protocol acceptance predicate against the current replay state. The checks include canonical parsing, body binding, valid-anchor-window membership, nullifier uniqueness, proof verification against the derived r_{anchor}, and the remaining replay rules of Section 15. If any check fails, e is rejected and the replay state is unchanged.

8. **Settlement and note detection.** If replay accepts e, it derives the ordered output note leaves, appends them to the note tree, assigns canonical positions, inserts the published nullifiers, and records the accepted transfer event. Recipients and senders then update their wallet views by scanning the accepted envelope and applying the decryption-and-consistency checks in Section 16. A decrypted output is not spendable because the wallet can recognize it. It becomes spendable only after replay has appended its leaf, assigned its canonical position, and the wallet can obtain a Merkle witness under an admissible anchor root that contains that leaf.

Definition 2 (Replay transition)**.** *Let $\mathcal{S} = (\mathcal{T}, \mathcal{N}, \mathcal{R}, \mathcal{H})$ be the current replay state while processing a block B. Let $\mathsf{Accept}(\mathcal{S}, B, e)$ be the deterministic acceptance predicate for a candidate envelope e under the replay rules. Let $\mathsf{NF}(e)$ be the set of nullifiers published by e. Let $\mathsf{Leaves}(e)$ be the ordered output leaves deterministically derived from the canonical public body of e. When $A_e := \mathsf{Accept}(\mathcal{S}, B, e)$, The per-envelope transition is*

$$\delta_B(\mathcal{S}, e) := \begin{cases} \mathcal{S}, & \text{if } A_e = \mathsf{false}, \\ (\mathcal{T}', \mathcal{N}', \mathcal{R}, \mathcal{H}'), & \text{if } A_e = \mathsf{true}, \end{cases}$$

where

$$\mathcal{T}' := \mathsf{Append}(\mathcal{T}, \mathsf{Leaves}(e)), \qquad \mathcal{N}' := \mathcal{N} \cup \mathsf{NF}(e), \qquad \mathcal{H}' := \mathcal{H} \,\|\, e.$$

Thus a rejected envelope has no partial effect on $\mathcal{T}$, $\mathcal{N}$, $\mathcal{R}$, or $\mathcal{H}$. The block-level root map $\mathcal{R}$ is updated only at block finalization, after every candidate envelope in B has been processed in canonical Bitcoin transaction order.

Consequences for wallets and replay. Definition 2 captures the state-level properties needed by the pipeline. Only accepted replay mutates shielded state, and acceptance is atomic: a candidate envelope either applies δ_B or leaves $\mathcal{S}$ unchanged. Wallet construction, proof generation, mempool relay, and Bitcoin inclusion are not shielded-state updates. They only provide inputs that replay may later accept or reject.

The remaining consequences are wallet-local. Before replay acceptance, consumed inputs are locked only by wallet policy, and created outputs are unconfirmed candidates without spendable positions. If the carrier transaction is not accepted, expires from the anchor window, or is invalidated by a reorganization, the wallet resynchronizes and rebuilds against the newly replayed state. A decrypted output becomes spendable only after replay appends its leaf, assigns its position, and the wallet can obtain a Merkle witness under an admissible anchor root that contains that leaf.

11 Publication Layer

Bitcoin provides publication and ordering for transfer envelopes. The carrier transaction does not execute shielded logic, and Bitcoin consensus does not interpret the envelope as a private transfer. Consensus only determines whether the carrier transaction is valid and where it appears in the chain. Shielded validity is checked by metaprotocol software that reads the published bytes, verifies the proof, and replays accepted envelopes in chain order.

This is the usual separation used by Bitcoin metaprotocols: the base chain publishes data, while external software assigns protocol-specific meaning to that data. A transaction may be valid on Bitcoin even if the metaprotocol payload it carries is ignored or rejected by a particular parser or replay implementation.

For this reason, the transfer protocol is defined over a canonical envelope encoding, not over a single Bitcoin transaction template. A publication method is suitable if indexers can locate candidate envelopes in accepted Bitcoin data, recover the complete envelope bytes, parse them deterministically, and order them by the carrier transaction's position in the active chain. Relay and mining policy still matter for deployment, since standardness policy and Bitcoin consensus validity are not the same thing.

Several carrier styles can provide this data-availability role. Counterparty documents `OP_RETURN` and other transaction-output encodings for metaprotocol data, while Ordinal inscriptions give a prominent example of w-carried data [Cou, Ord]. A Shielded Bitcoin envelope could be published through either style, or through another Bitcoin-valid structure that indexers can locate and parse deterministically. The choice affects footprint, fee behavior, standardness assumptions, failure modes, and deployability, but not

the transfer statement verified by the proof. The current implementation profile in Appendix A fixes an `OP_RETURN` carrier, while alternative carriers remain deployment choices discussed in Section 22.1.

Each envelope begins with protocol-identifying metadata such as a magic value, a version, and a transfer-type discriminator. These fields let indexers recognize candidate envelopes and route them to the appropriate parser and replay rules. The exact byte layout and canonical binary serialization are defined in Sections A.3 and A.5.

Table 2 summarizes the logical structure of one transfer envelope. The fields are independent of the carrier style; only the Bitcoin transaction path used to publish the bytes differs.

Field	Purpose
`header`	Fixed discriminator for protocol parsing: magic bytes, version, and transfer type.
h_{anchor}	Bitcoin block height selecting the block-level note-tree root against which the spend proof is built.
`input_count`, `output_count`	Explicit arity of the transfer. These counts determine how many nullifiers, ephemeral keys, and ciphertexts follow.
$\mathsf{nf}_{0..N-1}$	Public spend markers for the consumed input notes. Replay uses them to reject double spends.
$\mathsf{pk}_{\mathsf{eph},0..M-1}$	Per-output ephemeral public keys used by recipients and sender-side recovery to derive note decryption secrets.
$\mathsf{ct}^{\mathsf{note}}_{0..M-1}$	Recipient ciphertexts carrying the encrypted note plaintext for each output.
$\mathsf{ct}^{\mathsf{out}}$	Batched sender-recovery ciphertext under $\mathsf{vk}_{\mathsf{out}}$, carrying the per-output recovery material needed to reconstruct sent notes.
π	A succinct zero-knowledge proof showing that the inputs, outputs, nullifiers, and derived anchor root satisfy the transfer rules.

Table 2: *High-level structure of e. The exact field order, encoding rules, and replay requirements are fixed in the Transfer envelope and serialization appendix.*

12 Output Encryption

A transfer carries two encrypted channels: a sender-recovery channel over $\mathsf{vk}_{\mathsf{out}}$ and per-output recipient channels. The sender-recovery ciphertext $\mathsf{ct}^{\mathsf{out}}$ must be finalized before h_{body} is computed, since the canonical transfer body that feeds h_{body} includes $\mathsf{ct}^{\mathsf{out}}$. The per-output recipient channels encrypt each output note's plaintext using the same $\mathsf{sk}_{\mathsf{eph}}$ values already committed in the recovery batch, so both channels share a common per-output randomness source.

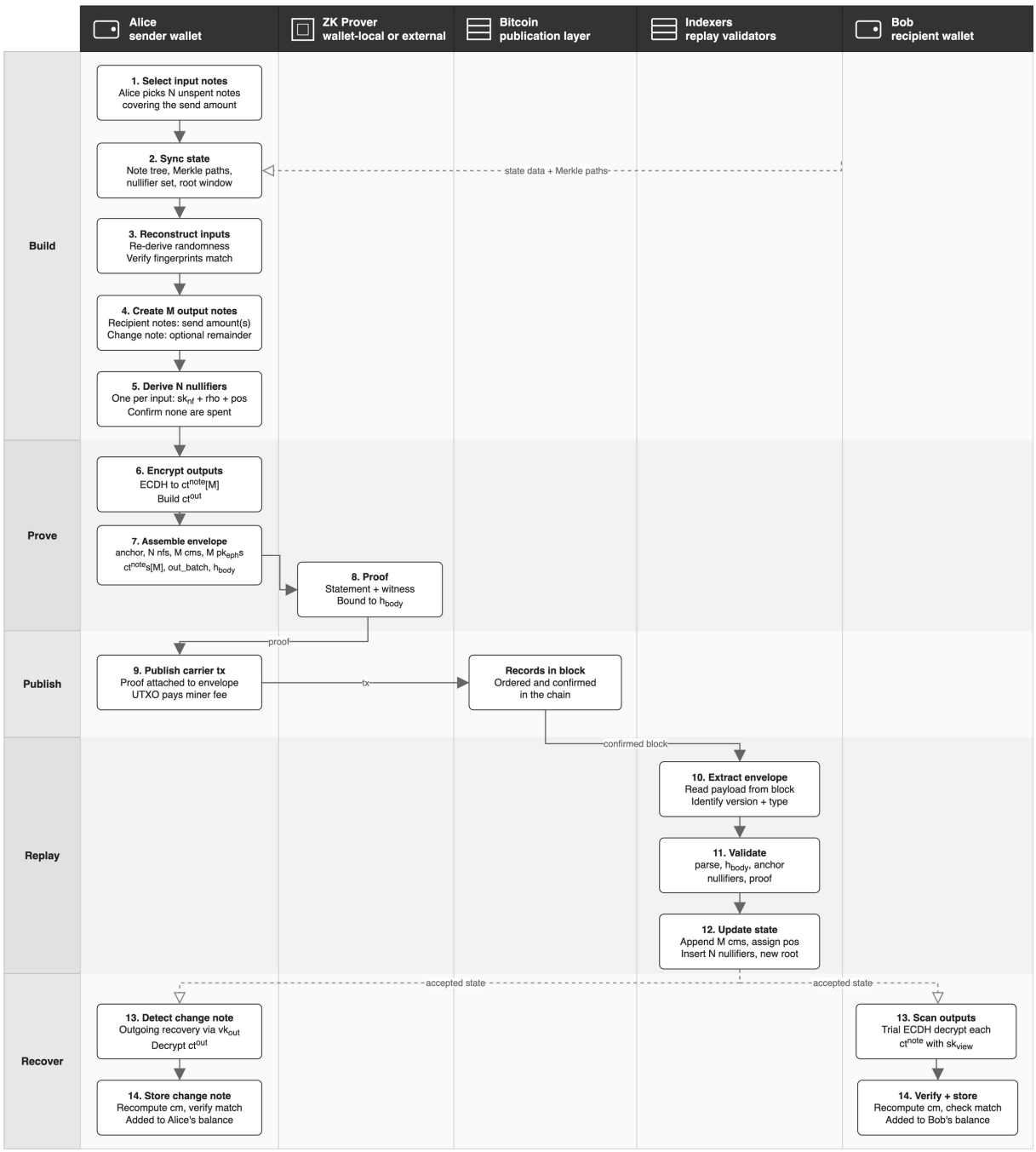

Figure 4: *Representative transfer lifecycle from note selection through publication, replay, and recipient-side note detection.*

12.1 Recipient Channels

Each output note is encrypted to its recipient through a per-output ephemeral channel. The sender uses $\mathsf{sk}_{\mathsf{eph}}$, the same per-output ephemeral secret already committed in $\mathsf{ct}^{\mathsf{out}}$, to compute $\mathsf{pk}_{\mathsf{eph}} = [\mathsf{sk}_{\mathsf{eph}}]G_d$ and derives the note shared secret as $\mathrm{ECDH}(\mathsf{sk}_{\mathsf{eph}}, \mathsf{pk}_d)$. A note key schedule is derived from that shared secret together with the published $\mathsf{pk}_{\mathsf{eph}}$, and the note plaintext $(v, \mathsf{d}, r_{\mathsf{seed}})$ is encrypted under that key to produce $\mathsf{ct}^{\mathsf{note}}$. The recipient derives the same shared secret from the published $\mathsf{pk}_{\mathsf{eph}}$ using $\mathrm{ECDH}(\mathsf{sk}_{\mathsf{view}}, \mathsf{pk}_{\mathsf{eph}})$, re-derives the note key schedule, and decrypts $\mathsf{ct}^{\mathsf{note}}$ to recover $(v, \mathsf{d}, r_{\mathsf{seed}})$. The concrete key schedule and AEAD constructions are fixed in Appendix A. The transfer circuit must prove ciphertext/plaintext consistency for every output: that the public $(\mathsf{pk}_{\mathsf{eph}}, \mathsf{ct}^{\mathsf{note}})_j$ pair encrypts the same output note used by the circuit for value conservation, and that $\mathsf{pk}_{\mathsf{eph},j} = [\mathsf{sk}_{\mathsf{eph},j}]G_{d_j}$. This constraint binds the public output fields to the private note plaintexts and makes the ciphertext-derived note leaf a valid public commitment to the output. Post-decryption consistency checks performed by the recipient

wallet are specified in Section 16.1. Figure 4 shows the transfer path from input selection to recipient-side note detection.

12.2 Sender Recovery Channel

$\mathsf{ct}^{\mathsf{out}}$ carries one encrypted record per output: the ordered list $(\mathsf{pk}_d,\ \mathsf{sk}_{\mathsf{eph}})_0, \ldots, (\mathsf{pk}_d,\ \mathsf{sk}_{\mathsf{eph}})_{M-1}$ indexed to match output order. A single AEAD tag covers the entire batch rather than one tag per entry. The batch is keyed from $\mathsf{vk}_{\mathsf{out}}$ and bound to the fixed public output data. A batch-binding digest is derived from the output count together with the ordered $\mathsf{pk}_{\mathsf{eph}}$ and $\mathsf{ct}^{\mathsf{note}}$ values, all of which are fixed before this ciphertext is constructed. That digest feeds a $\mathsf{vk}_{\mathsf{out}}$-based KDF producing the AEAD key and nonce for the batch, and also serves as AEAD associated data, binding the recovery ciphertext to the exact set of output ciphertexts it covers. This channel is finalized before h_{body} is computed because the canonical transfer body that feeds h_{body} includes $\mathsf{ct}^{\mathsf{out}}$. The recipient ciphertexts and $\mathsf{pk}_{\mathsf{eph}}$ values that serve as the batch-binding inputs are already fixed at that point, so no circular dependency arises. The concrete KDF and AEAD constructions are fixed in Appendix A. The procedure for using this channel to reconstruct sent-payment history is specified in Section 16.2.

13 Transfer Construction and Binding

This section specifies how a wallet assembles and publishes a e. Construction produces the canonical public body; h_{body} is derived from that body and supplied as a public input to the transfer proof, binding the proof to the exact envelope before publication.

13.1 Assembly order

The following five stages must be executed in order.

1. **Input preparation.**
   - Select confirmed input notes with nullifiers absent from $\mathcal{N}$.
   - Choose h_{anchor}, retrieve $r_{\mathsf{anchor}} = \mathcal{R}[h_{\mathsf{anchor}}]$ and the Merkle path for each input.
   - Reconstruct each input ℓ_i from $(v_i, d_i, r_{\mathsf{seed},i})$ and creation envelope fields, verify against the stored leaf.
   - Derive nf_i from $\mathsf{sk}_{\mathsf{nf}}$, ρ_i, and pos_i; verify no nf_i appears in $\mathcal{N}$ and all are distinct.
2. **Output construction.**
   - For each output j: form $(v_j, d_j, r_{\mathsf{seed},j})$ with fresh r_{seed}.
   - Verify $\sum_i \mathit{input}_i[v] = \sum_j \mathit{output}_j[v]$ (off-circuit).
   - For each output j: derive $\mathsf{pk}_{\mathsf{eph},j} = [\mathsf{sk}_{\mathsf{eph},j}]\, G_{d_j}$; encrypt into $\mathsf{ct}^{\mathsf{note}}_j$ (see sec. 12.1).
3. **Recovery ciphertext.**
   - Derive batch-binding digest from output count, the $\mathsf{pk}_{\mathsf{eph}}$ vector and the $\mathsf{ct}^{\mathsf{note}}$ vector.
   - Encrypt $[(\mathsf{pk}_d,\ \mathsf{sk}_{\mathsf{eph}})_0, \ldots, (\mathsf{pk}_d,\ \mathsf{sk}_{\mathsf{eph}})_{M-1}]$ under $\mathsf{vk}_{\mathsf{out}}$ to produce $\mathsf{ct}^{\mathsf{out}}$.
4. **Body serialization and binding.**
   - Serialize: `header` $\|$ h_{anchor} $\|$ `input_count` $\|$ `output_count` $\|$ nf vector $\|$ $\mathsf{pk}_{\mathsf{eph}}$ vector $\|$ $\mathsf{ct}^{\mathsf{note}}$ vector $\|$ $\mathsf{ct}^{\mathsf{out}}$ (proof excluded).
   - Compute $h_{\mathsf{body}} := H_{\mathsf{body}}$ (`const_salt` $\|$ `canonical_transfer_body`).
   - For each output j: derive ℓ_j from h_{body}, j, $\mathsf{pk}_{\mathsf{eph},j}$, $\mathsf{ct}^{\mathsf{note}}_j$, $h_{\mathsf{aux},j}$ (see 14.2).
5. **Proving and publication.**
   - Assemble public inputs and private witness (see 14.3).
   - Run the transfer proof.
   - Append proof bytes to the envelope.
   - publish.

After publication, treat consumed notes as locally locked until the envelope appears in accepted replay state. Output notes are not spendable until replay assigns a position. On reorganization, roll back local pending state and rebuild the transfer against the new chain history.

13.2 Serialization and Circuit Mapping

The serialized envelope and the circuit public-input layout need not be the same object. The envelope is count-prefixed and carries only the logically present entries. A concrete proving system may verify that statement directly or project it into a fixed-width verifier interface.

That mapping has to be fixed for every deployed verifier interface. `input_count` and `output_count` determine the logical lengths of the serialized vectors. If a concrete verifier interface uses padded slots, the dummy-value convention must also be fixed explicitly. If different implementations map serialized envelopes into circuit inputs differently, replay will diverge even if they agree on the byte-level envelope. See Section 14.3 for further details.

13.3 Transaction Binding

The transfer proof constrains only the fields that enter the circuit as public inputs: the derived r_{anchor}, the nf values, the per-output $\mathsf{pk}_{\mathsf{eph}}$ vector and $\mathsf{ct}^{\mathsf{note}}$ vector, and the derived h_{body}. The remaining published fields are not exposed individually to the verifier. These include the serialized header, h_{anchor}, the explicit `input_count` and `output_count`, and the batched sender-recovery ciphertext $\mathsf{ct}^{\mathsf{out}}$. With no explicit commitment, any of these unconstrained fields could be rewritten after proving without affecting proof verification, and even the recipient ciphertexts and ephemeral keys could be replaced with unrelated values. h_{body} closes this gap by committing to the entire public body as one object.

The hash is computed over the transfer body and supplied to the circuit as a public input:

$$h_{\mathsf{body}} := H_{\mathsf{body}}(\texttt{const_salt} \,\|\, \texttt{canonical_transfer_body}),$$

where H_{body} is the domain-separated transfer-binding hash fixed by the protocol, `canonical_transfer_body` is the ordered byte sequence of the header, h_{anchor}, the input and output counts, and the nf values, $\mathsf{pk}_{\mathsf{eph}}$ vector, $\mathsf{ct}^{\mathsf{note}}$ vector, and $\mathsf{ct}^{\mathsf{out}}$ fields, in the order fixed in the Transfer envelope and serialization Appendix A.5. The π field is excluded from the hashed bytes, since it is produced from h_{body} and appended only after proving completes; hashing it would be circular. The binding is therefore finalized once $\mathsf{ct}^{\mathsf{out}}$ is fixed, and proving runs against the resulting h_{body}.

Because the hash covers the whole body, the published envelope is non-malleable as a unit. Editing any byte of that body, whether by swapping a recipient ciphertext, replacing an ephemeral key, or rewriting h_{anchor}, changes h_{body}, and the proof generated for the original value no longer verifies. A third party cannot create a valid proof onto a modified body. This also binds h_{anchor} indirectly: the verifier statement exposes the derived $r_{\mathsf{anchor}} = \mathcal{R}[h_{\mathsf{anchor}}]$ rather than the raw height, but since h_{anchor} is part of the hashed body, an indexer that derived r_{anchor} from any other height would recompute a different h_{body} and reject the envelope.

The binding also reaches into the note tree. Each output's ℓ_j is derived from h_{body} together with the output index, $\mathsf{pk}_{\mathsf{eph},j}$, and $\mathsf{ct}^{\mathsf{note}}_j$ (Section 9), so the created leaves are tied to the exact transaction body that produced them. The same ciphertext appearing in a different transfer or at a different output position therefore yields a distinct leaf.

Replay checks the binding before mutating any state. Following the indexer acceptance order A.7, an indexer reconstructs `canonical_transfer_body` from the published carrier transaction A.4, recomputes h_{body}, and compares it against the value the proof was bound to before deriving the public statement, verifying the proof, and applying the envelope. A mismatch, from an alternate encoding or a substituted field, fails this check, and the envelope is rejected as a whole rather than partially applied. The byte-level format and replay rules are fixed in the Transfer envelope and serialization appendix.

14 Transfer Statement

One valid proof binds one private state transition to one canonical public transfer envelope. For replaying indexers, the proof establishes transfer validity without revealing which input notes were spent or which output belongs to which recipient. The protocol requires compact proofs and low verifier cost because every replaying indexer verifies every accepted transfer outside Bitcoin consensus. Output note leaves are derived by replay from public fields that the proof and h_{body} bind; spent input note leaves are reconstructed in circuit for Merkle membership. Everything that does not need arithmetic constraints stays outside the circuit and is handled by wallets and indexers: note scanning, recipient and outgoing recovery bookkeeping, note selection, Merkle-path retrieval, anchor height selection from the block-level valid-root window, canonical serialization, and duplicate nullifier rejection. The circuit definition is therefore the public statement (14.2), the witness (14.3), and the constraints connecting them (Appendix B)

14.1 Proved conditions

For one transfer, the proof establishes:

1. each input note exists in the note tree under the selected r_{anchor}
2. for each input, the prover knows the canonical position and Merkle path
3. for each input, the prover knows the secrets required to spend that note
4. the published nullifiers are exactly the values derived from the spent input notes
5. the public output $\mathsf{pk}_{\mathsf{eph}}$ and $\mathsf{ct}^{\mathsf{note}}$ values encrypt the same output notes used by the circuit
6. the public fields from which replay derives output note leaves are the same fields bound by h_{body} and checked by the proof
7. every input and output value is a canonical satoshi amount in the allowed range
8. the sum of input values equals the sum of output values without modular overflow
9. the public envelope posted to Bitcoin is the same envelope that the proof binds to

The output ciphertext relation binds the diversified transmission key pk_d. For spent input notes, that pk_d is reconstructed in circuit from $\mathsf{sk}_{\mathsf{spend}}$ and d. For created output notes, the recipient-side pk_d comes from the destination payment address chosen by the sender and is supplied directly as output address material in the witness; it is not derived from the sender's $\mathsf{sk}_{\mathsf{spend}}$. The constraint decomposition arithmetizing these conditions is in Appendix B.

14.2 Public statement

The public statement is the interface between the transfer proof and replay: the values the proof exposes to the verifier, which are exactly the values an indexer reads from the accepted envelope to verify the proof and derive a new state. Everything else is either a private witness or body-bound through h_{body}.

The statement comprises the derived r_{anchor}, the input and output counts, the nullifiers, the per-output $\mathsf{pk}_{\mathsf{eph}}$ and $\mathsf{ct}^{\mathsf{note}}$ values, and h_{body}. r_{anchor} is the root derived from a currently valid h_{anchor} in the block-level root window; h_{anchor} itself is not a separate proof input. It is bound through the h_{body} of the serialized body that contains that height. The nullifiers are the public spend markers for the input notes. $\mathsf{pk}_{\mathsf{eph}}$ vector and $\mathsf{ct}^{\mathsf{note}}$ vector are the public output fields from which replay derives new note leaves, and the circuit exposes them as verifier inputs because it proves ciphertext/plaintext consistency. h_{body} is the binding hash over the full public transfer body. Other envelope fields, including $\mathsf{ct}^{\mathsf{out}}$, are not exposed individually and stay body-bound through h_{body}. Nullifier uniqueness is enforced by replay rather than by the circuit: an accepted transfer must contain no duplicate nullifiers internally and none already present in the replayed nullifier set.

The canonical envelope order and byte packing used to derive these inputs are fixed in A.3 and A.5.

14.3 Private witness

The witness contains the private data from which the circuit reconstructs spent notes, derives nullifiers, verifies ciphertext relations, and reconstructs spent input note leaves. For each spent input note i, the witness includes $\mathsf{input}_i[v, \mathsf{d}, r_{\mathsf{seed}}, h_{\mathsf{body}}{}^{\mathsf{create}}, \mathsf{creation_output_index}, \mathsf{pk}_{\mathsf{eph}}, \mathsf{ct}^{\mathsf{note}}, \mathsf{pos}, \mathsf{path}^{\mathsf{Merkle}}]$ For

each created output note j, it includes $\mathsf{output}_j[v, \mathsf{d}, \mathsf{pk}_d, r_{\mathsf{seed}}]$. The witness also includes a single $\mathsf{sk}_{\mathsf{spend}}$ that authorizes all input spends; it is not a per-input value.

The witness excludes precomputed note leaves, precomputed nullifiers, and precomputed ρ or $\mathsf{sk}_{\mathsf{eph}}$. Those values should be derived inside the circuit so off-circuit helper logic cannot silently drift from circuit semantics. It also excludes $\mathsf{sk}_{\mathsf{nf}}$ as a free witness value because the circuit derives $\mathsf{sk}_{\mathsf{nf}}$ deterministically from $\mathsf{sk}_{\mathsf{spend}}$.

A concrete deployment must define the supported arities and the verifier interface used for each supported public statement shape. Unsupported counts are invalid. See Appendix A for the suggested implementation choices.

Outside the circuit, wallets and indexers handle note scanning, recipient and outgoing recovery bookkeeping, note selection, Merkle-path retrieval, anchor height selection from the block-level valid-root window, canonical serialization, duplicate nullifier rejection, and other replay checks that do not need arithmetic constraints.

15 Replay and Acceptance

Replay turns the ordered stream of published envelopes into shielded state. This section fixes the properties that acceptance must satisfy; the normative step-by-step order an indexer follows is the indexer acceptance order of Appendix A.7.

Shared validation. Every envelope must first pass a shared validation layer: the binary payload must parse, the magic bytes and version must match, vector lengths must equal the declared `input_count` and `output_count`, curve points must decode, and the h_{body} bound by the proof must equal the value recomputed from the canonical transfer body. For a transfer inside Bitcoin block H, the indexer then checks that h_{anchor} lies in the valid-root window, derives $r_{\mathsf{anchor}} = \mathcal{R}[h_{\mathsf{anchor}}]$ from the active-chain root map, rejects any nullifier that repeats within the envelope or already appears in $\mathcal{N}$, and verifies the proof against the derived root.

Acceptance boundary. Replay mutates canonical state only after every validity condition above has passed. An accepted transfer derives one ℓ per output, appends those leaves to $\mathcal{T}$ and assigns their canonical positions, inserts the published nullifiers into $\mathcal{N}$, and persists the transfer event. A rejected envelope, whether it fails parsing, the binding check, the anchor window, nullifier uniqueness, or proof verification, leaves $\mathcal{S}$ unchanged; it is never partially applied. Two correct indexers processing the same history deterministically apply the same mutations in the same order and derive the same $\mathcal{S}$ (Section 8).

Block finalization. The indexer mutates a working note tree and nullifier set as it processes the transactions of block H in canonical order, and stores $\mathcal{R}[H]$ only after the whole block has been replayed. Blocks with no accepted transfers carry the previous root forward, so every retained height has a well-defined root, as fixed in Section 8.

Reorganization. Replay is defined over the active chain, so a Bitcoin reorganization changes its input. Under a reorganization, $\mathcal{R}[h]$ may change for the first replaced height and every height after it. An envelope whose h_{anchor} referred to a root on the replaced chain is accepted on the new active chain only if its proof still verifies against the root now derived for that height; otherwise the wallet must rescan, recompute positions and Merkle paths, and rebuild the transfer against the new history. The mempool is never replay state: wallets build spend proofs only against roots obtained by deterministic replay of accepted blocks, never against transfers seen only in a mempool.

16 Recovery and Transfer Lifecycle

The wallet-side counterpart to replay is note detection and recovery. After a transfer has been accepted by deterministic replay, a recipient wallet scans published output fields to detect incoming notes using $\mathsf{vk}_{\mathsf{in}}$, while the sending wallet reconstructs outgoing history using $\mathsf{vk}_{\mathsf{out}}$ and $\mathsf{ct}^{\mathsf{out}}$.

Both paths require post-decryption consistency checks against the accepted envelope - successful decryption alone is not sufficient for acceptance. Given $\mathsf{seed}_{\mathsf{wallet}}$ and the accepted Bitcoin history, a wallet can reconstruct its complete state from replayed protocol history and canonical note positions. Those positions may come from local replay or from an indexer following the canonical replay rules. Cached note plaintexts are an optimization, not recovery-critical state.

16.1 Note Detection and Decryption Checks

Decryption itself follows the recipient-channel construction of Section 12.1; this section specifies the checks a wallet applies *after* a candidate ciphertext decrypts.

After decrypting a candidate note, the wallet uses the recovered $\mathtt{d}$ to identify the corresponding local diversified receive address and associated local pk_d, then re-derives the note leaf from the accepted envelope's h_{body}, $\mathtt{output_index}$, $\mathsf{pk}_{\mathsf{eph}}$, $\mathsf{ct}^{\mathsf{note}}$, and h_{aux}. The wallet accepts the note only if the recovered plaintext is consistent with the published ciphertext and the derived note leaf is the leaf assigned by replay.

The wallet also re-derives $\mathsf{sk}_{\mathsf{eph}}$ from r_{seed}, recomputes $G_d = \mathsf{DiversifyHash}(d)$, and checks that the published $\mathsf{pk}_{\mathsf{eph}} = [\mathsf{sk}_{\mathsf{eph}}]G_d$. That binds the recovered $\mathtt{d}$ to the actual diversified receive path used by the sender and prevents accepting malformed plaintext under an unrelated diversifier.

Malformed ciphertexts, malformed point encodings, or accidental decryptions therefore fail note detection even if they produce some byte string. Receipt depends on successful decryption plus ℓ and $\mathsf{pk}_{\mathsf{eph}}$ consistency against the accepted envelope.

16.2 Sender-Side Recovery

To reconstruct sent-payment history, the wallet re-derives the batch-binding digest from the output count and the ordered $\mathsf{pk}_{\mathsf{eph}}$ and $\mathsf{ct}^{\mathsf{note}}$ values of the accepted transfer envelope, re-derives the $\mathsf{vk}_{\mathsf{out}}$-based AEAD key and nonce from that digest, and decrypts $\mathsf{ct}^{\mathsf{out}}$ to obtain the ordered list $(\mathsf{pk}_d,\ \mathsf{sk}_{\mathsf{eph}})_0, \ldots, (\mathsf{pk}_d,\ \mathsf{sk}_{\mathsf{eph}})_{M-1}$.

For each output j, the wallet selects entry j from the decrypted batch, recomputes the note shared secret as $\mathrm{ECDH}(\mathsf{sk}_{\mathsf{eph},j},\ \mathsf{pk}_{d,j})$, re-derives the note key schedule from that shared secret and the published $\mathsf{pk}_{\mathsf{eph},j}$, and decrypts $\mathsf{ct}^{\mathsf{note}}_j$ to recover a candidate note plaintext $(v,\ d,\ r_{\mathsf{seed}})$.

The wallet performs the same post-decryption consistency checks as Section 16.1: it re-derives $\mathsf{sk}_{\mathsf{eph}}$ from the recovered r_{seed}, checks that the published $\mathsf{pk}_{\mathsf{eph},j}$ equals $[\mathsf{sk}_{\mathsf{eph},j}]G_d$, recomputes ℓ_j from h_{body}, output index j, $\mathsf{pk}_{\mathsf{eph},j}$, $\mathsf{ct}^{\mathsf{note}}_j$, and $h_{\mathsf{aux},j}$, and verifies that the result matches the leaf replay assigned to output j. A reconstructed sent note is accepted only if all checks pass. This record is outgoing history only; it does not by itself prove spend authority for that output.

16.3 Validity and Decryptability

Replay validity and wallet decryptability are related but distinct. As defined earlier, ℓ is the replay-derived Merkle leaf derived from proof-checked public output fields: h_{body}, output index, $\mathsf{pk}_{\mathsf{eph}}$, $\mathsf{ct}^{\mathsf{note}}$, and h_{aux}. Recipient-ciphertext consistency is part of transfer validity: the proof must show that the public output ciphertexts encrypt the same output notes used for value conservation.

An accepted transfer may therefore contain outputs that a particular wallet cannot decrypt, cannot recognize as its own, or rejects after local consistency checks, but it cannot contain an output ciphertext unrelated to the note plaintext proved by the transfer circuit. Protocol validity is established by replay and proof verification. Note receipt and sender-side reconstruction are established only by wallet-side decryption and consistency checks against the accepted envelope.

Fee availability for future Bitcoin publication remains a separate operational issue after value has entered the metaprotocol. It sits outside the normative transfer protocol and addresses L1 fee-paying capability separately from transfer validity. See section 22 for further details.

16.4 Wallet Recovery from Seed

Given $\mathsf{seed}_{\mathsf{wallet}}$ and the accepted protocol history replayed from the active Bitcoin chain, a wallet can reconstruct its complete state: received notes, outgoing history, note positions, spent status, and current spendable balance.

Incoming recovery scans accepted transfer envelopes for $(h_{\mathsf{body}}, j, (\mathsf{pk}_{\mathsf{eph}}, \mathsf{ct}^{\mathsf{note}})_j)$ tuples, attempts decryption using $\mathsf{vk}_{\mathsf{in}}$, and applies the detection and consistency checks of Section 16.1. Notes that pass are recorded with their replay-assigned positions. Outgoing recovery scans accepted transfer envelopes for $\mathsf{ct}^{\mathsf{out}}$ fields and applies the sender-side recovery procedure of Section 16.2. Sent notes that pass all consistency checks are recorded. Spendable balance is reconstructed only from unspent incoming notes for which the wallet derives spend authority.

Recovery depends on replayed canonical note positions, not on cached note plaintexts. A wallet recovering from a seed must either perform local replay or consume the output of an indexer treated as truthful.

Part IV

Analysis

This part fixes the ledger-and-replay model used by the security and privacy claims below. Bitcoin supplies publication, ordering, and probabilistic finality for carrier transactions; the shielded state is defined by deterministic replay and proof verification. The corresponding trust boundaries, assumptions, and goals are stated in the style used for privacy-preserving payment protocols.

Let λ be the security parameter. All adversaries are probabilistic polynomial-time algorithms in λ. A function $\mathsf{negl}(\lambda)$ denotes a negligible function.

For a Bitcoin chain view C, let $\mathsf{trim}_k(C)$ be the prefix obtained by deleting the last k blocks. The confirmation depth k is a wallet policy parameter. Let $\mathsf{Env}(C)$ be the ordered list of Shielded Bitcoin envelopes extracted from the Bitcoin transactions in C, using the canonical carrier and parsing rules. Replay is a deterministic partial function. When it is defined on the envelope sequence extracted from C, it returns

$$\mathsf{Replay}(\mathsf{Env}(C)) = \mathcal{S}_C := (\mathcal{T}_C, \mathcal{N}_C, \mathcal{R}_C, \mathcal{H}_C),$$

where $\mathcal{T}_C$ is the note tree, $\mathcal{N}_C$ is the accepted nullifier set, $\mathcal{R}_C$ is the block-root history, and $\mathcal{H}_C$ is the accepted transfer history. For an envelope e, let x_e be its public statement and let w_e be a candidate private witness. The transfer relation is $R_{\mathsf{tr}}(x_e, w_e) \in \{0, 1\}$.

Replay accepts e exactly when parsing succeeds, the public statement is canonical, the proof verifies for x_e, all replay checks pass, and the resulting state transition is defined.

Carrier transactions use the base chain as a publication and ordering service. After the wallet's confirmation policy, Bitcoin supplies only the probabilistic stability of the ordered byte stream from which replay is derived. Shielded validity remains outside Bitcoin consensus: note ownership, value conservation, nullifier correctness, and replay semantics are checked by the metaprotocol, not by miners or full nodes. Accordingly, custody and spend authorization remain at the wallet layer: possession of the user-controlled spending secret is required to authorize a note spend, while indexers and viewing capabilities do not confer spending authority.

Indexers sit outside that consensus boundary. They may provide replay state, envelope data, note positions, and roots, but they do not participate in canonical acceptance. A wallet that accepts an indexer response without replay verification has made a local trust assumption that the selected indexer is truthful for the claimed prefix: $\mathcal{S}_C = \mathsf{Replay}(\mathsf{Env}(C))$. No other user, unrelated indexer, or Bitcoin consensus rule attests to that wallet's synchronized view. A dishonest indexer may censor envelopes, delay updates, serve stale prefixes, or omit data; those failures affect wallets that rely on it, but they do not change the canonical state for a wallet that verifies replay against Bitcoin data.

For funds controlled by a wallet, the wallet remains part of the trusted computing base. Compromise of $\mathsf{sk}_{\mathsf{spend}}$ gives spend authority, and disclosure of viewing material changes visibility according to the key

model. Wallet-side guarantees also depend on fresh note randomness, replay verification where required, and spending only against anchors stable under the chosen confirmation policy.

Several deployed-system properties are intentionally outside the transfer-layer model. Mempool privacy, network-layer privacy, linkage between the carrier transaction and a fee-paying wallet, and entry or exit mechanisms remain outside the model unless specified elsewhere.

17 Assumptions

A1. Bitcoin ledger.

Bitcoin provides eventual publication and a probabilistic total order for carrier transactions. For two honest chain views C_1, C_2, the common-prefix property at depth k fails with probability at most

$$\Pr\big[\neg\big(\mathsf{trim}_k(C_1) \preceq C_2 \ \wedge\ \mathsf{trim}_k(C_2) \preceq C_1\big)\big] \le \epsilon_{\mathsf{btc}}(k),$$

where the bad event is that the two trimmed views are not prefixes of one another, and $\epsilon_{\mathsf{btc}}(k)$ decreases with k under the usual Bitcoin backbone assumptions.

A2. Data availability.

The bytes of accepted envelopes are available from the confirmed Bitcoin history. If an envelope is not retrievable, it is not part of the replay input for a wallet or indexer.

A3. Proof system and setup.

The common reference string is generated by the specified setup procedure. Its honest-generation or ceremony condition is part of the deployed protocol parameters, and the security claims below hold only when the setup is honestly generated and its trapdoor is not available to the adversary. Under this condition the proof system is knowledge-sound for R_{tr}: for every accepting proof an extractor recovers a witness w_e with $R_{\mathsf{tr}}(x_e, w_e) = 1$, except with advantage $\mathsf{Adv}^{\mathsf{ks}}_{R_{\mathsf{tr}}}$. The security lemmas below union this per-proof advantage over the at most q accepted proofs. For the privacy statements the proof system is zero knowledge in the CRS model: the challenger can generate the reference string together with a simulation trapdoor and, with $(\mathsf{SimSetup}, \mathsf{SimProve})$, produce a CRS and proofs computationally indistinguishable from honest ones. The trapdoor is held by the challenger inside the privacy reductions only (P1) and is never available to the adversary in any experiment.

A4. Hash and Merkle binding.

The domain-separated functions H_{body}, H_{leaf}, the KDF and commitment functions, and the nullifier derivation function are collision resistant and domain separated, with collision advantage $\mathsf{Adv}^{\mathsf{cr}}_H$. The note tree is position binding: an adversary cannot produce two leaf-and-path openings for the same position under one accepted root, except by a collision in H_{leaf}, which is counted in $\mathsf{Adv}^{\mathsf{cr}}_H$.

A5. Key derivation and spend binding.

The canonical key-derivation chain from $\mathsf{sk}_{\mathsf{spend}}$ to the diversified receive key pk_d and to $\mathsf{sk}_{\mathsf{nf}}$ is jointly binding: it is infeasible to find distinct spending secrets that induce the same receive authority but different nullifier authority, and infeasible to recover spending authority from public data or from disclosed $\mathsf{vk}_{\mathsf{in}}$ or $\mathsf{vk}_{\mathsf{out}}$. All KDF invocations are domain separated.

A6. Encryption.

Recipient note encryption is correct, confidential, ciphertext-integrity preserving, note-opening robust, and key committing for wallet recovery under the specified Diffie-Hellman, KDF, and AEAD construction; one accepted $(\mathsf{pk}_{\mathsf{eph}}, \mathsf{ct}^{\mathsf{note}})$ cannot satisfy the transfer relation for two different recipient keys or two different note plaintexts. Sender-recovery encryption is correct, confidential, ciphertext-integrity preserving, and key private (sender identity not revealed by the ciphertext), so a sender-recovery ciphertext does not reveal the producing $\mathsf{vk}_{\mathsf{out}}$.

All public points are canonically encoded and lie in the required prime-order subgroup. Two notions of this encryption are used separately below: $\mathsf{Adv}^{\mathsf{aead}}$ is the ciphertext-integrity and key-committing advantage used for recovery (Lemma 3), and $\mathsf{Adv}^{\mathsf{enc}}$ is the key-private semantic-security advantage, covering both the recipient and sender-recovery channels, used for privacy (Theorem 2).

A7. Nullifier derivation.

For secret $\mathsf{sk}_{\mathsf{nf}}$, nullifier derivation is a secure PRF over its domain-separated input containing at least ρ and the canonical note position. The output length is chosen so that collisions among polynomially many honestly or pseudorandomly generated nullifiers are negligible. Collision resistance of this function is covered by A4.

A8. Canonical implementation.

Correct implementations use the same serialization, parsing, transfer relation, proof-verification algorithm, nullifier derivation, note-leaf derivation, and replay ordering. There is no implementation-defined acceptance rule.

A9. Wallet behavior.

Honest wallets keep spending keys and viewing keys under the stated disclosure policy, sample fresh note randomness, verify replay-derived roots and note positions before spending, and choose a confirmation depth k before treating a note as final.

A10. Wallet synchronization.

An honest wallet constructs its shielded view from a claimed Bitcoin prefix C. The view used for spending, recovery, and balance display must equal canonical replay: $\mathcal{S}_C = \mathsf{Replay}(\mathsf{Env}(C))$. The wallet may establish this equality by local replay, by verification of an indexer response against Bitcoin data, or by the explicit indexer-trust assumption in A11.

A11. User-selected indexer trust.

An indexer is *truthful* for a claimed Bitcoin prefix C if it returns exactly $\mathsf{Replay}(\mathsf{Env}(C))$ and the corresponding envelope data. If a wallet consumes an indexer response without replay verification, that wallet treats the selected indexer as a local trusted component and assumes it is truthful for the accepted prefix. The trust assumption is local to the user of that wallet: Bitcoin consensus, other users, and unrelated indexers do not attest to the correctness of the wallet's synchronized view.

18 Security Goals

G1. Replay agreement.

For a correct implementation I on a confirmed Bitcoin prefix C, let $\mathcal{S}_C^I$ be the state it computes. For any two correct implementations I, J on the same prefix C, $\mathcal{S}_C^I = \mathcal{S}_C^J = \mathsf{Replay}(\mathsf{Env}(C))$.

G2. Double-spend resistance.

Let $\mathsf{NF}(e)$ be the ordered set of nullifiers published by an accepted envelope e. For every accepted history $\mathcal{H}$,

$$\forall e \neq e' \in \mathcal{H}: \quad \mathsf{NF}(e) \cap \mathsf{NF}(e') = \varnothing, \qquad |\mathsf{NF}(e)| = \texttt{input_count}(e).$$

Replay rejects repeated nullifiers both within one envelope and against the global accepted set $\mathcal{N}$.

G3. Balance.

For every accepted transfer witness, confidential values are canonical satoshi amounts and conservation holds over integers:

$$R_{\mathsf{tr}}(x_e, w_e) = 1 \quad \Longrightarrow \quad \sum_i v_i^{\mathsf{in}} = \sum_j v_j^{\mathsf{out}} \quad \text{and} \quad 0 \le v_i^{\mathsf{in}}, v_j^{\mathsf{out}} < 2^{b_{\mathsf{amt}}}, ,$$

where b_{amt} is the amount bit length.

For the transfer-only layer, this gives the history invariant $V_{\mathsf{active}}(\mathcal{H}_{\ell+1}) = V_{\mathsf{active}}(\mathcal{H}_\ell)$ whenever $\mathcal{H}_{\ell+1}$ is obtained from $\mathcal{H}_\ell$ by one accepted internal transfer. Entry and exit mechanisms require their own balance accounting.

G4. Body-to-statement binding.

The transfer proof is verified only against the public statement that replay derives from the canonical proof-excluded transfer body, bound through h_{body}, where $h_{\mathsf{body}}(e) := H_{\mathsf{body}}(\mathsf{ser}(e))$ and $\mathsf{ser}(e)$ denotes the salted canonical transfer body $\texttt{const_salt} \,\|\, \texttt{canonical_transfer_body}$ fixed in Section 13.3. Except with negligible H_{body}-collision probability, changing any body field changes the statement against which the proof is verified. The protocol does not require proof-byte non-malleability.

G5. Spend authorization.

If replay accepts a spend of a note n, then the accepted proof attests to a witness containing spending authority for the receive key of n:

$$\mathsf{Accept}(\mathcal{S}, e) = 1 \quad \Longrightarrow \quad \exists\, \mathsf{sk}_{\mathsf{spend}}, d, r_{\mathsf{seed}}, \mathsf{pos} : R_{\mathsf{tr}}(x_e, w_e) = 1.$$

The relation binds the witness to the note leaf, diversified receiving key, Merkle position, and nullifier derivation used by replay.

G6. Wallet recovery.

For an honest wallet seed s and confirmed history $\mathcal{H}$, let $\mathsf{WalletView}_s(\mathcal{H})$ be the canonical wallet view consisting of owned notes, sent-output records, replay positions, nullifier status, and spendable balance. Recovery from s and $\mathcal{H}$ must reproduce that view: $\mathsf{Recover}(s, \mathcal{H}) = \mathsf{WalletView}_s(\mathcal{H})$.

G7. Transfer soundness.

No adversary can make replay accept a transfer whose public statement has no valid witness.

G8. Transcript privacy.

Privacy is defined relative to an explicit leakage function. If two valid private histories X_0 and X_1 have the same leakage, $\mathsf{Leak}(X_0) = \mathsf{Leak}(X_1)$, then a public observer should not be able to tell which history produced the accepted public transcript. The published envelopes and replayed state should reveal only the information already included in Leak. The leakage function and the observable surface are detailed in Section 20.

19 Security

The security goals in Section 18 state the main properties. This section explains why the transfer layer satisfies those goals. The argument is about the history produced by canonical replay of confirmed Shielded Bitcoin envelopes. It is not a claim about an arbitrary wallet database, an arbitrary indexer response, or an unconfirmed mempool view. Peg-in/out, publication policy, networking, mempool behavior, and fee policy belong to deployment and are out of scope and briefly discussed in Section 22.

Fix a Bitcoin prefix C after the wallet's confirmation depth k. Security failure means that canonical replay accepts the history $\mathcal{H}_C$, but one of the non-privacy goals G1–G7 is false. The transcript-privacy goal G8 is treated separately in Section 20.

We argue each goal against an experiment in which a PPT adversary $\mathcal{A}$ outputs a confirmed prefix C; the experiment computes $\mathcal{S}_C = \mathsf{Replay}(\mathsf{Env}(C))$ and tests the stated bad event. Replay agreement is deterministic and stated without an adversary; recovery is stated against an honest-party oracle.

All experiments below share the following setup. The challenger runs the protocol trusted setup to sample public parameters pp (the proving and verifying keys for R_{tr} and the encryption parameters) and gives pp to $\mathcal{A}$. Replay, proof verification, and all derivations are computed relative to pp. Each experiment is parameterized by a PPT adversary $\mathcal{A}$ and the security parameter λ; $\mathcal{A}$ controls every published envelope unless an honest-party oracle is stated. Let $q = q(\lambda)$ bound the number of accepted transfer envelopes in $\mathcal{H}_C$; since $\mathcal{A}$ is PPT and each envelope consumes space in C, q is polynomial in λ. When an experiment states an honest-party oracle $\mathcal{O}_s$, the challenger fixes a wallet seed s, keeps s and its derived secrets unknown to $\mathcal{A}$, and lets $\mathcal{A}$ instruct $\mathcal{O}_s$ to create notes for s and to authorize and publish transfers spending s-owned notes; $\mathcal{A}$ controls all other envelopes. Each goal below is analyzed in its own experiment under this common setup: the non-recovery goals give $\mathcal{A}$ only the prefix C, while recovery (6) and ledger balance (5) additionally give $\mathcal{A}$ the honest-party oracle $\mathcal{O}_s$. The aggregate bound of Theorem 1 sums the advantages of these per-goal experiments.

Theorem 1 (Transfer-layer security)**.** *Fix a Bitcoin prefix C after confirmation depth k. Under A1–A10, for every PPT $\mathcal{A}$, by a union bound over Proposition 1 and Lemmas 1, 2, 3, and 4, together with Corollary 2, the probability that replay accepts $\mathcal{H}_C$ while any of goals G1–G7 fails, or an honest note is spent without authorization, is at most*

$$\epsilon_{\mathsf{btc}}(k) + c_H \cdot \mathsf{Adv}^{\mathsf{cr}}_H(\lambda) + q \cdot c_K \cdot \mathsf{Adv}^{\mathsf{ks}}_{R_{\mathsf{tr}}}(\lambda) + \mathsf{Adv}^{\mathsf{aead}}(\lambda) + \mathsf{Adv}^{\mathsf{auth}}(\lambda),$$

where $q = q(\lambda)$ bounds the number of accepted transfer envelopes in $\mathcal{H}_C$, and $c_H = c_K = 3$ collect the contributing lemmas: the collision terms of Lemmas 1, 3, and 4, and the knowledge-soundness terms of Lemmas 1, 2, and 4. The per-envelope extraction factor q multiplies only the knowledge-soundness term; the collision and AEAD reductions are tight, since any colliding or forging instance among the published objects breaks the single-instance game directly. The additional term $\mathsf{Adv}^{\mathsf{auth}}$ is the honest-note theft advantage of Corollary 2, negligible under A5; its knowledge-soundness contribution reuses the soundness instance already counted in c_K and is not added again.

19.1 Replay, Finality, and Synchronization

Proposition 1 (Replay agreement, G1)**.** *Under A8, any two correct implementations on the same prefix C compute the same $\mathcal{S}_C$. If a wallet builds against C and the active prefix C' satisfies $\mathsf{trim}_k(C) \preceq C'$, the replay inputs below depth k are stable except with probability $\epsilon_{\mathsf{btc}}(k)$ (A1).*

Proof sketch. Once the prefix C, carrier parser, serialization rules, proof-verification algorithm, note-leaf derivation, nullifier derivation, and replay ordering are fixed, A8 leaves no implementation-defined acceptance rule, so correct implementations compute the same $\mathsf{Replay}(\mathsf{Env}(C))$. The only probabilistic term is the prefix assumption: if the wallet later observes an active prefix C' with $\mathsf{trim}_k(C) \preceq C'$, the replay inputs below depth k remain stable except with probability $\epsilon_{\mathsf{btc}}(k)$. If the selected h_{anchor} is not in the stable prefix, the old proof is not assumed valid; the wallet rescans, recomputes positions and Merkle paths, and rebuilds against the active replay state. A verified indexer response is checked against Bitcoin data, while an unverified one is covered only by A11. □

19.2 Nullifiers and Double-Spend Resistance

Definition 3 (Double-spend experiment)**.** *In $\mathsf{Exp}^{\mathsf{ds}}_{\mathcal{A}}(\lambda)$ the PPT adversary outputs a prefix C; the experiment returns 1 $\iff$ replay is defined on $\mathsf{Env}(C)$ and either some distinct $e, e' \in \mathcal{H}_C$ satisfy $\mathsf{NF}(e) \cap \mathsf{NF}(e') \neq \varnothing$ or some e has $|\mathsf{NF}(e)| \neq \mathit{input_count}(e)$. Set $\mathsf{Adv}^{\mathsf{ds}}_{\mathcal{A}}(\lambda) := \Pr[\mathsf{Exp}^{\mathsf{ds}}_{\mathcal{A}}(\lambda) = 1]$.*

Lemma 1 (Double-spend resistance, G2)**.** $\mathsf{Adv}^{\mathsf{ds}}_{\mathcal{A}}(\lambda) \leq \mathsf{Adv}^{\mathsf{cr}}_{H}(\lambda) + q \cdot \mathsf{Adv}^{\mathsf{ks}}_{R_{\mathsf{tr}}}(\lambda)$.

Proof sketch. Each accepted input publishes a nullifier derived from the note-specific ρ, the nullifier key material $\mathsf{sk}_{\mathsf{nf}}$, and the replay-assigned pos. Replay rejects a repeated nullifier within one envelope and against $\mathcal{N}_C$, so disjointness of published nullifiers is exact under A8. The spender does not choose pos; it is assigned when the output leaf is appended to the note tree, so re-publishing the same spend against another h_{anchor} preserves the nullifier. The experiment therefore outputs 1 only if two distinct accepted notes yield equal nf on the distinct inputs forced by their distinct pos (a hash collision, $\mathsf{Adv}^{\mathsf{cr}}_{H}$)or a spend is accepted without a valid input witness, which by knowledge soundness (A3) costs $q \cdot \mathsf{Adv}^{\mathsf{ks}}_{R_{\mathsf{tr}}}$ over the at most q accepted envelopes. □

Because the nullifier binds the replay-assigned pos, distinct accepted notes have distinct derivation inputs even if a sender reuses r_{seed}. This avoids the serial-number reuse failure identified for the original Zerocash construction, where serial numbers combined the recipient key with sender-chosen randomness and so collided under randomness reuse [GGM16]. Disjointness of $\mathcal{N}_C$ is the operative form of double-spend resistance precisely because each note maps to a single nullifier. A witness satisfying R_{tr} fixes $\mathsf{nf} = H_{\mathsf{nf}}(\mathsf{sk}_{\mathsf{nf}}, \rho, \mathsf{pos})$, with $\mathsf{sk}_{\mathsf{nf}}$ re-derived in-circuit from $\mathsf{sk}_{\mathsf{spend}}$, ρ from r_{seed}, and pos the leaf's replay-assigned position; honestly re-spending a note therefore republishes its unique nullifier, which replay rejects against $\mathcal{N}_C$. Producing a second, distinct nullifier for the same note requires either a non-canonical witness, contradicting knowledge soundness ($q \cdot \mathsf{Adv}^{\mathsf{ks}}_{R_{\mathsf{tr}}}$), or two H_{leaf} openings at one position, a collision ($\mathsf{Adv}^{\mathsf{cr}}_{H}$); both are already bounded above. Lemma 1 is therefore equivalent to each accepted note being consumed at most once, the form used in Corollary 2.

19.3 Transfer Validity: Soundness, Balance, and Authorization

Definition 4 (Soundness experiment)**.** *In $\mathsf{Exp}^{\mathsf{snd}}_{\mathcal{A}}(\lambda)$ the PPT adversary outputs a prefix C; the experiment returns 1 $\iff$ replay is defined on $\mathsf{Env}(C)$ and some accepted $e \in \mathcal{H}_C$ has no witness w_e with $R_{\mathsf{tr}}(x_e, w_e) = 1$. Set $\mathsf{Adv}^{\mathsf{snd}}_{\mathcal{A}}(\lambda) := \Pr[\mathsf{Exp}^{\mathsf{snd}}_{\mathcal{A}}(\lambda) = 1]$.*

Lemma 2 (Transfer soundness, G7)**.** $\mathsf{Adv}^{\mathsf{snd}}_{\mathcal{A}}(\lambda) \le q \cdot \mathsf{Adv}^{\mathsf{ks}}_{R_{\mathsf{tr}}}(\lambda)$.

Corollary 1 (Per-transfer balance and spend authorization, G3, G5)**.** *Conditioned on the witness of Lemma 2, value conservation and range (G3) and spending authority for each consumed note (G5) hold, since R_{tr} checks integer conservation, value range, note membership, and re-derivation of the diversified key and $\mathsf{sk}_{\mathsf{nf}}$ from $\mathsf{sk}_{\mathsf{spend}}$. The same $q \cdot \mathsf{Adv}^{\mathsf{ks}}_{R_{\mathsf{tr}}}$ bound applies.*

Proof sketch. Replay does not inspect encrypted values; it accepts a transfer only after proof verification for a public statement whose witness satisfies R_{tr}, which checks note membership, Merkle-path consistency, input ownership, nullifier derivation, value range, and integer value conservation. Acceptance requires a verifying proof for x_e; by knowledge soundness an extractor recovers a witness with $R_{\mathsf{tr}}(x_e, w_e) = 1$ except with $\mathsf{Adv}^{\mathsf{ks}}_{R_{\mathsf{tr}}}$. Applying the extractor to each of the at most q accepted envelopes and union-bounding yields the aggregate $q \cdot \mathsf{Adv}^{\mathsf{ks}}_{R_{\mathsf{tr}}}$. For that witness the circuit re-derives the diversified transmission key from the spending witness and the witness-side diversifier d and constrains it to match the key bound to the note, and derives $\mathsf{sk}_{\mathsf{nf}}$ from the canonical serialization of $\mathsf{sk}_{\mathsf{spend}}$, so unrelated secrets cannot separately satisfy note ownership, spend authorization, and nullifier derivation. $\mathsf{vk}_{\mathsf{in}}$ and $\mathsf{vk}_{\mathsf{out}}$ are viewing capabilities only and are not accepted by R_{tr} as spend authority. An accepted transfer that is unbalanced, out of range, or unauthorized would therefore contradict knowledge soundness of R_{tr}, break key derivation and spend binding (A5), or rely on a non-canonical implementation. □

Definition 5 (Ledger-balance experiment)**.** $\mathsf{Exp}^{\mathsf{bal}}_{\mathcal{A}}(\lambda)$ *gives $\mathcal{A}$ the honest-party oracle $\mathcal{O}_s$ for a seed s whose secrets $\mathcal{A}$ does not see, while $\mathcal{A}$ controls every other published envelope and outputs the prefix C. Let $V_{\mathsf{active}}(\mathcal{H})$ be the total value of notes created and not yet spent in $\mathcal{H}$. The experiment returns* 1 $\iff$ *replay is defined on $\mathsf{Env}(C)$ and either*

1. *some accepted internal transfer in $\mathcal{H}_C$ breaks value conservation or range, so the invariant $V_{\mathsf{active}}(\mathcal{H}_{\ell+1}) = V_{\mathsf{active}}(\mathcal{H}_\ell)$ fails across an accepted internal step; or*
2. *some accepted $e \in \mathcal{H}_C$ spends a note created for s without authorization from $\mathcal{O}_s$.*

Set $\mathsf{Adv}^{\mathsf{bal}}_{\mathcal{A}}(\lambda) := \Pr[\mathsf{Exp}^{\mathsf{bal}}_{\mathcal{A}}(\lambda) = 1]$.

Corollary 2 (Ledger balance and theft resistance, G3, G5)**.** $\mathsf{Adv}^{\mathsf{bal}}_{\mathcal{A}}(\lambda) \le q \cdot \mathsf{Adv}^{\mathsf{ks}}_{R_{\mathsf{tr}}}(\lambda) + \mathsf{Adv}^{\mathsf{auth}}(\lambda)$, *where $\mathsf{Adv}^{\mathsf{auth}}(\lambda)$ is the probability that a PPT algorithm, given pp, the public chain, and the envelopes published by $\mathcal{O}_s$ but not s's secrets, produces a witness authorizing a spend of an s-owned note; it is negligible under A5, A9, and A10.*

Proof sketch. For event (1), Corollary 1 gives $\sum_i v_i^{\mathsf{in}} = \sum_j v_j^{\mathsf{out}}$ and the range bound for every accepted transfer except with $q \cdot \mathsf{Adv}^{\mathsf{ks}}_{R_{\mathsf{tr}}}$. Lemma 1 ensures each created note is consumed at most once, so no value is counted twice. The invariant then follows by induction over accepted internal transfers from the empty history: each accepted internal step removes the input values and adds equal output values, leaving V_{active} unchanged. Entry and exit are excluded by construction and carry their own accounting (G3). For event (2), an accepted spend of an s-owned note has, by Lemma 2, an extracted witness carrying spending authority bound to that note's receive key, position, and nullifier derivation (Corollary 1). Since $\mathcal{A}$'s view excludes s's secrets, producing such a witness either forges the proof ($q \cdot \mathsf{Adv}^{\mathsf{ks}}_{R_{\mathsf{tr}}}$) or recovers s's spending authority from public and oracle data ($\mathsf{Adv}^{\mathsf{auth}}$). Both are negligible under the stated assumptions, so theft of an honest note is negligible. □

19.4 Recovery Integrity

Definition 6 (Recovery experiment)**.** $\mathsf{Exp}^{\mathsf{rec}}_{\mathcal{A}}(\lambda)$ *gives $\mathcal{A}$ an oracle that creates notes and publishes envelopes for an honest seed s whose secrets $\mathcal{A}$ does not see, while $\mathcal{A}$ controls the rest of the published chain. On a confirmed history $\mathcal{H}$, it returns* 1 *iff* $\mathsf{Recover}(s, \mathcal{H}) \neq \mathsf{WalletView}_s(\mathcal{H})$. *Set* $\mathsf{Adv}^{\mathsf{rec}}_{\mathcal{A}}(\lambda) := \Pr[\mathsf{Exp}^{\mathsf{rec}}_{\mathcal{A}}(\lambda) = 1]$.

Lemma 3 (Recovery integrity, G6)**.** *Under A2, A9, and A10,* $\mathsf{Adv}^{\mathsf{rec}}_{\mathcal{A}}(\lambda) \le \mathsf{Adv}^{\mathsf{aead}}(\lambda) + \mathsf{Adv}^{\mathsf{cr}}_{H}(\lambda)$.

Proof sketch. Incoming recovery scans transfers and attempts decryption with the incoming viewing capability; outgoing recovery uses $\mathsf{vk}_{\mathsf{out}}$, the sender-recovery ciphertext, and the ordered output public data. A candidate note enters the wallet view only if the plaintext is consistent with the ciphertext, the

public $\mathsf{pk}_{\mathsf{eph}}$ equals $[\mathsf{sk}_{\mathsf{eph}}]G_d$, and the derived ℓ is the leaf assigned by replay. Malformed ciphertexts, accidental decryptions, or inconsistent plaintexts therefore enter only through AEAD failure ($\mathsf{Adv}^{\mathsf{aead}}$), hash/commitment failure ($\mathsf{Adv}^{\mathsf{cr}}_{H}$), or unavailable history (A2), or via a wallet that omits the checks required by A9 and A10. □

19.5 Envelope Binding and Malleability

Definition 7 (Binding experiment)**.** *In* $\mathsf{Exp}^{\mathsf{bind}}_{\mathcal{A}}(\lambda)$ *the PPT adversary outputs a prefix* C*; the experiment returns* $1 \iff$ *replay is defined on* $\mathsf{Env}(C)$ *and* $\mathcal{H}_C$ *contains distinct accepted envelopes* $e \neq e'$ *with* $x_e \neq x_{e'}$ *accepted under the same transfer proof. Set* $\mathsf{Adv}^{\mathsf{bind}}_{\mathcal{A}}(\lambda) := \Pr[\mathsf{Exp}^{\mathsf{bind}}_{\mathcal{A}}(\lambda) = 1]$.

Lemma 4 (Non-malleability and binding, G4)**.** $\mathsf{Adv}^{\mathsf{bind}}_{\mathcal{A}}(\lambda) \leq \mathsf{Adv}^{\mathsf{cr}}_{H}(\lambda) + q \cdot \mathsf{Adv}^{\mathsf{ks}}_{R_{\mathsf{tr}}}(\lambda)$.

Proof sketch. The public statement contains h_{body}, computed from the canonical serialization of the transfer body, and replay recomputes $h_{\mathsf{body}} = H_{\mathsf{body}}(\mathsf{ser}(e))$ from the published bytes before verification. Changing any public field after proof generation changes the public statement, so an adversary keeping the modified envelope accepted must find a collision in H_{body} ($\mathsf{Adv}^{\mathsf{cr}}_{H}$) or produce a valid proof for the modified statement, costing $q \cdot \mathsf{Adv}^{\mathsf{ks}}_{R_{\mathsf{tr}}}$ over the at most q accepted envelopes. These are the hash and proof-system terms of Theorem 1. □

Proof re-randomization. Lemma 4 bounds *statement* malleability: it forbids moving a proof onto a distinct public statement $x_{e'} \neq x_e$. It does not forbid *proof* re-randomization, which the proof system fixed in Appendix A admits, since from a verifying proof for a fixed x_e an adversary can derive a distinct proof $\pi' \neq \pi$ that still verifies for the same x_e. Because h_{body} is computed over the canonical transfer body with the proof bytes excluded (Section 13.3), π' yields the same body, the same h_{body}, and the same nullifiers as the original envelope. Because every internal transfer publishes at least one nullifier (transfer arity is fixed with `input_count` ≥ 1), the re-randomized copy republishes that nonempty nullifier vector, and replay rejects it as a repeated nullifier under Lemma 1; re-randomization therefore yields no double-spend, theft, or state divergence. It does change the carrier transaction identifier, so txid-level effects such as replacement, third-party rebroadcast before confirmation, or bookkeeping keyed on the txid are publication-layer concerns for the deployment profile (Section 22.3), not properties of the transfer proof.

20 Privacy Model

This section states privacy as a leakage boundary for the transfer protocol. It does not claim that Shielded Bitcoin removes all Bitcoin-level metadata, and it does not give an end-to-end privacy overview for `peg-in`, `peg-out`, fee funding, wallet networking, or relayer operation. The privacy question for this specification is narrower: what remains private inside the metaprotocol once a valid transfer envelope has been published to Bitcoin and accepted by deterministic replay. In definitions of privacy [PH20], the protocol targets *confidentiality* of transfer contents and *privacy* of transfer relations, but not *unobservability*: every transfer publishes an envelope to Bitcoin, so the fact that some transfer occurred, its timing, and its arity are observable by construction. Like Zcash and Monero, Shielded Bitcoin preserves the privacy of who paid whom and how much, not that a shielded transfer happened.

Privacy statements in this section based on a small set of cryptographic and operational assumptions, marked with [**P**·]. These are the assumptions a privacy reduction would actually consume: if any of them is false, the corresponding privacy property breaks.

P1. The transfer proof is zero knowledge for the fixed public statement, so proof bytes do not reveal witness values beyond that public statement.

P2. Recipient note encryption is key-private (recipient identity not revealed by the ciphertext) as well as semantically secure, and sender-recovery encryption is likewise key-private (sender identity not revealed by the ciphertext) as well as semantically secure, under the specified Diffie-Hellman, KDF, and AEAD.

P3. The nullifier derivation is pseudorandom and collision resistant for $\mathsf{sk}_{\mathsf{nf}}$ and ρ, so a public nf cannot be linked to a note leaf without the relevant wallet secrets or disclosed note data.

P4. Wallets use new r_{seed} values and do not reuse note randomness, ciphertext randomness, or deterministic retry outputs across transfers.

P5. Canonical serialization, h_{body} binding, proof verification, and replay rules are implemented consistently by all correct indexers.

P6. Wallet capabilities are not compromised or voluntarily disclosed, except where the disclosure is explicitly part of the participant's view.

The following two scope clarifications are important:

- **Observation model.** Privacy is evaluated with respect to the replayed history of the accepted active chain, after applying the wallet's chosen confirmation policy. Mempool observations, including unconfirmed transfers that may later be replaced or never confirmed, are outside this model.
- **Bitcoin-layer boundary.** This specification does not claim to prevent linkage of a shielded transfer from the Bitcoin wallet or transaction that funds its carrier transaction. Such linkage depends on the separately specified fee-funding, relayer, or unlock model; see paragraph 22.3 and 20.5.

20.1 Formal privacy definition

The target property is that the public transcript reveals nothing about a private history beyond an explicitly listed leakage function Leak. Two valid histories with the same leakage are then indistinguishable to a public observer.

Let X be a valid private transfer history for the metaprotocol, including note plaintexts, wallet secrets, note ownership, input-output relations, and counterparty information. Let $\mathsf{Pub}(X)$ be the public transcript seen by a passive adversary who observes only the accepted active chain after the wallet's confirmation policy: the carrier transactions that publish the corresponding envelopes to Bitcoin and the state produced by replaying that accepted active chain.

We split $\mathsf{Pub}(X)$ into two parts. The leakage function $\mathsf{Leak}(X)$ collects the *forced public schedule*: the facts an honest wallet cannot keep confidential by construction, which the indistinguishability experiment holds equal across both worlds. The remaining published objects are *honest outputs* derived from X, whose joint distribution the privacy theorem (Theorem 2) shows depends on X only through $\mathsf{Leak}(X)$, up to negligible advantage.

The leakage function $\mathsf{Leak}(X)$ is:

- Bitcoin publication metadata: block height, transaction order, carrier transaction structure, fees, feerate behavior, funding inputs, change outputs, and any wallet-clustering information visible at the Bitcoin layer.
- Envelope shape: protocol version, transfer type, h_{anchor} and therefore the root age $H - h_{\mathsf{anchor}}$, `input_count`, `output_count`, and envelope size.
- Replay schedule: acceptance or rejection under canonical replay, the note-leaf insertion order and the assigned tree positions as ordinal indices, and the in-window heights of the valid-root window.
- Boundary metadata: public information revealed by value entering or leaving the metaprotocol. In the current version this includes the public Bitcoin transaction and amount at entry, because `peg-in` and `peg-out` are outside the transfer specification.
- Disclosed capabilities: any $\mathsf{vk}_{\mathsf{in}}$, $\mathsf{vk}_{\mathsf{out}}$, note plaintext, report, or counterparty information that a user gives to the adversary.

The honest outputs not fixed by Leak are the secret-derived published objects: the ordered nullifiers, the ordered $\mathsf{pk}_{\mathsf{eph}}$ values, the ordered $\mathsf{ct}^{\mathsf{note}}$ values and $\mathsf{ct}^{\mathsf{out}}$, the transfer proof bytes, and the derived note-leaf values together with the block-level root values they induce and the resulting h_{body}. Each is computed honestly from the chosen history during the experiment. Under P1–P3 these objects are pseudorandom or simulatable given $\mathsf{Leak}(X)$, which is what Theorem 2 establishes.

Definition 8 (Transcript-privacy experiment)**.** *Use the experiment setup of Section 19: the challenger runs the protocol trusted setup, samples public parameters pp, and gives pp to the PPT adversary $\mathcal{A}$; replay, proof verification, encryption, and all derivations are relative to pp. In $\mathsf{Exp}^{\mathsf{priv}}_{\mathcal{A}}(\lambda)$, $\mathcal{A}$ outputs two valid private*

histories X_0, X_1 that are publicly consistent*: $\mathsf{Leak}(X_0) = \mathsf{Leak}(X_1)$, and X_0, X_1 agree on every component already in $\mathcal{A}$'s view, namely the notes $\mathcal{A}$ owns or controls, the plaintexts and capabilities $\mathcal{A}$ has been given, and the counterparty records $\mathcal{A}$ holds. The challenger samples $b \leftarrow \{0,1\}$, executes X_b honestly relative to pp, publishes the resulting carrier transactions, runs canonical replay, and returns the accepted active-chain transcript $\mathsf{Pub}(X_b)$ after the confirmation policy. $\mathcal{A}$ outputs b', and the experiment returns $1 \iff b' = b$; on a pair that is not publicly consistent it returns a uniform bit. Set $\mathsf{Adv}^{\mathsf{priv}}_{\mathcal{A}}(\lambda) := \left|\Pr[\mathsf{Exp}^{\mathsf{priv}}_{\mathcal{A}}(\lambda) = 1] - \frac{1}{2}\right|$.*

The target privacy statement is that $\mathsf{Adv}^{\mathsf{priv}}_{\mathcal{A}}$ is negligible for every PPT $\mathcal{A}$: for any two valid histories with the same leakage, the published envelopes and replayed state are computationally indistinguishable to a public-chain observer, so the transcript reveals only what is already contained in Leak.

20.2 Transcript-privacy theorem

This is the privacy counterpart to Theorem 1. It is kept separate because it is an indistinguishability claim about what the transcript reveals, not an acceptance claim about what replay admits.

Theorem 2 (Transfer-layer transcript privacy, G8)**.** *Under P1–P6, for every PPT $\mathcal{A}$ and every publicly consistent pair (X_0, X_1),*

$$\mathsf{Adv}^{\mathsf{priv}}_{\mathcal{A}}(\lambda) \le q \cdot \mathsf{Adv}^{\mathsf{zk}}(\lambda) + (n_o + q) \cdot \mathsf{Adv}^{\mathsf{enc}}(\lambda) + n_i \cdot \mathsf{Adv}^{\mathsf{prf}}(\lambda) + \binom{q_{\mathsf{nf}}}{2} 2^{-\ell_{\mathsf{nf}}},$$

where q bounds the number of accepted transfer envelopes, n_o the number of output notes, n_i the number of input notes, and q_{nf} the total number of published nullifiers in the executed history; each is polynomial in λ since $\mathcal{A}$ is PPT and every object consumes space on the carrier chain, and ℓ_{nf} is the nullifier output length. $\mathsf{Adv}^{\mathsf{zk}}$, $\mathsf{Adv}^{\mathsf{enc}}$, and $\mathsf{Adv}^{\mathsf{prf}}$ are the distinguishing advantages against transfer-proof zero knowledge (P1), key-private encryption, i.e. recipient and sender-recovery identity privacy together with semantic security (P2), and nullifier pseudorandomness (P3); the final term is the birthday bound for the uniform nullifier substitution in Game G_3. Assumptions P4–P6 are the freshness, canonical-implementation, and non-disclosure conditions under which these reductions apply rather than separate advantage terms.

Proof sketch. Fix a publicly consistent pair (X_0, X_1), so $\mathsf{Leak}(X_0) = \mathsf{Leak}(X_1)$. We transform the transcript handed to $\mathcal{A}$, through a sequence of games, into one that is a function of Leak alone, so that the challenge bit b becomes information-theoretically indeterminate.

Game G_0. The real experiment: the challenger executes X_b honestly relative to *pp* and returns $\mathsf{Pub}(X_b)$. Components of X_0 and X_1 that coincide on $\mathcal{A}$'s view, including every envelope $\mathcal{A}$ controls, are identical in both worlds and contribute nothing below; the hybrids touch only the components on which the two histories differ. Whenever a later game replaces a published object that feeds h_{body} (a recipient ciphertext, an $\mathsf{pk}_{\mathsf{eph}}$, or a nullifier), the challenger recomputes h_{body}, re-derives the dependent note leaves and block-level roots, and re-simulates the affected proofs for the updated public statement; this re-simulation is well defined because, after G_1, proofs are produced without a witness.

Game G_1 (proofs). Replace each transfer proof with one produced by the zero-knowledge simulator for its public statement. The challenger generates the reference string together with its simulation trapdoor (A3) and uses that trapdoor to simulate; the adversary never sees it. By zero knowledge (P1), a hybrid over the at most q proofs gives $|\Pr[G_0{=}1] - \Pr[G_1{=}1]| \le q \cdot \mathsf{Adv}^{\mathsf{zk}}$. After G_1 the proof bytes are a function of the public statement only and carry no witness information.

Game G_2 (ciphertexts). Re-sample each output's $\mathsf{pk}_{\mathsf{eph}}$ as a fresh ephemeral public key, and replace each recipient ciphertext and each sender-recovery batch with the encryption of a fixed equal-length zero plaintext, produced independently of the underlying key: a recipient ciphertext independently of the recipient pk_d, and a sender-recovery batch independently of the producing $\mathsf{vk}_{\mathsf{out}}$. Recipient and sender identity privacy follow from key privacy, and plaintext confidentiality from semantic security, of the specified Diffie-Hellman and AEAD encryption (P2), with fresh per-output randomness (P4) making each instance independent; a hybrid over the at most n_o recipient ciphertexts and q sender-recovery batches gives $|\Pr[G_1{=}1] - \Pr[G_2{=}1]| \le (n_o + q) \cdot \mathsf{Adv}^{\mathsf{enc}}$, where $\mathsf{Adv}^{\mathsf{enc}}$ bounds the key-private encryption advantage for both channels. After G_2 the $\mathsf{pk}_{\mathsf{eph}}$ values, recipient ciphertexts, and sender-recovery batches depend only on output count and ciphertext length, both fixed by Leak, and in particular no longer depend on the sender's $\mathsf{vk}_{\mathsf{out}}$.

Game G_3 (nullifiers). Replace each published nullifier with a uniform string of the right length. By pseudorandomness of the nullifier derivation keyed on $\mathsf{sk}_{\mathsf{nf}}$ (P3), with fresh ρ per note (P4), a hybrid over the at most n_i inputs gives $|\Pr[G_2{=}1] - \Pr[G_3{=}1]| \le n_i \cdot \mathsf{Adv}^{\mathsf{prf}}$. The substituted strings collide with each other or with any other published nullifier only with the birthday probability $\binom{q_{\mathsf{nf}}}{2} 2^{-\ell_{\mathsf{nf}}}$, which we carry as an explicit additive term; outside that event the acceptance pattern fixed by Leak is preserved, and after G_3 the nullifiers are independent of which note leaf was spent.

In G_3 every secret-derived published object has been replaced by a value that depends on X_b only through $\mathsf{Leak}(X_b)$: the simulated proofs depend on the public statement, the ciphertexts on output count and length, and the nullifiers on nothing beyond their count. The derived note-leaf values, the block-level roots, and h_{body} are deterministic functions of these now world-independent objects under canonical replay (P5), so they too depend on X_b only through Leak; no collision-resistance term is consumed, since the argument needs only that these derivations are deterministic. The remaining transcript fields are the schedule itself, equal by admissibility, and disclosure beyond the stated view is excluded by P6. Hence the G_3 transcript is a function of $\mathsf{Leak}(X_0) = \mathsf{Leak}(X_1)$ alone and is independent of b, so $\Pr[G_3{=}1] = \frac{1}{2}$. Collecting the three hybrid gaps and the G_3 collision term yields the stated bound. □

Inside one accepted transfer, the public statement exposes the selected r_{anchor}, input and output counts, nullifiers, output ciphertext fields, and h_{body}. It does not expose the input note plaintexts, output note plaintexts, input note Merkle paths, $\mathsf{sk}_{\mathsf{spend}}$, $\mathsf{sk}_{\mathsf{nf}}$, $\mathsf{vk}_{\mathsf{in}}$, $\mathsf{vk}_{\mathsf{out}}$, recipient pk_d values, which public note leaves are used as inputs, or the mapping between consumed notes and created notes.

Sender and spent-note privacy. A public nullifier reveals that some note spend was accepted in a particular Bitcoin block, but it should not reveal which earlier note leaf was spent. The anchor root leaks a constraint on the spent note: every input must already exist in $\mathcal{R}[h_{\mathsf{anchor}}]$. A transfer accepted in block H can therefore spend only notes that existed before block H began and, more specifically, notes present under the selected anchor root. The value of $H - h_{\mathsf{anchor}}$ is public, so unusual anchor ages can fingerprint wallets or signing workflows. (P1, P3, P5, P6).

Recipient and output privacy. Output ciphertexts should keep confidential the recipient, value, diversifier, and r_{seed} from an outside observer. The payment address itself is not serialized in the transfer envelope. However, output order and transfer arity are public, and the sender necessarily knows the destination address material and value for the outputs it creates. The protocol therefore protects the confidentiality of recipient information from third-party observers, not from the sender of a payment or from a recipient who receives and decrypts a note. There is also a within-transfer linkage: because each `note` output is connected with transfer's h_{body} (see Section 13.3), all outputs of a single transfer share that h_{body}. Two recipients of the same transfer who compare the h_{body} they each observed can thus tell that they were paid together, and hence that they share a common sender. Wallets that need to avoid it must split unrelated payments across separate transfers. (P1, P2, P4, P6)

Amount and graph privacy. Amounts are encrypted note plaintext fields and are checked by the proof rather than by public replay. For a passive observer, the transfer graph is not disclosed up to the public leakage function above. The observer still sees how many notes were consumed and created, when the transfer was published, which anchor height was used, and which Bitcoin transaction carried the envelope. Thus a `1->2` transfer and a `2->1` transfer are distinguishable even when both keep confidential sender, recipient, and amount. (P1-P6)

20.3 On-chain observable surface

A passive observer of Bitcoin sees more than the abstract transfer statement. The observer sees the raw carrier transaction, its inputs and outputs, fee behavior, transaction identifier, block placement, and the payload. From the payload, the observer can parse every public envelope field and can independently replay the metaprotocol state. Rejected envelopes are also visible as Bitcoin data, even though they do not mutate accepted Shielded Bitcoin state.

The public envelope reveals:

- the protocol header, version, and transfer type;

- h_{anchor} and therefore root age at the accepting block;
- the number of input notes and output notes;
- all nullifiers, $\mathsf{pk}_{\mathsf{eph}}$ values, recipient ciphertexts, and the sender-recovery ciphertext batch;
- proof bytes and total envelope size;
- derived note leaves and assigned positions after replay;
- whether another envelope later repeats a nullifier and is rejected.

The fields above don't reveal confidential note plaintexts or input-output mappings by themselves. They reveal stable metadata that may be combined with Bitcoin wallet clustering, timing analysis, counterparty information, or disclosed viewing capabilities.

20.4 Boundary Events

The transfer protocol begins after value has entered the shielded state and ends before value is released back to ordinary Bitcoin ownership. It therefore cannot keep confidential the public facts created at the boundary. A `peg-in` may reveal the funding transaction, amount, timing, fee behavior, and wallet cluster that supplied the Bitcoin liquidity. A `peg-out` may reveal the exit timing, amount, recipient Bitcoin address, fee behavior, and structure of the unlock transaction. Such information is part of Leak, not part of the confidential transfer history.

For Shielded Bitcoin, the expected boundary construction is PIPE-based. The `peg-in`, `peg-out`, and related unlock rules are meant to be expressed as Bitcoin-anchored metaprotocol transitions rather than as changes to Bitcoin consensus. The transfer protocol specified here then operates on the shielded note state between those boundary transitions. This separation keeps the transfer layer independent of a particular bridge realization, but it also prevents transfer-layer arguments from proving boundary properties.

Boundary constructions differ in what they expose and whom they require users to trust. A federation or multisignature bridge exposes different custody and liveness assumptions from a BitVM-style bridge; a covenant- or vault-based construction, if available, would expose a different set of unlock and policy constraints. A PIPE-based construction must likewise specify its own authorization rules, replay semantics, failure cases, and leakage function. Accordingly, this paper does not claim that entry or exit is trustless, private, free from transaction linkage, or censorship-resistant. Those claims belong to the concrete boundary protocol.

A distinctive `peg-in` amount followed shortly by shielded activity, or a later `peg-out` with correlated amount and timing, can support linkage without breaking note encryption. Repeated use of the same bridge operator, unlock pattern, fee source, or exit service can have the same effect. The transfer layer protects the confidentiality of note contents and input-output relations only relative to this boundary leakage.

20.5 Fee Wallet Linkage

The Bitcoin transaction that carries a transfer envelope must pay miner fees. If those carrier transactions are repeatedly funded from the same transparent Bitcoin wallet cluster, they can be linked at the Bitcoin layer even when the shielded transfer contents remain confidential. This leakage is outside the transfer proof statement: fee inputs, change outputs, and funding history are properties of the publication transaction, not of the shielded witness.

Thus transfer privacy and publication privacy are distinct. The proof may keep confidential the spent note, output plaintexts, amount, and input-output relation, while the carrier transaction still exposes a stable publication pattern. A chain observer can combine fee-input clustering with timing, boundary events, and disclosed counterparty information. Such linkage does not affect transfer validity, but it belongs to Leak.

Carrier fees can be funded separately from the user's ordinary Bitcoin wallet. A dedicated fee wallet creates a distinct funding path, although repeated use or predictable top-ups may still reveal a link. Relayers, batching services, and sponsors move publication or fee payment into shared infrastructure. These arrangements can disassociate the direct wallet link, but the model must state who pays the fee, what transaction and network metadata each service observes, and how publication proceeds when that service is unavailable.

The intended PIPE-based direction is different from treating fees as an external relayer problem. A user may prove authorization to a PIPE-controlled fee vault whose Bitcoin liquidity is reserved for, or otherwise attributable to, that user's publication costs. When the vault rule verifies, the vault releases the Bitcoin needed to fund the carrier transaction. In this design, fee-paying capability is derived from metaprotocol state rather than from repeated use of the user's ordinary transparent wallet cluster.

This fee-unlock path does not require the publisher to learn the private transfer contents. The user constructs the transfer envelope and proof locally; the publisher sees only the public envelope and the data required by the fee-unlock rule. It need not know the spent notes, output plaintexts, values, recipient addresses, or wallet secrets. Its role is publication, not custody of shielded value or validation of the private transfer beyond the public rules it can check.

A PIPE-based fee vault therefore reduces direct reuse of a transparent fee wallet, but it does not eliminate publication leakage. Vault access patterns, reimbursement structure, publication timing, and interactions with a publisher or relayer may still become observable metadata. The fee-unlock mechanism therefore requires its own leakage analysis.

20.6 Privacy set

A privacy set is always defined relative to an adversary, an observation model, and a secret of interest [PH20]. For Shielded Bitcoin, the relevant secret is not a user identity in isolation. On the spend side, it is the note leaf consumed by a public nullifier. On the output side, it is the recipient-controlled note plaintext and the private history that produced it. The privacy set is therefore not the set of all Shielded Bitcoin users, nor the set of all notes ever created. It is the set of private histories that remain compatible with the public transcript, the adversary's auxiliary information, and any disclosed viewing keys, note plaintexts, reports, or counterparty records.

This distinction is important in a note-based system. A large global note tree gives only a syntactic upper bound on the size of the spend privacy set. It may substantially overstate user-level privacy if a small number of actors created most notes, if the adversary controls or has learned many notes, or if a wallet repeatedly uses distinctive publication patterns. Conversely, note-level privacy may remain meaningful even when the observer already knows that all plausible notes belong to the same user: in that case the protocol may still keep confidential note plaintexts and input-output links, but it provides no participant privacy among users.

Spend-side candidates. Consider a transfer accepted in Bitcoin block H with $h_{\mathsf{anchor}} = a$. For one spent input, the protocol-level candidate set is bounded by the note leaves already committed under $\mathcal{R}[a]$. This is analogous to the public commitment set accumulated before a Zerocoin spend [MGGR13] and to the Zcash note traceability set for notes under a selected anchor [HBHW25]. The proof establishes membership of some leaf under that root, and the nullifier marks one such note as spent, but the public transcript does not identify which leaf was used. [**P1**,**P3**,**P5**,**P6**] Under the transfer assumptions, a passive observer with only the accepted public transcript cannot link the nullifier to a particular leaf inside this candidate set.

The candidate set is reduced by auxiliary knowledge. Leaves controlled by the adversary, leaves whose plaintexts or viewing data have been disclosed, leaves known through off-chain counterparty records, and leaves inconsistent with boundary events, timing, fee behavior, or wallet policy should not be counted as plausible candidates. For a transfer with N public inputs, the spend-side candidate space is a set of eligible N-tuples rather than a single note set 20.2.

Output-side candidates. For a created output, the relevant candidate set is not the anchor tree. It is the set of outputs, recipients, and private histories that could have produced the same observable output transcript. This transcript includes transfer arity, output index, publication time, carrier behavior, fee source, $\mathsf{pk}_{\mathsf{eph}}$, ciphertext length, and any disclosed counterparty context. [**P1**,**P2**,**P4**,**P6**] Recipient ciphertext privacy keeps confidential the recipient, diversifier, value, and r_{seed} only among outputs that remain compatible with this leakage. The sender and the recipient are excluded from this public-observer privacy claim, since they know the transfer they created or received.

Bounds. Let $\mathcal{T}_a$ denote the set of all note leaves already present in the global note tree under the anchor root selected by $h_{\mathsf{anchor}} = a$. For a single spent input, the largest possible spend-side privacy set is bounded by this tree prefix:

$$|\mathcal{A}^{\mathsf{spend}}_{\max}(a)| \leq |\mathcal{T}_a|.$$

For N inputs, the tuple-level upper bound depends on whether the concrete circuit and wallet policy treat input order as meaningful. Since this specification does not fix a normative input-ordering policy, it does not claim a stronger closed-form tuple count.

This upper bound may collapse to a singleton. If only one eligible note exists under the selected root, or if side information rules out all candidates except one, the privacy set has size one. The same phenomenon occurs at the user level when all plausible notes belong to one known actor. Such cases do not contradict note confidentiality or ciphertext privacy; they show that cryptographic confidentiality and deployed privacy are different claims.

Effective privacy. Cardinality alone is a weak measure when candidates are not equally likely. Following entropy-based privacy-set metrics [SD03], the effective privacy set is

$$|\mathcal{A}_{\mathsf{eff}}| := 2^{H(P)},$$

where P is the adversary's posterior distribution over candidates and $H(P)$ is Shannon entropy in bits. The effective size equals the raw candidate count only when the posterior is uniform. If public metadata or auxiliary information concentrates probability mass on a small subset of candidates, the effective privacy set shrinks accordingly. This is the same practical lesson found in empirical analyses of shielded systems: a large nominal pool does not by itself imply a large effective privacy set [KYMM18].

For Shielded Bitcoin, the adversary's posterior distribution P is conditioned on the accepted transfer transcript and on auxiliary observations captured by Leak. The transcript exposes the transfer arity and the selected anchor height. Publication timing, fee funding, boundary activity, and network observations can further distinguish candidate histories.

The timing and traffic-shape issues are not unique to payment systems. Mix-network analyses such as `Loopix` show that privacy depends on the distribution of concurrent traffic and, where available, on cover traffic and delays [PHE+17]. This specification does not define cover traffic, delay mechanisms, batching rules, arity normalization, or a normative wallet policy. Those mechanisms may improve deployed privacy, but they cannot be counted as properties of the transfer layer unless they are specified and analyzed as part of a deployment profile.

The resulting claim is conditional. **[P1–P6]** Shielded Bitcoin keeps confidential the spent note and output plaintext within the set of histories compatible with the leakage function. The size and distribution of that set are deployment properties. A stronger claim, such as a uniform lower bound on user privacy, would require adversary model, and adoption assumptions.

20.7 Out-of-scope

This section's privacy claims cover only the shielded note state between boundary transitions: the creation, spending, and recovery of notes, and the leakage of the accepted transfer transcript characterized by Leak. Several surrounding components materially affect deployed privacy but are not covered by the transfer proof and do not follow from Theorem 2: entry and exit, fee publication, network-layer observation, wallet policy, and implementation parameters. The boundary and fee-publication leakage surfaces are characterized in Sections 20.4 and 20.5; the full out-of-scope agenda for these components is collected in Section 22.

Part V
Closing Remarks

21 Summary

The protocol represents value as shielded notes. Each sender uses their wallet to construct a transfer envelope and submit it for publication on Bitcoin, directly or through a publication service. Indexers reconstruct the protocol state by replaying accepted envelopes. This specification is focused on private value transfer, without assuming in-consensus proof verification by Bitcoin or treating `peg-out` as trustless.

It defines an architecture for private Bitcoin-denominated transfers based on notes, ciphertext-derived note leaves, nullifiers, viewing keys, recoverable wallets, and replayable state anchored in Bitcoin publication. Concrete implementation parameters, including proving-system choice, hash parameters, point encodings, and byte-level formats, are deferred to the normative appendices and implementation-specific materials. `Peg-in` and `peg-out` are relevant to a full deployment, but they are left to a separate PIPE v2-focused treatment.

22 Future Work

The current version (see Appendix A) of Shielded Bitcoin fixes one transfer statement, one canonical envelope encoding, and an `OP_RETURN` publication path. Several of those choices are deployment parameters rather than properties of the transfer relation, and their best values depend on analysis that is out of scope here. This section collects the open deployment parameters, the question of light-client sync, and the surrounding components whose security does not follow from the transfer proof.

22.1 Deployment parameters

The proof system, publication carrier, transfer arity, and the contents of the public statement are not independent. They trade off envelope footprint, proving cost, and the size of the privacy set, and a deployment profile fixes them together rather than one at a time.

Proof system. The transfer relation does not depend on the concrete succinct proof system, and the main differences between candidates are proof size and setup assumptions. Groth16 produces proofs of about 192 bytes, while more recent constructions such as Polymath [Pol24] and Pari [DMS24] report about 176 and 160 bytes respectively. Because a proof is published in every envelope, these differences feed directly into the footprint figures and interact with the carrier choice below. Each candidate also carries its own setup and security assumptions, and a circuit-specific system needs one trusted setup per supported arity, which links this choice to the arity decision. The choice can affect the analysis as well: a system without Groth16's proof re-randomization would remove the re-randomization caveat in the security analysis (Section 14.3). The implementation appendix records the current default.

Publication carrier. The current implementation profile specifies a single `OP_RETURN` output as the carrier for the complete transfer envelope. A witness-carried alternative moves the envelope bytes into transaction witness data at roughly a quarter of the weight cost, at the price of a two-transaction prepay-and-reveal flow and its own standardness assumptions. The carrier choice is coupled to the proof system, since both determine how the envelope fits inside the available data-carrier budget, and to the deployment risk that relay or mining policy tightens around large carriers. None of these options change the transfer statement; they change footprint, fee behavior, failure modes, and deployability. The footprint appendix compares concrete publication profiles.

Transfer arity and the standard format. Input and output counts are public, so observers can distinguish a 1-to-2 transfer from a 2-to-1 transfer. Using one standard arity would prevent classification by these counts, but transfers with fewer notes would require padding.

The supported arities also determine the circuit shape and verifier interface (Section 14.3). A 2-to-2 format is a plausible default because it supports a payment with change, but we do not select a standard arity here. That choice requires measurements of expected transfer patterns and the concrete cost of padding.

Statement scope. A related choice is how much to bind into the public statement and prove in-circuit. Binding more structure reduces the number of separately published fields and the work a wallet redoes during local replay, which helps footprint and local storage. It pulls against proving cost, since a larger relation means a heavier circuit and proving key, and against constructions layered on top of the transfer relation. The right balance depends on the chosen proof system and on which extensions a profile intends to support.

22.2 Light clients and verified sync

A wallet needs note positions, Merkle paths, and a correct block-level root to build a spend proof, and today it can obtain a trustworthy view in only three ways: full local replay of the Bitcoin chain, verification of an indexer response against Bitcoin data, or an explicit trust assumption on a selected indexer (assumption A11). Full replay is expensive, and the trust assumption weakens the guarantee for wallets that cannot replay. A user who only wants to find and spend their own notes scans accepted outputs with the incoming viewing key, but still depends on a root they cannot cheaply check.

The open question is whether a light client can close this gap with a succinct proof that $\mathcal{R}[h]$ is the correct deterministic replay of the accepted envelopes up to height h, verifiable without re-executing replay. Such a proof, for example a recursive or proof-carrying construction over the replay function, would let a view-only wallet verify root correctness for its own notes while preserving replay agreement (goal G1) and without elevating an indexer to a trusted component. A light client of this kind reduces sync and trust cost; it does not reduce the on-chain cost of spending, since a spend still publishes a carrier transaction and pays its Bitcoin fee. The proving cost, update frequency, and proof size of such a construction are open.

22.3 Analysis Out-of-scope

This specification defines the transfer layer of Shielded Bitcoin, not a complete deployed payment system. Its claims apply to the representation of Bitcoin-denominated value as shielded notes after entry into the metaprotocol, to the creation and spending of those notes, to nullifier-based double-spend prevention, to wallet recovery from encrypted outputs, and to deterministic replay of accepted transfer envelopes into a common shielded state. The surrounding system contains additional components whose design choices affect trust, privacy, liveness, and deployability, but whose security analysis does not follow automatically from the transfer proof.

Peg-in and peg-out. The `peg-in` and `peg-out` constructions are not specified here. As discussed in Section 20.4, the expected direction for Shielded Bitcoin is to use PIPE-based boundary transitions: ordinary Bitcoin liquidity is committed on L1, while entry, exit, and unlock rules are represented as Bitcoin-anchored metaprotocol transitions rather than as changes to Bitcoin consensus. Federated or multisignature custody, BitVM-style bridges, and future covenant- or vault-based constructions remain useful comparison points, but the intended architecture is PIPE-based. The trust, liveness, censorship-resistance, and privacy properties of entry and exit therefore belong to the boundary protocol, not to the transfer proof.

Fee publication and fee privacy. The transfer proof does not specify how the carrier transaction is funded. Possible models include self-funding from a transparent wallet, third-party fee payment, batching, and sponsored publication. The intended PIPE-based direction is a fee-unlock vault 20.5: the user proves authorization to access Bitcoin liquidity reserved for publication costs, so that fee-paying capability is derived from metaprotocol state rather than repeated use of the user's ordinary transparent wallet cluster. The fee is public and scales with envelope size, so its magnitude depends on the transfer arity, the proof system, and the carrier, multiplied by the prevailing Bitcoin fee rate. This ties the fee to the same

parameters of Section 22.1, and an unusual fee or funding pattern can fingerprint a wallet just as anchor age or arity can. The incentive to pay is direct, since publication is what buys ordering and acceptance into replay, but the vault access pattern, publication timing, reimbursement structure, and publisher or relayer interaction still require their own leakage analysis.

Network observation, mempool, and retry behavior. The accepted-chain privacy model covers envelopes accepted by deterministic replay of the active Bitcoin chain. It does not cover a stronger adversary that observes transaction origination, mempool propagation, failed broadcasts, replacement attempts, or envelopes that appear only in orphaned blocks. Such observations can reveal timing, anchor age, arity, fee strategy, and retry patterns even when the corresponding transfer is never accepted into replay. The mempool is never replay state, but it is still an observation surface. An observer can predict pending transfers from mempool contents before they are accepted, and an adversary can spam the mempool to fingerprint a wallet's publication behavior or to perturb the fee and anchor-age conditions under which honest transfers are rebuilt and rebroadcast.

This leakage is a network-layer problem. Cryptocurrency-specific broadcast schemes such as Dandelion and Dandelion++ attempt to reduce transaction-origin leakage in peer-to-peer propagation [VFV17, FVB+18]. More general privacy-enhancing communication systems, including onion-routing systems such as Tor and mixnet designs such as Loopix, provide different trade-offs between latency, traffic-analysis resistance, cover traffic, and deployment complexity [DMS04, PHE+17]. Similar mechanisms could be used around Shielded Bitcoin publication, for example through delayed broadcast, proxy submission, relayer networks, mixnet-style transport, or privacy-preserving publication services. However, such mechanisms require a separate network adversary model and are not implied by the zero-knowledge transfer proof.

Wallet policy. Wallet behavior can materially affect privacy even when the cryptographic transfer protocol is sound. Input selection, output arity, change placement, anchor selection, publication timing, relayer choice, batching, and retry construction all shape the adversary's posterior over candidate histories. A wallet that uses distinctive arities, stale anchors, deterministic change positions, repeated fee sources, or deterministic rebuilt transactions can reduce the effective privacy set without violating any transfer-validity rule. A deployment profile may standardize common arity patterns, fix or randomize change placement, select anchors according to a shared policy, require fresh randomness for every rebuilt transfer, and avoid publication behavior that fingerprints a particular wallet. Arity is the most consequential of these levers and is treated on its own in Section 22.1, because its effects reach beyond wallet policy into the circuit and the trusted setup.

Implementation, serialization, and upgrades. Beyond the parameters above, a deployment profile must fix the remaining concrete encodings, the curve and hash parameters, and the verifier interfaces, and must define an upgrade and versioning procedure with compatibility rules across versions. These are necessary for an implementation, but their analysis belongs to the implementation profile or to separate specifications rather than to the transfer proof.

22.4 Opt-in KYC and Certified-Recipient Lineage

A future deployment could let wallets attach an encrypted lineage attestation to an output created for a Trust-Authority-certified payment address. Without revealing earlier addresses or the transfer graph, the proof would show that each lineage root came from an authenticated peg-in whose actual Bitcoin inputs were approved, that every later preserving transfer consumed only notes carrying the same claim, and that each claim-preserving transfer output used the certified `PaymentAddress`. An institution could verify this evidence locally and bind it to a fresh operation challenge. Bitcoin consensus and base replay would not interpret it, and notes without the evidence would remain valid. This direction is discussed further in Appendix D.

22.5 Protocol Extensions via the Reserved Leaf Slot

The note leaf reserves an unused commitment slot, h_{aux} (Section 9), which fixed to `aux_null` and does not serialize. This slot is the designated binding point for extensions that must commit auxiliary data to

a note at creation time without placing that data in the note plaintext. The transfer layer specified here does not interpret the slot: an extension defines its own opening relation, proves it in an extended circuit, and carries any additional public data in its own envelope format rather than in the base e.

The motivating example is an atomic swap between a shielded note and transparent Bitcoin, mediated by a liquidity provider and a standard Bitcoin HTLC. Such a swap commits a hash-lock in h_{aux}, where one secret unlocks both the shielded note and the HTLC and an off-chain binding proof ties the circuit-native and Bitcoin hash contexts to a single preimage.

A Implementation Profile and Recommended Defaults

This appendix fixes one concrete implementation profile for Shielded Bitcoin: a bundle of choices that, taken together, let independent implementations interoperate. The transfer relation itself does not depend on these choices, it is agnostic to the concrete proof system, publication medium, and byte-level encodings. Interoperability, however, requires every replaying implementation on a given network to share the same bundle, because replay's acceptance predicate includes the canonical serialization (A.5), the envelope body and arity layout (A.3), the carrier-location rule (A.4), proof verification under a fixed verifying key (A.1), the anchor rule W fixed in Section 8 with its network-wide minimum depth $K_{\min}$, and the deterministic acceptance order (A.7). Two implementations that differ on any of these, including which proof system they verify, accept different envelope sets and derive divergent shielded state from the same chain. A different deployment may define an alternative profile (say, another proof system or a witness-carried envelope), but that is a separate, non-interoperable network, not a per-implementation option. Only a few choices are genuinely local and never affect interoperability: internal storage layout, wallet seed entropy (A.2), and the wallet-side anchor depth K_{wallet} (A.6). The normative transfer-circuit decomposition is given in Appendix B.

A.1 Operational Context

Groth16 requires a circuit-specific setup. For `Transfer`, that means freezing the arithmetic circuit, running a trusted setup ceremony, publishing the proving key, publishing the verifying key, and associating that key material with the corresponding verifier identifier. Any circuit modification changes the setup and therefore changes both keys.

The prover is the wallet, or an external proving service used by the wallet, which means every indexer maintaining a protocol state acts as a verifier. Bitcoin consensus does not verify the `Transfer` proof. The wallet assembles witness data and public inputs, runs Groth16 proving, and publishes the transfer envelope to Bitcoin. Indexers then extract the envelope, recompute h_{body} from the canonical body, validate the selected h_{anchor}, derive r_{anchor}, derive the public statement, verify the Groth16 proof, derive output note leaves, and only then update the note tree and nullifier set. Indexers enforce transfer validity by recomputing the public statement, verifying the proof, and mutating replay state only after all checks pass.

Proof generation happens only after the wallet has synchronized protocol state, selected input notes, fetched note positions and Merkle paths, chosen a valid h_{anchor} and corresponding block-level root, constructed output notes and ciphertexts, derived the nullifiers, assembled the final public envelope, and computed h_{body}.

The protocol statement still depends on concrete implementation choices, including the exact in-circuit treatment of h_{body}, the note-encryption gadget used for ciphertext/plaintext consistency, the Merkle-tree hash function and path layout, and the curve and backend choices used for Groth16. Those parameters should be fixed by the implementation specification and associated with the corresponding verifier identifier. The AEAD constructions must meet the encryption assumptions of A6, which the security and privacy proofs consume. A vanilla AEAD is insufficient: the recipient channel must be a committing AEAD (key committing and note-opening robust, so one ciphertext cannot open under two keys or to two plaintexts) and key private (recipient identity not revealed by the ciphertext), and the sender-recovery channel must be key private ((sender identity not revealed by the ciphertext) as well as confidential and ciphertext-integrity preserving. The profile should therefore fix a context-committing AEAD together with a key schedule whose output is pseudorandom in the recipient and sender-recovery keys.

A.2 Wallet Seed and Diversifier Size

Wallet root input is a uniformly random 32-byte $\mathsf{seed}_{\mathsf{wallet}}$. Low-entropy passwords are not suitable wallet seeds.

Payment addresses use an 11-byte diversifier d. The diversifier selects the diversified receive base used to form pk_d, and the note plaintext carries d so the recipient can map a decrypted note back to the local receive path. The size of d is an address-namespace and encoding choice. It gives wallets enough

variability to allocate many receive paths under one viewing key while keeping payment addresses and note plaintexts compact. It is not the standalone security parameter for note detection or spending.

A.3 Transfer Envelope Body

Let N be the number of notes spent and M new created notes. A transfer envelope is an ordered pair $e := \big(\mathsf{Body}(e),\ \mathsf{proof}(e)\big)$, where

$$\mathsf{Body}(e) := (\mathsf{header},\ h_{\mathsf{anchor}},\ N,\ M,\ \mathsf{nf}_{0..N-1},\ \mathsf{pk}_{\mathsf{eph},0..M-1},\ \mathsf{ct}^{\mathsf{note}}_{0..M-1},\ \mathsf{ct}^{\mathsf{out}}).$$

Order	Field	Amount	Replay role
0	`header`	one	Magic bytes, version, and addition data.
1	h_{anchor}	one	Selects the block-level root used as the spend anchor.
2	`input_count`	one	Defines N, the number of published nullifiers.
3	`output_count`	one	Defines M, the number of published output channels.
4	nf values	N entries	Public spend markers checked against the nullifier set.
5	$\mathsf{pk}_{\mathsf{eph}}$ vector	M entries	Per-output ephemeral public keys for note decryption.
6	$\mathsf{ct}^{\mathsf{note}}$ vector	M entries	Recipient ciphertexts for encrypted note plaintexts.
7	$\mathsf{ct}^{\mathsf{out}}$	one batch over M outputs	Sender-recovery ciphertext authenticated as one batch.
8	π	one	Proof for the statement derived from the body.

Table 3: *Logical transfer-envelope body and proof fields.*

Each $\mathsf{ct}^{\mathsf{note}}_j$ encrypts a recipient plaintext $(v,\ d,\ r_{\mathsf{seed}})$. The sender-recovery plaintext is the ordered per-output list $(\mathsf{pk}_d,\ \mathsf{sk}_{\mathsf{eph}})$, authenticated by one AEAD tag for the full batch.

A.4 Publication Carrier

The carrier rule tells indexers where the envelope bytes are located in a Bitcoin transaction. It does not give Bitcoin consensus any shielded validation role. Under this profile, exactly one `OP_RETURN` output carries the complete binary envelope. Fragmentation across several outputs, witness-carried publication, and alternative carrier locations are not part of this profile. Indexers ignore witness bytes when extracting shielded transfer envelopes. The envelope must fit inside the selected `OP_RETURN` carrier. An oversized envelope is invalid under this profile. Bitcoin orders and publishes the carrier transaction; parsing, proof verification, and shielded-state mutation remain replay rules.

Single-output `OP_RETURN` publication is a relay and mining-policy assumption. Bitcoin consensus does not guarantee relay or mining of such outputs. The assumed policy environment supports `OP_RETURN` transport with the relaxed `-datacarriersize` default introduced in `Bitcoin Core v30.0`, which raised the default from 83 to 100,000 bytes (effectively uncapped, since the standard transaction-size limit binds first) and additionally permits multiple `OP_RETURN` outputs per transaction. This default remains node-configurable and is not universally adopted: operators may restore the historical limit with `-datacarriersize=83`, so single-output `OP_RETURN` relay of envelopes above the legacy cap depends on a sufficient share of relay and mining nodes running the relaxed policy. As with any standardness assumption, this is a deployment property to monitor, not a consensus guarantee.

Current size estimates. For the 2-input, 2-output configuration considered here, the serialization uses a 6-byte header, a 4-byte anchor, two one-byte counts, 32-byte nullifiers and compressed ephemeral keys, a 67-byte recipient ciphertext per output, a 144-byte sender-recovery batch (two 64-byte entries and one 16-byte tag), and a 192-byte proof. The complete serialized protocol envelope is therefore

$$E_{2\to 2} = 6 + 4 + 2 + 2(32) + 2(32) + 2(67) + 144 + 192 = \mathbf{610\ bytes}.$$

For this envelope size, the minimally pushed `OP_RETURN` script is 614 bytes and its complete serialized Bitcoin output is 625 non-witness bytes. Each envelope byte therefore costs four weight units, or one virtual byte, in this carrier. Virtual size applies to a Bitcoin transaction, not to the stand-alone envelope itself.

For comparison, fix a concrete witness-carrier template. A prepay/commit transaction spends one P2TR key-path input and creates two P2TR outputs: the script-path commitment and change. The reveal spends the commitment through a single-leaf P2TR script path and creates one P2TR output. Both transactions use 64-byte Schnorr signatures with `SIGHASH_DEFAULT`, empty `scriptSig` fields, and no annex. The tapscript checks a 32-byte x-only public key and places the envelope in an unexecuted `OP_0 OP_IF ... OP_ENDIF` branch. Splitting the envelope into 520-byte and 90-byte pushes gives a 652-byte tapscript. The single-leaf control block is 33 bytes.

Object	Raw bytes	Weight	Virtual size
Protocol envelope	**610**	—	—
Complete `OP_RETURN` output	**625**	2,500 WU	**625 vB**
Witness prepay/commit transaction	**205**	616 WU	**154 vB**
Witness reveal transaction	**851**	1,133 WU	**284 vB**
Two-transaction witness total	**1,056**	1,749 WU	**438 vB**

Table 4: *Carrier size for the 2-input, 2-output envelope.*

A.5 Canonical Serialization

Canonical serialization is an injective map from logical envelopes to bytes. A parser accepts a byte string only if it has one canonical decoding.

$$\begin{aligned}\mathsf{body_bytes}(e) := \mathsf{ser}(\mathsf{header}) \\ \| \ \mathsf{ser}(h_{\mathsf{anchor}}) \| \mathsf{ser}(N) \| \mathsf{ser}(M) \\ \| \ \mathsf{ser}(\mathsf{nf}_{0..N-1}) \| \mathsf{ser}(\mathsf{pk}_{\mathsf{eph},0..M-1}) \\ \| \ \mathsf{ser}(\mathsf{ct}^{\mathsf{note}}_{0..M-1}) \| \mathsf{ser}(\mathsf{ct}^{\mathsf{out}}).\end{aligned}$$

The canonical encoding rules are:

- fixed integers use the assigned little-endian width for the field;
- vector counts use Bitcoin-style canonical `CompactSize`;
- field elements use fixed-width canonical field encodings;
- curve points use one compressed canonical encoding;
- ciphertext fields use the fixed length selected by the envelope format;
- vectors are serialized in increasing index order.

A parser rejects shorter, longer, non-minimal, reordered, undecodable, or otherwise non-canonical encodings. It also rejects vector lengths that do not match N or M. The π field is serialized after $\mathsf{body_bytes}(e)$.

A.6 Anchor Window

Section 8 defines the valid-anchor window in terms of W and $K_{\min}$. The recommended constants for this implementation profile are

$$W := 100, \qquad K_{\min} := 1.$$

With these constants, replay accepts a candidate transfer in block H only when its anchor height $a = h_{\mathsf{anchor}}$ satisfies

$$H - 100 \le a \le H - 1.$$

The value $W = 100$ gives wallets a bounded publication window after proof generation and before inclusion. At Bitcoin's target block interval, this is about sixteen hours. This covers ordinary fee estimation, rebroadcast, delayed inclusion, and simple relayer workflows. It also keeps the retained root history small.

The value $K_{\min} = 1$ is the smallest depth for which the anchor root is already defined when block H is replayed. It permits $\mathcal{R}[H-1]$ as an anchor. It forbids $\mathcal{R}[H]$, which is written only after block H has finished replaying. This also prevents a note created in block H from being spent later in the same block.

For block H, an indexer checks h_{anchor} against the window before proof verification. It then derives r_{anchor} from $\mathcal{R}[h_{\mathsf{anchor}}]$. An expired anchor invalidates the transfer even if the arithmetic proof verifies against the old root.

Wallet software should still use a stricter local anchor policy. The recommended default is to choose anchors at depth $K_{\mathsf{wallet}} > K_{\min}$. The wallet selects K_{wallet} according to its reorganization risk tolerance. This is wallet policy, not a replay rule. $K_{\min} = 1$ defines what an indexer may accept. K_{wallet} defines what a wallet is willing to build. Wallets should use a common anchor-selection policy. Unusual anchor ages create avoidable metadata differences.

A.7 Indexer Acceptance Order

Replay logic has to be deterministic at the acceptance boundary and in state transition behavior. Replay state is mutated only after all envelope validity conditions have passed. Figure 5 gives the normative dependency order for replay.

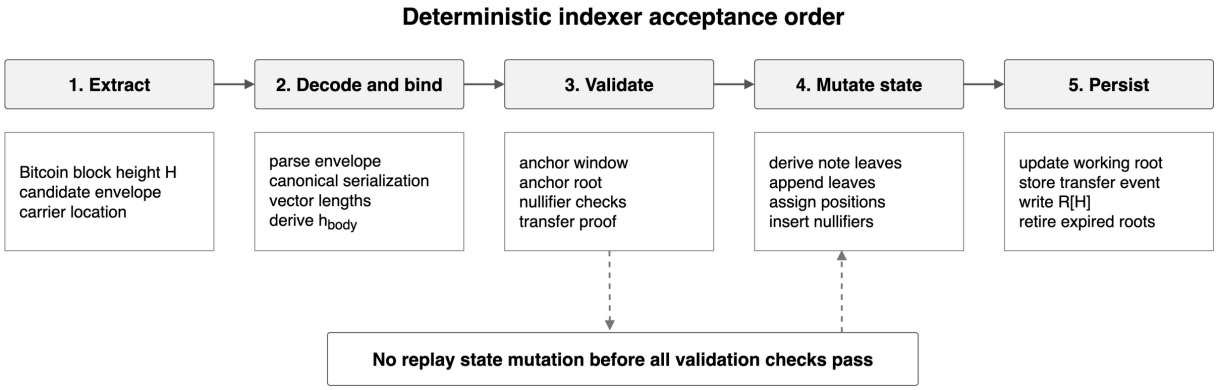

Figure 5: *Deterministic dependency order at the indexers acceptance boundary. State mutation begins only after parsing, body binding, anchor validation, nullifier checks, and proof verification have all succeeded.*

After replaying block H, the indexer stores the working root as $\mathcal{R}[H]$. Storage layout may vary, while replay outcome and the mutation boundary must not. The proof appendix defines circuit semantics.

B Transfer Constraint Decomposition

As defined in Section 14.2, the logical public statement for a transfer with N inputs and M outputs is

$$x_{\mathsf{tr}} = \left(r_{\mathsf{anchor}}, N, M, (\mathsf{nf}_i)_{i=0}^{N-1}, \left(\mathsf{pk}_{\mathsf{eph},j}\right)_{j=0}^{M-1}, \left(\mathsf{ct}_j^{\mathsf{note}}\right)_{j=0}^{M-1}, h_{\mathsf{body}}\right),$$

. The tuple above gives the logical verifier-input order. A concrete deployment profile must also fix field encodings and any padding used by a fixed-width verifier interface.

Replay derives r_{anchor} from the serialized h_{anchor}. The header, h_{anchor}, and $\mathsf{ct}^{\mathsf{out}}$ are not separate public-input slots, they are part of the canonical proof-excluded body bound through h_{body}.

The circuit reconstructs input leaves, proves Merkle membership, enforces spend authorization and nullifier derivation, reconstructs diversifier-bound output ciphertexts, checks value ranges and value conservation, and binds the proof to h_{body}.

Block 1: input-note reconstruction

For each spent input note i, the circuit derives ρ_i and $\mathsf{sk}_{\mathsf{eph},i}$ from $\mathsf{input}_i[r_{\mathsf{seed}}]$. It recomputes the input note leaf from $\mathsf{input}_i[h_{\mathsf{body}}{}^{\mathsf{create}}]$, $\mathsf{input}_i[\mathsf{creation_output_index}]$, $\mathsf{input}_i[\mathsf{pk}_{\mathsf{eph}}]$, and $\mathsf{input}_i[\mathsf{ct}^{\mathsf{note}}]$. It also checks that $\mathsf{input}_i[\mathsf{pk}_{\mathsf{eph}}]$ and $\mathsf{input}_i[\mathsf{ct}^{\mathsf{note}}]$ are consistent with $\mathsf{input}_i[v]$, $\mathsf{input}_i[\mathsf{d}]$, $\mathsf{input}_i[r_{\mathsf{seed}}]$, $h_{\mathsf{aux},i}$. , and the in-circuit-derived $\mathsf{pk}_{d,i}$.

Block 2: Merkle membership

Using the recomputed input ℓ_i, $\mathsf{input}_i[\mathsf{pos}]$, and $\mathsf{input}_i[\mathsf{path}^{\mathsf{Merkle}}]$, the circuit recomputes the Merkle root for each input and constrains it to equal public r_{anchor}.

Block 3: spend authorization

Each input note, the circuit derives the incoming viewing key, the viewing scalar, the diversified receive base for $\mathsf{input}_i[\mathsf{d}]$, and the corresponding diversified transmission key. It then constrains the spent note ciphertext relation to use that $\mathsf{pk}_{d,i}^{\mathsf{drv}}$. If a note ciphertext is proven consistent with the diversified transmission key implied by a given $\mathsf{sk}_{\mathsf{spend}}$ and d, only an entity that knows the corresponding $\mathsf{sk}_{\mathsf{spend}}$ can satisfy the constraint.

The viewing key governs receipt and decryption off circuit. $\mathsf{sk}_{\mathsf{spend}}$ governs spend authorization in circuit.

Block 4: nullifier derivation

For each spent input note, the circuit first derives $\mathsf{sk}_{\mathsf{nf}}$ from $\mathsf{sk}_{\mathsf{spend}}$. It then derives nf_i from that circuit-derived $\mathsf{sk}_{\mathsf{nf}}$, ρ_i, and $\mathsf{input}_i[\mathsf{pos}]$, then constrains it to equal the corresponding public nullifier.

Block 5: output-note reconstruction

For each created output note j, the circuit derives ρ_j and $\mathsf{sk}_{\mathsf{eph},j}$ from $\mathsf{output}_j[r_{\mathsf{seed}}]$, reconstructs the public $\mathsf{pk}_{\mathsf{eph},j} = [\mathsf{sk}_{\mathsf{eph},j}]G_{d_j}$, derives the note shared secret using witness-side $\mathsf{output}_j[\mathsf{pk}_d]$, and constrains the public $\mathsf{ct}_j^{\mathsf{note}}$ to be the encryption of $(\mathsf{output}_j[v], \mathsf{output}_j[\mathsf{d}], \mathsf{output}_j[r_{\mathsf{seed}}])$ under the derived note key. The output ℓ_j is not a separate public input. After proof verification, replay derives it from public h_{body}, output index, $\mathsf{pk}_{\mathsf{eph},j}$, $\mathsf{ct}_j^{\mathsf{note}}$, and $h_{\mathsf{aux},j}$, then appends that leaf.

Block 6: value range checks

For every note, the circuit constrains the value to the canonical unsigned satoshi range fixed by the implementation. At minimum, each value must be an integer in the representable transfer range and must not rely on field wraparound.

Block 7: value conservation

The circuit constrains $\sum_i \mathsf{input}_i[v] = \sum_j \mathsf{output}_j[v]$.

The sums range over all public inputs and outputs represented by the transfer counts. The equality is over the integer values admitted by the range checks, not merely over the ambient SNARK field. The implementation must also ensure that the accumulated sums cannot overflow the chosen integer range before equality is checked.

Block 8: envelope binding

The circuit constrains its internal statement to the derived public h_{body} so the proved transaction matches the published one.

Implementation details may differ depending on how much hashing happens inside the circuit and how much is represented through pre-hashed public inputs, but the proof must be bound to the exact public envelope. A proof that is valid for notes but not bound to the exact public envelope is insufficient for replay.

C Appendix: Alice-to-Bob Transfer Walkthrough

This walkthrough traces one accepted Shielded Bitcoin transfer from wallet state through publication, replay, and recipient detection. Derivation rules are referenced only where they affect transaction construction. The full pipeline view appears in Figure 4; this appendix follows the same flow at the level of concrete wallet state.

Reading convention. The walkthrough uses N for the number of consumed input notes and M for the number of created output notes. A value is called fixed once changing it would change the canonical transfer body, the proof statement, or replay-derived state.

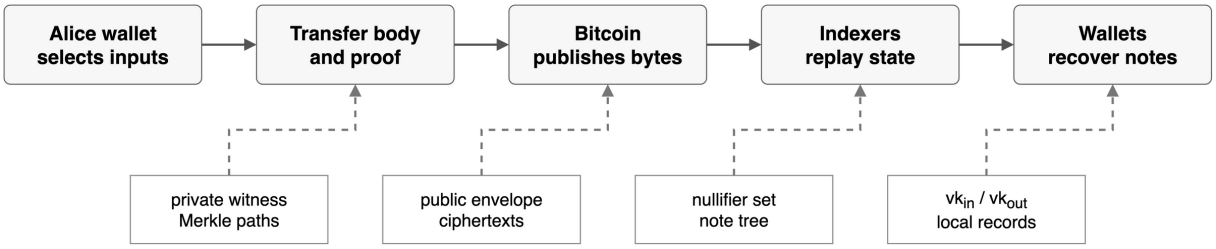

Figure 6: *Compact reading map for the Alice-to-Bob walkthrough. Bitcoin orders the envelope bytes; shielded validity and wallet recovery are determined by replay and local wallet checks.*

C.1 Initial state

Assume the system already has a global note tree, a nullifier set, and a rolling window of recent block-level valid roots. Alice already owns at least one spendable note. Bob already has a shielded address payload consisting of d_{Bob} and $\mathsf{pk}_{d,\text{Bob}}$.

Alice's wallet has already detected one or more notes that belong to her. For each one it knows the note plaintext, the creation h_{body}, output index, $\mathsf{pk}_{\mathsf{eph}}$, $\mathsf{ct}^{\mathsf{note}}$, the corresponding ℓ, the canonical tree position, and whether its nullifier has already appeared.

For each locally tracked note Alice stores the plaintext tuple $(v,\ d,\ r_{\mathsf{seed}})$, the diversified transmission key pk_d associated with that diversifier, the creation envelope fields needed to recompute the note leaf, the canonical tree position, and local spent-state metadata.

Bob has already derived his incoming and spending material from his wallet seed and use $(d_{\text{Bob}},\ \mathsf{pk}_{d,\text{Bob}})$ as his shielded address.

State at the start. Alice has spendable notes with known positions. Bob has a receive address. The indexers state provides the note tree, nullifier set, and recent root window.

C.2 Alice decides to pay Bob

Her wallet scans locally known unspent notes and picks a set whose total value covers $\mathtt{x}$.

If Alice has a single note with the exact value, there is no change output. More often, the wallet selects one or more notes whose total exceeds the send amount and creates a change output back to Alice.

The wallet chooses a set of input notes $in_0 \cdots in_{N-1}$, one recipient output note for Bob, and, if needed, one change note for Alice.

C.3 Alice synchronizes against the current protocol state

Before constructing the proof, Alice needs a fresh protocol state. Her wallet or a trusted indexer provides the current nullifier set, or at least a query interface over it, together with the current note tree, the recent block-level valid-root window, and Merkle paths for the chosen input note leaves.

The wallet chooses one h_{anchor} inside the current valid-root window, derives the corresponding $r_{\mathsf{anchor}} = \mathcal{R}[h_{\mathsf{anchor}}]$, and retrieves, for each selected input note, the canonical tree position and Merkle authentication path under that root.

For each input note i, it also obtains pos_i and $\mathsf{path}_i^{\mathsf{Merkle}}$ such that the input note leaf is a member of the tree under r_{anchor}.

C.4 Alice reconstructs the selected inputs

Before an input note is placed in the transfer witness, the wallet checks that its local record still describes the accepted output it intends to spend. For each input note, the wallet starts from the stored plaintext tuple $(v,\ d,\ r_{\mathsf{seed}})_i$. It re-derives ρ_i and $\mathsf{sk}_{\mathsf{eph},i}$ from $r_{\mathsf{seed},i}$, checks the stored $\mathsf{pk}_{\mathsf{eph},i}$ and $\mathsf{ct}_i^{\mathsf{note}}$ against that plaintext and $\mathsf{pk}_{d,i}$, and rebuilds the note leaf from the accepted creation envelope fields.

The rebuilt leaf must match the leaf and tree position used in the Merkle path. If it does not, the wallet is not holding the note described by its local record, and the transfer must not be built from that record.

C.5 Alice creates the output notes

The wallet next constructs the output notes. For Bob's recipient output, Alice forms a note plaintext $(\mathsf{send_amount},\ d_{\mathrm{Bob}},\ r_{\mathsf{seed},\mathrm{Bob}})$ with a new random $r_{\mathsf{seed},\mathrm{Bob}}$. The send amount and any change amount must be canonical satoshi values inside the range enforced by the transfer circuit. The destination address also contributes Bob's published transmission key $\mathsf{pk}_{d,\mathrm{Bob}}$, which is the recipient-side address material used in the output ciphertext relation for that note. If change is needed, Alice also forms a change note for one of her own addresses, again as a plaintext tuple $(\mathsf{change_amount},\ d_{\mathrm{change}},\ r_{\mathsf{seed},\mathrm{change}})$ with randomness.

The output note leaves are not known until the public body is finalized and h_{body} has been computed. At that point they will be derived from h_{body}, output index, $\mathsf{pk}_{\mathsf{eph}}$, $\mathsf{ct}^{\mathsf{note}}$, and h_{aux}.

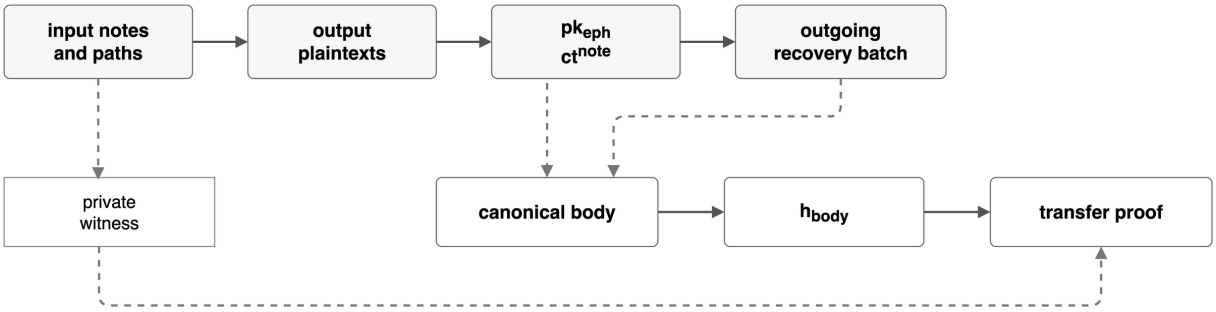

Figure 7: *Construction dependencies before publication. Recipient ciphertexts and the outgoing recovery batch must be fixed before h_{body}; the proof is generated only after the body is bound.*

C.6 Alice derives nullifiers for the inputs

For each input note, Alice derives $\mathsf{sk}_{\mathsf{nf}}$ deterministically from her $\mathsf{sk}_{\mathsf{spend}}$, then derives the nullifier from that $\mathsf{sk}_{\mathsf{nf}}$, the note-local ρ_i, and the note's canonical pos_i in the tree, exactly as the circuit will. Each nullifier is a function of the spending lineage, the note seed, and the canonical tree position; wallet-local identifiers do not enter the derivation.

Once the transaction is accepted, they enter the global nullifier set and make the input notes unspendable forever.

Before publication, Alice's wallet queries the replayed nullifier set and refuses to build the transfer if any derived nullifier is already present. It places the derived nullifiers in the transfer body. Replay later repeats the public nullifier-set check and verifies the proof that binds those published nullifiers to the private input notes.

The witness carries Alice's $\mathsf{sk}_{\mathsf{spend}}$, not an independently chosen $\mathsf{sk}_{\mathsf{nf}}$. Inside the circuit, $\mathsf{sk}_{\mathsf{nf}}$ is re-derived from $\mathsf{sk}_{\mathsf{spend}}$ before nullifier computation so the same note cannot be respent under a different nullifier by varying a second secret. Inside the circuit, spend authorization is checked against the pk_d used by the input note's ciphertext relation, derived from the same wallet secret lineage as $\mathsf{sk}_{\mathsf{spend}}$. Bob's pk_d lets Alice target the correct diversified receive path. Bob later detects the note with the corresponding $\mathsf{sk}_{\mathsf{view}}$

against the published $\mathsf{pk}_{\mathsf{eph}}$. Alice uses $\mathtt{d}$ to reconstruct the diversified base for $\mathsf{pk}_{\mathsf{eph}}$, but she uses the published pk_d for Diffie-Hellman. Bob's $\mathsf{sk}_{\mathsf{spend}}$ determines who can later spend it.

C.7 Alice encrypts the output notes

For Bob's output, Alice derives the ephemeral secret directly from the note seed and uses Bob's diversifier d_{Bob} to recover the diversified base for that address. The sequence is as follows. Alice starts from Bob's published payment address $(d_{\text{Bob}},\ \mathsf{pk}_{d,\text{Bob}})$. She maps d_{Bob} through $\mathsf{DiversifyHash}$ to recover the diversified base for that address, derives $\mathsf{sk}_{\mathsf{eph},\text{Bob}}$ from $r_{\mathsf{seed},\text{Bob}}$, and publishes $\mathsf{pk}_{\mathsf{eph},\text{Bob}} = [\mathsf{sk}_{\mathsf{eph},\text{Bob}}]G_{d_{\text{Bob}}}$. She then computes the note shared secret with $\mathsf{ECDH}(\mathsf{sk}_{\mathsf{eph},\text{Bob}},\ \mathsf{pk}_{d,\text{Bob}})$, runs the note KDF from that shared secret together with the published $\mathsf{pk}_{\mathsf{eph},\text{Bob}}$, and finally encrypts the plaintext tuple $(\mathsf{send_amount},\ d_{\text{Bob}},\ r_{\mathsf{seed},\text{Bob}})$ into $\mathsf{ct}^{\mathsf{note}}_{\text{Bob}}$.

C.8 Alice creates outgoing recovery ciphertexts

Outgoing transaction history also needs to remain recoverable under $\mathsf{vk}_{\mathsf{out}}$. In this format, that recovery path stores the exact material needed to recompute the note shared secret and decrypt the recipient ciphertext later.

For all outputs in the transfer, Alice builds one ordered batch. Its plaintext is the ordered list $(\mathsf{pk}_d,\ \mathsf{sk}_{\mathsf{eph}})_0, \ldots, (\mathsf{pk}_d,\ \mathsf{sk}_{\mathsf{eph}})_{M-1}$, indexed exactly like the outputs themselves.

The wallet first fixes the output-level public data that the recovery path will bind to and then derives the batched outgoing recovery ciphertext before h_{body} is computed.

C.9 Alice assembles the public envelope

At this stage, the public fields needed before sender-recovery encryption are fixed: h_{anchor}, $\mathtt{input_count}$, $\mathtt{output_count}$, nf vector, $\mathsf{pk}_{\mathsf{eph}}$ vector, and $\mathsf{ct}^{\mathsf{note}}$ vector. Alice derives one batch-binding digest from the fixed public output data, namely the output count together with the ordered $\mathsf{pk}_{\mathsf{eph}}$ vector and $\mathsf{ct}^{\mathsf{note}}$ vector. That digest feeds a single $\mathsf{vk}_{\mathsf{out}}$-based KDF, which yields the AEAD key and nonce for the whole batch. She then encrypts the ordered recovery list under that key, with the batch-binding digest as associated data, producing $\mathsf{ct}^{\mathsf{out}}$.

The canonical transfer body is complete only after this $\mathsf{ct}^{\mathsf{out}}$ is included. It consists of the fixed header, h_{anchor}, $\mathtt{input_count}$, $\mathtt{output_count}$, the ordered vectors of nf values, $\mathsf{pk}_{\mathsf{eph}}$ vector, and $\mathsf{ct}^{\mathsf{note}}$ vector, followed by $\mathsf{ct}^{\mathsf{out}}$. The proof is appended only after proving.

The assembly order is:

1. get the public fields and output ciphertexts
2. derive and include the final $\mathsf{ct}^{\mathsf{out}}$
3. hash the canonical serialized body into the derived h_{body}
4. derive each output ℓ from h_{body}, output index, $\mathsf{pk}_{\mathsf{eph}}$, $\mathsf{ct}^{\mathsf{note}}$, and h_{aux}
5. feed r_{anchor}, $\mathtt{input_count}$, $\mathtt{output_count}$, nf vector, $\mathsf{pk}_{\mathsf{eph}}$ vector, $\mathsf{ct}^{\mathsf{note}}$ vector, and h_{body} into proof generation

Why this order matters.

The recovery batch $\mathsf{ct}^{\mathsf{out}}$ is constructed from $\mathsf{vk}_{\mathsf{out}}$ and public output data that is fixed before h_{body} is computed. The recipient ciphertexts are constructed separately from the note shared secret and the published $\mathsf{pk}_{\mathsf{eph}}$.

Once the recipient ciphertexts and $\mathsf{ct}^{\mathsf{out}}$ are fixed, the canonical transfer body is complete, and Alice computes h_{body} over it. The proof takes h_{body} as a public input. Indexers recompute the same value from the published body, so changing a ciphertext or any byte of $\mathsf{ct}^{\mathsf{out}}$ after proving would make verification fail.

During sender-side restore, the wallet decrypts $\mathsf{ct}^{\mathsf{out}}$ and obtains the ordered list $(\mathsf{pk}_d,\ \mathsf{sk}_{\mathsf{eph}})_0, \ldots, (\mathsf{pk}_d,\ \mathsf{sk}_{\mathsf{eph}})_{M-1}$. For output j, it selects $(\mathsf{pk}_d,\ \mathsf{sk}_{\mathsf{eph}})_j$, reconstructs the note shared secret, re-derives the note key schedule, and decrypts $\mathsf{ct}^{\mathsf{note}}_j$. The wallet then checks that the recovered r_{seed} derives

the published $\mathsf{pk}_{\mathsf{eph},j}$, that the plaintext is consistent with $\mathsf{ct}_j^{\mathsf{note}}$, and that the recomputed ℓ_j matches the leaf assigned by replay.

C.10 Alice proves and publishes the transfer

The wallet has the full witness for the `Transfer` circuit: public inputs (r_{anchor}, `input_count`, `output_count`, nf vector, $\mathsf{pk}_{\mathsf{eph}}$ vector, $\mathsf{ct}^{\mathsf{note}}$ vector, h_{body}) together with the private witness derived from Alice's input notes, creation ciphertext data, Merkle paths, output notes, and spending secrets. The serialized body contains h_{anchor}; h_{body} binds that selected block height to the proof.

The wallet, or an external proving service acting on Alice's behalf, runs the selected prover for the `Transfer` circuit. The resulting proof shows that the private witness is consistent with the published transfer body under the transfer rules fixed elsewhere in the specification.

After proving, Alice publishes the final envelope through the protocol's carrier format. Bitcoin records the payload in chain order without interpreting h_{anchor}, nf values, $\mathsf{pk}_{\mathsf{eph}}$ vector, $\mathsf{ct}^{\mathsf{note}}$ vector, $\mathsf{ct}^{\mathsf{out}}$, or the proof.

From the wallet's perspective, the transfer is still pending. Consumed notes should be treated as locally locked, not permanently spent, until the envelope appears in accepted replay history. Likewise, any recipient note visible only through a mempool transaction has no canonical tree position and no canonical block-level root yet, so it is not spendable protocol state. If the publishing transaction is later removed by a reorganization, that local pending state must be rolled back.

C.11 Indexers replay the transfer

An indexer extracts the e from the new Bitcoin transaction in Bitcoin block H and applies deterministic validation.

It checks that the payload parses, the format byte matches the transfer format, the envelope serialization is canonical, the derived h_{body} recomputes from that canonical body, h_{anchor} satisfies $H - W \leq h_{\mathsf{anchor}} \leq H - 1$, no published nullifier has already appeared, the nullifiers are pairwise distinct inside the envelope, and the transfer proof verifies under the transfer verifying key against $r_{\mathsf{anchor}} = \mathcal{R}[h_{\mathsf{anchor}}]$.

The indexer mutates canonical state only after all of those checks have passed. It derives ℓ_j for each output from h_{body}, output index, $\mathsf{pk}_{\mathsf{eph},j}$, $\mathsf{ct}_j^{\mathsf{note}}$, and $h_{\mathsf{aux},j}$; appends those leaves to the note tree; assigns canonical positions to the new leaves; inserts nf vector into the nullifier set; and updates the working root for block H. Only after the whole Bitcoin block has been replayed, the indexers store that working root as $\mathcal{R}[H]$ and retire retained block roots older than the valid window.

Inclusion in a Bitcoin block means the data was published and can be replayed. Wallet finality still depends on confirmation policy and reorganization handling. Replay rules determine whether the data corresponds to a valid shielded transfer on the accepted chain history.

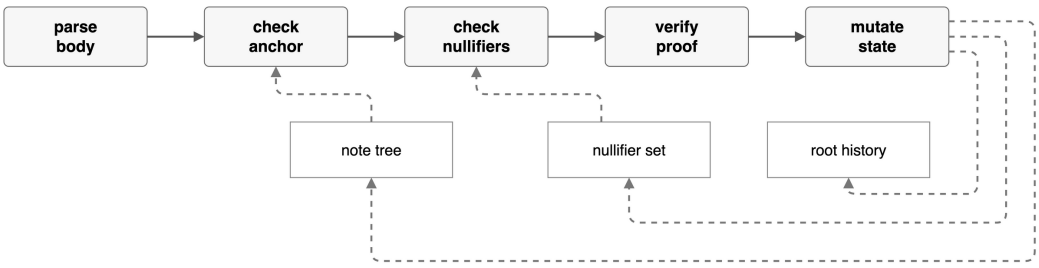

Figure 8: *Replay acceptance path. State changes occur only after parsing, anchor checks, nullifier checks, and proof verification all succeed.*

C.12 Bob detects the incoming note

After the transfer has been accepted by indexers, Bob's wallet eventually sees the new output ciphertexts.

For each output it uses h_{body}, output index, $\mathsf{pk}_{\mathsf{eph}}$, $\mathsf{ct}^{\mathsf{note}}$, and h_{aux} to run note detection with its incoming capability and derive the candidate ℓ. Bob first derives a candidate shared secret from his viewing material and the published $\mathsf{pk}_{\mathsf{eph}}$, and only after decryption does the recovered $\mathtt{d}$ identify which diversified receive path matched. The diversifier is recovered from the note plaintext and then used to bind that plaintext back to the correct diversified address.

The wallet derives a candidate shared secret from $\mathsf{ECDH}(\mathsf{sk}_{\mathsf{view},\mathrm{Bob}},\ \mathsf{pk}_{\mathsf{eph},\mathrm{Bob}})$. It runs the same note KDF from that shared secret and the published $\mathsf{pk}_{\mathsf{eph},\mathrm{Bob}}$, then attempts AEAD decryption of $\mathsf{ct}^{\mathsf{note}}_{\mathrm{Bob}}$.

The wallet then recovers $(v,\ d,\ r_{\mathsf{seed}})$, re-derives $\mathsf{sk}_{\mathsf{eph}}$ from r_{seed}, recomputes the diversified base for the recovered $\mathtt{d}$, checks that the published $\mathsf{pk}_{\mathsf{eph}}$ is consistent with that $\mathtt{d}$, and only then uses $\mathtt{d}$ to identify the corresponding local diversified receive address and associated pk_d. At that point it re-derives ρ from the recovered r_{seed}, checks ciphertext consistency against the corresponding local pk_d, and verifies that the derived ℓ is the leaf assigned to that output by replay.

If it does, Bob has learned the note value, the short recipient diversifier, the note seed, the associated local diversified transmission key for that diversifier, and the note's canonical position once his wallet syncs with indexer state. Bob stores the new note locally and it becomes part of his spendable set.

End state. Alice records the spent inputs, sent-output metadata, and any change note. Bob records the incoming note, its value, seed, diversifier, leaf, and position. Public observers see the envelope, nullifiers, ciphertexts, proof, and derived note leaves, but not the confidential note plaintexts.

D Opt-in KYC Evidence: Certified-Recipient Lineage

Know Your Customer (KYC) is the off-chain process by which a regulated institution identifies a customer and applies its risk policy. Such an institution may ask whether an incoming Shielded Bitcoin note descends from approved Bitcoin inputs and whether subsequent private transfers remained within certified recipients. The ordinary transfer proof does not answer these questions, while publishing the transfer path would defeat the privacy of the shielded layer.

An optional zero-knowledge lineage layer can provide this evidence. A Trust Authority attests to approved Bitcoin inputs and certifies exact shielded `PaymentAddress` values. A certified-lineage output carries a recursive proof that its roots are authenticated peg-ins, that every preserving transfer used only inputs with valid lineage, and that the output was created for its certified recipient address. The recipient checks the proof locally. Bitcoin consensus and base replay do not interpret it, and notes without lineage evidence remain valid Shielded Bitcoin.

Credentials and attestations. A KYC credential associates review under a named policy with a holder commitment. For each approved peg-in input, the authority signs the network and policy context together with the exact Bitcoin outpoint, value, locking condition, and depositor holder commitment. This prevents a prover from combining one customer's credential with another customer's approved inputs.

An address certificate covers the hash of one diversified address (d, pk_d), its policy and validity period, and the credential under which it was issued. Certification must also show that pk_d is derived from the recipient's `spend_sk` under the protocol key hierarchy.

A certified payment begins with a recipient request containing the one-time address. The sender verifies the authority signature and certificate status, recomputes the certified address hash, and uses the same d and pk_d in the ciphertext and lineage proof. The certificate remains a private witness.

Lineage proofs. At peg-in, the lineage-issuance proof checks the depositor's credential and an authority attestation for every entry in a canonical, duplicate-free Bitcoin input list. A PegInExecutionProof, evaluated under a rule fixed by the deployment, authenticates the accepted Bitcoin anchor, the inputs and amount actually used, and the shielded output. The issuance proof requires the approved and executed input lists to agree and binds the lineage root to the first certified output.

For a later transfer, the lineage-step proof verifies the package of every consumed input and connects it to the corresponding note and nullifier. It also checks the recipient's address certificate and binds the new claim to the specific base transfer and to the specific output within it. An input therefore cannot be omitted, nor can a proof be copied to a sibling output or another transfer.

The lineage relation adds a policy claim to the ordinary transfer proof, and both must refer to the same inputs and outputs. A preserving output carries a fixed-size encrypted package containing the current recursive proof and a compact statement commitment; earlier proofs and certificates remain private witnesses.

The certified-lineage property is preserved only when every input consumed by the transfer has a valid package. A transfer that consumes both a certified-lineage note and an ordinary note remains valid under the base protocol, but none of its outputs can retain the certified-lineage claim. If all inputs qualify, each output may preserve the claim or opt out - preserving change requires a certified address just as the payment output does.

Figure 9 shows when an output keeps the optional lineage claim and when it loses it.

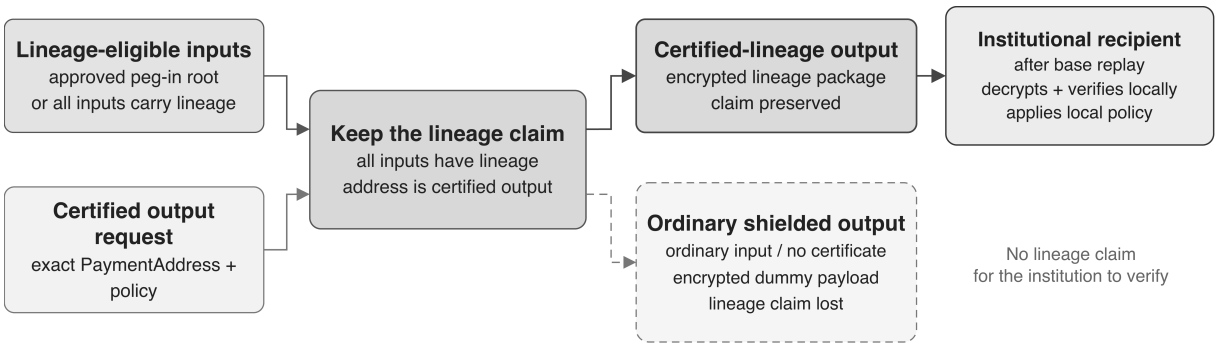

Figure 9: Optional KYC lineage. After base replay, an institutional recipient can verify a certified-lineage output locally when every consumed input has valid lineage and the output address is certified. If any input lacks lineage, no output can preserve the claim; when all inputs qualify, an uncertified or opted-out output alone loses it. The ordinary output remains valid Shielded Bitcoin and carries a same-length encrypted dummy payload, so the branches are not distinguishable on chain.

Institutional verification. After base replay accepts the transfer, the institution decrypts the output and verifies the lineage proof against its fresh challenge and accepted issuer, policy, and certificate status. It can then establish that every root refers to an authenticated peg-in whose complete Bitcoin input list was approved, that no preserving step omitted an input, and that the final proof concerns the exact note, certified `PaymentAddress`, and operation under review.

Under the zero-knowledge and encryption assumptions, the lineage proof need not reveal earlier addresses, holders, or the transfer graph. The statement still exposes the issuer, policy or epoch, and the claim being verified. It says nothing about Bitcoin history before the approved inputs, the quality of the authority's review, or the present owner of an address certified in the past.

Atomic swaps. An atomic swap can produce deal-specific proofs alongside the lineage-step proof. The client's proof ties a certified Bitcoin key or script to the payment or HTLC, while the liquidity provider proves the lineage of its shielded inputs and the output created for the client's certified `PaymentAddress`. Both proofs use the same swap identifier, defined as a commitment to the deal terms (roles, amounts, hash-lock, timeout, and nonce) together with the Bitcoin and shielded outputs of the two legs. They show that the two legs were constructed for the same deal, not that the swap has settled. The received note retains the lineage of the liquidity provider's shielded inputs; the client's Bitcoin payment does not create a new lineage root.

Privacy and policy scope. The lineage package is carried within the recipient-encrypted note data. Genuine lineage evidence and dummy payloads use the same encoding and length, so an observer without the recipient's viewing key cannot distinguish them from the ciphertext alone.

An authority certificate records a decision at a particular policy epoch, not the current controller or beneficial owner. A claim about current status requires an authority-authenticated status snapshot and policy epoch in the recursive statement; without one, the evidence establishes issuance-time approval only.

References

[ACG+26] Michel Abdalla, Brent Carmer, Muhammed El Gebali, Handan Kilinc-Alper, Mikhail Komarov, Yaroslav Rebenko, Lev Soukhanov, Erkan Tairi, Elena Tatuzova, and Patrick Towa. Bitcoin PIPEs v2. Cryptology ePrint Archive, Paper 2026/186, 2026.

[Azt23] Aztec Labs. Introducing aztec.nr: Aztec's private smart contract framework, 2023.

[Azt26] Aztec Labs. Aztec documentation: Privacy-first zkrollup, 2026.

[BSCG+14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. Cryptology ePrint Archive, Paper 2014/349, 2014.

[Cou] Counterparty. Counterparty data encoding methods.

[D+19] Nicolas Dorier et al. BIP 78: A simple payjoin proposal, 2019. Bitcoin Improvement Proposal.

[DMS04] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The Second-Generation onion router. In *13th USENIX Security Symposium (USENIX Security 04)*, San Diego, CA, August 2004. USENIX Association.

[DMS24] Michel Dellepere, Pratyush Mishra, and Alireza Shirzad. Garuda and pari: Faster and smaller SNARKs via equifficient polynomial commitments. Cryptology ePrint Archive, Paper 2024/1245, 2024.

[Ele20] Electric Coin Company. ZIP 224: Orchard shielded protocol, 2020.

[FVB+18] Giulia Fanti, Shaileshh Venkatakrishnan, Surya Bakshi, Bradley Denby, Shruti Bhargava, Andrew Miller, and Pramod Viswanath. Dandelion++: Lightweight cryptocurrency networking with formal anonymity guarantees. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2:1–35, 06 2018.

[G+24] Adam Gibson et al. BIP 77: Payjoin version 2, 2024. Bitcoin Improvement Proposal.

[GGM16] Christina Garman, Matthew Green, and Ian Miers. Accountable privacy for decentralized anonymous payments. In *International conference on financial cryptography and data security*, pages 81–98. Springer, 2016.

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. Cryptology ePrint Archive, Paper 2016/260, 2016.

[HBHW25] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash protocol specification. Technical report, Electric Coin Company, 2025.

[KYMM18] George Kappos, Haaroon Yousaf, Mary Maller, and Sarah Meiklejohn. An empirical analysis of anonymity in zcash. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 463–477, Baltimore, MD, August 2018. USENIX Association.

[LAA+25] Robin Linus, Lukas Aumayr, Zeta Avarikioti, Matteo Maffei, Andrea Pelosi, Orfeas Thyfronitis Litos, Christos Stefo, David Tse, and Alexei Zamyatin. Bridging bitcoin to second layers via BitVM2. Cryptology ePrint Archive, Paper 2025/1158, 2025.

[Lig] Lightning Labs. Taproot assets protocol.

[Lin23] Robin Linus. BitVM: Compute anything on bitcoin, 2023.

[LNP] LNP/BP Standards Association. RGB protocol documentation.

[Max13] Gregory Maxwell. Coinjoin: Bitcoin privacy for the real world. Bitcoin Forum post, 2013.

[MGGR13] Ian Miers, Christina Garman, Matthew Green, and Aviel D. Rubin. Zerocoin: Anonymous distributed e-cash from bitcoin. In *2013 IEEE Symposium on Security and Privacy*, pages 397–411, 2013.

[MO15] Sarah Meiklejohn and Claudio Orlandi. Privacy-enhancing overlays in bitcoin. In *Financial Cryptography and Data Security*. Springer, 2015.

[MPJ+16] Sarah Meiklejohn, Marjori Pomarole, Grant Jordan, Kirill Levchenko, Damon McCoy, Geoffrey M. Voelker, and Stefan Savage. A fistful of bitcoins: characterizing payments among men with no names. *Commun. ACM*, 59(4):86–93, March 2016.

[NEL25] Jonas Nick, Liam Eagen, and Robin Linus. Shielded CSV: Private and efficient client-side validation. Cryptology ePrint Archive, Paper 2025/068, 2025.

[Noe15] Shen Noether. Ring signature confidential transactions for monero. Cryptology ePrint Archive, Paper 2015/1098, 2015.

[Ord] Ordinals Project. Ordinal theory handbook: Inscriptions.

[PH20] Andreas Pfitzmann and Marit Hansen. A terminology for talking about privacy by data minimization: Anonymity, unlinkability, undetectability, unobservability, pseudonymity, and identity management. Technical Report Version v0.34, TU Dresden, 2020.

[PHE+17] Ania M. Piotrowska, Jamie Hayes, Tariq Elahi, Sebastian Meiser, and George Danezis. The loopix anonymity system. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1199–1216, Vancouver, BC, August 2017. USENIX Association.

[Pol24] Polymath. Cryptology ePrint Archive, Paper 2024/916, 2024.

[SB24] Ruben Somsen and Josie Baker. BIP 352: Silent payments, 2024. Bitcoin Improvement Proposal.

[SD03] Andrei Serjantov and George Danezis. Towards an information theoretic metric for anonymity. In Roger Dingledine and Paul Syverson, editors, *Privacy Enhancing Technologies*, pages 41–53, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[SSH+22] Rainer Stütz, Johann Stockinger, Bernhard Haslhofer, Pedro Moreno-Sanchez, and Matteo Maffei. Adoption and actual privacy of decentralized coinjoin implementations in bitcoin, 2022.

[VFV17] Shaileshh Venkatakrishnan, Giulia Fanti, and Pramod Viswanath. Dandelion: Redesigning the bitcoin network for anonymity. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 1:1–34, 06 2017.

[WAA+26] Robin Linus Woll, Ioannis Alexopoulos, Lukas Aumayr, Zeta Avarikioti, Matteo Maffei, and David Tse. BitVM3: Efficient bitcoin bridges via garbled circuits. Cryptology ePrint Archive, Paper 2026/933, 2026.