



ZECBIT

PIXEL NFTS ON ZCASH

THE PRIVATE NFT MARKET

Technical whitepaper - the ZecBit protocol stack: shielded assets, the ZMD-1 metadata standard, a key-less indexer, privacy-correct minting, wallet integration, and non-custodial settlement on Zcash.

VERSION

V1.0

DATE

AUGUST 2026

CHAIN

ZCASH - ZSA

SITE

ZECBIT.IO

CONTENTS

ZecBit

Part I – Foundations

1. Introduction
2. Design Principles and System Overview
3. Background: Zcash and the Orchard Shielded Protocol

Part II – The Asset Layer

4. Zcash Shielded Assets: the Asset Layer
5. The ZecBit NFT Model
6. Protocol Staging and Compatibility

Part III – Metadata, Registry, and Indexing

7. ZMD-1: The ZecBit Metadata Standard
8. The Asset Registry
9. The ZecBit Indexer

Part IV – Creation, Wallets, and Privacy

10. Mint Studio and the Issuance Pipeline
11. The Wallet Layer and ZecBit Connect
12. Privacy Model and Selective Disclosure
- 12A. Independence: Removing ZecBit from the Trust Path

Part V – Trading

13. Trading Architecture Overview and the Escrow Beta
14. Non-Custodial Settlement: the ZIP 228 Matcher
15. Marketplace Feature Set on the Signed-Order Model

Part VI – Security, API, and Deployment

16. Security Model and Threat Analysis
17. Public API and Agent Interface
18. Deployment, Operations, and Technical Roadmap

Part VII – Assets on Chains That Do Not Know About Them

- 19. Three Architectures for Assets on a UTXO Chain
- 20. The Inscription Record on Zcash Mainnet
- 21. What Ordinals Requires, and Why Zcash Cannot Supply It

Part VIII – What Can Be Built on Mainnet Today

- 22. Nullifier-Sealed Client-Side Validation
- 23. The Boundary: Why Public Ownership Cannot Follow
- 24. Data-Carrying Surfaces and Their Economics

Part IX – The Specification That Was Orphaned

- 25. ZSA: A Forensic Status Report
- 26. What the Asset Layer Would Actually Provide
- 27. Settlement: Trading Without Trusting

Part X – What Other Chains Built, and What It Cost Them

- 28. Ordinals, Dissected
- 29. The Other Four Designs

Part XI – The Machine Underneath

- 30. The Action
- 31. The Key Hierarchy
- 32. Note Encryption and the Cost of Finding Your Own Money
- 33. The Commitment Tree
- 34. The Nullifier

Part XII – The Transition Nobody Wrote About

- 35. What Sealing the Old Pool Changed
- 36. Pool Boundaries and What Crossing Them Costs
- 37. The Turnstile, and What It Proves About Auditability
- 38. Proving, and What It Costs in Practice

Part XIII – The Adversary's View

- 39. The Observer Model

40. Attacks That Do Not Work, and Why

41. The Leak That Remains

42. Threats Outside the Protocol

Part XIV — Building It

43. The Issuer's Procedure

44. The Transfer Procedure

45. Failure Modes, Ranked

Part XV — A Formal Treatment

46. Definitions

47. Uniqueness

48. Transfer Integrity

49. Ownership Privacy

50. Anchoring

Part XVI — Economics

51. What Each Operation Costs

52. The Venue's Economics

53. Why the Numbers Favour This Design

Part XVII — The Record on This Chain

54. What Was Actually Deployed

55. What the Record Implies

Part XVIII — Content, and Its Binding

56. What Is Being Committed

57. Storage and Availability

58. Mutable Metadata, and Why It Is Excluded

Part XIX — Infrastructure

59. What the Wallet Must Do

60. What the Indexer Must Do

61. Settlement Infrastructure

Part XX – What Would Change This

- 62. The Three Upgrades That Matter
- 63. What Would Not Change This
- 64. The Circuit That Would Close the Gap

Part XXI – What the Inscription Ecosystem Learned

- 65. Sat Theory, and the Problem It Solves
- 66. Provenance, Parents, and Delegates
- 67. Fungible Tokens on an Asset Protocol
- 68. The Market Structure That Emerged

Part XXII – Why the Account Model Does Not Port

- 69. What the Account Model Provides
- 70. Why the Port Fails
- 71. What the UTXO Model Provides Instead
- 72. The Honest Scorecard

Part XXIII – What Is Actually Known About Linkability

- 73. The Figure That Is Misquoted
- 74. What the Defensible Results Say
- 75. Applying This to the Construction

Part XXIV – The Layer Below

- 76. Network-Level Exposure
- 77. Light Clients

Part XXV – How the Specifications Got Here

- 78. The Process, and What It Explains
- 79. What This Implies About Building

Part XXVI – The Primitives

- 80. The Curve Cycle
- 81. The Hash Functions, and Why There Are Several
- 82. Commitments

- 83. Signatures and Re-Randomisation
- 84. The Proof System
- 85. The Key Agreement
- 86. Where the Assumptions Actually Live

Part XXVII – Running It

- 87. The Operational Commitments
- 88. Migration and Versioning
- 89. What This Project Owes Its Users

Appendix A – A Complete Lifecycle

- A.1 Setup
- A.2 Genesis
- A.3 First Sale
- A.4 Second Sale
- A.5 Third Sale
- A.6 What an Observer Saw

Appendix B – Verification, Specified

- B.1 Types
- B.2 Chain Interface
- B.3 The Procedure
- B.4 The Checks, and What Each Stops
- B.5 What Callers Must Do With the Outcome
- B.6 Re-verification Before Settlement
- B.7 Constructing a Transfer

Appendix C – Conformance

- C.1 Derivation
- C.2 Transfer
- C.3 Verification
- C.4 Independence
- C.5 Recovery
- C.6 Privacy

C.7 Interface

Appendix D – Open Questions

D.1 Two-Party Transaction Co-Construction

D.2 Succinct Nullifier Non-Membership

D.3 Anchor-Age Fingerprinting

D.4 Action-Count Distributions

D.5 Fungibility

D.6 Light-Client Verification

D.7 Multi-Item Disclosure

D.8 Issuer Equivocation

Part XXVIII – Specific Adversaries

91. The Chain Analytics Firm

92. The Compelled Venue

93. The Malicious Issuer

94. The Patient Global Adversary

95. What the Four Have in Common

Part XXIX – The Same Collection, Five Ways

97. Way One: Inscriptions on a Transparent Chain

98. Way Two: Account-Model Contract

99. Way Three: Transparent Seals on This Chain

100. Way Four: Client-Side Validation on a Transparent Chain

101. Way Five: The Construction of Part VIII

102. Choosing

Part XXX – Objections

103. "This Is Just RGB With Extra Steps"

104. "You Have Not Solved Anything, You Have Just Hidden It"

105. "The History Leak Makes This Useless"

106. "Nobody Will Run a Node"

107. "Timing Analysis Destroys This"

108. "The Construction Is Unaudited and Novel"

109. "Why Not Just Wait for the Asset Layer"

110. "Custody Means You Are Just a Bank"

111. "This Does Not Scale"

Appendix E – Notation

E.1 Protocol Values

E.2 Construction Values

E.3 Conventions

E.4 Properties

Appendix F – Glossary

Appendix G – Reading the Specifications

G.1 The Order to Read In

G.2 The Four Structural Checks

G.3 The Errors This Project Made

G.4 A Caution About Third-Party Documentation

Appendix H – Index of Claims

H.1 Protocol Facts

H.2 Derived Results

H.3 Economic Figures

H.4 Comparative Claims

H.5 Specification Status

H.6 Privacy Claims

H.7 The Record

H.8 Open

Appendix I – Deployment Checklist

I.1 Before Any Code

I.2 Key Management

I.3 Issuance

I.4 Wallet

I.5 Verifier

I.6 Venue

I.7 Recovery

I.8 Disclosure

Appendix J – Every Number in One Place

J.1 Sizes

J.2 Costs

J.3 Proof and Verification

J.4 Steady-State Operation

J.5 The Record

Appendix K – Error Taxonomy

K.1 Anchor Errors

K.2 Chain Errors

K.3 Outcomes, Not Errors

K.4 Environmental

Appendix L – Timeline

L.1 Foundations

L.2 The Current Pool

L.3 Inscriptions

L.4 The Asset Layer

L.5 The Transition

L.6 What Is Not On This Timeline

Appendix M – Specification Map

M.1 Load-Bearing for This Construction

M.2 Referenced, Not Depended On

M.3 Dependency Structure

M.4 How to Check This Yourself

Part XXXI – If the Open Questions Resolve

113. If Co-Construction Works

114. If Co-Construction Does Not Work

- 115. If Succinct Non-Membership Is Solved
- 116. If Anchor Age Turns Out to Fingerprint
- 117. If the Asset Layer Ships
- 118. If Nothing Resolves

Appendix N – Worked Derivations

- N.1 Fee Per Logical Action
- N.2 The Data Output Premium
- N.3 The Memo Per-Byte Figure
- N.4 Issuance Cost for a Collection
- N.5 Consignment Growth
- N.6 Scanning Cost
- N.7 Verification Cost

Part XXXII – Reading This Document

- 120. If You Are Auditing the Construction
- 121. If You Are Implementing
- 122. If You Are Issuing a Collection
- 123. If You Are Buying
- 124. If You Are Deciding Whether Any of This Applies

Part XXXIII – Paying From Outside

- 126. The Constraint
- 127. What Was Verified
- 128. Where the Venue's Fee Goes
- 129. Settlement Through the Same Path
- 130. Refunds Are in the Wrong Asset
- 131. Minting
- 132. What This Part Does Not Solve

Appendix O – Order Lifecycle

- O.1 States
- O.2 Transitions
- O.3 The Two Transitions That Matter

O.4 What Cannot Be Reversed

O.5 Where the Race Remains

Appendix P — Data Model

P.1 Collections

P.2 Listings

P.3 Bids

P.4 Cross-Asset Orders

P.5 What Must Not Be Stored

Appendix Q — Document Map

Q.1 The Shortest Honest Path Through

Q.2 The Load-Bearing Citation

Appendix R — Data Channels Across Chains

R.1 Public Channels

R.2 Private Channels

R.3 The Inversion

R.4 Direction of Travel

Appendix S — Auditing a Claim Like This One

S.1 The Six Questions

S.2 Warning Signs in Any Such Document

S.3 What This Document Would Fail On

Part XXXIV — What This Changed

134. Designs Abandoned

135. Claims Withdrawn

136. What Was Added Because of the Analysis

137. What Remains Undone

Appendix T — Wire Formats

T.1 The Anchor Payload

T.2 The Manifest

T.3 Metadata

T.4 The Consignment

T.5 Reassembly

T.6 Versioning

T.7 What Is Deliberately Absent

Appendix U – Trace of a Seal Chain

U.1 Genesis

U.2 The First Transfer

U.3 The Second Transfer

U.4 The Chain, Assembled

U.5 Why a Second N_3 Cannot Exist

U.6 What Alice Discloses to Bob

U.7 The Same Trace, From the Observer's Side

Appendix V – The Construction on One Page

V.1 The Primitive

V.2 The Lifecycle

V.3 The Properties

V.4 The Numbers

V.5 The Three Things to Check Before Trusting This

138. Closing: What This System Is Betting On

Appendix W – Revision Notes

W.1 Structural

W.2 Claims That No Longer Stand

W.3 Debts Not Discharged

Glossary

References

ZecBit

Technical Whitepaper — Version 1.0

zecbit.io

Abstract

Every major digital-collectible platform exposes the full ownership graph of its users: wallets, holdings, purchase history, and social connections are public by construction. ZecBit takes the opposite starting point. Built on Zcash Shielded Assets (ZSA) — the multi-asset extension of the Zcash Orchard shielded protocol — ZecBit combines a fully public, key-less-indexable asset catalog with fully private ownership and transfer, and gives holders cryptographic tools to disclose exactly what they choose and nothing more.

This document is the technical specification of the ZecBit stack: the asset model built on the ZIP 227 issuance protocol and the ZIP 226 transfer protocol; ZMD-1, the first metadata standard for Zcash Shielded Assets; a deterministic, key-less asset indexer; a mirror-friendly metadata registry; a batch-minting studio with privacy-correct issuance patterns; a staged wallet layer culminating in a browser signing SDK; and a trading architecture that is non-custodial by design, built on the ZIP 228 swap protocol, in which the marketplace assembles atomic settlements from user-signed orders without ever taking custody of assets or learning user identities.

The design is protocol-minimal: no new chain, no bridges, no wrapped assets, and no smart-contract layer. Every claim the system makes about an asset is either read directly from the Zcash chain or verifiable offline against chain data. This paper covers the technical architecture only; token economics, pricing, and business terms are out of scope.

Document Status

This is version 1.0, the reference specification of the system as designed and, where stated, as already built and exercised on the public ZSA test network. The Zcash Shielded Assets protocol is complete, independently audited, and deployed on a dedicated public test network; its mainnet activation is expected in a future Zcash network upgrade and is outside ZecBit's control. Sections that depend on protocol components not yet scheduled for activation (notably the ZIP 228 swap protocol) are marked accordingly. Empirical results referenced in this paper were produced by the ZecBit team on the public ZSA test network and are reproducible from the referenced open-source tooling.

What Already Exists

A specification that describes only intentions is a wish list. The following components of this document are built and running at the time of writing, on the public ZSA test network:

Component	State	Reference
Batch issuance of finalized assets	Working. Ten unique finalized assets (supply 1, <code>finalize = 1</code>) issued in a single transaction and confirmed	\$5.3

Issuance tooling	Working. Collection-level mint command over the ZSA branches of the core libraries	\$10
Deterministic art and trait generation	Working. Full collection derived from one published seed, with rarity ranks computed from trait frequencies	\$5.2, \$7
ZMD-1 metadata convention	Specified and in use in issued asset descriptions	\$7
Public-facing quest and allowlist, with review tooling	Live in production	\$10.5
Indexer, registry, mint studio UI	Specified; implementation in progress	\$8, \$9, \$10
Trading rehearsal engine	Test network only — never deployed on mainnet	\$13
Custody of user assets or funds	None, at any stage on mainnet	\$2.4, \$13, \$14
On-chain state anchors (non-equivocation)	In development	\$12A.1
Published snapshots (survives operator loss)	In development	\$12A.2
Non-custodial matcher	Specified; blocked on ZIP 228 activation	\$14

Everything else in this document is design, and is labeled as such where it matters.

Intended Audience

Protocol engineers, wallet developers, Zcash ecosystem contributors, security reviewers, and technically minded creators and collectors evaluating the system's guarantees.

Part I — Foundations

1. Introduction

1.1 Purpose and Scope

This document is the technical specification of ZecBit, a privacy-preserving marketplace and asset-infrastructure layer for non-fungible and semi-fungible digital assets issued as Zcash Shielded Assets (ZSAs). It specifies the ZecBit protocol stack end to end: the on-chain asset layer built on the ZSA issuance and transfer protocols (ZIP 227, ZIP 226); the metadata, registry, and indexing layers that make shielded assets discoverable without compromising holder privacy; the creation, wallet, and disclosure tooling required for practical use; and the trading and settlement architecture, including its staged path from operator-mediated settlement to non-custodial signed-order settlement.

The document is written for protocol engineers, wallet implementers, auditors, and technically sophisticated users. It assumes familiarity with public-key cryptography and blockchain data structures, but not with Zcash internals; Chapter 3 supplies the necessary background. Wherever ZecBit's behavior depends on a property of the Zcash protocol, we cite the governing Zcash Improvement Proposal (ZIP) inline, and we distinguish explicitly between properties that hold on Zcash mainnet today, properties that hold on the dedicated ZSA testnet, and properties that are contingent on a future network upgrade.

The following are explicitly **out of scope** for this document:

- Token economics of any kind, including the existence or design of any ZecBit-associated token.
- Pricing, fee schedules, revenue models, and commercial terms.
- Legal and regulatory analysis, except where a protocol mechanism exists specifically to enable compliance workflows (e.g., selective disclosure), in which case we specify the mechanism and not its legal sufficiency.
- Marketing claims of any kind. Where this document makes a comparative statement, it is a technical statement about visibility, trust assumptions, or protocol capability, and it is falsifiable against the cited specifications.

Statements about the future (in particular, about Zcash network upgrades) are forecasts about third-party processes that ZecBit does not control. Section 3.6 and Chapter 18 state these dependencies plainly, together with the behavior of the system while they remain unresolved.

1.2 The Problem: Total Ownership Transparency

Every major NFT platform in production today operates on a chain whose ledger is fully transparent at the account level. On such chains, the asset graph — who owns what, who bought it from whom, at what time, and in what quantity — is public, permanent, and trivially machine-readable. This is not an incidental property; it is the default data model, and the entire surrounding tooling ecosystem (explorers, portfolio trackers, analytics APIs, airdrop-targeting services) is built to exploit it.

The consequences for asset holders are well documented and structural rather than anecdotal in nature:

Wallet doxxing. A single link between an address and a real-world identity — an ENS-style name, a social-media post, a marketplace username, a shipping address leaked by a merchant, an exchange withdrawal — retroactively deanonymizes the address's entire history and, via clustering heuristics, frequently the holder's other addresses as well. Deanonymization is cumulative and irreversible: the ledger does not forget.

Portfolio surveillance. Because holdings are public, any party can continuously monitor the complete portfolio of any collector: acquisitions, disposals, unrealized positions, and behavioral patterns such as time-of-day activity. This information is asymmetric ammunition in any negotiation and is harvested at scale by commercial analytics services without holder consent.

Targeted attacks. Publicly visible high-value holdings convert directly into a target list. Documented attack classes include precision phishing against holders of specific collections (the attacker knows exactly which approval prompt to counterfeit), poisoned-NFT airdrops, social-engineering campaigns referencing the victim's actual holdings to establish false legitimacy, and — at the extreme — physical coercion against individuals whose on-chain wealth is publicly attributable.

Trade surveillance and front-running of known participants. When a recognizable collector's bid or listing activity is publicly attributable in real time, other participants systematically trade against that information: sniping listings the collector is known to be accumulating, repricing against their revealed reservation prices, and mirroring their entries. On transparent chains this requires no privileged access whatsoever; the mempool and the ledger provide it to everyone.

Set against these harms is a genuine tension: *provenance is a feature*. The verifiable history of an asset — that it was issued by a particular creator, that its supply is what the issuer claims, that a given item is authentic and not a copy — is a substantial part of what makes digital assets meaningful. Naive privacy designs that hide everything destroy provenance along with surveillance; fully transparent designs preserve provenance by surveilling everyone. Prior attempts at "private NFTs" on purpose-built privacy chains generally inherited a second problem: they required users to migrate to a new chain with no existing economy, no liquidity, and no installed wallet base, and none of those attempts found a sustained market. ZecBit's design begins from the position that both failure modes — total transparency and total darkness on an empty chain — are avoidable, and that the ZSA protocol's specific asymmetry between issuance and ownership is the mechanism that avoids them.

1.3 The ZecBit Thesis

ZecBit is built around one architectural fact of the ZSA protocol and one strategic fact about the Zcash network.

The architectural fact: issuance is transparent; ownership is shielded. Under ZIP 227, the issuance of a Zcash Shielded Asset is a public, on-chain event: the asset's identifier, its issuer's issuance-verification key, the quantity issued, and whether supply has been finalized are all visible to any observer without any keys, and are indexable by ordinary chain-scanning infrastructure. Under ZIP 226, transfers of that same asset occur entirely inside the Orchard-based shielded pool: holders, balances, transfer amounts, and counterparties are cryptographically hidden. The protocol thus natively separates the *existence and authenticity* of an asset (public) from the *ownership and movement* of an asset (private).

This asymmetry is precisely the split that a marketplace needs. A marketplace requires a public catalog — assets must be discoverable, their issuers verifiable, their supplies auditable — and a catalog can be built from issuance data alone. A holder requires private ownership — and the shielded pool provides it by default, not as an opt-in mixing step. Between the two sits **selective disclosure**: Zcash's viewing-key and payment-disclosure primitives (Section 3.5) allow a holder to prove facts about their own ownership — to a buyer, an auditor, a grant committee, or a court — without revealing anything to anyone else. ZecBit's thesis, in one sentence:

Public catalog. Private ownership. Disclosure on the holder's terms.

The strategic fact: the shielded economy already exists. Earlier privacy-asset systems asked users to bootstrap a new privacy chain from zero. ZecBit instead targets Zcash, an L1 in continuous operation since 2016, whose shielded usage as of mid-2026 is at historic highs: roughly 31% of circulating supply is shielded, and a majority of network transactions have been shielded since February 2026. Assets issued on Zcash inherit this economy — its anonymity set, its wallet infrastructure, its node software, and its settlement asset — rather than recreating it. ZecBit adds no chain, no bridge, and no wrapped representation of anything; it adds a catalog, an index, tooling, and a market on top of a protocol that already exists and has already been audited.

1.4 Design Goals

The following goals govern every design decision in this document. Where goals conflict, they are listed in priority order.

- 1. Privacy by default.** Ownership, balances, transfer amounts, and counterparties are never visible to ZecBit, to other users, or to public observers, except where a key holder deliberately discloses them. There is no "private mode" to enable; the shielded pool is the only ownership layer.
- 2. Verifiability on demand.** Every privacy affordance is paired with a disclosure affordance. Issuer authenticity, asset supply, and supply finalization are publicly verifiable by construction. Individual ownership and transaction history are provable by the holder using standard Zcash disclosure primitives, at the holder's sole discretion, with recipient-scoped granularity.
- 3. Protocol minimalism.** ZecBit introduces **no new chain, no bridge, and no wrapped assets.** Assets are native ZSAs on Zcash; ZEC is the settlement asset; consensus rules are Zcash's consensus rules. ZecBit-defined structures (metadata manifests, registry records, order formats) live strictly above consensus and can be independently reimplemented from this specification.
- 4. Decentralizable metadata.** Asset metadata is content-addressed and replicable. No ZecBit-operated server is a single point of failure for the interpretation of any asset: given the chain and the content-addressed metadata, any party can reconstruct the full catalog (Chapters 7–9).
- 5. Non-custodial settlement, and nothing else on mainnet.** The trading design requires no custody of assets or funds by any operator. Where a protocol capability is missing, the answer is to wait for it on mainnet and rehearse on a test network — not to ship custody and describe it as temporary.
- 6. Graceful protocol staging.** ZSA is complete and audited but not yet activated on Zcash mainnet (Section 3.6). ZecBit is therefore specified and built in stages that track the protocol's own deployment: full-stack operation on the dedicated ZSA testnet today, with a

mainnet activation path that requires reconfiguration, not redesign. No component of this specification assumes protocol behavior beyond what is specified in the cited ZIPs.

- 7. Reconstructibility.** The chain is the sole source of truth for asset state. Every off-chain component — indexer, registry, API — must be rebuildable from genesis by an independent operator using only public data and this specification (Section 2.2).

1.5 A Working Venue Today, and What Mainnet Changes

The obvious first question about anything built on Zcash Shielded Assets is whether it is real yet, given that ZSA has not activated on Zcash mainnet. The answer is that the venue is real and running now — and that mainnet activation is a migration event for it, not a birth.

What exists and operates today. The ZSA protocol is complete, independently audited, and live on a dedicated public network where its consensus rules run from the genesis block. On that network ZecBit already issues assets with real issuance bundles, real Halo 2 proofs, real consensus-enforced finalization, and real shielded transfers: ten unique finalized assets in one confirmed transaction, proofs generated in about three seconds on an ordinary CPU. Around that, the parts of this system that are not chain-specific — the catalogue, the metadata standard and registry, the indexer, the minting pipeline, the collection and item interfaces, the allowlist — are operating as a product, not as a prototype waiting for permission to exist.

So the position is not “a marketplace that will work someday.” It is a marketplace that works, on the network where shielded assets currently live, with a migration path to the network where they will live next.

Why the work is not throwaway — the property that makes this possible. Asset identity under ZSA is *derived*, never assigned by a platform:

```
AssetId = f( issuer key , hash(asset description) )
```

Both inputs are ours, and both are published. The same issuer key and description therefore yield the same asset identity on any ZSA chain — including one that does not exist yet. Every item committed today already has its mainnet identity fixed and computable by anyone, from public data, with no bridge, no snapshot, and no trust in us required. At activation the collection is re-issued from the same inputs and every observer can check that what appeared on mainnet is exactly what was committed months before. That is why building now is not a rehearsal: the artifacts survive the transition by arithmetic (Chapter 6).

What the ecosystem can use right now. The components most useful to other people are the ones that do not depend on which network is underneath, and they are available immediately rather than at some future launch:

- **ZMD-1**, the metadata standard (Chapter 7). ZIP 227 deliberately puts only a hash of an asset description on chain and warns wallets not to display raw descriptions as names. Something has to fill that gap before assets reach users, or every wallet invents its own convention and users are phished through asset names. ZMD-1 is hash-bound, signed by the issuance key,

verifiable offline, and specified so any implementation can adopt it against any registry — including none of ours.

- **The registry and indexer** (Chapters 8–9). Key-less, deterministically rebuildable from chain data, published open-source with public dumps and mirrors. A wallet or explorer can resolve asset names and supplies without asking our permission or trusting our servers.
- **The wallet integration path** (§11.6). Payment intents over ZIP 321 work with wallets as they exist today; PCZT covers application-built transactions using machinery wallets already ship for hardware signing. Neither requires ZSA to be live, and both are useful to any Zcash application that needs to charge for something.
- **Operational experience.** Issuance, batching, proof timings, failure modes, and the fee questions still unsettled between ZIP 227 and ZIP 317 are documented here rather than rediscovered later by everyone independently.

We would rather these become shared ecosystem infrastructure than ZecBit-only advantages, which is why the standard is offered to the ZIP process, the code is open, and nothing above requires a partner to depend on our uptime.

What genuinely waits for activation, stated plainly.

Capability	Today, on the ZSA network	At mainnet activation
Issuing assets, finalized supply, batch minting	Yes — running	Yes, on mainnet
Public catalogue, metadata resolution, indexing	Yes — same code, same standard	Yes
Shielded ownership and transfers	Yes	Yes
Wallet integration via ZIP 321 / PCZT	Available to build against now	Unchanged, plus assets
Assets carrying mainnet value	No — testnet assets are testnet assets	Yes
Trustless atomic trading	No — needs ZIP 228 (Chapter 14)	When ZIP 228 activates
Custody in the holder's own mainnet wallet	No — registry-tracked until claims open	Claimed to the holder's wallet

What we will not pretend. Testnet assets carry no mainnet value and no continuity of ownership, and anyone telling you otherwise about any project is lying. Activation timing belongs to the Zcash network upgrade process; this document contains no dates because we have none to offer. What we do control is that when the upgrade lands, the ecosystem does not start from zero: there is already a venue, a standard, an index, a tooling chain, and a set of collections whose mainnet identities were fixed and published long before the block that makes them real.

1.6 Document Roadmap

Part I — Background (Chapters 2–3) establishes the system's design principles, layered architecture, actor model, and terminology (Chapter 2), and provides a self-contained technical

primer on Zcash and the Orchard shielded protocol — notes, nullifiers, actions, Halo 2 proving, the key and address hierarchy, and the disclosure primitives — at the depth required by the rest of the document (Chapter 3).

Part II — The Asset Layer (Chapters 4–6) specifies how ZecBit uses the ZSA protocols: asset issuance under ZIP 227, including issuer keys, asset identifiers, supply semantics, and finalization (Chapter 4); shielded transfer and burn under ZIP 226 (Chapter 5); and ZecBit’s asset-class conventions for non-fungible and semi-fungible assets built on those primitives (Chapter 6).

Part III — Metadata, Registry, and Indexing (Chapters 7–9) defines the ZMD-1 metadata manifest format (Chapter 7), the ZecBit registry that binds on-chain asset identifiers to content-addressed metadata and issuer identity attestations (Chapter 8), and the indexer that transforms raw chain data into the public catalog (Chapter 9).

Part IV — Creation, Wallets, and Privacy (Chapters 10–12) covers the creator pipeline from media to issued asset (Chapter 10), wallet integration requirements including scanning, proving, and key management (Chapter 11), and the full selective-disclosure model: viewing-key scoping, payment disclosure, and provenance proofs (Chapter 12).

Part V — Trading (Chapters 13–15) specifies the order model and the matcher (Chapter 13), settlement in both the escrow-beta stage and the signed-order end state (Chapter 14), and market-integrity and privacy-preserving analytics constraints (Chapter 15).

Part VI — Security, API, and Deployment (Chapters 16–18) presents the threat model and security analysis (Chapter 16), the public API surface (Chapter 17), and the deployment architecture and mainnet-activation plan (Chapter 18).

2. Design Principles and System Overview

2.1 Layered Architecture

ZecBit is a stack of five layers above the Zcash chain, with the trading engine positioned alongside the read path rather than inside it. Each layer consumes only the layer below it, and each layer boundary in the diagram corresponds to a chapter boundary in this document.



Reading the diagram bottom-up:

- **Chain.** Zebra full nodes (the Zcash consensus implementation that succeeded `zcashd`, which is now fully retired) validate consensus and expose block data. ZSA issuance events are

public in block data; ZSA transfers are shielded and expose only the ciphertexts, commitments, nullifiers, and proofs described in Chapter 3.

- **Indexer.** A deterministic pipeline that consumes blocks and materializes the public catalog: every asset ever issued, its issuer key, cumulative supply, finalization status, and burn events. The indexer holds no viewing keys and, by construction, cannot see ownership (Section 2.3).
- **Registry.** A thin binding layer between on-chain asset identifiers and off-chain, content-addressed metadata manifests, plus issuer identity attestations. The registry adds interpretation, never authority: an asset absent from the registry still exists and is still tradable; a registry entry that contradicts the chain is invalid by definition.
- **API.** A stateless query surface over indexer and registry state, plus the order-submission endpoint for the trading engine. Anything served by the API is either public chain data, registry data, or data the requesting client submitted.
- **Applications.** The ZecBit web marketplace is one client among several possible; wallets and third-party applications consume the same API. No application holds a privileged position: the API exposes nothing that an independent indexer-plus-registry operator could not serve identically.
- **Trading engine.** The matcher and settlement components sit beside the read path. They receive orders from applications, and they produce (in the end state) fully-formed shielded transactions that settle directly on the chain, bypassing every intermediate layer on the write path. The trading engine is the only component whose trust model changes across deployment stages (Section 2.4).

2.2 Principle: The Chain Is the Source of Truth

Every fact that ZecBit asserts about an asset must be either (a) directly read from the Zcash chain, (b) deterministically derivable from chain data plus content-addressed data whose hash is anchored in chain-derived records, or (c) explicitly labeled as an off-chain attestation with a named attester.

Concretely:

- **Asset existence, issuer, supply, and finalization** are category (a): they are properties of issuance bundles in blocks and are recomputable by anyone running a node. The ZecBit indexer's asset table must be bit-for-bit reproducible by an independent implementation of Chapter 9 against the same chain.
- **Asset metadata** (names, media, collection structure) is category (b): manifests are content-addressed, and the binding from asset identifier to manifest hash is constructed so that a third party can verify the binding without trusting ZecBit (Chapter 8 specifies the binding; where the current protocol stage forces any component of the binding to be an operator attestation, that component is labeled as such and scheduled for elimination).
- **Issuer identity** ("this issuance key belongs to this artist") is irreducibly category (c): a chain cannot know who a person is. Registry identity records are therefore signed attestations with explicit attesters and are presented as such in every API response — never blended into chain facts.

The operational corollary is the **reconstruction requirement**: a party possessing only (i) a Zcash node, (ii) this specification, and (iii) access to the content-addressed storage network must be able to rebuild the entire public catalog and verify every category-(a) and category-(b) claim ZecBit has ever served. ZecBit’s own infrastructure is required to be rebuildable this way from genesis, and this reconstruction is exercised as a matter of operational routine rather than assumed. No component may develop write-side state that cannot be regenerated — with the sole exception of the trading engine’s open-order book, which is ephemeral by nature and whose loss cancels rather than corrupts (Chapter 13).

This principle is what makes the indexer and registry *decentralizable* rather than merely replicable: independent operators are not mirrors of ZecBit; they are peers deriving the same state from the same root.

2.3 Principle: The Observer Model

Privacy engineering fails most often not at the cryptographic layer but at the systems layer, when a design is evaluated only against an external adversary and not against its own operators. ZecBit is therefore specified against an explicit observer model: for each class of party, we state exactly what that party can see, and every subsequent chapter must conform to this matrix. A claim of the form “X is private” in this document always means “X is not visible to any party not listed as seeing X below.”

The parties:

- **Public observer.** Anyone with a Zcash node and this specification; no keys, no ZecBit account.
- **ZecBit indexer.** The catalog pipeline of Section 2.1 (equivalently: any independent indexer operator).
- **Matcher.** The trading engine component that receives and matches orders (Chapter 13). Listed separately because it necessarily sees order flow, which is more than the indexer sees.
- **Holder.** A party holding the spending and viewing keys for their own wallet.
- **Issuer.** A party holding an issuance key; sees everything a public observer sees plus their own key material. An issuer has *no* privileged view of who holds their asset.

Visibility matrix (✓ = visible by construction, ● = visible only for the party’s own activity or with keys the party holds, ✕ = not visible; D = visible only under holder-initiated disclosure, Chapter 12):

Fact	Public observer	ZecBit indexer	Matcher	Holder	Issuer
Asset exists; asset identifier	✓	✓	✓	✓	✓
Issuer issuance key; issuance history	✓	✓	✓	✓	✓
Total supply; finalization status; burns (aggregate)	✓	✓	✓	✓	✓
Metadata manifest content	✓	✓	✓	✓	✓

That <i>some</i> shielded ZSA transfer occurred (tx exists)	✓	✓	✓	✓	✓
Which asset a shielded transfer moved	✗	✗	● (own-settled trades)	● (own txs)	✗
Holder identity / addresses of an asset's owners	✗	✗	✗	● (self)	✗ / D
Any wallet's balance or portfolio	✗	✗	✗	● (self)	✗ / D
Transfer amounts and counterparties	✗	✗	● (own-settled trades)	● (own txs)	✗ / D
Open orders (asset, price, side)	✗	✗	✓	● (own orders)	✗
Linkage between a holder's successive market actions	✗	✗	● (bounded; Ch 13/15)	● (self)	✗
Full history of a specific item	✗	✗	✗	● (own custody span)	✗ / D

Three rows deserve emphasis. First, the indexer column equals the public-observer column everywhere: the catalog pipeline is *definitionally* keyless, so operating an indexer confers zero ownership visibility — this is what makes independent indexer operation safe and uninteresting as an attack. Second, the matcher is the only ZecBit-operated component that learns non-public information (order flow and, during settlement, trade details), which is why Chapters 13–15 impose data-minimization, retention, and unlinkability constraints on it specifically, and why its visibility is bounded per trade rather than cumulative per user. Third, the issuer row is deliberately weak: issuing an asset grants no surveillance rights over its holders. Every **D** entry is reachable only through the holder-controlled disclosure mechanisms of Chapter 12.

2.4 Principle: Staged Decentralization, Stated Honestly

The end-state settlement design — atomic swap of a shielded asset against shielded ZEC via jointly-constructed transactions signed by both counterparties, with no intermediary custody (Chapter 14) — depends on transaction-construction capabilities whose availability tracks the ZSA protocol's own maturation and wallet-side tooling. This document therefore states plainly what runs where: a free, non-custodial distribution today, a test-network rehearsal of trading interfaces, and non-custodial settlement as the only mainnet trading design.

Stage 0 — Catalog and testnet operations (current). The read stack (indexer, registry, API, applications) operates against the dedicated public ZSA testnet. No mainnet assets exist; no value is at risk. Trust assumptions: availability only.

Stage 1 — Distribution without custody (current). The Genesis collection is distributed free, so no payment leg exists and nothing needs to be held in trust: items are issued and delivered directly to the addresses their recipients supplied. Trust assumptions: availability, and the correctness of the allocation — both of which are checkable, because the collection is

committed on-chain before allocation and the draw is recomputable by anyone from a published block height. ZecBit never holds a user's funds and never holds an asset a user already owns.

Where an operator-mediated exchange is exercised at all, it is confined to the ZSA test network as an engineering rehearsal for the interfaces in Chapter 13 — on assets with no monetary value, never on mainnet, and never as a step users are asked to trust with something they own.

Stage 2 — Signed-order matcher (end state). Orders are signed authorizations to trade under stated terms. The matcher pairs compatible orders and assembles a single settlement transaction that both parties' authorizations jointly render valid; the transaction is atomic on-chain — either both legs execute or neither does. Trust assumptions: the matcher can censor (refuse to match), reorder within protocol-permitted bounds, and observe order flow; it *cannot* steal, alter trade terms, or settle without both signatures. Custodial risk is eliminated; the residual trust is availability and fairness, both of which are auditable and both of which are further reducible by running multiple independent matchers against the same order format (Chapter 13).

The commitment made by this document is one of engineering direction, not schedule: no Stage-1 mechanism may be built in a way that impedes Stage 2, and the order and settlement formats of Chapters 13–14 are specified for Stage 2 from the outset, with Stage 1 implemented as a degenerate case behind the same interfaces.

2.5 Actors and Roles

Creator / issuer. Generates and safeguards an issuance key; issues assets under ZIP 227; authors metadata manifests; optionally registers an identity attestation. An issuer's ongoing powers are limited to further issuance of their own non-finalized assets and finalization itself; issuers cannot freeze, claw back, or surveil issued assets. A single real-world creator may operate multiple issuance keys; the registry models this (Chapter 8).

Collector / holder. Holds assets in their own shielded wallet under their own spending keys at all times outside a Stage-1 in-flight trade. Interacts with the catalog anonymously (the read API requires no account), and with the market pseudonymously per-order. Controls all disclosure.

Marketplace operator (ZecBit). Operates reference instances of the indexer, registry, API, applications, and trading engine; publishes this specification and its reference implementations. On mainnet the operator holds no user assets and no user funds at any point; its distinguishing capability is order-flow visibility plus registry curation, whose limits are set by Section 2.2 (curation can annotate and rank; it cannot alter chain facts or render an asset nonexistent).

Indexer operators. Independent parties running the Chapter-9 pipeline. Because the pipeline is keyless and deterministic, independent operators serve as live verification of the reconstruction requirement: divergence between two honest indexers indicates a specification or implementation defect, not a trust decision.

Matchers (future, plural). Stage-2 order-matching operators. The signed-order format deliberately contains everything a matcher needs and nothing bound to ZecBit's identity, so

that multiple matchers can compete over the same asset universe; holders choose where to send orders.

2.6 Terminology

The following terms are used with fixed meanings throughout Parts I–VI.

Term	Definition
Asset	A distinct Zcash Shielded Asset: the unit of on-chain identity, identified by its ZIP 227 asset identifier and issuable only by its issuance key.
Collection	A ZecBit-level grouping of assets under one issuer and one collection manifest (Chapter 7). Collections are a metadata construct; the chain knows only assets.
Item	The marketplace-level unit of ownership and trade: for non-fungible classes, an asset with supply one; for semi-fungible classes, one unit of a bounded-supply asset (Chapter 6 defines the classes precisely).
Manifest	A content-addressed metadata document (format ZMD-1, Chapter 7) describing an asset or collection: media references, attributes, and structural relations.
Registry	The binding layer of Chapter 8: records associating asset identifiers with manifest hashes, and issuance keys with identity attestations.
Matcher	The component (Stage 2: any operator) that accepts orders and pairs them into settlements.
Order	A signed, self-contained expression of intent to trade: side, asset, quantity, price bounds, expiry, and the authorization material required for settlement (Chapter 13).
Settlement	The on-chain transaction (or, in Stage 1, transaction pair) that executes a matched trade.
Shielded / transparent	Used strictly in the Zcash protocol sense (Section 3.1), never metaphorically.
Disclosure	Any holder-initiated revelation of otherwise-shielded facts via the Chapter-12 mechanisms; always scoped to a named recipient and stated content.

Normative keywords **MUST**, **SHOULD**, and **MAY** are used in their RFC 2119 senses in specification chapters (4–9, 13–14, 16–17); Chapters 1–3 are descriptive.

3. Background: Zcash and the Orchard Shielded Protocol

This chapter is a self-contained technical primer on the parts of Zcash that the remainder of the document builds on. It describes the protocol as deployed and specified as of August 2026. Readers familiar with the Zcash protocol specification may skim to Section 3.5; readers requiring full formal detail should consult the Zcash Protocol Specification and the cited ZIPs, which are normative where this summary simplifies.

3.1 Zcash Architecture Primer

Zcash is a proof-of-work L1 blockchain in continuous operation since October 2016, originating as a fork of the Bitcoin codebase and retaining Bitcoin’s UTXO accounting for its **transparent** value pool: transparent addresses, inputs, and outputs behave essentially as in Bitcoin and are fully public. Alongside the transparent pool, Zcash defines **shielded pools**, in which value is carried by encrypted *notes* rather than public outputs, and transaction validity is established by zero-knowledge proofs (zk-SNARKs) rather than by public script evaluation. A single Zcash transaction may contain transparent components, shielded components, or both; value crossing between pools is visible in aggregate (the transaction’s net value balance per pool is public), while activity wholly inside a shielded pool reveals no amounts, senders, or recipients.

Four shielded pools have been deployed, each superseding its predecessor’s cryptography while remaining governed by pool-migration (“turnstile”) rules that keep cross-pool value flows publicly auditable:

Pool	Activated	Proving system	Trusted setup	Notes
Sprout	2016 (launch)	BCTV14 / Groth16	Yes	Original pool; long deprecated for new use.
Sapling	2018 (Overwinter/Sapling)	Groth16	Yes	Order-of-magnitude proving improvements; enabled mobile shielded wallets.
Orchard	2022 (NU5)	Halo 2	No	Pallas curve; action model; the basis of ZSA. (ZIP 224)
Ironwood	2026-07-28 (NU6.3)	—	No	New shielded pool with its own turnstile, introduced by the Ironwood upgrade; carried by the v6 transaction format. (ZIP 229)

Two operational facts about the 2026 network matter for this document. First, the original node implementation, `zcashd`, has been fully retired; the network runs exclusively on **Zebra**, the Rust consensus implementation, which is the node software all ZecBit infrastructure builds against. Second, shielded usage is no longer a minority behavior: approximately 31% of ZEC supply is shielded — a historic high — and shielded transactions have constituted the majority of network transactions since February 2026. The anonymity set that shielded assets would join is therefore

large and growing, which bears directly on the privacy quality any asset layered on the pool can deliver (an asset transfer hides among *all* shielded activity, not merely among other asset transfers).

The remainder of this chapter concerns Orchard specifically, because the ZSA protocol (Chapters 4–5) is defined as an extension of Orchard.

3.2 Orchard in Depth

Orchard (ZIP 224) is the shielded protocol introduced at NU5. Its data model consists of notes, commitments, nullifiers, and actions, bound together by Halo 2 proofs over the Pallas elliptic curve.

Notes. A note is the shielded analogue of a UTXO: a record entitling the holder of particular keys to spend a value. An Orchard note comprises:

- the recipient’s diversified payment address — a diversifier d and diversified transmission key pk_d (Section 3.4);
- a value v (a 64-bit integer of satoshi in base Orchard; the ZSA extension generalizes what the value denominates, per Chapters 4–5);
- a uniqueness element ρ (rho), derived from the nullifier of a note spent in the same transaction, which guarantees global note uniqueness and anchors nullifier non-forgeability;
- a random seed $rseed$, from which the note’s commitment randomness rcm and nullifier-blinding element ψ are deterministically derived (per the derivation discipline introduced in ZIP 212).

A note exists on-chain only in two derived forms: its *commitment* and (to its recipient) its *ciphertext*. The plaintext note travels to the recipient inside the transaction’s encrypted payload, decryptable with the recipient’s incoming viewing key; the ciphertext also carries a 512-byte memo field, an authenticated free-form channel from sender to recipient that ZecBit’s trading protocol uses as a structured envelope (Chapter 13).

Note commitments and the commitment tree. Each new note is committed to with a hiding, binding commitment cm computed with the Sinsemilla hash function (chosen for efficiency inside Halo 2 circuits). The x-coordinate extract of cm is appended to the **note commitment tree**, an incremental Merkle tree of depth 32 (capacity 2^{32} notes) whose internal nodes are Sinsemilla hashes. The tree is append-only; notes are never removed when spent. Every block’s chain state includes the current tree root; historical roots are called **anchors**.

Nullifiers. Spending a note publishes its **nullifier** nf , a value deterministically computable only with the note’s contents and the holder’s nullifier-deriving key nk , via a PRF-based derivation incorporating ρ , ψ , and cm . Nullifiers have two properties that jointly implement double-spend prevention without ownership disclosure: (i) the same note always yields the same nullifier, so consensus can reject a second spend by set membership; (ii) without nk , a nullifier is computationally unlinkable to any note commitment, so observers see that *some* note was spent but not *which*. The chain maintains the set of revealed nullifiers; a transaction revealing a duplicate is invalid.

Actions. Where Sapling had separate Spend and Output descriptions, Orchard uses a unified **action**: each action simultaneously spends exactly one note and creates exactly one note. A transaction's Orchard component is a bundle of $n \geq 1$ actions. Each action publishes:

- the nullifier nf of the spent note;
- the commitment cmx of the new note;
- the ephemeral key and note ciphertext for the recipient (plus an outgoing-viewing-key ciphertext enabling the *sender* to later recover what they sent);
- a value commitment cv , a homomorphic (Pedersen-style) commitment to the action's net value;
- a randomized verification key rk and a **spend authorization signature** under it.

Actions with nothing real to spend or create use *dummy* notes — zero-value notes indistinguishable from real ones — so the action count reveals only an upper bound on activity, not the actual number of real spends or outputs.

Proof, balance, and binding. A single Halo 2 proof covers the entire action bundle, attesting for every action that: the spent note's commitment is in the tree at the stated **anchor** (Merkle membership); the nullifier is correctly derived; the new note's commitment is correctly formed; the value commitment commits to the true net value; and rk is a correct re-randomization of the spender's authorization key (so each action's signature is unlinkable to the account, yet only the key holder could have produced it). Balance across the bundle is enforced homomorphically: the sum of value commitments must open to the bundle's declared net value balance ($valueBalanceOrchard$, the only amount made public, and only insofar as value enters or leaves the shielded pool), which is proven by a **binding signature** whose validity is possible only if the commitments actually sum correctly. The anchor mechanism means a spender proves membership against a historical tree root without revealing *which* leaf, giving each spend an anonymity set of the entire pool history up to that anchor.

This action model is the direct substrate of ZSA: ZIP 226 generalizes the action so that a note carries an *asset base* alongside its value, with balance enforced per-asset — but that specification belongs to Chapter 5, and nothing in it disturbs the structure just described.

3.3 Halo 2 Proving

Orchard's proofs are produced in **Halo 2**, a proving system with three properties that materially shape what ZecBit can ask of end-user devices.

No trusted setup. Halo 2's polynomial commitment scheme (an inner-product-argument construction over the Pallas/Vesta curve cycle) requires no secret-parameter ceremony. Earlier pools depended on multi-party ceremonies whose compromise would have permitted undetectable counterfeiting; Orchard removes this class of risk entirely. For an asset platform this is not an abstract nicety: the soundness of *supply* — the guarantee that exactly the issued quantity of an asset exists — rests on proof soundness, and Orchard's rests on discrete-log hardness plus the protocol's hash assumptions, with no ceremony trust anywhere in the chain of custody.

Recursion-friendliness. Halo 2 descends from the accumulation techniques of the original Halo work: proof verification can be deferred and composed, with the expensive verification component amortized across many proofs. The Pallas/Vesta two-cycle exists to make this efficient. ZecBit’s design does not currently rely on recursive aggregation, but the availability of efficient proof composition is relevant to future work on batch settlement (Chapter 14) and succinct catalog proofs (Chapter 9).

Client-side proving on consumer hardware. Orchard proving was engineered for wallets, not servers: producing the proof for a typical action bundle takes on the order of seconds on ordinary consumer hardware, including current mobile devices, with memory requirements compatible with mobile constraints. The consequence for ZecBit is architectural and absolute: **proving never leaves the holder’s device.** No ZecBit component ever receives spending keys or note plaintexts to “prove on behalf of” a user; the settlement designs of Chapter 14 are constructed so that each party proves locally and contributes proofs, never secrets. Any design that would require server-side proving over user secrets is rejected in this document as out of contract with the observer model of Section 2.3.

Verification is likewise cheap enough (milliseconds per bundle, and batchable) that full validation by Zebra nodes — and by ZecBit’s indexer, which validates rather than trusts its node’s output where feasible — imposes no meaningful throughput constraint at the transaction volumes contemplated in this document.

3.4 Keys and Addresses

Orchard defines a layered key hierarchy in which each level confers strictly less capability than the one above it. ZecBit’s disclosure model (Chapter 12) is essentially an exercise in handing out the *right* level of this hierarchy to the *right* party, so the hierarchy is worth stating precisely.

```

spending key (sk)
  | full control: authorize spends
  ├── ask  - spend authorization key (signs actions, re-randomized per action)
  ├── nk   - nullifier-deriving key
  └── rivk  - ivk commitment randomness
      |
      ▼
full viewing key FVK = (ak, nk, rivk)
  | sees ALL activity: incoming notes, and outgoing
  | spends (can derive nullifiers → detect own spends);
  | CANNOT spend
      |
      ▼
incoming viewing key IVK
  | detects and decrypts INCOMING notes only;
  | cannot see spends, cannot spend
      |
      ▼
diversified addresses addr_d = (d, pk_d)  for many diversifiers d
      one account → up to 288 mutually unlinkable receiving addresses

```

The **spending key** is the root secret; from it derive the spend-authorization key `ask`, the nullifier key `nk`, and the randomizer `rivk`. The **full viewing key (FVK)** — `(ak, nk, rivk)` — can decrypt all notes received by the account, recover what the account *sent* (via outgoing-viewing-key ciphertexts), and, because it contains `nk`, compute the nullifiers of the account's own notes and thereby recognize when they are spent on-chain. An FVK is therefore a complete read-only audit capability: its holder sees the account's entire history and live balance but cannot move a single zatoshi. The **incoming viewing key (IVK)**, derived from the FVK, is strictly weaker: it detects and decrypts incoming notes but cannot derive nullifiers, so it sees deposits without seeing spends or balances-net-of-spends. Hierarchical derivation of these keys from wallet seeds follows ZIP 32's Orchard extension.

Diversified addresses. From one IVK, the account holder can derive a payment address for any of $\sim 2^{88}$ *diversifiers* `d`: each address `(d, pk_d)` is a valid, independent-looking receiving address, and no observer — including someone who possesses two of the addresses — can determine that they belong to the same account without the viewing key. All diversified addresses of an account are scanned by the same key with no per-address cost.

Unified addresses (ZIP 316) package receivers for multiple pools (e.g., an Orchard receiver plus a transparent fallback) into a single encoded address, and correspondingly define unified viewing keys. ZecBit-facing wallet integrations use unified encodings throughout (Chapter 11).

Diversified addresses matter to marketplace UX far more than their brief specification suggests, and they are load-bearing in three places later in this document. *Per-counterparty isolation*: a

seller can give every buyer, every trade, and every deposit flow a fresh address; counterparties who compare notes learn nothing about whether they dealt with the same seller. *Per-listing reception without account sprawl*: Stage-1 and Stage-2 settlement (Chapter 14) can route each trade's proceeds to a distinct address of the same account, so the wallet remains one account with one backup seed while the market sees no address reuse at all. *Contact-point unlinkability*: a creator can publish a "storefront" receiving address publicly while using unlinkable diversified addresses for private dealings, all against a single spending key. On transparent-chain marketplaces, address reuse is the primary deanonymization vector; Orchard makes non-reuse free, automatic, and invisible.

3.5 Selective Disclosure Primitives

Everything in Sections 3.2–3.4 concerns keeping data hidden. Zcash also ships the inverse capability set — proving, on purpose, to a chosen party, that specific hidden things are true — and these primitives are the raw material of ZecBit's opt-in provenance system. Chapter 12 specifies their use in full; here we introduce the three primitives and their scoping properties.

Viewing-key disclosure. A holder may hand a counterparty their FVK or IVK (or the ZIP 316 unified encodings thereof). This is *account-scoped, ongoing* disclosure: an FVK recipient can audit the account's complete past and future activity (all incoming notes, all spends, hence running balances); an IVK recipient sees incoming activity only. Viewing keys are the correct primitive when the desired relationship is standing auditability — an artist's grant administrator, an institution's own compliance function, an estate executor — and the wrong primitive for one-off proofs, because they cannot be scoped to a single transaction and cannot be revoked (only future unlinkability can be recovered, by migrating funds to a fresh account). ZecBit tooling treats FVK hand-over as the *maximal* disclosure act and requires interfaces to present it as such.

Payment disclosure. For *transaction-scoped* proof, the holder of the relevant keys can produce a per-payment disclosure package: a verifiable statement, transmitted out-of-band, that a specific transaction paid a specific value (and memo) to a specific shielded address, without revealing anything about the account's other activity. Verification checks the disclosed data against the on-chain ciphertext and commitment, so a disclosure cannot assert a payment that did not happen. This primitive — one transaction, one recipient of the proof, zero residual visibility — is the workhorse of ZecBit provenance: proof of purchase, proof of ownership transfer to a dispute arbiter, and the holder-driven provenance chains of Chapter 12 are all built from payment disclosures composed across a custody history.

The memo field. Each shielded output carries a 512-byte encrypted memo readable only by the note's recipient (and by viewing-key holders). The memo is sender-authenticated in the sense that it arrives inside the note ciphertext bound to the transaction; it therefore serves as a private, on-chain-anchored messaging channel between exactly the two parties to a payment. ZecBit uses structured memos for trade-protocol signaling and for embedding disclosure-relevant references (asset context, order references) at settlement time, so that a later payment disclosure of the settlement transaction carries its own interpretation with it.

The design consequence to carry forward: Zcash's disclosure primitives are **holder-initiated, recipient-scoped, and granular** (account-level via viewing keys, transaction-level via payment

disclosure). Nothing in the protocol allows an issuer, an operator, or an observer to *compel* disclosure; equally, nothing prevents a holder from proving exactly as much as a given counterparty legitimately requires. ZecBit adds workflow, packaging, and verification tooling around these primitives — it does not add, and could not add, any disclosure capability that the protocol itself does not grant to key holders.

3.6 Network Evolution, Governance, and the Status of ZSA

Zcash evolves through discrete **network upgrades**: coordinated hard forks specified in ZIPs, activated at predetermined block heights, with consensus rules versioned by branch identifiers so that nodes unambiguously agree on which rule set governs which height (ZIP 200). Upgrades pass through public specification, implementation in Zebra, audit, testnet activation, and finally mainnet activation. This process is deliberate and conservative — a property that cuts both ways for ZecBit: it delays ZSA availability, and it is also precisely why the shielded pool ZecBit builds on has an unbroken soundness record worth building on.

The recent and pending upgrades relevant to this document:

- **NU5 (2022)** activated Orchard (ZIP 224) and unified addresses (ZIP 316) — the substrate of everything in this chapter.
- **NU6.3 “Ironwood” (activated 28 July 2026)** introduced the Ironwood shielded pool with its own turnstile, carried by the version 6 transaction format (ZIP 229). Separately, 2026 completed the consolidation of the node ecosystem: zcashd is fully retired and the network is Zebra-only, which simplifies ZecBit’s infrastructure story to a single node implementation.
- **NU7** is, at the time of writing, in its coinholder-vote window (25 August – 14 September 2026). The vote’s scope **explicitly excludes transaction-format changes**.

That exclusion is the single most consequential external fact for ZecBit’s schedule. The ZSA protocol requires additions to the transaction format — an issuance bundle and the OrchardZSA action structure carrying per-asset data (ZIPs 226, 227) — beyond the fields deployed in the current format. Because NU7 will not change the transaction format, **ZSA activation on Zcash mainnet is deferred to a subsequent network upgrade, and no mainnet activation date exists**. This document does not predict one, and no ZecBit plan in Part VI is conditioned on a specific date.

What exists today, stated plainly:

1. **The protocol is complete.** ZIPs 226 and 227 specify ZSA transfer/burn and issuance in full. The design and implementation were led by QEDIT as a multi-year effort funded through Zcash community grants.
2. **The protocol is audited.** Least Authority performed a security audit of the ZSA protocol; the final report was published in January 2025. Chapter 16 discusses the audit’s scope and findings as they bear on ZecBit’s threat model.
3. **The protocol is running on a dedicated public testnet.** A ZSA testnet — a Zebra fork maintained by QEDIT, with a public endpoint — operates with NU7-generation ZSA rules active

from height 1. This is not a simulation: it is the ZSA consensus code validating real issuance and shielded transfer of assets. All of ZecBit's Stage-0 infrastructure (Section 2.4) runs against this network, and every protocol-dependent claim in Chapters 4–6 is exercised there.

4. **Mainnet activation is a governance outcome, not an engineering one.** When a future upgrade carries the ZSA transaction-format changes, ZecBit's migration consists of retargeting node endpoints, re-anchoring the registry, and enabling mainnet settlement per the activation plan of Chapter 18 — reconfiguration of a system already in production against the testnet, not a rewrite.

This staging discipline — build the full stack against the real protocol on its real testnet, hold mainnet plans strictly contingent on Zcash governance, and say so — is design goal 6 of Section 1.4 in operation. The next three chapters now descend into the asset layer itself: issuance (Chapter 4), shielded transfer and burn (Chapter 5), and the asset-class conventions that make ZSAs function as non-fungible items (Chapter 6).

Part II — The Asset Layer

4. Zcash Shielded Assets: the Asset Layer

ZecBit's entire asset model is inherited from a single protocol substrate: Zcash Shielded Assets (ZSA), specified principally in ZIP 226 (transfer and burn) and ZIP 227 (issuance), with transaction-level encoding defined by the version 6 transaction format. This chapter describes that substrate in the depth required to understand every design decision in the remainder of this document. Nothing in ZecBit's NFT model is an application-layer simulation of asset semantics; uniqueness, supply finality, and transfer privacy are properties enforced by Zcash consensus and by the Orchard proving system. The marketplace layers described in later chapters are strictly additive — they interpret protocol facts, they never substitute for them.

4.1 Overview of OrchardZSA: what changes relative to vanilla Orchard

The Orchard shielded protocol, deployed with Zcash NU5, defines a pool of shielded notes in which senders, receivers, and amounts are hidden by zero-knowledge proofs over the Pallas curve. Vanilla Orchard supports exactly one asset: ZEC. Every note commitment and every value commitment implicitly refers to the native asset, and the homomorphic balance check that a transaction neither creates nor destroys value is performed against a single fixed value base point.

OrchardZSA (ZIP 226) generalizes this construction to a multi-asset shielded pool. The generalization is deliberately narrow. Rather than introducing a separate pool per asset — which would fragment the anonymity set — OrchardZSA places all assets, including ZEC, inside the *same* pool, and extends the note structure with one additional element: the Asset Base.

An OrchardZSA note is the tuple:

```
(d, pk_d, v, AssetBase, ρ, ψ, rcm)
```

where `d` and `pk_d` are the diversifier and diversified transmission key of the recipient, `v` is the value, and `AssetBase` is a point in the Pallas group `P*` that uniquely identifies the asset the note carries. For the native asset, the Asset Base is defined to be the pre-existing Orchard value base point, so a ZEC note under OrchardZSA is structurally the same object it was under vanilla Orchard.

Three concrete mechanisms distinguish OrchardZSA from Orchard:

Mechanism	Specified in	Summary
Asset Base in notes and value commitments	ZIP 226	Note commitments and value commitments are parameterized by <code>AssetBase</code> ; balance is checked per asset via homomorphic Pedersen commitments over asset-specific base points

Issuance bundle	ZIP 227	A new transaction bundle in which an issuer, authenticated by a BIP-340 Schnorr signature, creates supply of custom assets via plaintext issue notes
Burn field	ZIP 226	A public list of (AssetBase, value) pairs by which shielded custom-asset value is verifiably destroyed

The value commitment change is the mathematical heart of the design. In OrchardZSA the net value commitment of an action is:

$$cv_net := [v_net] \cdot \text{AssetBase} + [rcv] \cdot R$$

where R is the Orchard randomness base. Because Pedersen commitments are homomorphic only with respect to a common base point, commitments to values of *different* assets cannot be conflated: the transaction-wide binding-signature check partitions balance verification per Asset Base. A transaction that attempted to consume one asset and emit another would fail the balance equation, even though no observer can tell which assets are involved. Cross-asset value conservation is thus enforced blindly — the verifier confirms that every asset balances without learning which assets are present.

Equally important is what does *not* change. The note commitment for a ZEC note is computed identically to vanilla Orchard, and the commitment scheme is constructed so that OrchardZSA note commitments for custom assets are indistinguishable from those for ZEC (ZIP 226). All notes share one commitment tree, one nullifier set, and one anonymity set. This property — a single undifferentiated pool — is what allows ZecBit to claim that holding or trading an NFT is externally indistinguishable from ordinary shielded ZEC activity. Chapter 5 builds directly on it.

4.2 Asset identity: the derivation pipeline

Every ZSA asset is identified by a deterministic chain of derivations that begins with a human-authored description and ends with a curve point embedded in shielded notes. The pipeline, defined in ZIP 227, is:

```

asset_desc          assetDescHash          AssetId
(byte string, -----> (32 bytes) -----> (issuer key ik,
off-chain)          BLAKE2b on-chain          assetDescHash)
                    -256                      |
                                                | EncodeAssetId
                                                v
                                0x00 || ik || assetDescHash
                                (65 bytes)
                                |
                                | BLAKE2b-512
                                v
                                AssetDigest (64 bytes)
                                |
                                | GroupHash^P
                                v
                                AssetBase (Pallas point,
                                32-byte representation)

```

Each step is summarized below.

Step	Input	Function	Personalization / domain	Output size
1	asset_desc (non-empty byte string; SHOULD be well-formed UTF-8)	BLAKE2b-256	"ZSA-AssetDescCRH"	32 bytes (assetDescHash)
2	issuance key ik , assetDescHash	pair construction	—	AssetId := (ik, assetDescHash)
3	AssetId	EncodeAssetId = 0x00 ik assetDescHash	—	65 bytes
4	encoded AssetId	BLAKE2b-512	"ZSA-Asset-Digest"	64 bytes (AssetDigest)
5	AssetDigest	GroupHash^P	"z.cash:OrchardZSA"	Pallas group point (AssetBase , 32-byte encoding)

Three properties of this pipeline deserve emphasis, because ZecBit's asset registry (chapter 7) and provenance model (section 5.5) rest on them.

Identity is bound to the issuer. The AssetId is the *pair* of issuer key and description hash. Two different issuers publishing byte-identical descriptions produce different AssetIds, different AssetDigests, and different Asset Bases. There is no global namespace to squat and no registration race: an asset is cryptographically inseparable from the key that issued it. This is the

foundation of ZecBit's collection model (section 5.2) — imitation of a collection requires forging a Schnorr signature, not merely copying a name.

Only the hash of the description is on-chain — by design. ZIP 227 deliberately places the 32-byte `assetDescHash` on chain rather than `asset_desc` itself, and distributes the full description off-chain. The ZIP characterizes on-chain publication of arbitrary issuer-supplied byte strings as an "attractive nuisance": a consensus-replicated store of unvetted data invites abuse, bloats every full node forever, and creates content-liability surface for node operators. The hash commitment achieves the same binding — anyone holding the preimage can verify it against the chain — at fixed cost. A useful corollary is that `asset_desc` has no practical length ceiling: earlier drafts limited it to 512 bytes, but under the hash-commitment design a description may be arbitrarily long without incurring any chain cost. ZecBit exploits this by placing a structured, versioned description convention inside `asset_desc`; the convention itself is specified in chapter 7.

Derivation is fully deterministic. Given `(ik, asset_desc)`, every party — issuer, wallet, indexer, auditor — computes the identical `AssetBase` with no trusted intermediary. ZecBit's infrastructure never assigns identifiers; it only recomputes and verifies them. This determinism is also the linchpin of the forward-compatibility strategy in section 6.3: any artifact derived on the ZSA testnet is exactly reproducible on mainnet from the same inputs.

A worked sketch: suppose an issuer holds issuance validating key `ik` and authors the description string `zecbit:v1;...;item=0007` (illustrative — the normative convention is defined in chapter 7). The issuer computes `h = BLAKE2b-256("ZSA-AssetDescCRH", asset_desc)`, forms `AssetId = (ik, h)`, and derives `AssetBase = GroupHash^P("z.cash:0rcharZSA", BLAKE2b-512("ZSA-Asset-Digest", 0x00 || ik || h))`. The chain sees `h` inside the issuance action and `AssetBase` implicitly inside every subsequent shielded note; the string itself travels through ZecBit's off-chain distribution layer, where any recipient can re-hash it and confirm the binding.

4.3 Issuance mechanics

4.3.1 ISSUANCE KEYS

Issuance authority is held by a dedicated key pair, separate from spending keys. ZIP 227 defines a hardened ZIP-32 derivation path:

```
m / 227' / coin_type' / 0'
```

with purpose `227'` (mirroring the ZIP number), the standard coin type, and the account index fixed at 0. Master key generation uses the domain string `"ZcashSA_Issue_V1"`. The issuance authorizing key is a 32-byte secret key; the issuance validating key `ik` is the corresponding 32-byte BIP-340 x-only public key. Issuance authorization is a BIP-340 Schnorr signature — the scheme is named `ZSASchnorr` in ZIP 227 — computed over the transaction SIGHASH, so the signature commits to the entire issuance content and cannot be transplanted onto a different transaction.

The choice of BIP-340 is pragmatic: it is a widely reviewed, widely implemented signature standard with mature hardware and multiparty-signing support. For ZecBit creators this means

issuance keys can, in principle, be held in the same operational custody arrangements used for Bitcoin Taproot keys; wallet-level key management is discussed in chapter 11.

4.3.2 BUNDLE AND ACTION STRUCTURE

A transaction that issues assets carries an *issuance bundle* comprising the issuer identification, a vector of issuance actions, and a single authorizing signature covering all of them:

Field	Type / size	Description
issuer	length-prefixed bytes; encoding $0x00 \parallel ik$ (33 bytes)	The issuance validating key
nIssueActions	compactSize	Number of issuance actions
vIssueActions	variable	The issuance actions (layout below)
issueAuthSig	$0x00 \parallel \sigma$ (65 bytes); present iff $nIssueActions > 0$	BIP-340 signature over the transaction SIGHASH

Each *IssueAction* creates supply of exactly one asset:

Field	Size	Description
assetDescHash	32 bytes	BLAKE2b-256 hash of <code>asset_desc</code> (section 4.2)
nNotes	compactSize	Number of issue notes in this action
vNotes	$115 \times nNotes$ bytes	Issue note descriptions (layout below)
flagsIssuance	1 byte	Carries the <code>finalize</code> flag

And each *issue note description* is a fixed 115-byte record:

Field	Size	Description
recipient	43 bytes	Orchard raw address: diversifier <code>d</code> (11 bytes) $\parallel$ <code>pk_d</code> (32 bytes)
value	8 bytes	Note value <code>v</code> (unsigned 64-bit)
<code>p</code>	32 bytes	Derived deterministically (<code>DeriveIssuedRho</code>), binding the note to its issuance context
<code>rseed</code>	32 bytes	MUST be sampled uniformly at random by the issuer

One bundle may contain many IssueActions — issuing many distinct assets — and each action may contain many notes, distributing one asset’s new supply across many recipients. All of it is authorized by the single `issueAuthSig`. This batching is what makes atomic collection genesis possible (section 5.3).

4.3.3 PLAINTEXT ISSUE NOTES AND THEIR PRIVACY BOUNDARY

Issue notes are the one deliberately transparent element of the ZSA design. Unlike Orchard transfer outputs, which appear on chain only as commitments and ciphertexts, issue notes are recorded *in plaintext*: the recipient address components (`d`, `pk_d`), the value, and (via the

enclosing action) the asset are all publicly visible at issuance. There is no proof to generate and no ciphertext to decrypt; supply creation is auditable by direct inspection.

The privacy consequence is precise and worth stating carefully: **privacy for a ZSA asset begins with the first shielded transfer, not at issuance.** The issuance event publicly links an asset to a recipient address chosen by the issuer. Once that note is spent into the Orchard pool — including a self-transfer by the recipient — the asset disappears into the common anonymity set and all subsequent custody is hidden. ZecBit’s minting flows are therefore structured so that issuance recipients are single-use diversified addresses, and wallet guidance (chapter 11) treats the post-mint shielding transfer as a standard hygiene step. For provenance purposes (section 5.5) this transparency is a feature rather than a defect: the issuance event is the asset’s public birth certificate.

4.3.4 REFERENCE NOTES

The first issuance of every asset **MUST** include a *reference note*: an issue note with value 0 addressed to the default diversified address derived from the all-zero Orchard spending key. The reference note carries no economic value and is spendable by no one in practice; its role is structural. It is recorded in global state as `note_ref` and serves as a canonical, consensus-recognized existence anchor for the asset — a fixed on-chain object that indexers, wallets, and proof systems can point to as “the” origin of the AssetId, independent of how the economically meaningful notes were distributed. ZecBit’s registry keys every collection item to its reference note’s issuance transaction.

4.3.5 THE GLOBAL ISSUANCE STATE MACHINE

Zcash consensus maintains a global map:

```
issued_assets : AssetBase → ( balance ∈ {0 .. MAX_ISSUE},  
                               final   ∈ {0, 1},  
                               note_ref )
```

with `MAX_ISSUE =  $2^{64} - 1$` . Every issuance action updates this map, and consensus enforces a simple, irreversible state machine per asset:

```

                IssueAction                IssueAction
                (finalize=0)                (finalize=0)
                +-----+                +-----+
                |                |                |                |
                v                |                v                |
(nonexistent) -----> [ UNFINALIZED ] -+                |
|   first issuance        |                |                |
| (reference note        | IssueAction        |                |
| required)              | (finalize=1)        |                |
|                        v                |                |
|                        [ FINALIZED ] <-----+                |
|                        |                |                |
+--- first issuance        | no further issuance of this
    with finalize=1 -----+ AssetId is ever valid
    (single-action birth    (consensus rule:
        and finalization)    issued_assets(AssetBase).final ≠ 1
                             must hold for any new issuance)

```

While an asset is unfinalized, its issuer — and only its issuer, since the `AssetId` embeds `ik` and the bundle signature is checked against it — may issue further supply in later transactions. Setting the `finalize` bit in `flagsIssuance` moves the asset to the finalized state, after which any transaction attempting further issuance of that `AssetId` is invalid by consensus. The transition is permanent: there is no un-finalize operation, no issuer override, and no governance path that reopens supply. This single bit is the entire enforcement mechanism behind ZecBit’s non-fungibility guarantee in chapter 5: an asset issued with total supply 1 and `finalize = 1` is unique forever, as a matter of consensus validity rather than of contract code or platform policy.

4.3.6 SUPPLY ACCOUNTING

Because issuance is plaintext and burns are public (section 4.5), the circulating supply of every ZSA asset is exactly computable by any chain observer:

```
supply(AssetBase) = Σ issued values - Σ burned values
```

No trusted indexer is required, and the shielded pool cannot distort the figure: transfers within the pool conserve per-asset value by the homomorphic balance check of section 4.1. ZSA thus occupies an unusual point in the design space — *fully transparent supply, fully opaque distribution*. For collectibles this is precisely the desired split: scarcity is a public fact; ownership is a private one.

4.4 Transfers: ZSA notes in the Orchard pool

Once issued, a ZSA note moves exactly as an Orchard ZEC note moves: it is committed into the global note commitment tree, spent by revealing a nullifier and a zero-knowledge proof of a valid Merkle path, and replaced by new output commitments. The transfer transaction reveals:

Observable	Revealed?
Number of Orchard actions	Yes
Which asset(s) are transferred	No
Amounts	No
Sender address(es)	No
Recipient address(es)	No
Whether any custom asset (vs ZEC) is involved at all	No

The last row is the strongest claim and follows from the commitment-scheme design noted in section 4.1: ZSA note commitments are indistinguishable from ZEC note commitments, and both note types inhabit the same tree. An external observer of a ZecBit trade sees an Orchard-bearing Zcash transaction with some action count — the same shape as any shielded ZEC payment of similar complexity.

Split notes. One subtlety of multi-asset transfers requires a dedicated mechanism. Orchard actions pair one spend with one output. When a sender needs to produce more output notes of a custom asset than they have input notes of that asset — for example, paying a seller and returning change from a single input note, or padding a transaction to a uniform size — the extra actions need input slots. Vanilla Orchard fills such slots with *dummy* spends of self-generated fictitious notes. For custom assets, dummies are unsafe: a fictitious input note could declare an arbitrary Asset Base, allowing an attacker to conjure outputs of an asset that was never issued. ZIP 226 therefore introduces *split notes*: a padding input is a copy of a *real, previously issued* note with a `split_flag` set, and the circuit enforces that (a) the note has a valid Merkle path to the commitment tree, proving the Asset Base is one that genuinely exists on chain, and (b) the input value contributed to the balance equation is constrained to zero, so the split input adds no spendable value and its reuse cannot double-spend. The wallet may source the copied note from other notes it is spending, from unspent notes of the same asset, or even from already-spent notes — the zeroed value makes all three safe. Split notes matter to ZecBit chiefly as a wallet-engineering concern (chapter 11), but they also close what would otherwise be a fatal forgery channel against NFT integrity: without them, “issuance” outside the issuance bundle would be possible.

Balance verification. The verifier computes a binding verification key from the sum of all action value commitments, minus a commitment to the transaction’s net ZEC balance, minus commitments to each public burn entry (section 4.5). For a well-formed transaction, all per-asset value components cancel — inputs equal outputs for every hidden asset, with burns deducted —

leaving only the randomness term, which the binding signature validates. Balance is thereby enforced per asset without revealing the asset partition.

4.5 Burn semantics

ZIP 226 provides a single supply-reduction mechanism: the burn field. A transaction may include a set of pairs

Field	Size	Description
AssetBase	32 bytes	The asset being burned
value	8 bytes	Amount destroyed (unsigned 64-bit)

subject to the consensus rules: the Asset Base must not be the native (ZEC) base — ZEC cannot be burned via this mechanism; the value must satisfy $0 < \text{value} \leq \text{MAX_BURN_VALUE}$ where $\text{MAX_BURN_VALUE} = 2^{63} - 1$ (kept within signed-64-bit range for compatibility with value-balance arithmetic and potential future turnstiles); and each Asset Base may appear at most once per transaction. Burned value enters the balance equation of section 4.4 as a public deduction: the prover demonstrates, in zero knowledge, that shielded inputs covered the burned amount.

Burning is deliberately transparent — the asset identifier and amount are revealed — because its purpose is public: it moves value from the opaque distribution side of the ledger to the transparent supply side. The party performing the burn is *not* revealed; only the fact of destruction is.

For collectibles, burn semantics compose with finalization into a property no permissioned platform can offer: **verifiable, irreversible destruction**. Consider a ZecBit NFT — supply 1, finalized. If its (anonymous) holder burns it, every observer can compute $\text{supply} = 1 - 1 = 0$, and because the asset is finalized, supply can never rise again. The item is provably gone, globally and permanently, on the authority of consensus arithmetic alone. Applications include collection-wide supply reductions by holders, redemption flows in which burning an asset evidences the exercise of a claim, and clean retirement of obsolete series. ZecBit surfaces burn events in its registry as terminal lifecycle states (section 5.6).

4.6 Fees

ZSA transactions pay fees in ZEC under the conventional-fee rule of ZIP 317:

```
conventional_fee = 5000 zatoshi × max(2, logical_actions)
```

where `logical_actions` aggregates the transaction's logical workload: `ceil`-scaled transparent input and output sizes, twice the JoinSplit count, the maximum of Sapling spend and output counts, and the Orchard action count. For the Orchard-only transactions that dominate ZecBit usage, the rule reduces to $5000 \times \max(2, \text{nActionsOrchard})$ zatoshi. A typical two-action shielded transfer — the shape of most single-item NFT movements — carries the 10,000-zatoshi conventional fee. Because fee is a function of action count and action count reveals nothing

about assets, fee analysis does not erode the indistinguishability claims of section 4.4, though wallets should follow the ecosystem-standard fee rule uniformly to avoid becoming distinguishable by fee behavior.

One aspect remains explicitly unsettled: **the fee weight of issuance actions**. ZIP 227 states that the conventional fee is to account for issuance actions and for the creation of new custom assets, and defers the calculation to ZIP 317; as of mid-2026 the harmonization between the two documents is still pending. ZecBit treats this as an open protocol parameter: minting-cost figures are not quoted anywhere in this paper, the fee module in ZecBit’s transaction builder is isolated behind a versioned interface (section 6.5), and creators are advised that issuance pricing on mainnet will follow whatever rule the final specifications adopt. It is a reasonable expectation — but only an expectation — that per-action and per-new-asset weights will make large mints linear in cost, which is compatible with the batch-minting model of section 5.3.

5. The ZecBit NFT Model

Chapter 4 described the raw protocol. This chapter defines what ZecBit builds from it: a precise, minimal mapping from ZSA primitives to the familiar vocabulary of non-fungible tokens. The mapping is intentionally thin. Wherever a property can be obtained from consensus, ZecBit takes it from consensus; application conventions are reserved for properties the protocol deliberately does not provide.

5.1 Definition

Definition. A *ZecBit NFT* is a ZSA AssetId whose total issuance is exactly 1 at the fundamental unit level and whose issuance is finalized (`finalize = 1`).

Both clauses are consensus-checkable facts about the global `issued_assets` map (section 4.3.5). Supply 1 means exactly one indivisible unit of the asset exists; finalization means no transaction can ever create another. Uniqueness and non-reissuability are therefore properties of Zcash validity rules — not of a smart contract’s code, not of an indexer’s interpretation, and not of ZecBit’s infrastructure. ZecBit can neither mint a second copy of a creator’s item nor prevent an item from existing; it can only *recognize* items whose on-chain state satisfies the definition.

This construction follows the NFT recipe described in ZIP 227 itself: several issuance actions in a single bundle, each with an asset value of 1 at the fundamental unit level and `finalize` set for each, with a collection formed by a common issuer key. ZecBit is an implementation of that recipe, not a deviation from it.

It is instructive to place this against the two dominant NFT models:

Property	ZecBit / ZSA	ERC-721 (EVM)	Ordinals (Bitcoin)
Uniqueness guarantee	Consensus-enforced: finalized supply of 1 in global issuance state	Contract-enforced: uniqueness of <code>tokenId</code> is a property of the contract’s code; upgradeable or buggy contracts can violate it	Indexer-enforced: inscription identity and transfer tracking are conventions applied by off-chain indexers to Bitcoin transactions
Metadata binding	32-byte hash of <code>asset_desc</code> embedded in the AssetId itself; immutable by construction	<code>tokenURI</code> per contract; may point to mutable off-chain data unless the contract freezes it	Inscribed content is on-chain and immutable, at full on-chain storage cost
Ownership visibility	Hidden: custody is a shielded note in the Orchard pool	Public: <code>ownerOf</code> is world-readable state	Public: controlling UTXO is world-readable
Transfer privacy	Sender, receiver, asset, and amount all hidden; transfers indistinguishable from ZEC payments	None; full transfer graph public	None; full transfer graph public

Royalty enforcement	None at protocol level; handled at ZecBit's trading layer (chapters 13–15)	Not enforced at protocol level; ERC-2981 is advisory and marketplace compliance is voluntary	None
Chain / status	Zcash, transaction format v6; live on dedicated ZSA testnet, mainnet pending a future network upgrade (chapter 6)	EVM networks, deployed at scale	Bitcoin, deployed at scale

Two rows merit comment. On *metadata binding*, ZSA is distinctive in fusing the metadata commitment into the asset's identity: changing one byte of `asset_desc` yields a different `assetDescHash`, hence a different AssetId, hence a different asset. There is no pointer that can rot or be repointed; there is only a commitment that either verifies or does not. The off-chain availability of the preimage is a real operational concern, and chapter 7 specifies ZecBit's distribution and pinning conventions for it. On *royalty enforcement*, ZSA offers nothing, and this paper claims nothing: shielded transfers are invisible, so protocol-level royalty extraction is structurally impossible. ZecBit's approach operates entirely within its trading protocols and is described in chapters 13–15; it is out of scope here.

5.2 Collections: the issuer key as the collection

The ZSA protocol has no native concept of a collection. What it has is stronger in one dimension and weaker in another: every AssetId cryptographically embeds its issuer's validating key `ik` (section 4.2). ZecBit therefore defines:

Definition. A *ZecBit collection* is the set of ZecBit NFTs whose AssetIds share a common issuer key `ik`.

Membership in a collection is thus a verifiable cryptographic fact. Given any item's AssetId, anyone can read off the issuer key and check it against the creator's published key (creator identity and key publication are handled by the registry, chapter 7). Counterfeiting a collection would require producing a valid issuance bundle signed under the genuine `ik` — that is, forging a BIP-340 signature. A copycat issuing visually identical items under a different key produces assets whose AssetIds differ at the first derivation step; no indexer confusion or marketplace policy can blur the distinction, because the distinction is arithmetic.

Within a collection, individual items are distinguished by their `asset_desc` preimages. ZecBit adopts a deterministic item-numbering convention carried inside `asset_desc` — a structured field identifying the collection, the item index, and item-specific content commitments — so that the ordered item list of a collection is reconstructible by hashing candidate descriptions and matching them against on-chain `assetDescHash` values. The normative grammar of this convention is specified in chapter 7 and is only forward-referenced here.

Honesty about the boundary requires listing what the protocol does **not** provide, and what therefore lives in upper layers:

- **No on-chain enumeration order.** The chain records issuance actions in transaction order, but nothing marks "item 7 of 100" as such; ordering is a property of the `asset_desc` convention

and of the registry's interpretation of it.

- **No collection-level metadata object.** There is no on-chain record naming the collection, describing it, or declaring its intended size. These are registry-layer facts (chapter 7), attested by the creator under the same issuer key but not enforced by consensus.
- **No consensus notion of "collection completeness."** An issuer key remains capable of issuing new AssetIds forever (each individually finalized or not). A creator's declaration that a collection is closed at N items is a signed off-chain commitment, surfaced and checked by the registry, not a chain rule. Collectors for whom this distinction matters can verify that all N items are finalized — that much *is* consensus-enforced — while treating "no future sibling items" as a reputational commitment.

This division of labor is deliberate: consensus enforces the expensive properties (uniqueness, issuer binding, supply finality), and the application layer supplies the cheap, evolvable ones (naming, ordering, presentation).

5.3 Batch minting and atomic collection genesis

Because an issuance bundle may contain many IssueActions (section 4.3.2), an entire collection can be born in a single transaction: one bundle, N actions, each action carrying its own `assetDescHash`, its reference note, one value-1 note to the creator's chosen address, and `finalize = 1`. A single `issueAuthSig` authorizes the whole set.

Atomicity here is not cosmetic. A collection minted in one transaction either exists in full or does not exist at all; there is no partially-minted state in which early items circulate while later items are pending, no window in which a mempool observer can front-run the "rest" of a collection, and no reorganization scenario that splits a collection's genesis across diverging histories. The collection's birth is one event with one block height, one transaction identifier, and one signature — which also makes the provenance record of section 5.5 maximally simple.

Implementation validation (ZecBit team, 29 August 2026, public ZSA testnet). A batch of 10 unique finalized assets — 10 IssueActions, each issuing a single value-1 note and setting `finalize = 1` — was constructed as a single transaction, submitted to the public ZSA testnet (QEDIT's Zebra fork, chapter 6), and confirmed at height 251. Proof generation for the transaction completed in approximately 3 seconds on a consumer-grade CPU. This is the only performance measurement reported in this paper; it is offered as evidence that the batch-genesis pattern is practical with current tooling, not as a benchmark of eventual mainnet performance.

The transaction is public and independently checkable, against a node this project does not operate:

```
txid      611ddfd4459e813b86b935fc05217f929e29c37cdc96ba8f23fc647e30c26317
height    251
node      https://dev.zebra.zsa-test.net

curl -s -X POST https://dev.zebra.zsa-test.net \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc": "2.0", "id": 1, "method": "getrawtransaction",
    "params": ["611ddfd4459e813b86b935fc05217f929e29c37cdc96ba8f23fc647e30c26317", 1]}'
```

Each item's `asset_desc` is `zmd1|zecbit-genesis|<index>`; recomputing `BLAKE2b-256` over that string with personalization `ZSA-AssetDescCRH` (section 4.2) yields an `assetDescHash` that appears verbatim in the transaction's issuance bundle. A reference verifier that performs this check end to end, using only the description string and a public node, is published alongside this document.

For collections too large for a single transaction (whether by policy, by transaction-size limits, or by fee considerations under the eventual issuance-fee rule of section 4.6), ZecBit's minting pipeline shards the collection into a sequence of bundles issued by the same key, and the registry records the genesis as a transaction *set*. Sharded genesis weakens per-collection atomicity to per-shard atomicity; the registry marks such collections accordingly so that the provenance record remains exact.

5.4 Editions and semi-fungibles

The value-1-finalized pattern is one point in a spectrum the protocol supports. Issuing an asset with value $N > 1$ and `finalize = 1` creates a *capped edition*: exactly N interchangeable units, supply frozen forever. This is the natural representation for numbered print runs, event passes, or any artifact whose instances are intentionally identical.

The trade-offs against per-item AssetIds are structural:

Property	N distinct AssetIds (value 1 each)	One AssetId, value N
----------	--------------------------------------	------------------------

Per-unit identity	Yes — each unit is a distinct asset	No — units are mutually fungible
Per-unit provenance / burn	Yes — each unit's destruction is individually visible	Aggregate only — supply decreases, units are not distinguishable
Issuance cost	N issuance actions	One issuance action
Partial transfers	Not applicable (indivisible)	Yes — a holder of k units may transfer any subset, shielded
Suitable for	Unique works, 1-of-1s, distinguishable items	Uniform series, passes, editions

Both patterns coexist in the same shielded pool with identical privacy properties. ZecBit's application layer treats large uniform series as a distinct asset class from unique items — they are catalogued, displayed, and traded through interfaces appropriate to fungible-within-series objects rather than through the NFT interfaces of chapters 13–15. The distinction is purely an application-layer classification reflecting the structural differences in the table above.

An unfinalized asset (issuer retains the right to mint more) is also expressible and is occasionally appropriate — open editions, ongoing series — but ZecBit's registry displays finalization status prominently and its NFT definition (section 5.1) excludes unfinalized assets by construction: an item whose supply *could* grow is not, in ZecBit's vocabulary, an NFT.

5.5 The provenance model: public birth, private life

Provenance under ZecBit decomposes into three layers with sharply different visibility:

- 1. Public origin.** The issuance event is fully transparent (section 4.3.3): the issuer key, the block height and time, the set of `assetDescHash` values, the finalization flags, the initial recipient addresses, and the reference notes are all on-chain plaintext. This is the item's birth certificate, and it is as auditable as any transparent-chain mint — arguably more so, since supply arithmetic is closed under consensus (section 4.3.6).
- 2. Private chain-of-custody.** From the first shielded transfer onward, custody is invisible. No public record links the item to its current holder, to the number of times it has changed hands, or to the prices paid. Where traditional NFT provenance is a public transfer graph, ZecBit provenance deliberately is not: the *existence* and *origin* of the item are public facts, while its *history of hands* is nobody's business by default.
- 3. Optional selective disclosure.** Zcash's viewing-key and payment-disclosure mechanisms allow a holder to prove facts about their own custody — to a counterparty, an auditor, an insurer, or a court — without publishing them to the world. A seller can demonstrate current ownership to a prospective buyer; a collector can evidence acquisition to an appraiser. The disclosure toolkit ZecBit standardizes for these flows is specified in chapter 12; here it suffices to note that the privacy of layer 2 is a default, not a prison.

The resulting model — *public origin, private custody, provable-on-demand ownership* — inverts the assumption, inherited from transparent chains, that provenance requires a public ledger of owners. What collecting actually requires is (a) certainty that the item is genuine and unique,

supplied by layer 1 plus consensus; and (b) the current holder’s ability to prove possession when they choose, supplied by layer 3. The public transfer graph of transparent chains turns out to be an artifact of their architecture, not a requirement of provenance — and it is the layer that leaks collector identity, wealth, and trading behavior. ZecBit removes exactly that layer.

5.6 Lifecycle

The complete lifecycle of a ZecBit NFT, annotated with the visibility of each stage:

```
[ CREATED ]                                off-chain: creator authors asset_desc,
|                                           derives AssetId under issuance key ik
| issuance bundle                          (nothing on-chain yet)
| (IssueAction, value=1,
|   finalize=1, reference note)
v
[ ISSUED / FINALIZED ]                    on-chain, PUBLIC: birth certificate --
|                                           issuer, time, assetDescHash, recipient,
| first shielded transfer                  supply=1 frozen forever
| (often a self-transfer)
v
[ PRIVATELY HELD / TRADED ] <----+ on-chain, SHIELDED: custody and trades
|           |                           | hidden in the Orchard pool; transfers
|           +--- shielded sale ---+ indistinguishable from ZEC payments;
|               or transfer          selective disclosure at holder's option
|                                   (ch. 12); trading protocols ch. 13-15
| burn (optional, terminal)
v
[ BURNED ]                                on-chain, PUBLIC: (AssetBase, 1) in the
                                           burn field; supply provably 0, forever
                                           (finalization forbids reissuance)
```

The lifecycle has exactly two public transitions — the first and, if it ever occurs, the last — bracketing an arbitrarily long private middle. Every later chapter of this paper elaborates one region of this diagram: metadata and registry semantics attach to the CREATED→ISSUED edge (chapter 7), wallets operate the private middle (chapter 11), disclosure instruments pierce it selectively (chapter 12), and trading protocols animate it (chapters 13–15).

6. Protocol Staging and Compatibility

ZecBit is built on a protocol layer that is complete, audited, and running — but not yet on Zcash mainnet. This chapter states that situation plainly, describes the transaction-format substrate ZSA arrives with, and specifies the engineering strategy by which ZecBit remains correct across the transition.

6.1 The staging reality

The ZSA protocol stack (ZIPs 226 and 227, over transaction format v6) has the following deployment status, stated without embellishment:

- The specification and implementation are **complete and externally audited**, with a security audit by Least Authority concluded in January 2025.
- ZSA functionality is **live today on a dedicated public testnet**, operated on a fork of the Zebra node implementation maintained by QEDIT.
- ZSA requires **transaction format v6**, whose mainnet activation is tied to a Zcash network upgrade. ZSA was **deferred beyond NU7**, and **no mainnet activation date exists** at the time of writing.

ZecBit’s deployment plan mirrors this staging rather than fighting it:

```

Protocol:  [ ZSA testnet (live) ] -----> [ future NU: v6 + ZSA on mainnet ]
           |                               |
ZecBit:    [ full stack on testnet ] -----> [ same stack, mainnet endpoints ]
           |                               |
           |                               |
           | minting, registry, wallet      | creators re-issue collections
           | tooling, trading protocols     | (sec. 6.3); registry marks
           | exercised end-to-end           | testnet-era provenance as such

```

Everything described in this paper — minting pipelines, the registry, wallet integrations, trading flows — is built and exercised against the live testnet now. This is not a compromise position: it means that by the time mainnet activation occurs, every ZecBit component will have an operational history against real protocol rules, and the mainnet transition reduces to re-pointing infrastructure at new endpoints plus the deterministic re-issuance process of section 6.3. The empirical validation reported in section 5.3 is a product of this posture.

The corresponding honesty requirement runs in the other direction: assets issued on the testnet are testnet artifacts. They carry no mainnet existence and no continuity of ownership into mainnet, and ZecBit’s registry labels them as such permanently (section 6.3).

6.2 Transaction format v6, at the level ZecBit depends on it

Transaction format v6 is the container within which all ZSA operations travel. ZecBit interacts with it through a serialization layer, so only its structural outline matters here:

- **An issuance bundle**, encoded as described in section 4.3.2: issuer encoding, a compactSize-counted vector of IssueActions (each: 32-byte `assetDescHash`, compactSize note count, 115-byte issue note descriptions, 1-byte `flagsIssuance`), and the 65-byte-encoded `issueAuthSig` when actions are present.
- **Burn fields** within the Orchard-side bundle: a vector of 40-byte entries (32-byte `AssetBase`, 8-byte value) per section 4.5.
- **Action groups**, a v6 structural innovation that organizes Orchard/ZSA actions into groups, each carrying an `nAGExpiryHeight` field — a per-group expiry height (or 0 for no constraint) after which a transaction containing the group cannot be mined. In the initial ZSA deployment this structure functions chiefly as forward-compatibility scaffolding: it reserves room for later protocol work in which independently authorized action groups with independent expiries could be composed — machinery relevant to future atomic-swap designs (forward-referenced in chapters 13–15) without being load-bearing for ZecBit’s launch functionality.
- **Housekeeping changes** relative to v5, including explicit fee representation in the transaction header and a restructured memo bundle.

One meta-observation belongs in this chapter rather than a footnote: the v6 encoding was originally specified in ZIP 230, and at the time of writing that document has been withdrawn in favor of a successor specification, with field-level details migrating between documents as the format is finalized. The encodings cited above were verified against the specifications current as of mid-2026, and the fact that they are still in motion is itself a datum — it is precisely why ZecBit isolates all serialization behind the versioned layer described in section 6.5, and why this paper cites structure rather than byte-offsets wherever the distinction matters.

6.3 Forward-compatibility strategy: determinism as the bridge

ZecBit’s central compatibility principle is:

Everything ZecBit stores is either (a) derived from chain data, or (b) derived from deterministic conventions over creator-supplied inputs. Nothing depends on infrastructure-assigned identifiers.

The consequences are mechanical. An `AssetId` is a pure function of `(ik, asset_desc)` (section 4.2); a collection is a pure function of an issuer key (section 5.2); item ordering is a pure function of the `asset_desc` convention (chapter 7). None of these depend on which network the derivation is executed against. Therefore every artifact created on the ZSA testnet is **exactly reproducible on mainnet**: a creator who holds their issuance key and their description preimages can re-derive the identical `AssetIds` and re-issue the identical collection at the genesis of mainnet ZSA activation, and every party can verify the correspondence by recomputing the hashes.

ZecBit’s mainnet transition policy follows directly:

1. **Creators re-issue.** Mainnet collections are created by the creators themselves, signing with the same issuance keys (or with fresh keys, at their option — key migration is a supported, registry-attested event). ZecBit performs no custodial migration and asserts no bridging claims; there is no mechanism by which it could, and the absence of such a mechanism is a feature of the trust model.
2. **The registry preserves history with labels.** Testnet-era collections remain permanently visible in the ZecBit registry, marked as testnet artifacts. Where a mainnet re-issuance corresponds to a testnet original (same `ik`, same `asset_desc` set — a verifiable correspondence), the registry links the two records, giving early collections a documented pre-mainnet history without conflating the two networks' assets.
3. **No continuity of ownership is claimed at the protocol layer.** Testnet custody does not, by itself, entitle anyone to mainnet assets: no consensus rule carries a holding across the boundary, and no cryptographic construction can manufacture one. Distribution of re-issued mainnet collections is therefore a creator decision, outside protocol and outside ZecBit's authority.

A creator may nonetheless commit to one, and the distinction matters because ZecBit is itself a creator for its own collections. Where such a commitment is made — as it is for the Genesis collection — it is a **published policy backed by an auditable record**, not a protocol guarantee: the pre-mainnet assignment is committed in the anchored log (12A), each entry is claimable exactly once, and the re-issuance signed by the same issuance key is verifiable against what was committed. What that buys a holder is a promise whose breach is publicly detectable rather than deniable. What it does not buy is consensus enforcement, and no honest document should describe it as such.

This is the least-trust transition available: it requires believing no bridge, no snapshot, and no operator — only BLAKE2b and BIP-340.

6.4 Dependencies and their maturity

ZecBit's stack rests on a small number of external components whose maturity varies. The table below states each honestly, with the mitigation ZecBit maintains:

Component	Status (mid-2026)	ZecBit mitigation
ZSA protocol specification (ZIPs 226/227, v6 format)	Complete; audited (Least Authority, Jan 2025); encoding documents still consolidating; deferred beyond NU7, no mainnet date	Versioned serialization and consensus-rules layer (sec. 6.5); testnet-first deployment so specification drift is caught operationally
Zebra ZSA fork (QEDIT)	Live, operating the dedicated public ZSA testnet; a fork, not yet merged into mainline Zebra	ZecBit operates its own node instances pinned to specific reviewed revisions; upgrades are staged against an internal environment before adoption; node access abstracted behind an internal RPC interface

librustzcash ZSA branches	Feature branches; not part of mainline releases	Dependencies vendored and pinned by commit; ZecBit tracks upstream merge progress and maintains a thin compatibility shim so a future switch to mainline releases is a dependency bump, not a rewrite
ZSA-capable wallet ecosystem	Effectively one ZSA-capable wallet exists today	ZecBit ships its own wallet tooling (chapter 11) and does not gate launch on third-party wallet adoption; wallet interfaces follow the standard ZSA note-management model so future wallets can interoperate
Proving/parameter stack (Orchard circuit, ZSA variant)	Implemented and covered by the protocol audit; performance validated only anecdotally (sec. 5.3)	No performance claims made beyond the single reported measurement; proving executed client-side, so improvements accrue without ZecBit changes

The one-wallet reality deserves a plain sentence: today, a user's practical access to ZSA assets runs through a single wallet implementation or through ZecBit's own tooling. This is the thinnest point of the ecosystem and the strongest argument for the wallet-layer investment described in chapter 11.

6.5 What breaks if X changes: risk enumeration and isolation layers

A protocol that has not activated on mainnet can still change. ZecBit enumerates the plausible changes and maintains an isolation layer for each, so that any single change is absorbed in one module:

Protocol change risk	Plausibility & impact	Isolation layer
Issuance fee rule finalized differently than anticipated (ZIP 227 / ZIP 317 harmonization, sec. 4.6)	Expected to be resolved before mainnet; affects minting cost, potentially batch-mint economics	Fee computation is a single versioned module in the transaction builder; no fee constants appear anywhere else in the stack or in this paper
asset_desc constraints tightened (e.g., a consensus MUST on UTF-8 well-formedness, or a size bound reinstated)	Possible; current rule is non-empty + SHOULD-UTF-8, with historical drafts having had a 512-byte limit	ZecBit's description convention (ch. 7) already mandates well-formed UTF-8 and keeps the on-chain-hashed preimage compact by placing bulk content behind hash references inside <code>asset_desc</code> ; a size bound would require no convention change
v6 encoding details move again (ZIP 230 withdrawal/successor, field reshuffles, action-group semantics)	Ongoing, observed in practice (sec. 6.2)	All (de)serialization confined to one library keyed by transaction version and consensus branch; upper layers consume parsed structures, never raw bytes
Swap/trading ZIP evolution (future ZIPs for atomic ZSA swaps, action-group composition)	Actively evolving area; affects chapters 13–15, not the asset layer	Trading protocols specified against capability interfaces ("atomic two-asset settlement") rather than against specific ZIP mechanics; chapter 13 details fallback settlement paths that require only the ch. 4 primitives

Testnet resets or fork divergence	Routine for a dedicated testnet	Registry treats network instances as first-class: every record is scoped to a network/branch identifier, so a reset produces a new scope rather than corrupted history; determinism (sec. 6.3) makes re-population mechanical
NU scheduling changes (further deferral, or earlier-than-expected activation)	Outside anyone's control; date risk only	No component assumes a date; mainnet enablement is a configuration change plus the sec. 6.3 policy, rehearsed against the testnet

The pattern across all six rows is the same: ZecBit binds itself to the *stable core* of the protocol — the derivation pipeline of section 4.2, the issuance state machine of section 4.3.5, the indistinguishability and balance properties of section 4.4, and burn arithmetic of section 4.5 — and treats everything else as replaceable plumbing behind narrow interfaces. Those core properties are the ones the audit covered, the ones the ZIPs define as consensus semantics rather than encoding, and the ones every chapter that follows depends on. The next chapter builds the first application layer on top of them: the metadata and registry conventions by which a 32-byte hash on a private chain becomes a legible, verifiable catalogue of collections.

Part III — Metadata, Registry, and Indexing

7. ZMD-1: The ZecBit Metadata Standard

This chapter specifies **ZMD-1** (ZecBit Metadata, version 1), the normative standard by which ZecBit binds a Zcash Shielded Asset to its human-facing metadata: name, description, media, traits, and creator attestation. ZMD-1 is the first ecosystem metadata standard for ZSA-based non-fungible assets; nothing in it depends on ZecBit infrastructure, and any wallet, indexer, or marketplace MAY implement it independently.

Conventions. The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this chapter and throughout the remainder of this document are to be interpreted as described in RFC 2119 and RFC 8174 when, and only when, they appear in all capitals, as shown here.

7.1 Motivation and Requirements

7.1.1 THE HASH-ONLY ON-CHAIN REALITY

The ZSA protocol (ZIP 227) does not place asset descriptions on chain. For each asset, the chain records only a 32-byte BLAKE2b-256 digest of the issuer-chosen `asset_desc` byte string — the *assetDescHash*. The full `asset_desc` preimage is distributed off-chain by whatever means the issuer and the ecosystem arrange. The asset's protocol-level identity is the pair

```
AssetId = (issuer_ik, assetDescHash)
AssetBase = 32-byte Pallas point encoding derived from AssetId
```

where `issuer_ik` is the issuance validating key. A *collection*, in ZecBit terms, is the set of assets sharing a common issuer key.

Two consequences follow. First, the chain cannot answer "what is this asset called?" or "what does it look like?"; it can only confirm or refute a *candidate* answer, by hashing. Second, the binding between an asset and its metadata is exactly as strong as the scheme used to construct `asset_desc` — the only issuer-controlled input to the on-chain commitment. ZMD-1 is a discipline for constructing `asset_desc` so that the resulting hash commits, transitively, to a complete and independently verifiable metadata document.

7.1.2 ANTI-PHISHING CONSTRAINTS INHERITED FROM ZIP 227

ZIP 227 imposes a display rule that ZMD-1 treats as foundational: wallets **MUST NOT** present the raw `asset_desc` string to users as the asset's name. The rationale is anti-phishing — `asset_desc` is chosen unilaterally by the issuer, so any issuer can produce a string that reads "Official Wrapped ZEC" or imitates a well-known collection. ZIP 227 anticipates *petname systems* and *trusted registries* as the mechanisms by which wallets attach safe display names to assets, but defines neither. At the time of writing, no ecosystem metadata standard for ZSAs exists; ZMD-1

fills that gap for the non-fungible case, and Chapter 8 specifies the registry and petname layers that complete the picture.

ZMD-1 therefore distinguishes sharply between three things that other NFT ecosystems tend to conflate:

1. the **identity string** (`asset_desc`), which is machine-facing, hash-committed, and never displayed raw;
2. the **manifest**, a signed JSON document carrying names, descriptions, media references, and traits;
3. the **display name**, which is assigned by the client layer according to petname rules (§8.4) and is always accompanied by verification state.

7.1.3 REQUIREMENTS

ZMD-1 is designed against the following requirements. Implementations claiming ZMD-1 conformance MUST satisfy R1–R5; R6 and R7 are design goals that shape the standard but are enforced at the registry and client layers respectively.

#	Requirement
R1	Deterministic binding. Given an <code>AssetBase</code> observed on chain, there is at most one valid <code>asset_desc</code> , and (in the full form) at most one valid manifest, bound by cryptographic hash. Two honest verifiers presented with the same inputs MUST reach the same accept/reject verdict.
R2	Offline verifiability. Verification of the item↔metadata binding MUST be possible using only: the on-chain issuance record, the candidate <code>asset_desc</code> , the candidate manifest, and the media bytes. No network access, trusted server, or registry query is required for <i>verification</i> , even where servers are used for <i>distribution</i> .
R3	Mirror-ability. All metadata artifacts MUST be self-contained byte strings whose integrity is checkable from their identifiers, so that any party can mirror them without being trusted (§8.2).
R4	Media integrity. Every media object MUST be bound by a cryptographic hash of its exact bytes, independent of the transport (IPFS, HTTP gateway, local file) over which it was fetched.
R5	Immutability of committed metadata. Where the full form is used, metadata is frozen at issuance time; no party, including the creator, can alter it after the fact without changing the asset's identity.
R6	No trusted publisher. The standard MUST NOT require the existence of any particular registry, gateway, or company for correctness. Registries accelerate lookup; they never adjudicate truth.
R7	Safe display. The standard MUST be consumable by clients in a way that satisfies the ZIP 227 display rule: raw <code>asset_desc</code> is never a name, and displayed names carry verification context.

7.2 The `assetdesc` Encoding

7.2.1 GRAMMAR

A ZMD-1 `asset_desc` is a UTF-8 byte string in one of two forms: **minimal** or **full**. Both are pipe-delimited, with a leading version tag. The grammar below is given in ABNF (RFC 5234); note that because every permitted character is in the ASCII range, the UTF-8 encoding is byte-identical to ASCII.

```

asset-desc      = minimal-form / full-form

minimal-form    = tag SEP collection SEP index
full-form       = tag SEP collection SEP index SEP manifest-cid SEP content-hash

tag             = %s"zmd1"                ; lowercase, exact
SEP             = %x7C                    ; "|"

collection      = slug
slug            = slug-edge [ *62slug-char slug-edge ]
                                   ; 1..64 chars total, see rules below
slug-edge       = %x61-7A / DIGIT         ; a-z 0-9
slug-char       = slug-edge / "-"

index           = "0" / ( nz-digit *9DIGIT )
                                   ; decimal, no leading zeros, <= 10 digits
nz-digit        = %x31-39

manifest-cid    = 1*112cid-char          ; CIDv1, base32 lowercase (RFC 4648 §6,
cid-char        = %x61-7A / %x32-37      ; no padding), multibase prefix "b"

content-hash    = 64hex-lower            ; BLAKE2b-256 digest, lowercase hex
hex-lower       = DIGIT / %x61-66

```

Field rules:

- **tag**. The literal `zmd1`. Parsers MUST match case-sensitively. A string beginning with any other tag (including `zmd2`, see §7.7) MUST NOT be interpreted under this specification.
- **collection** (the *collection slug*). Between 1 and 64 characters from `[a-z0-9-]`. It MUST NOT begin or end with a hyphen and MUST NOT contain two consecutive hyphens. The slug is a machine identifier, not a display name; uppercase, whitespace, and non-ASCII characters are deliberately excluded to eliminate homoglyph and normalization ambiguity in the hashed identity string. The human-readable collection name lives in the manifest.
- **index**. The item's ordinal within the collection, as a decimal integer without leading zeros, in the range 0 to 4294967295 inclusive. Issuers SHOULD assign indices sequentially from 0 or 1 and MUST NOT reuse an index within a collection. Note that the protocol does not enforce index uniqueness — two distinct `asset_desc` strings with the same `(collection, index)` but different trailing fields would yield distinct assets. Clients therefore MUST treat `(issuer_ik, collection, index)` collisions across distinct AssetBases as a flaggable anomaly (§8.3).
- **manifest-cid** (full form only). The IPFS CIDv1 of the canonical manifest bytes (§7.3.3), in base32 lowercase multibase encoding (prefix `b`). Issuers MUST use CIDv1; CIDv0 (`Qm...`, base58) is NOT permitted, because its mixed-case alphabet conflicts with the lowercase byte

discipline of this grammar and its implied dag-pb wrapping is ambiguous for small documents. The CID SHOULD use the `raw` multicodec so that the addressed block is exactly the canonical manifest byte string.

- **content-hash** (full form only). The BLAKE2b-256 digest of the canonical manifest bytes, as 64 lowercase hexadecimal characters. This field, not the CID, is the normative binding: verifiers MUST check `content-hash` and MAY additionally check the CID. The CID exists for retrieval; the hash exists for truth. Using BLAKE2b-256 here keeps the entire verification chain — `assetDescHash`, `content hash`, `media hashes` — on a single primitive already required by every ZSA validator.

7.2.2 MINIMAL VERSUS FULL FORM

The two forms trade immutability against flexibility:

Property	Minimal form	Full form
On-chain commitment covers	collection slug + index	slug + index + entire manifest + (transitively) all media bytes
Manifest bound by	creator signature only (§7.3)	cryptographic hash and creator signature
Metadata mutable after issuance	Yes — creator can publish, revise, and re-sign a manifest at any time	No — any change to the manifest changes <code>content-hash</code> , hence <code>assetDescHash</code> , hence the asset identity
Manifest must exist at issuance time	No	Yes
Typical <code>asset_desc</code> length	≤ 80 bytes	≈ 145–200 bytes

The full form is RECOMMENDED for all art and collectible issuance: it delivers requirement R5 (frozen metadata) and makes the entire metadata document as tamper-evident as the chain itself. The minimal form is appropriate when metadata legitimately cannot be final at issuance time (e.g., reveal mechanics), with the explicit caveat — which clients MUST surface — that minimal-form metadata is authenticated by the creator’s ongoing signature, not by the chain.

Both forms fit comfortably inside the protocol’s bound on `asset_desc` length (512 bytes under ZIP 227 at the time of writing). ZMD-1 additionally requires that a conformant `asset_desc` MUST NOT exceed 256 bytes, reserving headroom for the protocol bound and discouraging abuse of the identity string as a data channel. Data belongs in the manifest; the identity string is a pointer and a commitment.

7.2.3 PARSING RULES

Parsers MUST:

1. verify the byte string is valid UTF-8 containing only bytes permitted by the grammar;
2. split on `|` (0x7C) into exactly 3 or 5 fields — any other field count is invalid;
3. validate each field against its production above;
4. reject, rather than repair, any violation (no trimming, case-folding, or zero-stripping).

A string that fails ZMD-1 parsing is not thereby an invalid *asset* — the protocol imposes no format on `asset_desc` — it is simply outside ZMD-1, and clients **MUST** treat such assets as **non-conformant**: displayable only under generic fallback naming (§8.4), never under any name derived from the raw string.

7.2.4 WORKED EXAMPLE

A creator operating the collection `pallas-wanderers` issues item 42 in full form. The manifest of §7.3.4 canonicalizes (per §7.3.3) to a byte string whose BLAKE2b-256 digest is:

```
7c1f0d0a9e5b34d2c6a1f8e09b47d3125fa6c08842e9d17b30c5a4f6d28e91bb
```

and whose CIDv1 (base32, raw codec) is:

```
bafkreid4d4gqtzs3gtjmnip6qjwupuysl6tmbcefvhixmyyljjhw2i5eo5m
```

The `asset_desc` is therefore the 145-byte string:

```
zmd1|pallas-  
wanderers|42|bafkreid4d4gqtzs3gtjmnip6qjwupuysl6tmbcefvhixmyyljjhw2i5eo5m|7c1f0d0a9e5b34d2c6a1f8e0  
9b47d3125fa6c08842e9d17b30c5a4f6d28e91bb
```

At issuance, the chain records only `assetDescHash = BLAKE2b-256(asset_desc)`; the string above travels off-chain (registry, dumps, IPFS, or hand to hand), and any holder of it can re-derive the hash and confirm the match without consulting anyone. (All digests and CIDs in this example are illustrative values with the correct format, not outputs of a real computation.)

7.3 The Item Manifest

7.3.1 OVERVIEW

The item manifest is a JSON document that carries everything human-facing about an item. It is identified by its canonical-form hash (`content-hash` in the full-form `asset_desc`) and authenticated by a creator signature made with the collection's issuance key — the same key whose signature the protocol requires on the issuance bundle itself. This choice is deliberate: it means manifest authorship is provable by exactly the party the chain already recognizes as the issuer, with no additional key infrastructure.

7.3.2 SCHEMA

The normative schema, in JSON Schema (draft 2020-12):

```

{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://zecbit.io/schemas/zmd1/item-manifest.json",
  "title": "ZMD-1 Item Manifest",
  "type": "object",
  "required": ["schema_version", "collection", "item", "media", "creator"],
  "properties": {
    "schema_version": {
      "const": "zmd1"
    },
    "collection": {
      "type": "object",
      "required": ["slug", "name", "issuer_vk", "network"],
      "properties": {
        "slug": {
          "type": "string",
          "pattern": "^[a-z0-9]([a-z0-9]|-(?!-)){0,62}[a-z0-9]$|^^[a-z0-9]$" },
        "name": {
          "type": "string", "minLength": 1, "maxLength": 128 },
        "issuer_vk": {
          "type": "string", "pattern": "^[0-9a-f]{64}$",
          "description": "Issuance validating key, lowercase hex" },
        "network": {
          "enum": ["main", "test"] }
      },
      "additionalProperties": true
    },
    "item": {
      "type": "object",
      "required": ["index", "name"],
      "properties": {
        "index": {
          "type": "integer", "minimum": 0, "maximum": 4294967295 },
        "name": {
          "type": "string", "minLength": 1, "maxLength": 128 },
        "description": {
          "type": "string", "maxLength": 4096 }
      },
      "additionalProperties": true
    },
    "media": {
      "type": "array",
      "minItems": 1,
      "items": {
        "type": "object",
        "required": ["type", "cid", "hash", "mime"],
        "properties": {
          "type": {
            "enum": ["image", "animation", "audio", "video",
              "model", "document"] },

```

```

"hash": { "type": "string", "pattern": "^[0-9a-f]{64}$",
          "description": "BLAKE2b-256 of the exact media bytes" },
  "mime": { "type": "string", "maxLength": 128 },
  "bytes": { "type": "integer", "minimum": 1 },
  "dimensions": {
    "type": "object",
    "properties": {
      "width": { "type": "integer", "minimum": 1 },
      "height": { "type": "integer", "minimum": 1 }
    }
  },
  "role": { "enum": ["primary", "thumbnail", "full", "extra"] },
  "additionalProperties": true
},
"traits": {
  "type": "array",
  "items": {
    "type": "object",
    "required": ["trait_type", "value"],
    "properties": {
      "trait_type": { "type": "string", "minLength": 1, "maxLength": 64 },
      "value": { "type": ["string", "number", "boolean"] },
      "display": { "enum": ["string", "number", "percentage",
                           "date", "boost", "hidden"] }
    }
  },
  "additionalProperties": true
},
"creator": {
  "type": "object",
  "required": ["sig"],
  "properties": {
    "name": { "type": "string", "maxLength": 128 },
    "sig": { "type": "string", "pattern": "^[0-9a-f]+$",
            "description": "IssueAuthSig by the issuance key over the signing digest (see
canonicalization rules)" }
  },
  "additionalProperties": true
},
"created_at": { "type": "string", "format": "date-time",
                "description": "UTC, RFC 3339; informational only" }
},

```

```
"additionalProperties": true
}
```

Notes on individual fields:

- **collection.issuer_vk** MUST equal the issuance validating key that appears in the on-chain issuance bundle for this asset. This is what welds the manifest to a specific issuer even when the manifest circulates detached from any chain context.
- **collection.network** guards against a testnet manifest being replayed against a mainnet asset of identical construction. Verifiers MUST check it against the chain they are validating on.
- **media[].hash** binds bytes regardless of transport (requirement R4). **media[].cid** is a retrieval hint. Verifiers MUST check **hash**; checking **cid** is OPTIONAL.
- **traits** follows the field naming that the broader NFT ecosystem has standardized around (**trait_type** / **value**), easing tooling reuse, but the semantics here are advisory display data with no protocol meaning.
- **created_at** is informational and unverifiable; clients MUST NOT present it as a proven timestamp. The proven timestamp of an item is its issuance block (Chapter 9).
- **additionalProperties: true** throughout is deliberate; see §7.7.

7.3.3 CANONICALIZATION

The **content-hash** must be reproducible by independent implementations, so the manifest's byte serialization is fixed. ZMD-1 adopts the JSON Canonicalization Scheme (JCS, RFC 8785):

1. Object members are sorted by lexicographic order of their names' UTF-16 code units.
2. No insignificant whitespace is emitted.
3. Strings are UTF-8 encoded with the minimal escaping JCS prescribes.
4. Numbers are serialized in JCS canonical number form. To avoid the corner cases of floating-point canonicalization entirely, ZMD-1 additionally requires that every number in a conformant manifest MUST be an integer representable in 53 bits; fractional trait values MUST be expressed as strings.
5. The result is a single line of UTF-8 with no trailing newline.

Define, for a manifest **M**:

```
canonical(M)      = JCS serialization of M
signing_input(M)  = JCS serialization of M with the member
                    creator.sig REMOVED (the creator object remains,
                    minus its sig member)
signing_digest(M) = BLAKE2b-256( "ZMD1-ItemSig" || signing_input(M) )
```

Then:

- `creator.sig` MUST be a valid signature by the collection's issuance key over `signing_digest(M)`, using the issuance authorization signature scheme defined by ZIP 227 (IssueAuthSig). The domain-separation prefix `ZMD1-ItemSig` (ASCII, no delimiter) ensures a manifest signature can never be confused with an issuance-bundle signature made by the same key.
- `content_hash` (the `asset_desc` field) MUST equal `BLAKE2b-256(canonical(M))` — i.e., the hash of the *complete* manifest, signature included.

This two-step construction resolves the obvious circularity (the signature cannot cover itself) while ensuring the hash covers everything, including the signature.

7.3.4 ANNOTATED EXAMPLE

The manifest committed by the worked example of §7.2.4 (shown pretty-printed for readability; the canonical form is the same document in JCS serialization):

```

{
  "schema_version": "zmd1",
  "collection": {
    "slug": "pallas-wanderers",
    "name": "Pallas Wanderers",
    "issuer_vk": "9f3e21c88a07b6d45f10ea92c3db7541806cfd2ab9e3541708c6f2d1e05a4b73",
    "network": "test"
  },
  "item": {
    "index": 42,
    "name": "Wanderer #42 — The Cartographer",
    "description": "Forty-second of the Pallas Wanderers. Hand-finished linework over a generative
base composition."
  },
  "media": [
    {
      "type": "image",
      "role": "primary",
      "cid": "bafkreib6mf3yqbcqz3xh2pkgwl4onv5irxg2tqok5s2m3hfnczvyd7q2ru",
      "hash": "a95c31e8f2d07b6a4c19d5e8203f7b1c64da0e92b8f13c57a6d40e19c2385f70",
      "mime": "image/png",
      "bytes": 1730442,
      "dimensions": { "width": 2000, "height": 2000 }
    },
    {
      "type": "image",
      "role": "thumbnail",
      "cid": "bafkreicu2rq5kz3m4x6ejwqhtplf7n2bqvxiw3y5dcen4mahoxbvms6oaq",
      "hash": "13b7f6a02c9de845b1a0347c6f9e2d18e5c4a97022bd68f3915ac0d47e8b26c1",
      "mime": "image/webp",
      "bytes": 48213,
      "dimensions": { "width": 512, "height": 512 }
    }
  ],
  "traits": [
    { "trait_type": "Archetype", "value": "Cartographer" },
    { "trait_type": "Palette", "value": "Verdigris" },
    { "trait_type": "Edition", "value": 42, "display": "number" }
  ],
  "creator": {
    "name": "Studio Meridian",
    "sig":

```

```
75dce8b09341fa6c25d87e13b04a9d6"  
  },  
  "created_at": "2026-08-20T14:03:11Z"  
}
```

Annotations: `issuer_vk` must match the on-chain issuance bundle; `item.index` must match the `index` field of the `asset_desc`; `collection.slug` must match the `collection` field; the two `media[].hash` values are what verifiers check against downloaded bytes; `creator.sig` is verifiable by anyone holding the manifest, because the verifying key is *in* the manifest and cross-checked against the chain.

7.4 The Collection Manifest

Item manifests describe items; the **collection manifest** describes the collection as a whole. It is published under the issuer key (i.e., discoverable via the issuer key in the registry, §8.2) and signed with the issuance key using the same canonicalization discipline as §7.3.3, with domain-separation prefix `ZMD1-CollSig`.

Required and notable members:

```

{
  "schema_version": "zmd1",
  "type": "collection",
  "collection": {
    "slug": "pallas-wanderers",
    "name": "Pallas Wanderers",
    "issuer_vk": "9f3e21c88a07b6d45f10ea92c3db7541806cfd2ab9e3541708c6f2d1e05a4b73",
    "network": "test",
    "description": "A 256-piece study of shielded provenance."
  },
  "media": [
    { "type": "image", "role": "banner",
      "cid": "bafkrei...", "hash": "44e1...", "mime": "image/png" },
    { "type": "image", "role": "avatar",
      "cid": "bafkrei...", "hash": "9a02...", "mime": "image/png" }
  ],
  "links": {
    "website": "https://example.org",
    "contact": "mailto:studio@example.org"
  },
  "royalty": {
    "advisory": true,
    "bps": 500,
    "note": "Requested resale royalty. Advisory only; see below."
  },
  "item_policy": {
    "planned_count": 256,
    "index_base": 1,
    "finalization": "finalize-each"
  },
  "creator": { "name": "Studio Meridian", "sig": "..." },
  "created_at": "2026-08-18T09:00:00Z"
}

```

Normative points:

- **Binding.** A collection manifest is bound to a collection solely by `(issuer_vk, slug)` plus the signature. Unlike an item manifest in full form, it is not hash-committed on chain, and is therefore *replaceable by the creator*: a newer validly signed collection manifest for the same `(issuer_vk, slug)` supersedes an older one. Registries **MUST** retain superseded versions in their append-only log (§8.2).
- **Royalty is advisory.** The `royalty` object declares the creator's requested resale royalty in basis points. Because ZSA transfers are fully shielded, no on-chain mechanism can observe —

let alone enforce — secondary sales. The `advisory: true` member is REQUIRED and MUST be `true` in ZMD-1; clients MUST NOT present the royalty as protocol-enforced. Whether and how the ZecBit trading layer honors advisory royalties is specified in Chapters 13–15.

- `item_policy` declares the creator’s intent: the planned item count, the index base (0 or 1), and the finalization policy — `finalize-each` (each asset’s issuance is finalized immediately, making per-item supply provably fixed), `finalize-at-close` (finalization deferred until the collection completes), or `open` (no finalization commitment). This is a *declaration*; the chain’s finalize flags (Chapter 9) are the ground truth, and indexers MUST surface discrepancies between declared policy and observed flags.
- `links` values MUST be well-formed URLs; clients SHOULD treat them as untrusted external content.

7.5 The Verification Algorithm

Given: an `AssetBase` and its on-chain issuance context (issuer key `ik`, `assetDescHash`, `network`), a candidate `asset_desc`, a candidate manifest `M`, and (optionally) media byte strings. The following algorithm decides ZMD-1 validity. It requires no network access beyond having previously obtained the inputs (requirement R2).

```

function VERIFY_ITEM(chain_ctx, asset_desc, M, media_bytes[]):

    // Step 1 – identity binding
    assert utf8_valid(asset_desc) and len(asset_desc) <= 256
    assert BLAKE2b256(asset_desc) == chain_ctx.assetDescHash
    // Optionally, re-derive AssetBase from (chain_ctx.ik, assetDescHash)
    // per ZIP 226/227 and assert it equals the observed AssetBase.

    // Step 2 – grammar
    fields = PARSE_ZMD1(asset_desc)           // §7.2.3; abort on failure
    form   = fields.form                      // MINIMAL or FULL

    // Step 3 – manifest hash binding (full form only)
    canon = JCS(M)
    if form == FULL:
        assert BLAKE2b256(canon) == fields.content_hash
        // OPTIONAL: assert CID(canon, raw, base32) == fields.manifest_cid

    // Step 4 – cross-field consistency
    assert M.schema_version == "zmd1"
    assert M.collection.slug == fields.collection
    assert M.item.index == fields.index
    assert hex(M.collection.issuer_vk) == chain_ctx.ik
    assert M.collection.network == chain_ctx.network

    // Step 5 – creator signature
    digest = BLAKE2b256("ZMD1-ItemSig" || JCS(M without creator.sig))
    assert IssueAuthSig.verify(chain_ctx.ik, digest, M.creator.sig)

    // Step 6 – media integrity (for whichever media were fetched)
    for (m, bytes) in zip(M.media, media_bytes):
        if bytes != null:
            assert BLAKE2b256(bytes) == m.hash

    return VERIFIED(form)

```

Outcome semantics: `VERIFIED(FULL)` means the chain itself commits to every byte of the metadata and media; `VERIFIED(MINIMAL)` means the chain commits to the identity (`slug`, `index`) and the *creator* vouches for the manifest via signature. Clients MUST distinguish these outcomes in their UI treatment. Any assertion failure yields `INVALID`, and clients MUST NOT partially trust an invalid bundle (e.g., by displaying its name while discarding its media).

Verifiers SHOULD perform Step 1 before Step 3 so that an attacker cannot cause expensive manifest processing for a preimage that does not even match the chain.

7.6 Media Storage and Retrieval

ZMD-1 is transport-agnostic by construction: because every media object is bound by `hash` (R4), *where* bytes come from has no bearing on *whether* they are authentic.

- **IPFS-first.** Creators SHOULD publish media (and canonical manifests) to IPFS and SHOULD arrange pinning through at least two independent operators. The `cid` field makes content retrievable by any IPFS node.
- **HTTP gateways as accelerators.** Clients MAY fetch via HTTP gateways (public or ZecBit-operated) for latency; a gateway is a cache, not an authority. Clients MUST verify `hash` on fetched bytes regardless of transport, and MUST NOT special-case “trusted” hosts to skip verification.
- **Registry dumps as fallback.** Manifest bytes (though not necessarily large media) also travel in registry full dumps (§8.2), so the metadata layer survives even a total IPFS outage.

Size guidance (SHOULD-level, enforced as submission policy by the ZecBit registry, §8.2, not by the standard itself): thumbnail $\leq$ 256 KiB; primary image $\leq$ 20 MiB; animation/video $\leq$ 100 MiB; canonical manifest $\leq$ 64 KiB. Creators SHOULD always include a small `thumbnail` role so that clients can render galleries without pulling full-resolution assets. The `bytes` member lets clients budget downloads before fetching.

7.7 Extensibility

- **Unknown-field tolerance.** Every object in a ZMD-1 manifest admits additional members. Verifiers MUST NOT reject a manifest for containing unknown members, and MUST include unknown members in canonicalization and hashing (they are part of the committed document even if not understood). Clients MUST ignore unknown members for display purposes. This is the standard robustness posture: strict in what the grammar fixes, tolerant in what it leaves open.
- **Reserved namespaces.** Member names beginning with `zmd_` are reserved for future revisions of this standard and MUST NOT be used for private extensions. Private extensions SHOULD be namespaced under a single vendor member (e.g., `"x-studiometidian": { ... }`).
- **Versioning.** The leading `zmd1` tag in `asset_desc` and the `schema_version` member version the standard as a whole. A future incompatible revision will use the tag `zmd2` with its own grammar; the tag position (first field, pipe-delimited) is guaranteed stable across versions, so a parser can always dispatch on the prefix up to the first `|`. Backward-compatible clarifications may be published as errata to ZMD-1 without a tag change, but MUST NOT alter the canonicalization or hashing rules — those are frozen for the lifetime of the `zmd1` tag, because deployed hashes depend on them.
- **What ZMD-1 deliberately excludes.** Mutable state (the chain and Chapter 9’s indexer own supply and finalization), pricing and order data (Chapters 13–15), holder attestations and disclosure proofs (Chapter 12), and any notion of on-chain enforcement of royalties.

Exclusion is a feature: everything inside the manifest is exactly the set of facts that can be frozen at issuance and verified offline forever.

8. The Asset Registry

8.1 Role

ZIP 227 leaves a gap between the chain and the user: the chain stores `assetDescHash`; users need names, images, and provenance; something must map one to the other. The ZIP anticipates a “common trusted registry” without specifying one. The ZecBit Asset Registry occupies that role while minimizing the trust the phrase implies. Its function is **preimage distribution**: given an `assetDescHash` observed on chain, return the `asset_desc` preimage and associated ZMD-1 manifests, together with any registry-layer annotations (verification state, moderation flags).

The essential property, inherited from Chapter 7, is that the registry distributes *self-verifying* artifacts. Every record it serves can be checked against the chain by the recipient (§7.5). The registry can therefore be fast, centralized, and convenient in operation while remaining unable to lie about content — its honest-behavior surface is reduced to *availability* and *labeling*, analyzed in §8.5.

8.2 Architecture

8.2.1 THE APPEND-ONLY LOG

The registry’s core data structure is an append-only log of records:

```
{
  "seq": 184201,
  "recorded_at": "2026-08-20T14:07:55Z",
  "kind": "item",
  "asset_desc_hash": "e3b1c9f4a8d02765e1bc03fa9d5c47218e60b3d4f7a2c185d9e04b6f13a7c520",
  "asset_desc": "zmd1|pallas-wanderers|42|bafkreid4d4...|7c1f0d0a...",
  "manifest": { "...": "ZMD-1 item manifest, verbatim" },
  "submitter_sig": "...",
  "registry_annotations": {
    "verification": "registered",
    "flags": []
  }
}
```

Records are never modified or deleted; corrections and supersessions (e.g., a creator replacing a collection manifest, §7.4; a moderation flag being added or lifted) are expressed as *new* records referencing the same `asset_desc_hash` or collection. The current state served by the read API is a deterministic fold over the log. This design makes the entire registry reproducible from its log alone, which is what makes mirroring meaningful.

8.2.2 APIS

The registry exposes:

- a **public read API** (REST; no authentication, no API keys for read paths): lookup by `assetDescHash`, by `AssetBase`, by issuer key, and by collection slug; plus log tailing (`/v1/log?since_seq=`) so mirrors can follow in near-real time. Read responses always include the raw record so the caller can re-verify.
- a **submission API** for creators. Authentication is a signature by the issuance key over the submitted record (domain-prefix `ZMD1-RegSubmit`), which makes registration *permissionless for one's own assets*: whoever controls the issuance key of a collection — and only that party — can register preimages and manifests for it, with no account, approval, or relationship with ZecBit required. The registry validates submissions by running the full §7.5 algorithm (minus media fetch, which runs asynchronously) before appending; invalid submissions are rejected, never stored.

8.2.3 FULL DUMPS AND MIRRORS

The registry publishes reproducible snapshots of its complete log:

- a **git repository** containing the log as line-delimited JSON, partitioned by sequence range, so that history, diffs, and provenance of registry state are auditable with commodity tools;
- **torrent snapshots** of the same content at regular intervals, for bulk transfer;
- a signed **snapshot manifest** per release, listing every partition file with its BLAKE2b-256 hash and the covered sequence range.

Anyone MAY run a mirror by replaying the dump and following the log endpoint. A mirror is a full peer for read purposes: because records self-verify against the chain, a client resolving metadata through a mirror loses nothing versus the origin. Decentralization here is by design rather than by protocol — there is no consensus among mirrors, and none is needed, because there is nothing to reach consensus *about*: content is hash-bound, and annotations (§8.3) are explicitly per-operator opinions.

8.2.4 CONFLICTS AND THE UNRESOLVED STATE

Conflict handling in a hash-keyed store is unusually simple. A record is keyed by `asset_desc_hash`, and its `asset_desc` must hash to that key; under the collision resistance of BLAKE2b-256, two *different* valid preimages for one key cannot exist. “First-valid-wins” is therefore not an arbitration rule but a triviality — any later valid submission for the same key is byte-identical in its bound content and is recorded as a duplicate confirmation.

The interesting failure mode is *absence*, not conflict. An asset can be issued on chain without anyone ever registering its preimage — the chain does not require it. The registry (and Chapter 9’s indexer) represents such assets explicitly as **unresolved**: `assetDescHash` known, preimage unknown. Unresolved assets are first-class objects: they appear in the catalog with generic identification (issuer fingerprint, hash prefix, issuance height), can be traded (the trading layer needs only `AssetBase`), and transition to resolved the moment a valid preimage is submitted by anyone — resolution requires no permission, since validity is checkable. Clients MUST render unresolved assets as such and MUST NOT infer anything (positive or negative) from unresolvedness beyond “no preimage has been published here yet.”

8.3 Verification Levels and Copymint Detection

8.3.1 VERIFICATION LEVELS

The registry annotates collections with a verification level. These are **operational, registry-layer judgments** — they live in annotations, not in the chain and not in ZMD-1 documents — and every mirror MAY apply its own policy and publish its own levels. The ZecBit-operated registry uses four:

Level	Meaning	Criteria (operational)
unverified	Default	Asset observed on chain; preimage may or may not be registered. No judgment expressed.
registered	Self-registered	Valid preimage and manifests submitted with a correct issuance-key signature (§8.2.2). Proves <i>key possession</i> , i.e., that the registrant is the on-chain issuer — nothing about real-world identity.
verified	Verified creator	registered , plus an off-chain identity attestation reviewed by the registry operator (e.g., control of a long-standing public identity that publicly claims the collection). Subjective by nature; the attestation evidence is summarized in the annotation.
flagged	Adverse finding	Abuse review concluded the collection is deceptive: copymint, impersonation, or malicious media. The finding and its basis are recorded in the log (append-only, so a lifted flag remains historically visible).

Clients **MUST** display verification state alongside any collection name (this is half of the ZIP 227 display rule; §8.4 is the other half) and **SHOULD** visually subordinate **unverified** / **flagged** collections in discovery surfaces.

8.3.2 COPYMINT DETECTION

Copyminting — re-issuing someone else’s media under a new issuer key — cannot be prevented at the protocol layer: issuance is permissionless and the chain sees only hashes. The registry addresses it at registration time:

- 1. Ingest.** For each newly registered item, media bytes are fetched (per the manifest CIDs) and hash-verified.
- 2. Perceptual fingerprinting.** Each image is reduced to perceptual hashes (difference-hash and frequency-domain hash variants) that are stable under re-encoding, resizing, and mild recompression.
- 3. Corpus comparison.** Fingerprints are compared against the corpus of all previously registered media. Matches within a conservative distance threshold, where the issuer keys differ, raise a candidate copymint event.
- 4. Human review.** Candidates are queued for review; automated action is limited to *withholding the **registered** badge*, never to blocking log inclusion — the log records what was submitted regardless, because suppressing records would violate append-only auditability.

5. **Outcome.** Review can clear the candidate (legitimate reuse: same creator on a new key, licensed derivatives, commonly reused public-domain elements) or flag the later collection.

This pipeline is stated with its honest limitations. Perceptual hashing is evadable by determined adversaries (heavy crops, style transfer, adversarial perturbation) and produces false positives on legitimately similar art (generative collections sharing a base composition, homage works, minimal-style pieces). Therefore: detection is best-effort and **MUST NOT** be represented to users as a guarantee; earlier registration is evidence of priority *in the registry*, not proof of authorship; and a documented **appeal path** exists — a flagged party may submit counter-evidence (e.g., demonstrating control of the original creator identity, or earlier publication elsewhere), reviewed by a person distinct from the original reviewer, with the appeal and outcome appended to the log. Mirrors inherit none of these judgments automatically; they receive the annotations in the log and **MAY** re-derive, override, or ignore them.

8.4 The Petname Layer

The registry exists so that wallets and clients can display names *safely*. ZMD-1 plus the registry yields the following client-side display rules, which together implement the petname discipline ZIP 227 calls for:

1. **Never the raw string.** Clients **MUST NOT** display raw `asset_desc` (or any substring of it) as an asset's name. This holds even for ZMD-1-conformant strings: the slug inside `asset_desc` is an identifier, not a name.
2. **Name + state, atomically.** A registry-derived display name **MUST** always be rendered together with its verification level. A name without state is indistinguishable from an endorsement.
3. **Collision-safe canonical form.** The canonical short identification of an item is:

```
<collection-slug>#<index> · <issuer-fingerprint>  
e.g.    pallas-wanderers#42 · zi:9f3e21c8
```

where the issuer fingerprint is the first 8 hex characters of `BLAKE2b-256(issuer_ik)` with a `zi:` prefix. Because slugs are not unique across issuers (anyone can issue under any slug), the fingerprint is what disambiguates; clients **MUST** include it whenever two known collections share a slug, and **SHOULD** include it always in contexts where assets from multiple collections appear together.

4. **Human names are decoration.** `item.name` and `collection.name` from manifests **MAY** be displayed, but only alongside the canonical form or with the verification state, never as the sole identifier in a confirmation or signing context. In transaction-confirmation UIs specifically (wallet layer, Chapter 11), the canonical form **MUST** appear.
5. **Local petnames.** Clients **MAY** let users assign local petnames to collections they interact with; local petnames are stored client-side, never transmitted, and rendered in a visually distinct style so a user's own label cannot be spoofed by a manifest.

6. Unresolved and non-conformant fallback. Assets that are unresolved (§8.2.4) or whose `asset_desc` is not ZMD-1-conformant are displayed by issuer fingerprint and `assetDescHash` prefix only, with explicit “unresolved metadata” state.

8.5 Trust Analysis

What can a malicious or compromised registry actually do?

It cannot forge. Serving a false preimage requires a second BLAKE2b-256 preimage for an on-chain hash; serving a tampered manifest breaks `content-hash` (full form) or the creator signature (both forms); serving substituted media breaks `media[].hash`. Any client that runs §7.5 — and conformant clients **MUST**, at minimum on first resolution and before any signing context — detects all of these. Registry compromise therefore cannot convert into metadata forgery against verifying clients.

It can withhold. The registry can refuse to serve or accept particular records (censorship by omission), or go offline entirely. Mitigations: the full-dump mechanism means the historical log is widely replicated and any mirror restores availability; permissionless submission means a censored creator can publish the identical self-verifying bundle anywhere else (their own site, IPFS, another registry), and any client or indexer can validate it without ZecBit’s involvement. Withholding degrades convenience, not truth.

It can mislabel. Verification levels and copymint flags are operator judgments and are the one place the registry’s opinion carries weight. A malicious operator could verify an impersonator or flag a legitimate creator. Mitigations are structural rather than cryptographic: judgments are recorded append-only with stated bases (auditable after the fact), mirrors apply independent policies (so a captured origin does not capture the ecosystem’s labeling), and the layer that *is* cryptographic — key-possession proof underlying `registered` — is machine-checkable by anyone from the log. Clients **SHOULD** treat `verified` as the opinion of a named operator, and the UI vocabulary reflects this (“verified by”).

Deterministic re-derivation as the backstop. Because registry state is a pure fold over an append-only, hash-verifiable log, and because every content-bearing record independently verifies against the chain, the registry’s entire truthful content can be reconstructed and audited by a third party from a dump plus a chain node. The trust that remains is confined to availability and labeling — precisely the residue that mirrors and append-only auditability are designed to cover.

9. The ZecBit Indexer

9.1 Requirements

The indexer is the component that turns raw chain data into a marketplace-grade public catalog. Its defining constraints:

- **Chain-only, key-free.** It MUST build the complete public asset catalog from raw block data alone. It holds no viewing keys, no spending keys, no user secrets of any kind; everything it knows, anyone can know. This is not a limitation to be engineered around but the design point: the indexer's output is exactly the public information of the ZSA protocol — issuance, burn, finalization — and nothing else.
- **Reproducible.** Two independent operators syncing from the genesis of the ZSA chain MUST arrive at byte-identical catalog state (§9.7). No indexer state may depend on wall-clock time, observation order across peers, or operator input.
- **Standard interfaces only.** It MUST consume the node's standard Zcash JSON-RPC surface (`getblock` , `getrawtransaction` , `getbestblockhash` , ...) and parse raw v6 transactions itself. It MUST NOT depend on nonstandard or in-development RPCs; in particular, while upstream Zebra is developing an asset-state RPC (`getassetstate`), the indexer treats such endpoints as optional cross-checks (§9.7) and never as inputs. This keeps the indexer runnable against any conformant node.
- **Marketplace-grade queries.** It serves the read paths the marketplace UI and third parties need: asset detail, collection listing, issuance/burn event history, catalog-wide statistics — with pagination and predictable latency.

9.2 Architecture

The indexer is a Rust service structured as a pipeline around an embedded relational store.



9.2.1 INGESTION AND PARSING

The ingestion driver maintains a persisted cursor `(height, block_hash)` and advances it block by block: fetch header and transactions via RPC, decode each v6 transaction, and extract from it (a) the issuance bundle, if present — issuer validating key, per-asset `assetDescHash`, finalize flag, and the plaintext issue notes, each carrying a recipient diversified address and a value (issue notes, unlike transfer outputs, are public by protocol design; this includes the value-0 *reference note* that is the first note of every asset); and (b) the burn fields, each a public `(AssetBase, amount)` pair. Every extracted fact becomes a row in an event table before it touches any aggregate — the event tables are the system of record, and all state tables are deterministic folds over them.

9.2.2 SCHEMA


```

-- Chain spine: enables reorg detection and rollback.
CREATE TABLE blocks (
  height          INTEGER PRIMARY KEY,
  block_hash      BLOB NOT NULL UNIQUE,      -- 32 bytes
  prev_hash       BLOB NOT NULL,
  time            INTEGER NOT NULL           -- consensus header time
);

-- Events: append-only within a canonical chain; deleted only on rollback.
CREATE TABLE issuance_events (
  id              INTEGER PRIMARY KEY,
  height          INTEGER NOT NULL REFERENCES blocks(height),
  tx_id           BLOB NOT NULL,
  tx_index        INTEGER NOT NULL,
  issuer_ik       BLOB NOT NULL,             -- issuance validating key
  asset_desc_hash BLOB NOT NULL,             -- 32 bytes
  asset_base      BLOB NOT NULL,             -- 32 bytes
  amount          INTEGER NOT NULL,          -- sum of issue-note values
  note_count      INTEGER NOT NULL,
  is_reference    BOOLEAN NOT NULL,          -- contains the value-0 first note
  finalize        BOOLEAN NOT NULL
);

CREATE TABLE burn_events (
  id              INTEGER PRIMARY KEY,
  height          INTEGER NOT NULL REFERENCES blocks(height),
  tx_id           BLOB NOT NULL,
  asset_base      BLOB NOT NULL,
  amount          INTEGER NOT NULL
);

-- State: pure folds over the event tables.
CREATE TABLE assets (
  asset_base      BLOB PRIMARY KEY,
  asset_desc_hash BLOB NOT NULL,
  issuer_ik       BLOB NOT NULL,
  first_seen_height INTEGER NOT NULL,
  finalized        BOOLEAN NOT NULL,
  supply_issued   INTEGER NOT NULL,
  supply_burned   INTEGER NOT NULL,
  reference_note_seen BOOLEAN NOT NULL
);

```

```

CREATE TABLE collections (
  issuer_ik          BLOB PRIMARY KEY,
  asset_count        INTEGER NOT NULL,
  first_issuance_height INTEGER NOT NULL,
  last_issuance_height INTEGER NOT NULL
);

-- Enrichment: registry-derived, rebuildable independently of chain state.
CREATE TABLE asset_metadata (
  asset_base          BLOB PRIMARY KEY REFERENCES assets(asset_base),
  resolution           TEXT NOT NULL,           -- 'unresolved' | 'resolved' | 'invalid'
  asset_desc           BLOB,                     -- preimage, once resolved
  zmd1_form            TEXT,                     -- 'minimal' | 'full' | 'nonconformant'
  collection_slug      TEXT,
  item_index           INTEGER,
  manifest_json         TEXT,
  verification          TEXT,                   -- registry annotation, §8.3
  last_checked_at      INTEGER NOT NULL
);

```

Circulating supply is not stored; it is the definition `supply_issued - supply_burned`, computed at read time so that it can never drift from the events. Both quantities are exact — issuance and burn are the two fully public flows of the protocol, so on-chain supply is exactly computable by anyone, forever.

9.2.3 REORG HANDLING

Chain reorganizations are handled by the spine table:

```

on new candidate block B at height h:
  if blocks[h-1].block_hash == B.prev_hash:
    append B; parse; commit          // common case
  else:
    // Reorg: walk back to the fork point.
    f = h - 1
    while node.getblockhash(f) != blocks[f].block_hash:
      f = f - 1
    BEGIN TRANSACTION
      DELETE FROM issuance_events WHERE height > f;
      DELETE FROM burn_events     WHERE height > f;
      DELETE FROM blocks          WHERE height > f;
      re-fold assets / collections for every asset_base
        touched by the deleted events;      -- fold is over remaining events
      cursor = (f, blocks[f].block_hash)
    COMMIT
    resume forward sync from f + 1 on the new branch

```

Because state tables are folds over events, rollback is *deletion plus re-fold* — there is no compensating-update logic to get wrong, and a rollback followed by re-sync provably converges to the same state as a fresh sync of the new branch. The cursor is updated in the same transaction as the data it describes, so a crash at any point leaves a resumable, consistent store: on restart, the indexer re-validates that the node still has `cursor.block_hash` at `cursor.height` (detecting reorgs that happened while it was down) and proceeds. The API additionally exposes each response's `as_of` height and marks facts within a small recency window as provisional, since the most recent blocks are inherently subject to reorganization.

9.3 What the Indexer Cannot See

The list is short and absolute: **transfers, holders, and balances**. Once issued, ZSA notes move inside the shielded pool; a transfer transaction reveals neither the asset, the amount, the sender, nor the recipient to any observer without keys — and the indexer, by requirement, holds none. Consequently:

- The indexer cannot say who owns an item, how many holders a collection has, when an item last changed hands, or whether it has *ever* changed hands since issuance.
- The only addresses it ever sees are the recipient diversified addresses inside public issue notes — the initial mint recipients. It records these as issuance facts but **MUST NOT** present them as *current* holders, which they may have ceased to be one block later, invisibly.
- All marketplace “activity” data — listings, sales, trade history, floor — therefore originates exclusively in the trading layer (Chapters 13–15), where participants disclose what a trade requires by taking part in it, and nowhere else.

The API is designed to make this asymmetry *legible* rather than papered over. Every asset and collection response carries an explicit `chain_visibility` object distinguishing `public` facts from `shielded` non-facts (see the example in §9.6), and endpoints for unknowable quantities do not exist — there is no `/holders`, returning empty, to be misread as “zero holders.” A client built naively against this API cannot accidentally invent ownership data, because the shape of the API never offers any.

9.4 Derived Data: Computable versus Not

Quantity	From chain?	Basis
Total issued per asset / collection	Yes, exact	Public issuance bundles
Total burned per asset / collection	Yes, exact	Public (AssetBase, amount) burn fields
Circulating supply	Yes, exact	issued – burned
Finalization status (supply provably fixed)	Yes, exact	Finalize flags
Mint timeline (per-block issuance history)	Yes, exact	Event heights + header times
Collection size, first/last issuance height	Yes, exact	Aggregation over issuance events
Reference-note presence per asset	Yes, exact	First (value-0) note at issuance
Initial mint recipients	Yes (issuance-time only)	Plaintext issue notes; historical fact, not current ownership
Holder count / distribution	No	Transfers shielded
Current owner of an item	No	Transfers shielded
Transfer volume / velocity	No	Transfers shielded
Floor price, sale history	No (from chain)	Prices exist only in the orderbook and settlement layer, Chapters 13–15

The upper half of this table is the honest, exact, and — unusually for an NFT platform — *cryptographically complete* statistical surface: supply figures on ZecBit are not sampled or estimated, they are computed. The lower half is structurally absent from the chain, and the marketplace derives its market data from the trading layer alone.

9.5 Registry Join

Chain state and metadata are joined by an enrichment pipeline that is strictly one-directional: registry data annotates chain data and never influences chain-derived facts.

1. Every asset enters the catalog as `unresolved` the moment its issuance is indexed — the chain never waits for metadata.

2. A **registry follower** tails the registry log (§8.2.2) or replays dumps. For each record whose `asset_desc_hash` matches an indexed asset, the indexer *independently re-runs* the ZMD-1 verification algorithm (§7.5) — it does not trust the registry’s own validation — and on success marks the asset `resolved`, storing preimage, parsed form, slug, index, and manifest. A record that fails verification marks nothing and is logged as a discrepancy.
3. A **recheck loop** revisits `unresolved` assets against the registry at intervals with exponential backoff (metadata frequently arrives days after issuance) and refreshes registry *annotations* (verification levels, flags) for resolved assets, since those are the one mutable input.
4. The enrichment tables can be dropped and rebuilt from registry dumps at any time without touching chain state — the two halves of the store fail, and recover, independently.

9.6 Public API Surface (Indexer-Owned Endpoints)

The complete ZecBit API is specified in Chapter 17; this section sketches only the endpoints the indexer itself owns. All are read-only, unauthenticated, and paginated; all responses carry `as_of` (indexed height and block hash).

```
GET /v1/assets/{asset_base}
GET /v1/collections/{issuer_ik}
GET /v1/collections/{issuer_ik}/items?cursor=&limit=
GET /v1/events?type=issuance|burn&issuer=&asset=&from_height=&cursor=
GET /v1/status
```

Example — asset detail:

```
GET /v1/assets/8b4e2f91c0d7a6531e8f04b2d9c75a1683f0e4d2b7a91c5f06d38e12a4c7b950
```

```
{
  "as_of": { "height": 412884, "block_hash": "0004a9d1..." },
  "asset_base": "8b4e2f91...b950",
  "asset_desc_hash": "e3b1c9f4...c520",
  "issuer_ik": "9f3e21c8...4b73",
  "issuer_fingerprint": "zi:9f3e21c8",
  "first_seen_height": 401220,
  "finalized": true,
  "supply": { "issued": 1, "burned": 0, "circulating": 1 },
  "metadata": {
    "resolution": "resolved",
    "zmd1_form": "full",
    "collection_slug": "pallas-wanderers",
    "item_index": 42,
    "manifest": { "...": "verbatim ZMD-1 manifest" },
    "verification": "verified"
  },
  "chain_visibility": {
    "public": ["issuance", "burn", "supply", "finalization"],
    "shielded": ["transfers", "holders", "balances"],
    "note": "Ownership and transfer data do not exist on chain; trading activity is served by the trading layer."
  }
}
```

Example — collection summary:

```
GET /v1/collections/9f3e21c88a07b6d45f10ea92c3db7541806cfd2ab9e3541708c6f2d1e05a4b73
```

```
{
  "as_of": { "height": 412884, "block_hash": "0004a9d1..." },
  "issuer_ik": "9f3e21c8...4b73",
  "issuer_fingerprint": "zi:9f3e21c8",
  "asset_count": 256,
  "first_issuance_height": 398101,
  "last_issuance_height": 405772,
  "supply": { "issued": 256, "burned": 3, "circulating": 253 },
  "finalization": { "finalized_assets": 256, "open_assets": 0 },
  "metadata": {
    "resolution": "resolved",
    "collection_manifest": { "...": "verbatim, latest superseding version" },
    "verification": "verified"
  },
  "chain_visibility": { "...": "as above" }
}
```

The `/v1/events` stream serves the issuance and burn history that powers mint timelines and audit tooling; `/v1/status` reports sync height, node connectivity, registry-follower lag, and the software version/commit for reproducibility audits (§9.7).

9.7 Operational Notes

- **Multi-node redundancy.** Production deployments run the ingestion driver against multiple independent ZSA nodes; block data is accepted when fetched from any node and cross-checked by hash against a second before the recency window closes. Divergence between nodes halts ingestion and alerts, rather than picking a side silently.
- **Deterministic rebuild guarantee.** Because all state is a fold over chain-derived events, a rebuild from genesis on the same software version MUST reproduce the chain-derived tables byte-for-byte. ZecBit operates this as a standing check: an independent rebuilder periodically resyncs from scratch and compares a Merkle-style digest of the chain-derived tables against production; any mismatch is a released bug, treated with corresponding severity. Where a node exposes an asset-state RPC (such as Zebra's in-development `getassetstate`), it is used as an additional cross-check of computed supplies — never as an input (§9.1).
- **Snapshot and restore.** The store snapshots as (database image, cursor, software version, table digests). Restoring a snapshot and replaying forward is equivalent to continuous operation; the digests let an operator verify a snapshot before trusting it. Snapshots are an availability optimization only — the authoritative recovery path is always resync from the chain.

- **Public runnability.** The indexer is published as source with pinned dependencies and a documented build. Anyone with a conformant ZSA node can run one, reproduce ZecBit’s catalog exactly, and detect any divergence between what ZecBit serves and what the chain says. Like the registry’s dumps (§8.2.3), this converts “trust ZecBit’s data” into “check ZecBit’s data” — the marketplace’s catalog layer is, end to end, an accelerator over public information rather than a custodian of private truth.

The layers above the catalog — wallet integration and key handling (Chapter 11), selective disclosure and privacy mechanics (Chapter 12), and the trading protocol that supplies all activity and price data (Chapters 13–15) — build on the foundation specified here: a metadata standard whose truth is checkable offline, a registry whose trust surface is reduced to availability and labeling, and an indexer whose every fact is recomputable by anyone from the chain alone.

Part IV — Creation, Wallets, and Privacy

10. Mint Studio and the Issuance Pipeline

10.1 Overview: Productizing ZSA Issuance

At the time of writing, the only practical route to issuing a Zcash Shielded Asset is a developer command-line workflow: patching or building the ZSA branches of the core Rust libraries, deriving an issuance key by hand, constructing an issuance bundle programmatically, and broadcasting the result against a node the operator runs themselves. There is no graphical tooling, no hosted service, and no wallet that exposes issuance as a user-facing operation. This is a normal state of affairs for a protocol capability that is still on a dedicated testnet, but it is a complete barrier for the population Mint Studio is designed to serve: creators who can produce a collection's media and metadata but who will never operate a Rust toolchain.

Mint Studio is the productization of that CLI workflow. It does not introduce any new protocol capability; every operation it performs — key derivation, `asset_desc` construction, issuance-bundle assembly, batch signing, broadcast — corresponds to a step that a developer could perform manually against the same libraries. The Studio's contribution is to make the workflow safe by construction: it canonicalizes metadata before hashing, it enforces the privacy-correct minting patterns described in Section 10.5 as defaults rather than as documentation, it makes finalization an explicit and prominent choice rather than a flag, and it handles the operational failure modes (partial batch confirmation, key ceremony mistakes, duplicate submissions) that a manual workflow leaves to the operator.

Two design commitments frame everything in this chapter. First, the Studio must never be the identity of a collection. A collection's identity is its issuance key (Section 10.3); the Studio is a construction tool, and in the Phase B architecture (Chapter 11) the signing operation moves entirely into the creator's own wallet, with the Studio backend reduced to a manifest pipeline and transaction assembler that never touches key material. Second, everything the Studio writes on-chain must remain interpretable without the Studio. The `asset_desc` of every item embeds the ZMD-1 manifest hash, and the manifest itself is content-addressed; a third party with a node and the published manifest can verify a ZecBit-minted collection with no reference to ZecBit infrastructure.

The remainder of this chapter walks the pipeline end to end: the creation flow and its trust boundaries (10.2), issuance key management and its unusual permanence properties (10.3), the mechanics of batching large collections across transactions (10.4), the privacy patterns that issuance-time transparency forces on any correct implementation (10.5), and the finalization and drop policies that determine what "supply" means for a ZecBit collection (10.6).

10.2 Collection Creation Flow

The flow from a creator's media files to a registered on-chain collection has seven stages. Each stage has a defined trust boundary, and the boundaries differ between Phase A (hosted wallets, testnet beta) and Phase B (creator-held keys via ZecBit Connect); the sequence below annotates both.

Stage 1 — Media upload. The creator uploads item media through the Studio browser interface. Media is stored in the content-addressed store described earlier in this paper; the store returns a content hash per file. Nothing at this stage touches the chain or any key material.

Stage 2 — Manifest generation. For each item, the Studio backend assembles a ZMD-1 manifest: item name, collection membership, media content hashes, creator-declared attributes, and the collection-level fields. The manifest is serialized in the ZMD-1 canonical form — a deterministic byte serialization in which field ordering, string encoding, and numeric representation are fixed — and hashed. Canonicalization matters because the manifest hash is about to become part of an immutable on-chain identifier: two byte-different serializations of the same logical manifest would produce two different asset identities. The creator reviews the rendered manifest in the browser; what is reviewed is rendered from the same canonical bytes that will be hashed, not from a parallel representation.

Stage 3 — `asset_desc` construction. Each item's `asset_desc` is constructed per the encoding defined in the metadata chapter: a versioned structure embedding the ZMD-1 manifest hash and the collection linkage fields. Because the ZSA asset identifier is derived from the issuance validating key together with the asset description hash, this step fixes the item's permanent on-chain identity. The Studio computes the resulting `assetDescHash` for every item at this stage and displays it; these hashes are the idempotency keys used throughout the rest of the pipeline (Section 10.4).

Stage 4 — Preview. The Studio renders the collection exactly as it will appear in the marketplace catalog, including the registry entry that will be auto-submitted at Stage 7. The preview also states, in fixed language, the finalization policy chosen (Section 10.6) and the recipient pattern chosen (Section 10.5). This is the last stage at which anything can be edited.

Stage 5 — Batch transaction construction. The backend assembles one or more issuance bundles. Each bundle carries N `IssueAction`s — one per item — under a single issuance authorization signature. Batch minting is standard ZSA behavior, not a ZecBit extension; ZecBit has empirically validated the pattern on the public ZSA testnet with ten finalized items issued in a single transaction, with proof generation completing in roughly three seconds on a consumer CPU. Collections larger than a single transaction's practical limits are chunked as described in Section 10.4.

Stage 6 — Signing and broadcast. In Phase A, the ZecBit wallet service signs with the issuance key it custodies for the creator's account (with the custodial caveats of Section 11.2). In Phase B, the unsigned bundle is passed to the creator's wallet through the ZecBit Connect bridge as a `sign_issuance` request; the wallet displays the human-readable summary mandated by Section 11.3 and signs locally with BIP-340 Schnorr under the creator's own issuance key. The signed transaction is broadcast through ZecBit's node infrastructure (or the creator's own node; the bridge does not require ZecBit's).

Stage 7 — Registry auto-submission. On confirmation, the Studio submits the collection's registry entry — issuer key, collection manifest, item list with `assetDescHash`es and confirming transaction identifiers — to the collection registry. Registry submission is a convenience, not a gate: the on-chain issuance is valid and complete whether or not the registry entry exists.

The full sequence, Phase B form:

Creator Browser (Studio UI)	Studio Backend (manifest/tx svc)	Wallet Layer (ZecBit Connect)	Node (zebrad, ZSA)	Registry
-- upload media ----->				
<- content hashes -----				
-- item fields ----->				
	-- canonicalize ZMD-1			
	-- hash manifests			
<- manifests+hashes ---				
-- approve preview --->				
	-- build asset_desc			
	per item			
	-- assemble issuance			
	bundle(s), N			
	IssueActions each			
	-- sign_issuance ----->			
	(unsigned bundle,			
	summary payload)			
<----- consent prompt (rendered by wallet) --				
-- user approves ----->				
		-- derive m/227'/		
		coin'/0' key		
		-- BIP-340 sign		
	<- signed bundle -----			
	-- broadcast tx ----->			
	<- txid, then confirmation -----			
	-- submit collection entry ----->			
	<- registry ack -----			
<- collection live ----				

The trust boundary to note: in this diagram the Studio backend sees manifests, media, and unsigned transactions, but never key material and never a signature it could not have obtained from the public chain. The wallet layer sees the unsigned bundle and the summary, and is the sole holder of the issuance key.

10.3 Issuance Key Management

The issuance key is derived at the ZIP-32 hardened path `m/227'/coin'/0'` from the creator's seed, and issuance bundles are authorized with BIP-340 Schnorr signatures under the corresponding validating key. Three consequences of the ZSA design make this key categorically more important than a typical signing key, and Mint Studio's key-management guidance follows directly from them.

The issuance key *is* the collection's identity. A ZSA asset identifier is derived from the issuance validating key and the asset description hash. There is no separate "collection contract" or registry-assigned identity; the same `asset_desc` issued under a different key is a different asset. It follows that a creator's issuance key is the permanent, unforgeable root of everything they will ever issue: continuations of a collection, future collections intended to share provenance, and any statement of the form "this item was issued by the same party as that one" all reduce to signatures under this one key. The registry's issuer records are attestations *about* this key, never substitutes for it.

Key loss. If the issuance key (or the seed from which it derives) is lost, no existing item is affected in any way: finalized items are complete, transferable, and verifiable forever without any further participation from the issuer. What is lost is the future — the ability to issue new items under the same issuer identity, to complete an unfinalized edition, or to sign issuer-authenticated registry updates. There is no recovery path at the protocol layer and ZecBit cannot provide one; a creator who loses the key and continues under a new one is, on-chain, a new issuer, and the registry will record the succession only as a human-attested claim, clearly distinguished from cryptographic continuity.

Key compromise. An attacker holding the issuance key **cannot** forge or duplicate any finalized item: finalization is consensus-enforced, and no signature can reopen a finalized asset. What the attacker *can* do is issue *new* assets — new `asset_desc`s — under the compromised issuer identity, producing items that are cryptographically indistinguishable from genuine new work by the creator. The mitigation is social and registry-level, since it cannot be cryptographic: the registry defines a **flagging protocol** under which a creator (authenticating through previously registered out-of-band channels) reports the compromise with a height; the registry marks the issuer record, continues to list pre-flag assets normally, and displays a prominent warning on any asset first issued at or after the flagged height. Marketplace surfaces inherit the flag. This does not prevent the forgery — nothing can — but it bounds its market impact to the window before detection.

Key ceremony guidance. Given the above, Studio enforces a short ceremony at collection creation rather than treating key generation as an implementation detail: (1) the seed is generated wallet-side (Phase B) and displayed exactly once for backup, with the Studio backend structurally unable to observe it; (2) the creator must confirm backup by re-entry before any issuance is enabled; (3) the Studio recommends — and the documentation states plainly — that the issuance branch be treated with cold-storage discipline between drops, since it is needed only at issuance time, unlike the spending account which sees routine use; (4) creators planning long-lived collections are advised to register out-of-band authentication channels with the registry *before* their first issuance, because those channels are the only usable input to the flagging protocol later. In Phase A the ceremony is necessarily weaker — the service custodies the key — which is one of the reasons Phase A is confined to the test network (Section 11.2).

10.4 Batch Construction

A single issuance bundle carries multiple `IssueAction`s under one authorizing signature, so small and medium collections mint in one transaction; ZecBit's testnet validation of ten finalized items per transaction at ~3 s proving time on consumer hardware indicates the per-transaction cost is

dominated by ordinary proving and fixed transaction overhead, not by per-action cost. For larger collections, the binding constraint on actions-per-transaction is practical — transaction size limits and mempool/relay policy — not a consensus rule specific to issuance. The Studio therefore treats the chunk size as a tunable operational parameter, chosen conservatively below observed relay limits, rather than a protocol constant to be maximized.

Chunking a collection across transactions introduces the failure modes any multi-transaction commit has, and the Studio addresses them with three mechanisms.

Deterministic ordering. Items are assigned to chunks by a fixed rule — sort by `assetDescHash`, partition sequentially — computed once at Stage 5 and recorded in the mint job. The plan is a pure function of the item set and the chunk size, so any party (including a creator resuming from a different machine, or an auditor reconstructing events) can recompute which items belong to which transaction.

Resumability. A mint job is a state machine persisted by the Studio backend (and, in Phase B, mirrored in the wallet so the creator can resume without the Studio):



A job interrupted in `PARTIAL` — browser closed, node unreachable, wallet offline — resumes by rescanning the chain for the job's known transaction identifiers and each chunk's `assetDescHash` es, reconciling confirmed state, and continuing from the first unconfirmed chunk. A reorg that drops a previously confirmed chunk returns the job from `PARTIAL` toward `STALLED` for that chunk; the chunk is simply re-broadcast (or rebuilt and re-signed if the original transaction has expired).

Idempotency via `assetDescHash`. The critical safety property is that retries must never double-issue. Two facts make this achievable. First, each item's `assetDescHash` is unique within the job by

construction (Stage 3 rejects duplicates), and the asset identifier it induces is globally unique under the issuer's key. Second, for finalized items, the chain itself enforces at-most-once: a second issuance of a finalized asset is invalid by consensus, so the worst outcome of an over-eager retry of a finalize-at-mint chunk is a rejected transaction, not a duplicate item. For the reissuable-edition exception (Section 10.6), where consensus does not forbid further issuance, the Studio enforces idempotency itself: before broadcasting any chunk it queries the indexer for issuance events matching the chunk's `assetDescHash` es under the issuer's key, and it refuses to broadcast a chunk any of whose hashes has already been issued, requiring explicit creator override with the duplicate items removed. `assetDescHash` thus serves as the end-to-end idempotency key from Stage 3 through Stage 7, including registry submission, which is keyed on the same hashes and is safe to repeat.

10.5 Privacy-Correct Minting Patterns

Issuance is the one transparent moment in a ZSA asset's life. Issue notes are plaintext: the recipient's diversified address and the issued value are public in the issuance transaction, permanently. Every subsequent transfer is fully shielded, but a marketplace that mints *directly to buyers* would write, into the permanent public record, an association between each item and the address that first received it — precisely the linkage the rest of the system is built to prevent. A diversified address is not a real-world identity, but it is a durable correlation handle: if the buyer ever reuses that address, discloses it, or receives to it in any other observable context, the issuance-time record retroactively identifies their acquisition. Mint Studio therefore treats "who is the issue-note recipient" as a privacy-critical design decision and supports exactly two patterns, with direct-to-buyer minting deliberately absent from the product.

Pattern A — issue to self, distribute shielded (the default). All issue notes in the batch are addressed to a *fresh* diversified address of the issuer's own account, generated for this mint job and never used elsewhere. The public record then says only: this issuer issued these items to themselves. Distribution to buyers happens afterward as ordinary shielded transfers, revealing nothing about recipients, prices, or timing beyond the existence of Orchard-ZSA actions in later blocks. This is the correct default for primary sales through the marketplace and for any collection where recipients are not known at mint time. Its residual leakage is examined honestly in Section 12.3: the world learns that N items were issued and can count subsequent shielded action volume, but cannot attribute any transfer to any item or recipient.

Pattern B — fresh diversified address per recipient (allowlist drops). When recipients *are* known at mint time — an allowlist, a free claim, collaborator copies — issuing directly to them saves a round of transfers, and can be done without doxxing anyone by exploiting the Zcash diversified-address model: one account exposes many unlinkable receive addresses. Each allowlisted recipient submits, through an authenticated marketplace session or their wallet, a diversified address *freshly derived for this drop and used for nothing else*. The public issuance record then shows N distinct addresses that are unlinkable to each other, to the recipients' other addresses, and to their marketplace identities. The Studio enforces the freshness discipline mechanically where it can: addresses submitted through ZecBit Connect are derived on request with a drop-scoped diversifier, and the Studio rejects any submitted address it has seen in any prior context

it knows about. What it cannot enforce — a recipient independently reusing the address elsewhere — is stated in the claim UI as the recipient's own residual risk.

Timing considerations. Even with correct address hygiene, timing correlates. A public mint announcement followed minutes later by an issuance transaction and then a burst of shielded transfers lets an observer bind the (public) issuance to the (private) distribution *as an event*, even though individual recipients stay hidden; and a Pattern-B claim flow in which each recipient's issue note lands in a per-recipient transaction at claim time would leak claim ordering. The Studio's mitigations are structural: Pattern-B drops are issued as batches at a scheduled height rather than per-claim; Pattern-A distribution transfers are, at the creator's option, spread across a randomized window rather than emitted in one burst; and the drop tooling never publishes per-item "minted just now" events, only collection-level confirmation. These reduce, but do not eliminate, timing inference; the residue is accounted in Section 12.3.

10.6 Finalization Policy

In ZSA terms, an NFT is an asset issued with value 1 whose `IssueAction` carries the finalize flag: after finalization, consensus permits no further issuance of that asset identifier, ever, by anyone — including the issuer. Finalization is what converts "the issuer promises this is unique" into "the chain enforces that this is unique," and it is the property on which every scarcity claim in this paper rests.

Default: finalize at mint. Mint Studio finalizes every item in the same `IssueAction` that issues it. Uniqueness is therefore a consensus fact from the item's first confirmed block, with no window — not one block — during which the issuer could quietly issue a second copy. The Studio surfaces this as the collection's default and describes any deviation in unambiguous terms at Stage 4 preview and in the registry entry.

The reissuable-edition exception. ZSA permits an asset to remain open: issued now, finalizable later or never. This supports a legitimate product shape — open editions, print-run-style releases where the edition closes at a date rather than a count — but an open asset is categorically different from an NFT, and conflating them is how buyers get defrauded. ZecBit's policy is labeling, not prohibition: an item from an unfinalized asset is displayed everywhere (catalog, item page, order flow, registry) with an explicit `REISSUABLE` state and the plain-language statement that the issuer can create further identical units until finalization; the state transitions to `FINALIZED` only on the confirming finalization action, and the marketplace never describes a reissuable item as "unique," "1 of 1," or an "NFT." The registry records the finalization status as a first-class field so third-party surfaces inherit the distinction.

Drop mechanics: commit-reveal fairness. Scheduled drops with hidden traits face a fairness problem: if manifests are public before mint, participants with better tooling snipe the desirable items; if manifests are simply withheld, the creator can reassign traits after seeing who minted what. The Studio's drop mode resolves this with a standard commit-reveal structure built on machinery this chapter has already defined. At commitment time — before the drop opens — the creator's full set of canonical ZMD-1 manifests is hashed, and those hashes are embedded in the items' `asset_desc`s and thus in the on-chain asset identifiers; the manifests themselves are withheld. Items mint (finalized, per the default) with their content hidden but their content-

hashes immutably committed. At the reveal block, the Studio publishes the manifests to the content-addressed store; anyone can verify that each revealed manifest hashes to the value committed in the corresponding `asset_desc` at mint. The creator therefore cannot alter, reorder, or reassign anything after commitment, and participants cannot inspect traits before reveal. The one honesty caveat, stated in the drop UI: commit-reveal proves the *assignment* was fixed in advance; it does not prove the *distribution* of traits was fair, which is a claim about how the creator generated the set before committing, addressed by registry disclosure of the generation procedure where creators choose to make it.

11. The Wallet Layer and ZecBit Connect

11.1 The Gap

A web marketplace presumes a wallet layer it can talk to. On Ethereum-family chains this layer is so mature it is invisible: injected providers, standardized connect flows, signature requests with structured payloads, hardware-wallet passthrough. In the Zcash ecosystem, as of August 2026, **none of this exists** — not in an immature form, but at all. There is no browser wallet-connect standard, no injected-provider convention, and no precedent for a Zcash dApp requesting a signature from a user's wallet.

The concrete inventory is short. Exactly one third-party wallet supports ZSAs on the dedicated testnet; it is a desktop application, and its integration surface is a server-side GraphQL API mode rather than anything a web page can reach. Mobile wallet SDKs have no ZSA support. The closest prior art for in-browser Zcash is WebZjs — a WASM SDK with an associated MetaMask Snap — which demonstrates that compact-block scanning and proving are feasible in a browser context, but it targets non-ZSA Zcash, and lacks both ZSA support and active funding. The one genuinely encouraging ecosystem asset is PCZT, a partially-created-transaction interchange format for cooperative signing, which establishes the pattern needed for hardware and multi-party signing — but it does not yet support the v6 transaction format that ZSAs require.

The consequences for ZecBit are structural, not cosmetic. A marketplace cannot ship a "connect wallet" button when there is nothing to connect; it cannot delegate signing to wallets that cannot parse the transactions; and it cannot wait, because every chapter of this design depends on users holding, transferring, and eventually trading shielded assets from a browser. ZecBit's answer is an explicitly staged architecture: **Phase A**, a hosted-account wallet service that is honest about being custodial and confined to the test network (11.2); and **Phase B**, ZecBit Connect, a browser SDK and wallet-bridge protocol intended to become the missing layer — for ZecBit first, and, by design, for any Zcash web application (11.3). The invariant that separates the phases: *in the target architecture, the marketplace never holds mainnet user keys*. Phase A is a bootstrapping concession that the target architecture exists to eliminate.

11.2 Phase A — Hosted Accounts (Testnet Beta)

Phase A is a server-side wallet service, written in Rust directly against the ZSA branches of the core Zcash libraries — the same code paths a native ZSA wallet would use, behind an internal API instead of a UI. Its design goals are, in order: correctness against the moving ZSA testnet protocol, honest presentation of its trust model, and a clean export path out of it.

Account derivation. The service holds a single master seed in an HSM-guarded environment; the seed does not leave the boundary, and signing operations execute inside it. Each user account is derived at a per-user hardened ZIP-32 path from this master seed: a spending account for holding and transferring assets and, for creators, an issuance branch at `m/227'/coin'/0'` - equivalent depth under the user's subtree. Derivation is deterministic from the user's account identifier, so the service's database stores no per-user key material at rest — only the mapping from authenticated users to derivation indices.

Session authentication and spending policy. Marketplace sessions authenticate to the wallet service with short-lived tokens bound to the user's login; the wallet service is a separate deployment from the marketplace application, so a compromise of the web tier does not directly reach signing. Every signing request is evaluated against a per-account policy engine before it reaches the HSM: rate limits (operations per unit time), height-window limits (a signed transaction must target the current chain tip within a small tolerance, preventing stockpiling of pre-signed transactions), and operation-type allowances (an account that has never issued cannot suddenly sign issuance bundles without an explicit enablement step). These policies bound the blast radius of a stolen session token to a rate, not a balance.

Custodial labeling. Phase A accounts are custodial and the interface says so without euphemism: a persistent, non-dismissable label on every wallet surface states that ZecBit operates the keys for this testnet account. There is no "semi-custodial" language anywhere in the product.

Export path. Any user can, at any time: (a) reveal their account's seed material (derived subtree exported in a standard encoding) after strong re-authentication, or (b) instruct a sweep — a shielded transfer of all held assets to an external address the user provides, typically in the one third-party ZSA-capable wallet. Export is a first-class, tested flow, because its Phase B analogue (migration off hosted accounts entirely) is the whole point.

Why this is acceptable — and where it stops. On a test network, the assets at stake have no monetary value; the protocol itself is subject to breaking changes that make long-lived self-custodial key material a liability; and the alternative (requiring every beta participant to run patched desktop software) would reduce the beta population to developers, defeating its purpose of exercising marketplace flows with real non-developer users. None of these arguments survives contact with mainnet: valuable assets under a service master seed are a honeypot, a regulatory status change, and a violation of the design goal stated in 11.1. Phase A is therefore scoped to the testnet beta by policy, and the mainnet launch criteria (Chapter 16 treats the security dimension) include the retirement of hosted signing for user funds.

The trust statement, plainly:

Property	Phase A (hosted, testnet)	Phase B (ZecBit Connect)
Who holds spending keys	ZecBit (HSM-guarded master seed)	User's wallet only
Who holds issuance keys	ZecBit, per-user derived	Creator's wallet only
Can ZecBit move user assets	Yes (policy-limited, logged)	No
Can ZecBit see user holdings	Yes (full visibility)	Only what user discloses
Can ZecBit censor operations	Yes	Can refuse service, cannot block signing or self-broadcast
What a full ZecBit compromise yields	All hosted testnet assets	No keys; metadata only (Ch. 12/16)

Recovery if ZecBit disappears Export/sweep while service lives User unaffected (keys and sync independent)

Appropriate network

Testnet only

Mainnet target

11.3 Phase B – ZecBit Connect

ZecBit Connect is two deliverables: a **WASM core** — the Rust ZSA wallet libraries compiled to WebAssembly, giving any browser page (or extension, or webview) full ZSA wallet capability — and a **bridge protocol** that lets an application request operations from a wallet without ever seeing key material. The split matters: the WASM core makes a *ZecBit-shipped* wallet experience possible immediately, while the bridge protocol is the piece intended to outlive ZecBit’s own implementation, defining how *any* Zcash web application talks to *any* wallet.

WASM core. The core compiles the ZSA branches of the core libraries to WASM and provides: compact-block note scanning over the lightwalletd protocol (Section 11.4); Halo 2 proof generation in-browser, which — consistent with the seconds-per-transaction figures observed on consumer CPUs natively — is a seconds-scale operation in WASM, acceptable for an explicit user action like confirming a purchase; and key storage in origin-isolated browser storage, with keys encrypted at rest and, where the platform supports it, wrapped by a passkey so that unlocking the wallet is a WebAuthn ceremony rather than a password, and raw key bytes exist unencrypted only transiently in WASM linear memory during signing. WebZjs is acknowledged prior art for the general approach; ZecBit Connect differs in targeting ZSA (v6 transactions, issuance, multi-asset notes) and in being funded as marketplace infrastructure rather than a grant artifact.

Bridge protocol. The bridge is JSON-RPC 2.0 carried over `postMessage` (same-device: extension or embedded wallet) or WebSocket (cross-device or desktop-wallet pairing). The request vocabulary is deliberately small:

Method	Request semantics	Wallet MUST display before approval
<code>get_addresses</code>	Fresh diversified address(es), optionally purpose-scoped (e.g. drop claim, Pattern B)	Requesting origin; purpose; that a fresh unlinkable address will be shared
<code>get_assets</code>	Disclose selected holdings to the requesting app	Exact asset list to be disclosed; scope and audience
<code>sign_issuance</code>	Sign an issuance bundle (Mint Studio Stage 6)	Item count; per-item asset names and <code>assetDescHash</code> es; finalize flags; recipient pattern (self vs. per-recipient)
<code>sign_transfer</code>	Sign a shielded transfer	Asset (registry petname + hash), amount, destination, fee
<code>sign_order</code>	Sign a trade order (future; Ch. 13–15)	Full order terms; venue; expiry
<code>request_disclosure</code>	Produce a selective-disclosure artifact (Ch. 12.4)	Exact statement to be proven; audience; expiry

Every request carries the requesting origin, a human-readable summary payload, *and* the raw payload; the wallet **MUST** render its summary from the raw payload, never trust the app-supplied summary, and **MUST** resolve asset identifiers to names via the registry's petname mechanism with the underlying hash always available — an unresolvable or registry-flagged asset is displayed as such, prominently. Blind signing (approving a payload the wallet cannot fully render) is a protocol violation, not a degraded mode. These MUST-display rules are normative for anything calling itself a ZecBit Connect wallet, because the consent screen is the entire security model of the bridge.

Third-party wallet path. The same JSON-RPC vocabulary runs over deep links (mobile wallets) and local sockets (desktop wallets), so the existing ZSA desktop wallet — and any future mobile wallet, once mobile SDKs gain ZSA support — can implement the bridge without adopting any ZecBit code. For hardware wallets and cooperative/multi-party signing, the bridge is designed to degrade into an interchange flow: the app produces a PCZT-style partially-created transaction, the bridge shuttles it to each signer, and the completed transaction returns for broadcast. PCZT does not yet support v6/ZSA transactions; ZecBit's position is to track and contribute to that support rather than invent a parallel format, and `sign_*` methods are specified to accept either a complete-and-sign flow or a PCZT round-trip so the upgrade is additive.

11.4 Sync Strategy In-Browser

A shielded wallet cannot ask a server "what do I own" — that question being unanswerable by third parties is the point of the system. Instead the wallet downloads compact blocks (trimmed transaction data over the lightwalletd protocol) and trial-decrypts each output with its incoming viewing key: outputs that decrypt are the user's notes; everything else is noise. This is the fundamental cost of shielded UX, and in a browser it must be engineered around, not wished away.

ZecBit Connect's sync design rests on one structural observation: **the marketplace catalog requires no scanning at all.** Everything the catalog shows — assets, issuers, supply, finalization, registry metadata, issuance events — is public data served instantly by the indexer, because issuance is transparent (Section 10.5) and asset existence is public by design (Section 12.1). Trial decryption is needed *only* to discover the user's own notes: what they hold and what they can spend. The UX consequence is stated honestly in the interface: **the catalog is instant; the wallet balance is not.** A user browses, searches, and inspects collections with zero sync; the "your items" view shows a sync progress state until scanning reaches the tip, which takes seconds to minutes depending on how far behind the wallet is.

The engineering around that residual cost: scanning runs in Web Workers, off the interface thread, with the trial-decryption inner loop in WASM; the indexer supplements lightwalletd with hints that let the wallet prioritize ranges where relevant activity exists (never *which* outputs are the user's — the server cannot know that — only where ZSA action density makes scanning worthwhile). **Checkpointing** bounds work: the wallet persists scan progress and note-commitment-tree state in origin storage continuously, so a returning user scans only the delta since last visit, and a fresh restore-from-seed can start from a trusted recent checkpoint rather than the ZSA activation height when the user attests the seed is newly created. **Background sync** keeps the delta small: while any marketplace tab is open, the worker follows the tip quietly,

so the common case for an active user is a wallet that is already synced when they look at it. A newly restored old seed remains the worst case — a full historical scan, minutes not seconds — and the interface presents it as a restore operation with progress, not a hang.

11.5 Account Model

A ZecBit Connect account is one seed with two jobs, kept deliberately apart.

One seed, two branches. From a single mnemonic seed the wallet derives (a) the **spending account** — Orchard-ZSA spending key, full viewing key, incoming viewing key, and the diversified-address space used for receiving, holding, and trading — and (b) for creators, the **issuance branch** at the hardened `m/227'/coin'/0'` path, holding the BIP-340 issuance key of Section 10.3. The branches never share keys; compromise of a device that has unlocked only the spending account does not expose the issuance identity, and the wallet treats the issuance branch as a separately unlocked, rarely touched compartment (surfaced only inside Mint Studio flows). One backup covers both: the mnemonic restores collector and creator identities together.

Multi-device. Devices share the account by seed import (manual or QR-mediated), after which each device syncs independently — there is no server-side account state to reconcile, because the chain plus the seed *is* the account state. Two devices spending concurrently can race to spend the same note; the wallet treats a detected double-spend rejection as an ordinary retry-with-fresh-notes event. A deliberate intermediate configuration is the **watch-only device**: importing only the full viewing key gives a device that sees all incoming and outgoing activity for the account but cannot spend or issue — suitable for a monitoring dashboard or a lower-trust machine, and a direct application of the Zcash viewing-key model rather than a ZecBit invention.

Recovery. Recovery is the mnemonic, full stop. It restores spending capability, the issuance identity, and — via rescan from checkpoint (11.4) — the complete holdings picture. What it does not restore is anything that never lived in the seed: bridge pairings, petname overrides, and disclosure artifacts (Chapter 12) are device-local conveniences, re-established on use. ZecBit holds nothing recoverable in Phase B by construction; the honest corollary, displayed at seed creation in the same breath as the backup ceremony, is that ZecBit can help with a lost password (passkey re-wrap, if the platform permits) but a lost seed with no backup is unrecoverable by anyone. Optional social/custodial recovery layers may be built *above* this model by third-party wallets implementing the bridge; the protocol neither requires nor precludes them.

11.6 Meeting the Ecosystem Where It Is

The two phases above describe wallet capability ZecBit can build for itself. They do not describe the situation a user is actually in, which is that they already have a Zcash wallet, they already keep their ZEC there, and they have no interest in a second one. A marketplace that can only be used through its own wallet has not solved the wallet problem; it has renamed it.

The integration path with existing wallets is therefore a first-class part of this design, and it is deliberately built out of what those wallets already do rather than out of what we would like

them to add.

Payment intents (available now). ZIP 321 payment request URIs are already parsed and constructed by the mainstream Zcash wallets. A ZecBit checkout is expressible as one: recipient, amount, and the order memo that binds an anonymous shielded payment to an order. The user taps, the wallet opens on its own confirmation screen with the fields filled, and the wallet — not ZecBit — decides what to display and whether to sign. Two conventions are missing at the ecosystem level and are worth standardizing rather than inventing privately: a **return target**, so that a user who completes a payment lands back in the application instead of stranded in the wallet, and a **result signal** (outcome code, and a transaction identifier only with explicit consent) so an application can say *confirmed* immediately instead of polling. ZecBit does not depend on either — our payment watcher detects incoming shielded payments with a viewing key and matches them by memo — but their absence is felt by every user, in every application, on every payment.

Application-built transactions via PCZT. Some operations are not payments and cannot be expressed as a payment URI: an issuance bundle (ZIP 227), and later one leg of a swap (ZIP 228). The ecosystem already has the right shape for these in PCZT, the partially-created-transaction format used in production for hardware signing, where an unsigned transaction is constructed on one device and signed on another. Substituting an application for the hardware wallet changes nothing about the security argument: the signer parses the transaction itself, renders its own summary from the parsed bytes rather than from any text the application supplied, and refuses anything it cannot fully render. The one real gap is that PCZT support at the version 6 transaction format — the format ZSA requires — does not exist yet anywhere; that is upstream work, and it is work ZecBit intends to help carry rather than route around.

Asset display, when ZSA lands. On the day shielded assets activate, every wallet inherits a problem it does not have today: a balance stops being one number, and assets need names and images that the chain does not carry. ZIP 227 puts only a 32-byte hash of the asset description on chain and explicitly warns wallets not to display the raw description as a name, because an issuer can write anything there. Absent a shared standard, each wallet will invent its own resolution scheme and users will be phished through asset names within a week. ZMD-1 (Chapter 7) is written to be that standard: hash-bound, signed by the issuance key, verifiable offline, and specified so that any wallet can implement it against any registry — including none of ours. ZecBit's registry is a convenience, not a dependency, and the standard is offered to the ecosystem rather than retained.

The security constraint on all three. Expanding what a URI or an external application can ask a wallet to do expands an attack surface that the ecosystem has already flagged as sensitive: any application can register a handler, and a malicious one can craft requests designed to mislead. The rules ZecBit holds itself to, and proposes as the shared ones, are the same rules already stated for our own bridge in §11.3: the wallet renders only what it parsed, application-supplied prose is never displayed as fact, return targets are consent-gated per origin rather than automatic, transaction identifiers are disclosures rather than defaults, and every application must remain functional when the user declines. The direction of trust never reverses — the wallet holds the keys, parses, displays, and decides; the application asks.

12. Privacy Model and Selective Disclosure

12.1 The ZecBit Privacy Contract

Privacy claims fail when they are vague. This section states ZecBit's precisely, as three tiers, and everything else in the chapter is derived from them.

Always public — visible to anyone with a node, forever, with no ZecBit involvement:

- The existence of every asset: its identifier, its `asset_desc` (hence its ZMD-1 manifest hash), and its issuer (the issuance validating key).
- Supply facts: every issuance event, the value issued, and the finalization status of every asset.
- Burns: the asset and amount of every burn.
- **Issuance-time recipients, as addresses** (Section 10.5): the diversified address in each plaintext issue note. An address is not an identity — this is the load-bearing distinction — but it is public.
- Aggregate activity shape: the number of shielded actions per transaction and per block (not their contents, assets, amounts, or parties).

Always private, absent deliberate user action:

- Holdings: what any address or person owns. There is no `balanceOf`, no owner enumeration, no "items held by X" query — not as a policy, but as a cryptographic property of shielded notes.
- Transfer history: sender, recipient, asset, and amount of every post-issuance transfer.
- Purchase prices and counterparties, now and under the trading designs of Chapters 13–15 (with the orderbook caveats noted in 12.3).
- The linkage between any two of a user's diversified addresses, and the linkage between addresses and marketplace accounts.

Disclosable by choice — private by default, revealable through the mechanisms of 12.4 with scope chosen by the holder: ownership of a specific item, receipt of a specific note, contents of a specific transaction, or (via viewing keys) broader account visibility.

The visibility matrix, by actor:

Fact	Public observer	ZecBit (Phase B)	Chosen counterparty	Holder
Asset exists; issuer; manifest hash	Yes	Yes	Yes	Yes
Supply, finalization, burns	Yes	Yes	Yes	Yes
Issue-note recipient <i>address</i>	Yes	Yes	Yes	Yes

Identity behind that address	No	No ¹	If holder discloses	Yes
Current holdings of a person	No	No ¹	Only disclosed items	Yes
Transfer parties/amounts/assets	No	No	Only disclosed notes	Yes
Sale price of a past trade	No	No ²	If a party discloses	Parties
Full account activity	No	No	Only via viewing key (12.4b)	Yes

¹ Except metadata ZecBit necessarily observes operationally (12.3). ² Trading-venue visibility is treated in 12.3 and Chapters 13–15. Phase A hosted accounts sit outside this matrix: there ZecBit sees everything, which is why Phase A is testnet-only (11.2).

12.2 Threats Addressed

The tiers above are worth stating only because they block concrete, observed harms. Each threat below is routine on transparent-chain NFT platforms; each maps to the specific mechanism that removes it here.

Threat	On transparent chains	ZecBit mechanism that blocks it
Collector doxxing — binding a person to their holdings, then to their wealth, location, or legal exposure	Trivial: one address label (an ENS name, an exchange withdrawal, a public tweet) unlocks the full holding set	No holding set exists to unlock. Holdings are shielded notes; even a fully deanonymized <i>address</i> reveals only what was issued to it at issuance time, not what is held now
Portfolio surveillance — continuously monitoring what a known party accumulates	Standard tooling (portfolio trackers, whale alerts)	Trial decryption requires the holder's incoming viewing key; without it, an observer cannot even detect that a given party received anything
Sale-price scraping — building per-item and per-collector price histories	Every sale is public; price history is a commodity dataset	Transfer amounts and any associated payment are shielded; no third party can construct a price history for an item or a person
Front-running of known wallets — watching a profitable trader's address and copying or racing their actions	Mempool and address monitoring; endemic	There is no known wallet to watch: transfers reveal no parties, and pending shielded transactions reveal no intent (trading-layer specifics in Ch. 13–15)
Social-graph inference — reconstructing who transacts with whom from the transfer graph	The transfer graph <i>is</i> the data model; clustering heuristics recover exchanges, friend groups, payroll	The transfer graph does not exist. Post-issuance transfers expose no endpoints, so there is no graph to run heuristics on — this is impossibility, not difficulty

The pattern across the rows: transparent-chain harms flow from one root — a public, append-only ledger of *who holds and moves what*. ZecBit does not mitigate that ledger; it never creates it. The only rows with residue are the ones touching issuance and infrastructure, which is the subject of the next section.

12.3 Residual Leakage Analysis

A privacy design earns trust by enumerating what it leaks. The following is the honest list.

Issuance-time addresses. The plaintext issue note is permanent. Pattern B (10.5) reduces it to per-recipient fresh addresses — unlinkable unless a recipient reuses one, which is in the recipient's control but not ZecBit's. Pattern A reduces it further: the public record shows only issuer-to-self. But Pattern A has its own residue: the world sees that the issuer issued N items, and can count shielded actions in subsequent blocks. It learns *that* distribution activity of some volume occurred following the mint — an issuer→? transfer count, roughly — without learning any recipient, item, or price. For most collections this is harmless; for a collection whose *sales volume* is itself sensitive, it is a real, unmitigated signal.

Timing correlation. Action counts per block are public. A mint announcement at a publicized time, followed by a measurable burst of Orchard-ZSA actions, binds the burst to the drop as an aggregate event. The mitigations of 10.5 (batched Pattern-B issuance, randomized distribution windows) flatten the signal; they cannot remove it, and a sufficiently distinctive burst against a quiet chain baseline remains attributable to the drop. What never leaks, even then, is any *individual* correspondence: which action, which item, which buyer.

Orderbook leakage (forward reference). Once trading exists, an order is a statement of intent — asset, price, side — necessarily visible to whatever matches it. Whoever operates matching (ZecBit, in the initial design of Chapters 13–15) sees order flow, and order flow is exactly the price-and-interest data that 12.2 celebrates removing from the public chain. The trading chapters treat this as their central privacy problem; this chapter's contract is scoped accordingly: 12.1's guarantees are guarantees about the *chain*, and the trading layer's additional trust is accounted where it is introduced (Ch. 14).

Registry metadata. Creator profiles are public *by choice*: a creator who registers a name, links social accounts, and signs registry updates has published their identity, and no shielding applies to what they published. The registry's design responsibility, met in its chapter, is that publication is opt-in per field and that pseudonymous creators are first-class.

Network layer. Shielded cryptography does not hide IP addresses. A lightwalletd or RPC operator — including ZecBit's own infrastructure — observes which client fetched which block ranges, when, and from where; correlated with request timing (a proof-generation pause followed by a broadcast), this is meaningful metadata even though note contents stay opaque. ZecBit's posture: infrastructure logs are minimized and retention-limited (Ch. 16 for specifics); ZecBit Connect supports connecting through Tor and third-party lightwalletd endpoints, and the documentation states plainly that users with network-level adversaries in their threat model should use them. Compact-block sync is already the right shape for this — the server learns which ranges were fetched, not which outputs mattered.

Phase A. During the testnet beta, none of this chapter's guarantees hold against ZecBit itself: the hosted wallet sees keys, holdings, and history. This is disclosed at every Phase A surface and is the primary reason Phase A terminates at mainnet (11.2).

12.4 Selective Disclosure — Verifiable Flex

Absolute privacy is not what collectors want; *default* privacy with *chosen* revelation is. The canonical case: prove to one party — or to everyone — that you own a specific item, without exposing anything else you hold. ZecBit calls this the verifiable flex, and supports it through four mechanisms, ordered by availability, each with its honest limitations.

(a) Demonstration transfers (available now; weak). The holder asks the verifier for context (a nonce, a fresh address), then performs an observable action only the item's holder could perform — for example, a shielded self-transfer of the item, followed by a payment-disclosure-style revelation of that transaction's contents to the verifier. This works with nothing but a wallet, but it is weak: it consumes a transaction, its guarantees are bespoke to the ceremony, and naive versions (transfer to a verifier-controlled address and back) temporarily surrender the item. Supported as a fallback; recommended for nothing.

(b) Incoming-viewing-key sharing (available now; NOT recommended — documented to be avoided). Zcash viewing keys make it trivial to give a verifier visibility: hand over an IVK (all incoming notes) or FVK (all incoming and outgoing activity) for the account. The reason ZecBit documents this pattern in order to reject it: **scope**. The statement to be proven is "I hold item X"; an IVK proves it by revealing *every* incoming note the account has ever received or will receive — the entire portfolio, ongoing, with no expiry and no revocation short of migrating to a new account. Viewing keys are the right tool for auditors, accountants, and the holder's own watch-only devices (11.5), where account-wide visibility *is* the requirement. As an ownership proof they are a scope violation, and ZecBit surfaces will never request one for that purpose — a rule stated normatively in 12.5 partly so that users learn to treat any third party requesting an IVK "to verify ownership" as a red flag.

(c) Per-note disclosure proofs (the target design). The Zcash ecosystem's payment-disclosure lineage — out-of-band proofs about specific transactions and notes — provides the right primitive shape: a self-contained artifact proving a fact about *one note* and nothing else. ZecBit's target mechanism is a per-note ownership proof with the following statement, spelled out because precision here is the product:

There exists a note with asset identifier X (value 1), whose commitment is in the global note-commitment tree, such that the prover knows the note's full contents (randomness, address) and holds keys authorized to spend it; and — for freshness — this proof is bound to challenge nonce N and asserts the note was unspent at anchor height H .

The unspent-at- H clause is the honest subtlety: nullifiers are private, so "not yet spent" cannot be shown against the public chain the way it could on a transparent one; freshness is anchored to a recent tree state and a verifier-chosen nonce, so a proof asserts ownership *as of* H , and a verifier wanting currency demands a recent H . Verification flow: the verifier obtains the artifact (from the holder directly, or rendered as a profile badge), checks the proof against the stated anchor via ZecBit's hosted verifier *or* a standalone open-source verifier requiring only a node —

ZecBit must not be a trust dependency for verifying flexes. The artifact reveals the asset, the anchor, and validity; it reveals no address, no other holdings, and no transaction history.

(d) ZK attestations over collections (research direction). The natural generalization: "I hold at least one item of collection C" without revealing *which* — membership proofs over the set of asset identifiers sharing an issuer, and beyond that, threshold statements ("≥ k items", "an item with trait T"). These follow the same skeleton as (c) with an additional set-membership layer, and nothing in the ZSA design forecloses them; but they are unimplemented, their proving costs in-browser are unmeasured, and ZecBit accordingly marks them as a research direction, not a roadmap commitment.

Profile integration. Verified-holdings badges on ZecBit profiles are rendered from mechanism-(c) artifacts and are governed by three rules: they are **user-initiated** (a badge exists because the holder generated a proof and attached it — the marketplace never derives holdings any other way, having no way to); **revocable as display objects** (the holder can remove the badge at any time; removal cannot un-reveal what the artifact proved to those who saw it, and the interface says so); and **never scraped** — ZecBit does not harvest disclosure artifacts from elsewhere, does not infer holdings from Phase-A operational visibility for display anywhere, and treats any holdings information outside an explicit user disclosure as nonexistent for product purposes.

12.5 Disclosure UX Rules (Normative)

Selective disclosure fails not at the cryptography but at the consent screen: users approve scopes they did not understand and cannot retract. The following rules are normative for ZecBit surfaces and for wallets implementing the ZecBit Connect `request_disclosure` method; they extend the MUST-display rules of 11.3.

- 1. Exact scope, before signing.** Every disclosure request MUST display, before approval: the precise statement to be proven (in the terms of 12.4c — asset, anchor height, nonce); the audience (named counterparty, or "anyone holding this artifact — effectively public"); and the duration (the expiry where the primitive supports one, or the words "cannot expire" where it does not). The wallet MUST render this from the raw request payload, never from requester-supplied text.
- 2. Single-purpose artifacts.** One disclosure, one statement, one audience. Requests bundling multiple notes, assets, or audiences into a single approval MUST be split or refused. There is no "select all" over holdings.
- 3. Expiry where the primitive allows.** Mechanism-(c) artifacts carry the challenge nonce and anchor height, giving verifiers a natural staleness bound; requests SHOULD specify the tightest anchor the use case tolerates. Where a mechanism cannot expire (a revealed fact is revealed), the consent screen MUST say exactly that, in those words.
- 4. Escalation is always a fresh ceremony.** No disclosure grants a standing capability. A counterparty wanting a second proof — a fresher anchor, a different item — starts a new request; wallets MUST NOT offer "always allow" for disclosure methods.
- 5. Viewing keys are never ownership proofs.** Per 12.4(b): ZecBit surfaces MUST NOT request IVKs or FVKs to establish ownership, and conformant wallets SHOULD warn when any origin

requests a viewing key, stating what account-wide, ongoing visibility the key confers.

6. **Disclosure history is local.** The wallet keeps a user-reviewable log of every disclosure made — statement, audience, date — stored locally with the account (11.5), so a user can always answer “what have I revealed, and to whom.” ZecBit servers hold artifacts only where the user published them (e.g., a profile badge).

12.6 Position Among Alternatives

Three architectures compete for this design space. Transparent-chain platforms (the incumbent NFT model) put everything public. Fully-dark designs shield everything, including asset existence and metadata — and thereby break the marketplace itself: if no one can enumerate what exists, there is no catalog, no search, no issuer reputation, and no way for a buyer to discover an item they have not been privately told about. This is the **catalog discoverability problem**, and it is why fully-dark asset systems remain curiosities: a market needs a commons of *what exists* even when *who holds it* is private.

ZecBit’s position is the deliberate hybrid the previous sections have assembled: **public catalog, private ownership**. Assets, issuers, supply, and finalization are public — fully indexable, searchable, and verifiable, giving the discoverability of a transparent platform. Holdings, transfers, prices, and counterparties are shielded — giving the ownership privacy of a dark design. Disclosure bridges the two, on the holder’s initiative and at the holder’s chosen scope.

Property	Transparent-chain platforms	ZecBit (public catalog / private ownership)	Fully-dark designs
Catalog: enumerate assets, issuers, supply	Public	Public	Not possible without out-of-band data
Scarcity verifiable by anyone	Yes	Yes (finalization, consensus-enforced)	Only to parties shown private proofs
Issuer identity / provenance	Public	Public (key-rooted; pseudonymous allowed)	Private; reputation cannot accrete
Holdings of a given person	Public	Private	Private
Transfer history, prices, counterparties	Public	Private	Private
Portfolio surveillance / doxxing risk	Endemic (12.2)	Blocked at the ledger	Blocked
Prove ownership to a chosen party	Trivial (too trivial — it is always proven to everyone)	By choice, scoped (12.4)	By choice, but verifier must also trust asset-existence claims
Search, discovery, price of entry for buyers	Excellent	Excellent (catalog is public data, 11.4)	Poor to nonexistent

Residual leakage	Everything	Issuance-time addresses, timing aggregates, venue/order flow, network metadata (12.3)	Network metadata; plus all trust moved into private channels
------------------	------------	---	--

The honest summary of the trade: ZecBit gives up the privacy of *what exists* — which no marketplace can use anyway — to keep the privacy of *who holds and pays what*, which is the privacy people are actually harmed by losing. The chapters that follow carry this contract into the adversarial setting: trading and matching (Chapters 13–15), where order flow creates the new visibility accounted in 12.3, and the security and threat model (Chapter 16), where the guarantees stated here are defended against active attackers rather than passive observers.

12A. Independence: Removing ZecBit from the Trust Path

A marketplace that keeps ownership in its own database has, whatever else it has built, made one promise it cannot keep: that it will still be there tomorrow. If the servers burn, the company folds, or the operator simply changes its mind, the ownership records are gone and the items with them. This chapter specifies how ZecBit removes itself from that position.

The right way to reason about this is not "are we decentralized" - a word that means everything and therefore nothing - but three separate powers an operator might hold, each of which can be removed independently:

Power	The question it answers	Mechanism	Status
We could lie	Can the operator rewrite history, or show different histories to different people?	On-chain state anchors, chained (12A.1)	In development
We could lose	If the operator disappears, does ownership survive?	Published signed snapshots + reconstructible index (12A.2)	In development
We could obstruct	Can the operator block a transfer a holder wants to make?	Consensus-verified transfer authorization (12A.3)	Path specified; gated on protocol maturity

The first two are ours to solve now and depend on no protocol change. The third is the hard one, and its honest treatment is the subject of 12A.3.

12A.1 We cannot lie: chained state anchors

Publishing a Merkle root of the ownership registry to a public chain is the obvious first move, and by itself it is not enough. A bare root proves only that *some* state existed at *some* height. It does not establish that the published state is the latest one, it does not prevent the same item appearing twice, and - most importantly - it does not stop an operator from maintaining two divergent histories and showing each to a different audience. That last failure, **equivocation**, is the one that matters, because it is exactly what a dishonest operator would do.

The construction that closes it is due to Catena (Tomescu and Devadas, IEEE S&P 2017), and it is elegant: **each anchor transaction spends the output of the previous anchor**. The chain of anchors is therefore a chain of spends, and forking the log requires double-spending an output - which Zcash consensus will not accept. Non-equivocation stops being a promise we make and becomes a property the network enforces. An auditor needs only block headers and a few hundred bytes per statement to verify the whole history.

Two refinements complete the design:

- **A sparse Merkle tree**, keyed by a hash of the item identifier, so that each item occupies exactly one structurally determined position. This gives both membership proofs ("this item

is yours”) and *non-membership* proofs (“this item was not assigned to anyone else”), which a plain list cannot provide.

- **Consistency proofs** between successive roots, in the manner of Certificate Transparency (RFC 9162), so that anyone can verify each published state is an extension of the previous one rather than a quiet rewrite.

What this buys a holder: a proof, checkable against Zcash mainnet, that their ownership was recorded at a specific height, and that we cannot retroactively deny or reassign it without producing an inconsistency anyone can detect. What it does not buy: protection against an operator who simply stops publishing. That is 12A.2’s problem.

Anchoring application state into Zcash is not novel - it is being done today by other projects using memo fields on mainnet, and ZIP 302 reserves a leading byte specifically for application-defined memo content. Our anchors are placed where they are *publicly discoverable*, which is a requirement the chained construction imposes: a memo is readable only by its recipient, so a chain of anchors that no third party can find cannot be audited by one.

12A.2 We cannot lose: reconstructible state

Anchors prove that a state existed. They do not, by themselves, tell anyone what that state *was*. Data availability is a separate obligation, and the failure mode is well documented elsewhere: client-side-validation systems on other chains have a property their own documentation admits, where losing your local proof archive means losing the asset, and a seed phrase is not sufficient to recover it. That is unacceptable for a consumer marketplace.

ZecBit’s answer is to make the full state redundantly public:

- **Signed snapshots**, published on a fixed schedule: the complete set of ownership *commitments*, the tree root, the anchor transaction carrying it, and a hash chaining it to the previous snapshot. Distributed by ordinary means - a public repository, content-addressed storage, and mirrors.

What a leaf contains matters more than that it exists, because publishing the assignment layer naively would recreate exactly the public ownership ledger this system exists to avoid. Each leaf is therefore a commitment, not an identity:

```
leaf = H( item_id || owner_secret || salt )
```

where `owner_secret` is known only to the holder and `salt` is fresh per assignment. A reader of the snapshot learns how many items exist, that each occupies exactly one position, and that the set has not been rewritten between anchors. A reader learns nothing about who holds any of them, and two snapshots do not reveal that the same party holds two items. The holder, possessing the preimage, can open their own leaf to a chosen recipient at a time of their choosing - the same disclosure discipline the rest of this document applies to shielded notes, applied to the assignment layer.

- **A holder-side certificate:** each holder can export a compact proof of their own item - the leaf, its Merkle path, and the anchoring transaction - so that one holder's claim survives independently of everyone else's copies, including ours.

The same structure bounds the mainnet claim of section 6.3. A leaf is claimable exactly once: opening it requires the preimage, the claim is recorded in the next snapshot, and the sparse tree's non-membership proofs (12A.1) make a second claim against an already-opened leaf detectable by any observer rather than only by us. Derivation guarantees that the mainnet asset is byte-for-byte the committed one; this is what guarantees it reaches one holder rather than two.

- **A deterministic index:** the catalogue itself is already rebuildable by anyone from chain data alone (Chapter 9), so the only thing snapshots must carry is the assignment layer.

The test we hold ourselves to is concrete: *if every ZecBit server disappeared tonight, could a third party reconstruct how many items exist and verify that no assignment was ever rewritten, and could an individual holder prove their own claim to their own item?* With snapshots and anchors published, the answer is yes to both, and it requires no cooperation from us.

Note what the test deliberately does not ask. It does not ask whether a third party can reconstruct *who owns what* - that would be a public ownership ledger, and building one would defeat the entire purpose of this system. Survivability here means every holder can prove their own claim and no claim can be silently erased or reassigned; it does not mean the holder set becomes public property when the operator vanishes. The two are frequently conflated, and the distinction is the whole reason the leaves are commitments.

12A.3 We cannot obstruct: the honest state of the art

The third power is the hardest, and this section will disappoint anyone hoping for a clean claim, because the honest position today involves a genuine trade-off.

For an operator to be unable to block a transfer, the *authorization* of that transfer must be verifiable by the network rather than by us. On chains with programmable contracts this is trivial. On Zcash today there are two paths, and each costs something.

The public path. Bind ownership to a transparent-layer output: the item lives on a specific output, transfer means spending it, and authorization comes free from Zcash consensus. This works now, it is fully verifiable by anyone, and the operator disappears from the trust path entirely. The cost is decisive for us: every owner and every transfer becomes public - precisely the surveillance surface this document exists to remove. Adopting it would mean building, on a privacy chain, the exact ledger of holders we criticize elsewhere. We are not willing to describe that as private ownership, and we will not ship it as the default.

The private path. Prove ownership and authorize transfers with zero-knowledge proofs over shielded notes, so that authorization is verifiable without revealing the holder. Until recently this was blocked by a basic gap: shielded addresses have no equivalent of signing an arbitrary message, so a holder had no way to attest to anything. That gap is closing. Work now exists - an independent library targeting Orchard, and a proposed protocol extension exposing a

randomized spend-authority key as a public input - that lets a holder prove control of a specific note bound to a specific message, without revealing the note, the address, or the holder. This is the shape of the correct answer.

It is also not ready. The library is young and unaudited; the protocol proposal is a draft. Building a transfer protocol on either is real research with real risk of not converging, and we will not present it as imminent.

Our position, stated plainly:

1. **Now:** anchors and snapshots (12A.1, 12A.2) remove the ability to lie and the ability to lose. These are the failures that actually destroy user holdings, and they are addressed without compromising privacy.
2. **Meanwhile:** we treat the ability to obstruct as an operator power we still hold, we say so rather than obscuring it, and we constrain it - every transfer request and every refusal is recorded in the same anchored log, so obstruction is at least detectable and provable after the fact.
3. **When ZSA activates:** the problem dissolves. Items become native shielded assets held in the holder's own wallet; transfer authorization is consensus-verified and private simultaneously, because that is what the protocol was designed to do. Our anchoring layer becomes redundant and is retired.
4. **If ZSA does not activate:** the zero-knowledge path above is the fallback, pursued as research rather than promised as a feature.

The reason this chapter exists is that most marketplaces never write it. The distinction between "we would not do that" and "we could not do that" is the entire difference between a custodian and infrastructure, and a reader deserves to know exactly which parts of a system have crossed that line and which have not.

Part V — Trading

13. Trading Architecture Overview and the Escrow Beta

13.1 Why Settlement Is the Hard Problem on a Shielded UTXO Chain

Every NFT marketplace ultimately reduces to one operation: the simultaneous exchange of an asset for payment between two parties who do not trust each other. On account-based smart-contract chains this problem is solved by a program: an exchange contract takes custody of both legs for the duration of a single atomic transaction, and the chain's execution semantics guarantee that either both transfers happen or neither does. Zcash offers no such instrument. There are no smart contracts, no on-chain programs, and no mechanism by which the chain itself can hold an asset "in trust" pending a condition. The chain validates transactions; it does not run marketplaces.

The natural fallback on a UTXO chain would be the technique used by Bitcoin Ordinals venues: partially signed transactions. A seller signs only their own input under a sighash mode that commits to their input and their desired output, and leaves the remainder of the transaction open for a buyer to complete. Zcash's shielded transaction format forecloses this path as well, for a structural reason. A shielded transaction carries a **single binding signature computed over the entire transaction** — it proves that the sum of all value commitments balances, and it can only be produced by a party who knows the commitment randomness of *every* spend and output in the transaction. There is no notion of a per-input signature that a counterparty can later complement. Two mutually distrusting parties therefore cannot incrementally assemble one shielded transaction: whoever finalizes it must know secrets belonging to both sides, which collapses the construction back into trust.

The consequence is stark and worth stating plainly, because it dictates the entire architecture of this chapter and the next: **trustless atomic settlement of an NFT-for-ZEC trade is not possible on Zcash in the current transaction format**. It becomes possible only when ZIP 228 ("Asset Swaps") activates, because ZIP 228 restructures the transaction format precisely at this point — it partitions a transaction into independently authorized Action Groups and makes the binding signature *aggregatable* across parties (Chapter 14).

Given that constraint, the design space for a marketplace operating today contains exactly three points:

1. **Operator-mediated escrow.** A marketplace-controlled account holds both legs briefly and performs the transfers itself. Atomicity comes from the operator's systems rather than the chain, so the design inherits operator trust in full. ZecBit uses this only as a test-network rehearsal (§13.2) and never on mainnet — it is listed here to complete the design space, not because we ship it.
2. **Trusted matcher over signed orders.** Parties pre-authorize their own leg cryptographically; a coordinator combines authorizations and submits the result. Trust in the coordinator is reduced to *liveness and privacy* — it can decline to settle or leak metadata, but it cannot

steal or alter terms, because the cryptography binds them. This is the ZIP 228 model. It is unavailable until ZIP 228 activates.

3. **Pure protocol swaps with no coordinator.** Parties discover each other and assemble bundles peer-to-peer. Even under ZIP 228 someone must perform matching and aggregation; removing the coordinator entirely removes the orderbook, price discovery, and the counterparty-privacy shield that a matcher provides (§14.2). This point exists in the design space but is a venue-less protocol, not a marketplace.

These three points sit on a trust spectrum:

FULL OPERATOR TRUST		MINIMAL TRUST	
[Custodial escrow]	[ZIP 228 matcher]	[P2P swaps]	
operator holds assets	operator holds only	no operator	
atomicity = database	signed orders	no orderbook	
theft possible	theft impossible	discovery	
(bounded by §13.3	terms immutable	unsolved	
accountability)	liveness + privacy		
	trusted only		
----- Phase T -----	----- Phase M -----		
(testnet)	(mainnet)		

ZecBit ships exactly one of them on mainnet: **Phase M**, the non-custodial matcher of Chapter 14, contingent on ZSA and ZIP 228 activation. **Phase T** is a test-network rehearsal — an engine that exercises the orderbook, wallet integration, registry, and API against assets with no monetary value, so that the surrounding system is proven before the settlement core arrives. It is scaffolding, built to be discarded (§13.4), and it is never promoted to a network where anything is worth stealing.

13.2 The Escrow Beta: Deposit-Based Trading on Testnet

The escrow engine implements the oldest marketplace design there is: both legs pass through an intermediary account, and the intermediary's own bookkeeping — not the chain — enforces that they travel together.

Escrow accounts and per-listing derivation. The engine controls a hierarchical shielded wallet. For every listing it derives a fresh diversified shielded address, so that each listing corresponds to a distinct receiving address and, after deposit, to an identifiable set of received notes. Per-listing derivation serves three purposes: incoming deposits are attributable to a specific listing without heuristics; the reconciliation procedure (below) can audit each listing's balance independently; and the disclosure-based proof-of-deposit of §13.3 can reveal exactly one listing's notes without exposing any other. The spending authority for all derived addresses resides in a single engine-controlled key tree; Phase T makes no claim to the contrary, and key-management hardening is discussed with the rest of the threat model in Chapter 16.

The trade flow. A sale proceeds as follows:

Seller	ZecBit Engine	Buyer	Chain
-- create listing ----->			
<- listing addr (L) ----			
== shielded transfer: NFT note -> L =====>[tx1]			
	<--- deposit detected (scan) -----		
	-- state: LISTED ----->	(visible in	
		orderbook)	
	<---- purchase intent ---		
	-- payment addr (P) ---->		
	<== shielded transfer: ZEC -> P =====[tx2]		
	-- state: FUNDED		
	== leg A: NFT -> buyer =====[tx3]		
	== leg B: ZEC -> seller =====[tx4]		
<-- settlement notice -- <-- settlement notice --->			
	-- state: SETTLED		

Note what the diagram makes explicit: settlement is **two independent on-chain shielded transfers** (tx3, tx4). The chain sees two unrelated transactions and would happily confirm one without the other. The atomicity the user experiences is manufactured inside the engine: the decision to emit both legs, and the recording of that decision, occur inside a single ACID database transaction that transitions the listing to a settling state and enqueues both transfers as a unit. If the process dies at any point after that commit, the recovery procedure (below) finds the settling record and completes whichever legs lack confirmed transaction IDs. Atomicity is thus a property of the engine's write-ahead state, not of the chain — which is exactly why this design is custodial and exactly why it is confined to testnet.

Listing state machine. Every listing is an instance of the following machine. Transitions marked with ***** are driven by chain observation (the engine's note scanner); the rest are driven by API calls or the engine's own scheduler.

```

+-----+
create ----->| CREATED |
+-----+
| * NFT deposit confirmed (depth >= d_min)
v
+-----+ seller cancels +-----+
| LISTED |----->| REFUNDING |
+-----+ or listing TTL +-----+
| elapses |
| * ZEC payment confirmed | NFT returned
| (full amount, depth >= d_min) v
v +-----+
+-----+ | REFUNDED |
| FUNDED | +-----+
+-----+
| engine opens settlement record
v (single DB transaction)
+-----+
| SETTLING |----- crash ----> recovery re-enters
+-----+ SETTLING idempotently
| both legs confirmed
v
+-----+
| SETTLED | terminal states: SETTLED,
+-----+ REFUNDED, EXPIRED

```

A listing whose deposit never arrives within its deposit window moves **CREATED** → **EXPIRED**. A funded listing cannot be cancelled by the seller: once **FUNDED** is recorded, settlement is committed, which is the escrow-world analogue of the “no free option” discipline formalized for the matcher in §14.5.

Failure handling at every edge. The engine’s correctness argument is only as good as its treatment of the unhappy paths. The material cases:

Failure	Detection	Recovery
Deposit orphaned by reorg. The NFT deposit (tx1) confirms, the listing goes LISTED , then a chain reorganization drops tx1.	The scanner tracks each relied-upon note’s block hash, not just height; a reorg that invalidates the containing block marks the note <i>unconfirmed</i> .	Listing reverts to CREATED and is hidden from the orderbook until the deposit reconfirms at depth d_min (deposits are only acted on after a confirmation-depth threshold precisely to make this rare). If the deposit never reconfirms, the listing expires; nothing was owed to anyone.

Partial payment. Buyer sends less than the listed amount to the payment address, or sends in multiple transactions.	Payment addresses are per-purchase-intent; the scanner sums confirmed notes received at the address.	FUNDED is entered only when the confirmed sum meets the full price before the intent's deadline. A shortfall at deadline triggers automatic refund of whatever was received, to a refund address the buyer supplied at intent creation (the engine never invents a return path — shielded funds have no discoverable sender). Overpayment: the excess is returned in the settlement batch.
Payment orphaned by reorg after FUNDED .	Same block-hash tracking as deposits.	If the reorg occurs before the settlement record is opened, the listing reverts to LISTED and the intent to unpaid. If it occurs after — the narrow ugly window — the engine has emitted or is emitting legs against payment that no longer exists. Mitigation is preventive (d_min chosen so that testnet reorg depth in practice never reaches it) and corrective: the reconciliation pass flags the listing for manual resolution, and Phase T's testnet-only scope caps the damage at test value. This window is one of the structural reasons the design is disposable.
Engine crash between settlement legs. tx3 broadcast, process dies before tx4.	On startup, recovery scans for SETTLING records with incomplete leg confirmations.	Each leg in the settlement record carries a deterministic idempotency key and, once built, the serialized transaction itself. Recovery re-broadcasts unconfirmed legs rather than rebuilding them, so a crash can delay settlement but cannot double-spend a leg or emit it twice with different notes.
Broadcast accepted, confirmation never observed (mempool eviction, expired transaction).	Per-leg confirmation timeout.	The leg is rebuilt from the same settlement record against current chain state and re-broadcast; the old serialized transaction is retained so that if <i>both</i> somehow confirm, reconciliation detects the double-send immediately (it cannot double-spend, since both spend the same notes, but the record must show which one won).

Reconciliation against wallet notes. Independently of the state machine, a periodic reconciler recomputes ground truth from the chain: it walks the engine wallet's full note set (received, spent, unspent, per derived address) and checks it against the sum of what the database says the engine *should* hold — the NFTs of all **LISTED** / **FUNDED** / **SETTLING** listings plus the ZEC of all **FUNDED** / **SETTLING** intents, minus everything recorded as settled or refunded. Any discrepancy halts new settlements and pages an operator. On a shielded chain this reconciliation is only possible *because* the engine holds the wallet's viewing keys; no outside party can perform it — which is the honest limitation that motivates the next section.

13.3 Accountability in the Custodial Phase

A custodial engine on a shielded chain is, from the outside, a black box: unlike a custodial marketplace on a transparent chain, users cannot even watch the escrow address on a block explorer. ZecBit does not pretend this trust gap away; it narrows it with the mechanisms that shielded infrastructure actually permits, and states plainly what residual trust remains.

What the chain does make public. ZSA issuance is the one transparent edge of the system: issuance bundles, and therefore the total supply ever issued per asset, are public chain data (Chapter 7). This yields a hard *upper bound* form of accountability: the escrow engine cannot conjure NFTs, and any claim it makes about holdings is checkable against public supply. It yields

nothing more — transfers are shielded, so escrow *holdings* are not publicly auditable. That is the transparency limit, and ZecBit’s remaining mechanisms are built to compensate for it.

Signed liability statements per listing. For every listing, on every state transition, the engine emits a statement — listing identifier, asset base, state, and the engine’s resulting obligation (“holds 1 unit of asset **A** for listing **L**, owed to the depositor of record”) — signed with a published ZecBit accountability key and delivered to the affected parties, timestamped and append-only. These statements do not prevent theft; they make repudiation impossible. An operator that misbehaves has, in the seller’s hands, its own signed admission of the liability it failed to honor.

User-visible proof-of-deposit. Because Phase T listing addresses are per-listing, the engine can prove receipt of a specific deposit without exposing anything else: it discloses the received escrow note for that listing to the seller (using the payment-disclosure machinery described in Chapter 11), letting the seller’s wallet verify against the chain that the note exists, is addressed to the escrow, carries the claimed asset base, and sits in a confirmed block. The seller thereby gets chain-grade evidence that “the escrow really holds my NFT” — the strongest statement possible without the escrow surrendering spend authority.

Why this never touches anything you own. An engine that moves assets on behalf of others is, by construction, only as trustworthy as its operator — which is precisely why this one is confined to a test network where assets carry no monetary value, and why no mainnet design in this document contains it. The rehearsal exists to prove out the orderbook, the interfaces, and the failure paths before ZIP 228 makes real settlement possible; it is not a product stage users are asked to trust with property. On mainnet there is one settlement design, and it is the non-custodial one in Chapter 14.

13.4 A Deliberately Disposable Design

The escrow engine’s most important design property has nothing to do with escrow: it is that **the engine exposes the same external contract the ZIP 228 matcher will expose**, so that when settlement becomes non-custodial, clients migrate without rewriting. Both phases present the same shape to a wallet: *orders in, settlement events out*. What changes is everything behind that boundary.

Interface element	Phase T (escrow engine)	Phase M (ZIP 228 matcher)	Client-visible change
Create listing	Listing object; engine returns a deposit address ; seller transfers NFT	Listing object; wallet attaches a signed swap order (\$14.1); no transfer occurs	Wallet builds an order instead of a deposit — a wallet-internal change (Ch. 11)
Listing becomes live	On confirmed deposit	On order validation	Same event name/schema; different latency profile
Buy	Purchase intent; engine returns payment address; buyer transfers ZEC	Purchase intent; wallet attaches signed taker order	Same substitution as listing

Cancel	Engine refunds NFT from escrow	Soft-cancel via orderbook, hard-cancel via self-spend (§14.4)	Same endpoint; response now distinguishes cancel tiers
Settlement notification	Emitted when both engine legs confirm	Emitted when the swap bundle confirms	Identical event schema; Phase M adds the bundle txid
Expiry	Engine TTL; engine refunds	nAGExpiryHeight ; order dies on chain, nothing to refund	Same event; Phase M expiry needs no refund leg
Asset custody during listing	Escrow wallet	Seller's wallet, uninterrupted	The point of the migration
Atomicity guarantor	Engine database transaction	Chain-validated aggregated binding signature	Invisible to the API, foundational to the trust model
Market data source	Engine settlement log	Matcher settlement log (§14.6)	Identical read API

Because the contract is shared, the orderbook service, registry, notification pipeline, and client SDKs built during Phase T carry over intact; Phase M replaces one component behind a stable boundary. The full API surface is specified in Chapter 17; the wallet-side order construction it presumes is Chapter 11's subject.

14. Non-Custodial Settlement: the ZIP 228 Matcher

ZIP 228 (“Asset Swaps”) is, at the time of writing, a **Draft** ZIP: it is not scheduled for a specific network upgrade and its reference implementation is pending. Everything in this chapter is therefore an architecture designed against a draft specification, and ZecBit tracks the ZIP’s evolution rather than assuming its finality. What the draft already provides, however, is precisely the primitive that §13.1 showed to be missing: a way for mutually distrusting parties to each authorize *their own fragment* of a transaction, such that a third party can assemble fragments into a valid whole without being able to alter, steal, or selectively use them.

The mechanism is the **Action Group**. Where a current Orchard transaction contains one flat set of actions under one anchor and one binding signature, a ZIP 228 transaction may contain two or more Action Groups, each carrying its own anchor root, its own flags, and its own expiry height (`nAGExpiryHeight`), and each independently authorized by its creator’s spend-authorization signatures over that group’s digest. Balance is still proved by a single binding signature over the whole transaction — but the key for that signature can now be *aggregated* from per-order contributions, which is what lets a coordinator finalize a transaction whose secrets it does not know.

14.1 Anatomy of a Swap Order for the NFT Case

A swap order is an off-chain object — it never appears on the chain in order form — with four components. For a maker listing an NFT at a price of N/ZEC :

```

SWAP ORDER (maker: sell NFT, want N ZEC)

=====
| ProposedInput  : SwapIO                                (40 bytes) |
|   --- asset_base : 32 bytes  -- the NFT's unique ZSA asset base   |
|   --- qty       : u64       -- 1  (an NFT is a qty-1 asset)       |
|-----|
| DesiredOutput  : SwapIO                                (40 bytes) |
|   --- asset_base : 32 bytes  -- ZEC (the native-asset base)       |
|   --- qty       : u64       -- N zatoshi                           |
|-----|
| bsk           : 32 bytes                                     |
|   -- the sum of the value-commitment randomness (rcv) across this |
|   order's actions; the maker's additive share of the eventual    |
|   transaction-wide binding signature key                          |
|-----|
| SwapActionGroup : ActionGroupDescription                  |
|   --- anchor (rt)      -- note-commitment tree root the maker's  |
|   |                     spend proof is valid against              |
|   --- flags                                                    |
|   --- nAGExpiryHeight : u32  -- settlement deadline (see §14.5)  |
|   --- actions (>= 2):                                          |
|   |   [A1] SPEND  : maker's NFT note          (zk spend proof)    |
|   |   [A2] OUTPUT : new note, N ZEC, addressed TO THE MAKER      |
|   |   [A3] fee-side action(s) (see §14.3)                       |
|   --- spend-auth signatures over THIS GROUP'S digest           |
=====

```

Three properties of this structure carry the whole design, and each deserves emphasis.

The order is self-addressed. Action A2 — the payment the maker will receive — is an ordinary shielded output note that the maker constructs *to their own address* at order-creation time. The counterparty does not appear anywhere in the order; in the ZIP's own framing, order senders do not know in advance with whom their orders will be matched. This is the inversion that makes counterparty-blind matching possible: instead of "pay Alice's leg to Bob," each party pre-writes their own incoming leg, and matching consists of finding groups whose declared inputs and self-addressed outputs mirror one another.

The order is fully proved and fully signed at creation. The spend of the NFT note carries its zero-knowledge proof; the group carries spend-authorization signatures over its digest, and under the v1 digest that coverage includes `nAGExpiryHeight`. Nothing about the group can change afterward without invalidating it. An order is not an intention — it is a complete, self-contained, cryptographically closed fragment of a future transaction.

The **ProposedInput / DesiredOutput** pair is the matcher-readable summary. The action group itself is shielded; the matcher cannot read notes out of it. The two **SwapIO** fields are the plaintext header the maker deliberately exposes so that matching is possible at all: *offering asset X, quantity q; requiring asset Y, quantity r*. This is the exact and total extent of voluntary disclosure, and §14.2 tabulates it.

The taker's order is the mirror image:

```

SWAP ORDER (taker: pay N ZEC, want the NFT)
-----
ProposedInput  = ( ZEC,                N zatoshi )
DesiredOutput  = ( NFT asset_base, 1      )
SwapActionGroup:
  [B1] SPEND   : taker's ZEC note(s) totaling >= N
  [B2] OUTPUT  : new note, the NFT, addressed TO THE TAKER
  [B3] change output back to the taker, if any
  [B4] fee-side action(s)
  spend-auth signatures; own anchor; own nAGExpiryHeight

```

Two structural notes for the NFT case specifically. First, ZIP 228 as drafted supports no partial fills — orders match exactly or not at all. For fungible pairs this is a real limitation; for NFTs it is no limitation whatsoever, since the asset leg is indivisibly **qty = 1**. Second, because each group carries its own anchor, maker and taker may prove against different note-commitment tree states; neither party's proof goes stale merely because the other constructed theirs later.

14.2 Matching and Assembly: What the Matcher Can and Cannot Do

The **matcher** is ZecBit's Phase M coordinator. It maintains the orderbook of signed swap orders and, when it finds complementarity, assembles a settlement bundle.

The complementary-order predicate. Orders SO_1 (maker) and SO_2 (taker) match when:

```

S01.ProposedInput.asset_base == S02.DesiredOutput.asset_base    (the NFT)
S01.ProposedInput.qty         == S02.DesiredOutput.qty           (= 1)
S02.ProposedInput.asset_base == S01.DesiredOutput.asset_base    (ZEC)
S02.ProposedInput.qty         == S01.DesiredOutput.qty           (price agrees)
current_height + safety_margin < min(S01, S02).nAGExpiryHeight

```

with the venue's tie-breaking policy among multiple candidates given in §14.4.

Bundle assembly. The matcher concatenates the two action groups (plus fee groups, §14.3) into one transaction, then produces the transaction-wide binding signature using a key formed by the modular addition of the orders' **bsk** values:

$$\text{bsk_bundle} = \text{bsk_1} + \text{bsk_2} + \text{bsk_fee_groups} \pmod{r}$$

The maker's `bsk1` is the sum of value-commitment randomness across the maker's actions; likewise the taker's. Their sum is exactly the randomness sum of the union of actions — so the aggregate key produces a binding signature valid for the whole bundle, *and for nothing else*. Consensus then checks what it always checks: every group's proofs and spend-auth signatures, every group's expiry against the current height, and the binding signature's balance proof over the entire transaction.

This one aggregation rule, combined with per-group spend-auth signatures, yields the full set of non-custodial guarantees. Each is a theorem about signatures, not a policy promise:

Attempted matcher misbehavior	Why it fails	Cryptographic mechanism
Alter any order term — price, asset, expiry, any action	The group's digest changes; the maker's spend-auth signatures over that digest no longer verify	Spend-authorization signatures over the Action Group digest (expiry included in the v1 digest)
Redirect an output — pay the seller's proceeds elsewhere	The output note's recipient is fixed inside the self-addressed output action; changing it is altering the group	Self-addressed outputs + the row above
Cherry-pick actions — e.g., take the NFT spend but drop the payment output	A subset of actions has a different randomness sum; <code>bsk</code> values are per-order contributions, so no subset's binding signature can be formed	Aggregated binding signature: valid only over the exact union of the contributed orders' actions
Steal the assets outright	The matcher holds signed orders, never notes; it has no spend authority over anything	Orders are authorizations, not custody
Hold an order forever as leverage	The group dies at <code>nAGExpiryHeight</code> ; consensus rejects expired groups	Signed, consensus-enforced expiry (\$14.5)
Learn who is trading	Notes, addresses, and memos are shielded inside the groups; the matcher reads only the <code>SwapIO</code> headers	Shielded action groups; disclosure limited to <code>ProposedInput / DesiredOutput</code>

And the matcher's knowledge, stated positively — because honest accounting of what a coordinator *does* see is as important as what it does not:

The matcher sees	The matcher does not see
Asset bases and quantities of both legs (hence: which NFT, at what ZEC price)	Any address, on either side
Fee amounts and fee-group structure	Any linkage between an order and an on-chain identity

Each order's <code>nAGExpiryHeight</code> and anchor	Note contents, memos, or the parties' other holdings
Submission metadata (timing, connection) — a Chapter 16 topic	Anything about the trade once settled, beyond its own logs (§14.6)

The residual trust in the matcher is therefore exactly two-dimensional: **liveness** (it could decline to match or to broadcast — analyzed as harmless-but-annoying in §14.7) and **privacy of the metadata it inherently sees** (formalized in Chapter 16). Value-safety requires no trust at all.

14.3 Fees at Assembly

Fee handling is where the assembling venue exercises the one power it legitimately has: adding *its own* action groups to the bundle. Two kinds are added, and it matters that they are venue-level constructs, honestly labeled as such.

The matcher fee. ZIP 228 provides for the matcher to be compensated in ZEC via a fee action or an additional action group contributed by the matcher itself: the participating orders are constructed so that the ZEC legs leave a defined surplus, and the matcher's own group claims that surplus to a matcher address, alongside the standard miner fee that balances the transaction. The mechanism's integrity properties are the same as everything else in the bundle: the fee the maker and taker will bear is fixed by the arithmetic of their *own signed groups* — the matcher cannot enlarge its take without breaking the balance equation that the aggregated binding signature must satisfy against the signed, unalterable orders. ZecBit's fee *schedule* is a published commercial term outside this document's scope; the *mechanism* is as described, and an order's signed terms are always sufficient to compute the worst-case fee its creator can bear.

Creator royalties. The asset registry (Chapter 9) lets an issuer declare a royalty: a beneficiary shielded address and a proportion. At assembly time, the matcher inserts a royalty action group routing the declared proportion of the ZEC leg to the beneficiary, and ZecBit's matching policy refuses to assemble bundles for royalty-declaring collections without it. ZecBit states the nature of this guarantee without decoration: **royalty enforcement is venue-level, not protocol-level**. This is the model transparent-chain PSBT venues converged on (the assembling venue inserts the creator-fee output; the guarantee holds only on that venue), and no stronger guarantee exists on a chain without contracts: a different assembler, or two parties settling peer-to-peer under ZIP 228 itself, can omit the royalty group entirely. What ZecBit guarantees is the conjunction "*settled through ZecBit* $\Rightarrow$ *royalty paid*," enforced by the same mechanism that pays the matcher, with the royalty transfer itself shielded like any other.

14.4 The Order Lifecycle

Wallet	ZecBit Matcher	Chain
(1) CREATE		
build actions, zk proofs,		
self-addressed output,		
choose nAGExpiryHeight,		
spend-auth sign, derive bsk		
--(2) SUBMIT order----->		
	(3) VALIDATE + STORE (private)	
<--- order accepted -----		
	(4) MATCH (price-time)	
	(5) ASSEMBLE bundle,	
	add fee + royalty groups,	
	aggregate bsk, bind-sign	
	--(6) BROADCAST----->	
	<----- confirmations -----	
<--(7) settlement event -----		
wallet detects incoming		
note independently (Ch. 11)		

(1) Creation is entirely wallet-side and is the expensive step: zero-knowledge proof generation for the spends, note construction for the self-addressed output, expiry selection (§14.5), spend-auth signing, and `bsk` derivation. The wallet internals are Chapter 11's subject; what matters architecturally is that after step (1) the user's device holds a finished bearer-grade artifact and needs no further involvement until settlement lands.

(3) Private storage — the bearer-instrument rule. A signed swap order is *live authorization*: anyone who possesses it and a complementary order (plus the modest fee-group contribution) can assemble a valid settlement, because the order's binding-signature share travels with it. PSBT marketplaces on Bitcoin learned the corresponding lesson in production: a "cancelled" or "repriced" listing whose old signed artifact had leaked remained executable at the old terms, because deleting a database row does not revoke a signature. ZecBit therefore treats the orderbook's confidentiality as a **security property, not merely a business one**: raw signed orders are never published, never returned by any read API (Chapter 17 exposes listing *metadata* only), and are stored and access-controlled as secrets (Chapter 16). The public can see *that* an NFT is listed and at what price; only the matcher's assembly path ever touches the signed object. That this same confidentiality is also the venue's principal defensible asset (§14.6) is a happy alignment, not the rationale.

(4) Matching policy. The complementary-order predicate (§14.2) is protocol; the choice among multiple satisfying candidates is venue policy. ZecBit's baseline is **price-time priority**: for a taker

crossing the book, the best-priced complementary order wins, ties broken by order-acceptance timestamp. The policy is published, versioned alongside the API (Chapter 17), and deliberately boring — a matcher whose value-safety is cryptographic competes on the fairness and transparency of exactly this remaining discretionary layer, and §14.6’s verifiable reporting exists partly so that policy conformance is checkable after the fact.

Cancellation semantics are where the signed-order model differs most sharply from user expectations formed on contract chains, and ZecBit surfaces the three tiers explicitly rather than papering over them:

Tier	Mechanism	Guarantee	Cost / latency	Against whom it protects
Soft-cancel	Orderbook delisting: matcher marks the order dead, excludes it from matching, hides the listing	Trust-based: only as good as the matcher’s promise and its custody of the signed artifact	Instant, free	An honest matcher (i.e., the normal case)
Hard-cancel	Self-spend: the wallet spends the order’s input note in an ordinary shielded transaction to itself; the order’s spend proof now references a spent note and no assembly of it can ever validate	Trustless and absolute — enforced by consensus, effective immediately upon the self-spend confirming	One on-chain transaction and its miner fee	Everyone, including a malicious or compromised matcher and any leaked copy of the order
Expiry	nAGExpiryHeight passes; consensus thereafter rejects any bundle containing the group	Trustless, but passive — protection arrives at the deadline, not on demand	Free; latency = time remaining to expiry	Everyone, eventually

Cancellation is *not a protocol message* — ZIP 228 defines no “revoke order” object, and none is possible, because a signature cannot be unsigned. An order dies in exactly two ways: its expiry passes, or its input note is spent elsewhere. The soft/hard distinction is ZecBit’s honest packaging of that reality: soft-cancel is the cheap everyday path whose integrity §13.3-style accountability and the never-publish rule make trustworthy in practice; hard-cancel is the escape hatch that means no user is ever *forced* to trust the venue’s cancel button. The recommended wallet behavior — surface soft-cancel by default, offer “revoke on-chain” for high-value listings and repricings — is specified in Chapter 11.

Repricing deserves one explicit sentence, again on PSBT-venue evidence: a repriced listing is a **new order**, and if the price moved downward materially, the wallet should hard-cancel the old one, because a soft-cancelled cheaper order is a standing discount to anyone who ever obtains the artifact.

14.5 Expiry Discipline

nAGExpiryHeight is a uint32 block height, chosen by the order’s creator, covered by the creator’s signature, and enforced by consensus: a bundle containing an expired group is invalid. Two subtleties govern how ZecBit uses it.

Expiry bounds settlement, not execution. The deadline constrains when the *bundle confirms on chain*, not when matching happens. A matcher must therefore apply a safety margin — it should not assemble against an order whose remaining lifetime is within the expected broadcast-to-confirmation window, or the settlement races the deadline and can die in the mempool. ZecBit’s matcher enforces a minimum remaining-lifetime threshold at assembly time and treats orders inside the margin as unmatched pending expiry.

Choosing the horizon is a real trade-off. Short expiries maximize the maker’s optionality and minimize the lifetime of the bearer artifact, at the cost of *re-listing churn*: every expiry that the maker still wants live must be replaced with a freshly proved, freshly signed order — wallet work (proof generation is nontrivial) and orderbook noise. Long expiries minimize churn but lengthen the window during which a leaked artifact matters and during which the maker’s price may go stale. ZecBit’s client defaults therefore scale the horizon with intent — short for aggressive repricing, longer for patient listings — with the venue-recommended values living in the API’s advisory parameters (Chapter 17), not in protocol.

Signed expiry kills the free option. The classic failure of cross-chain atomic swaps (HTLC-style) is well documented: the party who moves second holds, during the timeout window, a *free option* — settle if the price moved favorably, abandon if not — and the party who moved first bears that optionality as pure cost. The ZIP 228 construction dismantles this at both ends. There is no “second mover” inside a bundle: both legs are pre-signed and settle in one transaction or not at all. The only optionality left is the matcher’s discretion over *whether* to assemble — and that is bounded by an expiry the *maker* chose and signed, which the matcher cannot extend (v1 digest coverage), cannot exploit beyond its horizon, and cannot separate from the order. An unmatched order is not an open-ended dangling commitment; it is a firm quote with a firm, self-imposed, consensus-enforced end.

14.6 Market Data on a Shielded Chain

On a transparent chain, market data is a public good extracted from the ledger: anyone can compute any collection’s floor, volume, and history from confirmed transactions. On Zcash, a settled swap is an ordinary shielded transaction — **third parties cannot see that a trade occurred at all**, let alone its asset or price. The chain contributes exactly one public fact stream (ZSA issuance, §13.3) and nothing about secondary trading.

The consequence is structural: **the matcher’s settlement log is the only source of floor, volume, and history in the ecosystem.** This cuts two ways. Commercially, it means the defensible asset of a Zcash-native marketplace is not fee capture — a fee-free clone can re-assemble any *public* orders, which is one more reason orders are never public — but the private orderbook, the bid liquidity resident in it, and the market-data monopoly that shielded settlement creates. Epistemically, it means every statistic ZecBit publishes is a *claim by an interested party*, checkable by no outside observer against the chain. A venue that is the sole witness to the market must either be blindly believed or must make its reports verifiable. ZecBit chooses the latter.

Verifiable aggregate reports. For each reporting period (an epoch of block heights), the matcher publishes a signed report constructed as follows:

1. For every settled trade i in the epoch, the matcher forms a **per-trade commitment** $c_i = H(\text{salt}_i \parallel \text{canonical_trade_record}_i)$, where the canonical record contains the epoch, collection, asset base, price, fee and royalty amounts, settlement txid, and bundle position, and salt_i is a fresh random value generated per trade.
2. The commitments $\{c_i\}$ are the leaves of a **Merkle tree**; the report contains the root R , the leaf count, the epoch's aggregate statistics (trade count, volume, floor, per-collection breakdowns), and the matcher's signature over all of it, chained to the previous epoch's report hash so the sequence is append-only.
3. **Privately**, each trade's maker and taker receive $(\text{salt}_i, \text{canonical_trade_record}_i, \text{merkle_path}_i)$ over their authenticated channel.

The properties, and who checks them:

- **Inclusion, checked by participants.** Either party to trade i recomputes c_i from the record and salt they received, verifies the Merkle path to the published R , and — decisively — checks the record's txid and terms against *their own wallet's view of the settlement*, which they possess independently (their wallet detected the notes; Chapter 11). A matcher that omits or misstates their trade is caught by the very person with both the evidence and the incentive to look.
- **No public per-trade disclosure.** To everyone else, a leaf is an opaque salted hash: the per-trade salt defeats dictionary attacks over the (small, guessable) space of plausible trade records, so publishing the tree reveals the aggregate statistics and the leaf count and nothing per-trade. Privacy of individual trades survives the report.
- **Consistency, checked by anyone.** The signed, hash-chained epoch sequence means the matcher cannot rewrite history or show different reports to different audiences without producing conflicting signatures — self-incriminating artifacts in the §13.3 sense.
- **The honest residual.** Commitments prevent *understatement* and *falsification of real trades* against alert participants; they cannot prevent *fabrication* — the matcher inventing trades with itself to inflate volume — because a fabricated trade's "participants" are the matcher and will happily hold its salts. Wash-trading resistance on a shielded chain is a matter of fee economics (every fabricated settlement pays real miner fees and, structurally, the venue's own royalty routing) and of reputation, not cryptography, and ZecBit says so rather than implying otherwise.

Portability limits. The same structure bounds what users can take with them. A participant can *prove to a third party* that their specific trades happened (salt + record + path + signed root is a portable receipt — useful for provenance, Chapter 9), but nobody, including ZecBit, can hand the market's history to a successor venue in verifiable form beyond these signed aggregates; and reputation built on ZecBit activity is likewise attestation-based (venue-signed statements about a pseudonym) rather than chain-derived. Data on a shielded chain is not a commons, and architectures that pretend otherwise are describing a transparent chain. Chapter 16 returns to the matcher's own data handling obligations that this monopoly position creates.

14.7 Failure and Adversary Cases

The matcher's threat model is formalized in Chapter 16; here we close the settlement design by walking the four adversary cases specific to the swap machinery, applying the guarantees of §14.2.

Withholding. The matcher goes dark, or selectively refuses to match certain orders. Effect: no settlement occurs; every outstanding order dies at its own `nAGExpiryHeight`; no assets ever left any wallet. Loss: zero value; cost is the opportunity cost of the unfilled trade plus re-listing effort elsewhere. Withholding is the *worst* a purely malicious-but-non-colluding matcher can do to funds, which is the design's central achievement. (Selective withholding as unfairness — censoring a competitor's collection, say — is a policy-conformance issue, made evidence-generating by §14.6 reporting rather than impossible.)

Front-running. On contract-chain venues, a coordinator seeing an attractive incoming order can insert its own transaction ahead of it. Translate the attack here: the matcher sees taker order T crossing maker order M at a price it considers cheap, and wants the NFT for itself. It cannot redirect M 's NFT output — that output is self-addressed inside M 's signed group. Its only move is to *be the counterparty*: construct its own taker order T' and match M against T' instead of T . But T' must satisfy the same complementary predicate — the matcher must actually pay the maker's full signed price from its own funds, taking real inventory risk, exactly as any market participant could. What survives is not front-running in the extraction sense but **self-preferencing in queue priority**: matching itself ahead of an equal-priced T in violation of price-time policy. The maker is indifferent (identical signed terms either way); the displaced taker is harmed only in priority; and the venue's policy publication plus epoch reporting make systematic self-preferencing a detectable pattern rather than a silent tax. There is no riskless-profit insertion for the matcher to perform.

Matcher-counterparty collusion. The matcher conspires with a taker against a maker. Every lever is already welded shut by signatures: terms cannot be altered (spend-auth over the group digest), outputs cannot be redirected (self-addressing), actions cannot be subset-extracted (bsk aggregation), expiry cannot be stretched (v1 digest). The collusion surface reduces to *information*: the matcher leaking orderbook state — e.g., the existence and pricing of soft-cancelled orders, or book depth — to a favored trader. That is a genuine privacy/fairness harm, bounded in value terms by the fact that every order's worst case is its own signed terms, and it is treated as what it is: an insider-data problem, handled by the access-control, logging, and minimization obligations of Chapter 16, not a settlement-integrity problem.

Griefing via mass fake orders. An attacker floods the book with orders they intend to kill — priced to sit at the top, then self-spent (hard-cancelled) the moment a settlement is assembled against them, wasting matcher work and polluting price signals; or simply submitted in bulk to bloat storage and matching. On a contract chain the standard remedy is an order bond that misbehavior forfeits. **Order bonds are not possible here** — no contracts means no programmable slashing — and ZecBit does not pretend at a cryptographic remedy where only operational ones exist:

- *Validity at admission:* an order is only accepted after full validation — proofs verify, anchor is known, expiry is sane — so unfunded or malformed spam never enters the book, and (a point developed for offers in §15.5) a valid order **is** proof its input note existed at its anchor.
- *Priced admission:* per-identity and per-connection rate limits on order submission, with authenticated sessions (Chapter 17) so limits attach to accounts rather than IPs.
- *Behavioral scoring:* an account whose orders are repeatedly hard-cancelled at assembly time — the note found spent when the bundle is built — accumulates a reliability penalty: reduced submission limits, deprioritized matching, eventual suspension. Assembly-time note-status checks make each such event cheap to detect and attribute.
- *Assembly-time revalidation:* every order in a candidate bundle is rechecked (note unspent as of the latest chain state, expiry margin satisfied) immediately before broadcast, so a poisoned order costs the matcher a rebuild, not a failed on-chain settlement — the same isolation principle §15.3 applies to sweeps.

These measures manage griefing; they do not eliminate it, and the residual — a well-funded adversary rotating identities — is a rate-limiting arms race common to every off-chain orderbook, shielded or not. It is recorded as such in Chapter 16.

15. Marketplace Feature Set on the Signed-Order Model

Chapters 13 and 14 fixed the settlement substrate. This chapter derives the product from it: which marketplace features the signed-order model supports natively, which require an interactive protocol on top, and which ZecBit declines to build because the substrate cannot support them honestly.

15.1 Feature Matrix

Feature	Mechanism	Phase	Trust level	UX flow (summary)
Fixed-price listing	Maker order: $\text{ProposedInput} = \text{NFT}$, $\text{DesiredOutput} = \text{ZEC}$ (§14.1)	T (escrow analogue) / M	M: non-custodial	List → order signed in wallet → live; sale settles without seller present
Item offer (bid on a specific NFT)	Maker order, mirrored: $\text{ProposedInput} = \text{ZEC}$, $\text{DesiredOutput} = \text{the item's asset base}$	T / M	M: non-custodial	Bid → funded order rests in book → holder accepts → immediate settlement
Collection offer ("any item in collection C")	Intent object + interactive accept (§15.2) — not a single signed order	M	Non-custodial; intent is unsigned until accept	Bidder posts funded intent → holder accepts against their item → bidder wallet co-signs concrete order → settle
Trait offer ("any item in C with trait t")	Same as collection offer with a registry-validated trait predicate (metadata per Ch. 9)	M	As collection offer	As collection offer, with trait check at accept time
Sweep (buy N listed items at once)	One taker intent resolved against N maker orders → single multi-group bundle (§15.3)	M	Non-custodial	Select items → wallet builds/signs per-item taker orders → one settlement transaction
Private sale (directed listing)	Standard maker order + matcher routing restriction to a designated counterparty (§15.4)	T / M	Non-custodial; routing is venue-enforced	Seller names counterparty handle → only that account's order can match
Editions trading (semi-fungible ZSAs, $\text{qty} > 1$)	Same order anatomy with $\text{qty} = k$; no partial fills (§14.1), so each order is take-it-whole; venue may suggest denomination splitting into multiple orders	T / M	Non-custodial	List k units at total price → sold as a lot; or list as k separate $\text{qty}-1$ orders for granularity

Two entries in this table are not "just orders" and get their own sections: collection/trait offers (§15.2), because a single signature cannot range over N items, and sweeps (§15.3), because multi-group assembly has failure-isolation obligations.

15.2 Interactive Accept: Collection and Trait Offers

A collection offer says "I will pay P for *any* item in collection C." The signed-order model cannot express this in one artifact, for a reason visible directly in §14.1: `DesiredOutput` names a **concrete asset_base**. An NFT's asset base is unique to the item; there is no "any of these N bases" wildcard, and no disjunction a spend-auth signature could range over. Bitcoin PSBT venues hit the same wall in identical form — one signature cannot cover "any of N items" — and converged on the same resolution ZecBit adopts: the standing bid is an *intent*, and the holder's acceptance triggers an interactive step in which the bidder's wallet resolves the intent into a concrete signed order against the specific item being sold.

Bidder wallet	ZecBit	Holder wallet
--(1) post INTENT --->		
{collection C,		
[trait pred t],	-- intent visible to	
price P, TTL,	holders of C [with t] -->	
funding attest.		
(see §15.5)}		
	<--(2) ACCEPT (item X) -----	
	holder signs maker	
	order: NFT_X -> P ZEC	
	(concrete, self-addr.)	
<-(3) resolve req. ---		
"intent I matched		
against asset_		
base X; co-sign?"		
--(4) signed taker -->		
order: P ZEC ->		
DesiredOutput = X		
	--(5) assemble + broadcast (§14.2)	
<----- settlement event ----->		

Design points that make this construction sound rather than merely workable:

- **The intent is not a bearer instrument.** Unlike a signed order, an intent carries no spend authorization — leaking it leaks a price signal, nothing more. The bearer-instrument discipline of §14.4 applies only from step (4) onward, and the step-(4) order names one concrete item with a deliberately short expiry, so its bearer-risk window is minimal.
- **Step (3) is the policy gate.** The bidder's wallet — not the venue — verifies before co-signing that item X is genuinely in collection C under the registry's issuer-binding rules (Chapter 9) and satisfies the trait predicate against registry-attested metadata. A venue that lied at step (2) produces a co-sign refusal, not a bad trade. Wallet-side auto-accept policies ("co-sign

automatically if registry checks pass and $\text{price} \leq P$) make the interaction invisible for online wallets while keeping the trust boundary in the right place; the wallet behavior is Chapter 11's.

- **Latency honesty.** Interactive accept means a collection offer settles only when the *bidder's* wallet is reachable for step (4). ZecBit surfaces bidder responsiveness (recent auto-accept availability) on standing intents, because a bid that cannot complete is book pollution — the same hygiene concern as §14.7, solved by disclosure rather than pretense.
- **Holder's risk is nil.** The holder's step-(2) order is a normal maker order: self-addressed, exact-priced, expiring. If the bidder never completes, it expires or is hard-cancelled like any listing; the holder has granted a short option whose terms they signed, not shipped an asset.

15.3 Sweeps: Multi-Group Settlement

A sweep — buying N listed items in one action — is where ZIP 228's transaction structure pays for itself: since a transaction may contain arbitrarily many Action Groups, N maker orders plus the buyer's corresponding taker groups can settle as **one on-chain transaction**, atomically, in both directions — the buyer cannot end up with 3 of 5 items, and no seller can be settled while another is dropped, because a single aggregated binding signature covers the exact union of all included orders' actions (§14.2: subsets don't verify).

Assembly for a sweep of maker orders $M_1 \dots M_N$:

1. **Taker side.** The buyer's wallet constructs one taker order per item (`ProposedInput` = ZEC at M_i 's signed price, `DesiredOutput` = item's asset base), funding them from its note set with change handled across the taker groups. Per-item taker orders — rather than one monolithic group — are what make failure isolation possible in step 3.
2. **Balancing and fee aggregation.** The matcher computes the bundle-wide balance across all groups and attaches a *single* consolidated fee structure: one matcher-fee group covering the venue fee across all N legs, royalty groups per distinct collection represented (a sweep spanning collections routes each collection's declared royalty separately, §14.3), and one miner-fee provision for the whole transaction — N legs, one settlement's worth of overhead.
3. **Failure isolation — one bad order must not poison the bundle.** Every constituent order is revalidated at assembly time (§14.7): note unspent against current chain state, expiry margin satisfied, proofs and signatures verified at admission and group digest unchanged since. Any order failing revalidation is **ejected before signing**: the bundle is rebuilt from the surviving orders, the corresponding taker order is returned unused to the buyer's wallet (it names only the ejected item's asset base — it cannot be repurposed against anything else, so dropping it is safe), and fees are recomputed for the reduced set. The buyer's stated sweep policy — "all-or-nothing" versus "fill what survives" — decides whether a reduced bundle proceeds or the sweep aborts entirely. What is *never* acceptable is discovering the bad order on-chain: a bundle that reaches broadcast must already be internally valid, so the blast radius of a hard-cancelled or expired constituent is a rebuild, not a failed settlement or a burned fee.

The same machinery generalizes: nothing restricts a multi-group bundle to one buyer. The matcher may batch unrelated matched pairs into shared settlements for confirmation efficiency

— with the caveat that batching couples unrelated trades’ fates at the mempool margin, so ZecBit batches conservatively and never across expiry-margin boundaries (§14.5).

15.4 Private Sales

A private sale — “this listing may only be bought by *that* counterparty” — requires no new cryptography, only matcher policy: the maker order is standard (§14.1; recall the counterparty appears nowhere in the order format itself), and the seller attaches a directed-routing instruction naming the intended counterparty’s ZecBit account. The matcher then refuses to match the order against any taker order but one submitted by that account, and does not display the listing publicly.

The privacy accounting deserves the same honesty applied everywhere else in this paper. **On chain:** nothing — the settlement is an ordinary shielded swap bundle, indistinguishable from any public-book trade; no third party learns that a sale occurred, was private, or involved these parties (who, per §14.2, are not even visible to the matcher *as addresses*). **At the matcher:** the routing instruction is precisely a statement that two ZecBit accounts intend to trade with each other, so the matcher learns an account-level relationship it would not otherwise know. Directed routing is thus a *venue-level* confidence, enforced by policy rather than cryptography — a malicious matcher could ignore the restriction and match the order to anyone *at the same signed terms* (price safety is unconditional; §14.2), and the account linkage lives in matcher logs under Chapter 16’s data-handling regime. Sellers for whom the account linkage itself is sensitive are advised of the alternative that needs no venue at all under ZIP 228: exchange the signed orders bilaterally out-of-band and have either party assemble — at the cost of doing their own coordination, which is exactly the trade-off §13.1’s spectrum predicts.

15.5 Offers on Shielded Funds: Funded-Order Attestation

Transparent-chain PSBT venues fought a chronic plague: **unfunded bids**. A bid is cheap to sign; verifying it is *backed* required the venue to check that the referenced UTXO existed, was confirmed, was unspent, and remained so — polling chain state per bid, forever, and still losing the race when a bidder spent the UTXO after verification. Bid-spam and its “confirmed UTXO checker” infrastructure were the tax the model paid.

On ZecBit the problem dissolves, for a reason that is elegant rather than incidental: **a swap order is itself a validity proof of its funding**. The taker order behind an item offer (or behind a collection intent’s funding attestation, §15.2) contains the bidder’s actual spend action — a zero-knowledge proof that a note of the stated ZEC value exists in the note-commitment tree at the order’s anchor and that the bidder holds its spend authority, plus spend-auth signatures over the group. The matcher’s admission validation (§14.7) verifies this proof as a matter of course. There is no separate “proof of funds” step, no balance disclosure, no doxxing of the bidder’s wider holdings: the order proves *exactly* the note it spends and nothing else — the bidder’s balance, other notes, and address remain shielded even from the matcher. An accepted bid on ZecBit is a funded bid, by construction.

The one residual is the same one hard-cancel creates everywhere (§14.4): the bidder can spend the note *after* admission, killing the order. The proof pins funding at the anchor, not eternally. This is precisely the assembly-time-revalidation and reliability-scoring case of §14.7 — the venue detects the spent note the moment it tries to assemble, at zero on-chain cost, and prices repeat offenders out of the book. Compared to the transparent-chain situation, the shielded venue is strictly better off: verification is a one-time proof check plus a cheap assembly-time recheck, rather than continuous chain surveillance of every resting bid.

15.6 What ZecBit Deliberately Does Not Build

A feature list is also defined by its refusals, and each refusal below is a conclusion of this paper's own analysis, not a roadmap gap.

English auctions. An ascending auction requires bids that *bind* — a winning bidder who can walk away converts the auction into a free-option festival and invites shill-bidding without cost. Binding bids require either locked funds (escrow or a contract holding the bid) or a penalty mechanism; Zcash has no contracts (§13.1), ZIP 228 offers no bonds (§14.7), and a signed order is hard-cancellable by its own maker at will (§14.4) — so a "bid" in this model is structurally non-binding. Transparent-chain PSBT venues reached the same conclusion empirically and fell back to custodial escrow specifically for auctions; ZecBit will not reintroduce custody into Phase M for one feature. The decisive precedent is demand-side anyway: the largest transparent-chain venue, with contracts fully available, removed English auctions for low usage. ZecBit's expiring fixed-price listings plus item/collection offers cover the discovery function; timed price-descent (a maker replacing their order at intervals with lower-priced ones — operationally a Dutch-style schedule built from ordinary re-listing, §14.5 churn costs acknowledged) is the closest honest relative and needs no new trust.

AMM pools for NFTs. A pool is a contract: passive liquidity requires an autonomous party that holds inventory and quotes continuously. Without contracts, "pooled" liquidity is a custodian with a pricing formula — Phase T trust reintroduced permanently, for a mechanism whose fit to unique assets is dubious even where it is possible. Standing collection offers (§15.2) are the non-custodial expression of floor-level passive demand, and they suffice.

NFT-collateralized lending. Loans require enforceable liquidation — seizing collateral on default — which is escrow or contracts by definition. Out of scope for ZecBit as a venue; nothing prevents third parties from building custodial credit atop ZecBit-traded assets, but it will not be this venue and this paper will not dress custody as protocol.

15.7 Activity, Notifications, and Watchlists in a Shielded World

On transparent chains, "activity" is a public feed anyone can index. Here, §14.6 established that no such commons exists — so every feed ZecBit offers must be derived from some *particular* party's legitimate view, and the honest design question is: which event streams exist, for whom, from what source. The answer, by actor:

Stream	Recipient	Source	Privacy boundary
--------	-----------	--------	------------------

Own-order lifecycle (accepted, matched, settling, settled, expired, cancel-confirmed)	The order's creator	Matcher state + the recipient's own wallet detecting the settlement notes (Ch. 11) — two independent witnesses	Delivered only over the account's authenticated channel (Ch. 17); contains full detail because it is the party's own trade
Counterparty-facing events (your listing sold; your intent was accepted)	The other side of the same trade	Same trade record, filtered	Reveals the trade's terms — which the counterparty necessarily learns — and <i>no account identity</i> of the other side, which the matcher does not have (§14.2) and therefore cannot leak
Public listing feed (new listings, delistings, price changes)	Anyone	Orderbook metadata — never signed orders (§14.4)	Terms of <i>offered</i> trades are public by the maker's choice to list; maker identity is a ZecBit pseudonym at most
Aggregate market activity (floor moves, epoch volume)	Anyone	§14.6 signed epoch reports	Aggregates only; per-trade data exists publicly only as salted commitments
Watchlists (alert me on: collection floor change, new listing matching a trait, a specific item listed)	The subscribing account	Server-side evaluation against the public listing feed and aggregate stream	A watchlist is itself sensitive metadata — it declares interest, which fronts intent; watchlists are stored under the same access regime as the orderbook (Ch. 16) and are never used as matching or ranking input
Own-wallet events (incoming assets, confirmations)	The wallet owner	The user's own wallet scanning (Ch. 11) — no server involved	Fully private; ZecBit servers are not in this loop at all

The structural point, one last time in this chapter's terms: a shielded marketplace cannot offer, and should not counterfeit, the panopticon feed of a transparent chain — *"every trade, live, for everyone"* does not exist here, by design of the underlying chain and to the direct benefit of its users. What it offers instead is exact and sufficient: complete information about your own activity, corroborated independently by your own wallet; the public terms of everything currently for sale; and verifiable aggregates about everything already sold. The enforcement of these boundaries — channel authentication, matcher data handling, and the adversarial analysis of each stream — is Chapter 16's subject; the transport that carries them is Chapter 17's.

Part VI — Security, API, and Deployment

16. Security Model and Threat Analysis

This chapter defines what ZecBit protects, whom it protects it against, and where the residual risks lie. The analysis is deliberately conservative: wherever a ZecBit component holds custody or acts as a trusted intermediary, that trust is stated plainly, bounded, and paired with a user-side escape hatch wherever the protocol permits one.

16.1 Assets Under Protection and Security Goals

The system protects five distinct classes of assets, each with its own integrity or confidentiality goal:

#	Protected asset	Security goal	Primary guarantor
G1	Catalog integrity	The public asset catalog reflects exactly the issuance and burn events recorded on-chain; no fabricated assets, no suppressed assets, deterministic reproducibility from raw blocks	Indexer determinism + anyone-can-rebuild property
G2	Custody of user assets	Phase A (hosted): assets held by ZecBit are recoverable and withdrawable by their owner and by no one else. Phase B (Connect): ZecBit never holds keys; custody is entirely user-side	HSM boundary (Phase A); client-side signing (Phase B)
G3	Orderbook confidentiality	Signed swap orders held by the matcher are never disclosed, because a leaked maker signature is a bearer instrument that any party can execute	Matcher isolation, encryption at rest, hard-cancel escape hatch
G4	Metadata authenticity	A resolved manifest is either provably the one the creator committed to (hash-bound, signature-verified) or is rejected; the Registry cannot substitute content	assetDescHash binding + BIP-340 creator signatures + ZMD-1 verification
G5	User privacy	Ownership, transfer history, trading activity, and counterparty identity of shielded assets are not observable by ZecBit or by third parties beyond what the Zcash observer model inherently exposes (issuance and burn are public by protocol design)	Shielded transfers (ZIP 226), self-addressed orders, aggregate-only market data

Two structural principles govern the whole model:

- 1. Verifiability over trust.** Every claim ZecBit's services make — catalog contents, metadata resolution, market statistics — is either derivable from public chain data or accompanied by a cryptographic proof (signature, hash preimage, inclusion proof) that a client can check locally. Trust in ZecBit servers is an availability convenience, not a correctness requirement.
- 2. Honest custody labeling.** Where custody exists (Phase A wallets, Phase T escrow), it is not disguised. The security analysis for those phases assumes ZecBit is a fully trusted — and therefore fully attackable — party, and scopes deployment accordingly (testnet only for escrow; see §18.2).

16.2 Adversary Model

We consider seven adversary classes. Each is characterized by capability, not motive.

A1 — External network attacker. Controls no ZecBit infrastructure. Capabilities: submitting arbitrary API requests, running clients and wallets, mounting DoS, attempting web-layer exploits (XSS, CSRF, injection), phishing users, and operating malicious dApps that speak the ZecBit Connect bridge protocol.

A2 — Malicious creator. A legitimate user of Mint Studio and the Registry who issues assets designed to deceive: copymints of established collections, name-squats, manifests containing malicious media or markup, or assets whose issuance keys are engineered to collide with reserved namespaces. A2 holds valid issuance keys and can produce valid BIP-340 signatures over anything they issue.

A3 — Malicious user/trader. Holds funds and assets, participates in trading. Capabilities: submitting malformed or adversarial orders, attempting double-spend races between hard-cancel and matching, probing the matcher for order-existence side channels, and exploiting escrow reconciliation gaps in Phase T.

A4 — Compromised ZecBit component. The strongest internal adversary class: full control of exactly one component. Analyzed per component in §16.3, but the headline blast radii are:

Compromised component	Worst-case capability	Cannot do
Indexer	Serve a false or incomplete catalog	Forge chain data for anyone who rebuilds or cross-checks against a node
Registry	Withhold or mislabel metadata; revoke or misassign badges	Forge a preimage for an <code>assetDescHash</code> it does not know; forge creator signatures
Studio backend	Corrupt in-flight drafts; leak pre-reveal drop contents; inject manifest content before creator signing	Alter anything after the creator has signed and the issuance is on-chain
Wallet service (Phase A)	Spend all hosted assets (bounded by HSM policy and per-account limits)	Touch Phase B users, whose keys never exist server-side
Matcher	Leak or execute stored signed orders within their signed terms; manipulate matching policy; suppress trades	Alter signed order terms; extract action-group subsets; learn user addresses; execute expired or hard-cancelled orders

A5 — Network adversary. On-path attacker capable of interception, delay, and reordering of traffic between users, ZecBit services, and Zcash nodes. Cannot break TLS or forge signatures.

A6 — Chain-level adversary (testnet). The dedicated ZSA testnet is low-difficulty by construction. A6 can plausibly mount deep reorgs and 51% attacks cheaply. Consequence: **testnet chain data is treated as weakly final**. All testnet services apply elevated confirmation-depth policies (§16.3, "Reorg handling") and no economically meaningful value is placed at risk (custodial escrow is testnet-only precisely because the underlying settlement layer there is itself weak).

A7 — Chain-level adversary (mainnet). On Zcash mainnet, deep reorgs are assumed economically prohibitive but not impossible. Services treat mainnet finality probabilistically: a configurable confirmation depth gates "UI-final" status, and the indexer is reorg-aware to arbitrary depth (deterministic rebuild guarantees recovery even from reorgs deeper than the live-tracking window).

Out of scope: compromise of the Zcash protocol itself beyond the assumptions inventoried in §16.4; compromise of a user's own device (mitigated only partially, via signing-prompt design and hardware-wallet support in the bridge protocol); and legal/regulatory compulsion, which is addressed operationally by the transparency and disclosure policies of §16.5.

16.3 Component-by-Component Threat Matrix

Notation: **L-factors** lists the conditions that make the threat more or less likely; **Residual** states what remains after mitigation. Requirements marked **normative** use RFC 2119 keywords.

16.3.1 REGISTRY

Threat	Impact	Likelihood factors	Mitigation	Residual risk
Registry poisoning — attacker submits a false preimage or manifest for an existing <code>assetDescHash</code>	Users see wrong metadata for a real asset	Low: submission requires the preimage that hashes to the on-chain value	Preimage MUST hash (BLAKE2b) to the on-chain <code>assetDescHash</code> ; manifest MUST carry a valid BIP-340 signature under the issuance key; both checks are re-runnable by any client	Effectively none for content substitution; hash binding is the root of trust
Withholding / mislabeling by a compromised Registry (A4)	Censorship of specific collections; wrong verification badges	Medium: Registry is a single operational service	Full public dumps published on a fixed schedule; independent mirrors re-serve dumps; clients MAY resolve against any mirror; badge decisions are published with reasons and are appealable	A freshly submitted manifest can be suppressed until the next dump cycle; badge system remains a curated (trusted) layer by design
Name-squatting — A2 registers display names/tickers imitating known projects	User confusion, purchase of wrong asset	High: names are not consensus objects; anyone can claim any string	Display names are explicitly non-unique and non-authoritative; the only canonical identifier is the asset ID / issuance key; UI MUST render the verification badge state and the petname adjacent to any display name; search ranking demotes unverified look-alikes (confusable-script and edit-distance screening)	Sophisticated squats on unverified assets remain possible; mitigated socially via badges, not cryptographically

Copyminting — A2 re-issues media of an existing collection under a new issuance key	Buyers acquire counterfeits	High: media is copyable by definition	Registry copymint screening: perceptual-hash comparison of submitted media against the verified corpus; flagged matches are quarantined from search and marked in resolve responses; original-creator claim process	Screening is heuristic; novel transformations evade perceptual hashing. Copyminting cannot be prevented at consensus level — only labeled
Registry signing-key compromise	Attacker signs false dumps/badge attestations	Low: key in HSM, offline ceremony (\$16.5)	Dump signatures rotate via published key-transparency log; clients pin the current key and verify continuity	Window between compromise and rotation; bounded by monitoring

16.3.2 MINT STUDIO

Threat	Impact	Likelihood factors	Mitigation	Residual risk
Malicious media upload (A2) — crafted image/video exploiting decoder bugs in viewers' browsers	Client-side compromise of anyone viewing the asset	Medium: decoder CVEs recur	All media is transcoded server-side into a fixed allowlist of output formats; originals are never served; SVG is rasterized or rejected; media served from a sandboxed, cookie-less origin with Content-Security-Policy: sandbox and correct Content-Type (never sniffed)	Zero-days in the transcoder itself; contained by running transcoding in an isolated, unprivileged worker
Manifest injection / XSS via metadata — script or markup smuggled into ZMD-1 fields	Stored XSS in every frontend that renders the field	High: classic NFT-ecosystem failure mode	Normative sanitization rules: (1) All ZMD-1 string fields MUST be treated as plain text; renderers MUST NOT interpret HTML, Markdown, or URIs embedded in name/description fields. (2) Fields carrying URIs MUST be schema-restricted to declared URI fields, validated against an allowlist of schemes (https , ipfs), and rendered only as inert links. (3) Frontends MUST context-escape on output regardless of input validation. (4) The Registry MUST reject manifests containing control characters, bidirectional-override codepoints, or invalid UTF-8. (5) ZMD-1 canonicalization (JCS, RFC 8785) is applied before signing, so a sanitized-then-signed manifest cannot be "unsanitized" later without breaking the signature	Renderers outside ZecBit's control may ignore the rules; the normative spec and shipped verification libraries reduce, not eliminate, this

Studio backend compromise (A4) — manifest tampering before creator signature	Creator unknowingly signs attacker-chosen content	Medium	Signing happens client-side (Connect) or on an isolated signer (Phase A); the signing prompt displays the manifest digest and human-readable summary; the commit-reveal pipeline stores only commitments server-side pre-reveal, so a compromised backend cannot leak unrevealed drop contents it never held in plaintext	A creator who does not check the digest at signing time can be deceived; UX mitigations only
Commit-reveal front-running — backend or observer learns drop contents pre-reveal and snipes rare items	Unfair drop outcomes	Medium on public infrastructure	Reveal assignment is derived from a post-commit public randomness beacon (block hash at a pre-committed height), so neither ZecBit nor the creator can steer assignment after commitments are published	Beacon manipulation by A6 on testnet is cheap; on mainnet, miner influence over a single block hash is the standard residual for this construction
Issuance plaintext exposure — issue notes are plaintext at issuance by protocol	Initial recipient linkability	Certain (protocol property)	Mint-layer mitigations documented in Chapter 10 (issuance to creator-controlled address followed by shielded distribution); Studio defaults to the private distribution flow	The issuance event itself is public forever; this is a ZSA design fact, not a ZecBit defect

16.3.3 HOSTED WALLET SERVICE (PHASE A)

Phase A is honestly custodial. The analysis assumes A4 (full service compromise) and measures blast radius.

Threat	Impact	Likelihood factors	Mitigation	Residual risk
Full service compromise	Attacker attempts to drain all hosted accounts	Low-medium: standard web-service attack surface	Master seed exists only inside the HSM; the application server can request signatures, never key material. HSM policy enforces: per-account derivation isolation, per-account velocity limits, and a withdrawal policy requiring queued withdrawals above a threshold to pass a delay window with out-of-band user notification and operator co-approval	An attacker with persistent app-server control can drain up to the per-account velocity limits until detected; the HSM bounds rate, not intent
Insider abuse	Operator-level theft	Low	Dual-control on HSM administrative operations; append-only signed audit log of every signature request, reconciled daily against on-chain state (\$16.5 invariants)	Collusion of the full operator quorum

Account takeover (A1 phishing a user)	Single-account loss	Medium	Standard session hardening; withdrawal-address allowlisting with delayed changes; notification on every withdrawal	Users who approve attacker withdrawals; unavoidable in any custodial design
HSM policy misconfiguration	Limits silently ineffective	Low	Policy is version-controlled, reviewed, and covered by the pre-mainnet audit (§16.6); synthetic canary accounts continuously test that limits actually reject over-limit requests	Novel policy-bypass bugs in HSM firmware

Deployment scope: Phase A runs on testnet only. The custodial risk profile above is one of the two reasons (with A6) that hosted accounts do not carry mainnet value.

16.3.4 ZECBIT CONNECT (PHASE B)

Threat	Impact	Likelihood factors	Mitigation	Residual risk
WASM supply-chain compromise — malicious SDK build served to clients	Silent key theft at scale	Low frequency, catastrophic impact	Reproducible builds: the WASM artifact MUST be bit-for-bit reproducible from the tagged open-source revision; releases are signed and their hashes published in the Registry's transparency log; integrator pages MUST load the SDK with Subresource Integrity (SRI); third parties are encouraged to run independent build verification	An integrator that omits SRI and pulls from a compromised CDN reintroduces the risk on their origin
Origin isolation failure	A malicious page reads key material from the wallet context	Low: browser-enforced boundary	Keys live only in the wallet origin (extension or wallet-operated origin); the dApp origin receives only bridge messages; all <code>postMessage</code> traffic is target-origin-pinned and schema-validated; no key material, seed, or proving witness ever crosses the bridge	Browser same-origin-policy bugs
Confused-deputy via bridge protocol — malicious dApp (A1) induces the wallet to sign something the user did not intend, or replays a session belonging to another dApp	Unauthorized signatures	Medium: JSON-RPC bridges are a known attack surface	Every bridge session is bound to the requesting origin at <code>connect</code> time; each <code>sign_*</code> request carries the session ID and is checked against that origin; requests are typed (a <code>sign_order</code> can never be smuggled through a <code>sign_transfer</code> prompt); nonce-per-request prevents replay; capabilities are granted per-method, not wholesale	A user who approves a correctly-described but ill-advised request; the protocol authenticates provenance and content, not wisdom

Signing-prompt spoofing / phishing	User signs a transfer or order to an attacker while believing otherwise	High: primary real-world loss vector in comparable ecosystems	Registry-petname display rules (normative): the wallet prompt MUST render assets by (a) the user's locally assigned petname if one exists, else (b) the Registry-verified name with badge state, else (c) the raw truncated asset ID with an explicit "unverified" marker. The prompt MUST NOT render creator-supplied display names as the primary identifier for unverified assets. Order prompts MUST show both legs (give/receive), fee and royalty action groups, and expiry height. Prompts render in wallet-controlled UI (extension popup or wallet origin), never in dApp-controlled DOM	Users habituated to approve; look-alike petnames the user assigned themselves
Malicious dApp flooding the bridge	Prompt fatigue, denial of service on wallet UI	Medium	Per-origin rate limiting inside the SDK; automatic session suspension after repeated rejections	None significant
Compromised wallet-bridge relay (WebSocket transport)	Observation/tampering of bridge traffic when the postMessage path is unavailable	Low	Bridge payloads are end-to-end authenticated between SDK and wallet (session-key MAC established at pairing); the relay carries opaque frames; tampering fails MAC verification	Traffic-analysis metadata at the relay (timing, sizes)

16.3.5 MATCHER (PHASE M)

The matcher's defining property: **stored maker signatures are bearer instruments**. Anyone holding a fully signed order can assemble and broadcast an execution within the order's signed terms. Orderbook secrecy is therefore a security property, not merely a privacy preference — this is threat class one below.

Threat	Impact	Likelihood factors	Mitigation	Residual risk
--------	--------	--------------------	------------	---------------

Signature leak → ghost-listing execution — exfiltrated order set executed by attacker, possibly after the user believed a soft-cancel removed it	Users' standing orders fill against their current intent (though only ever within signed terms)	Medium: the matcher is the highest-value single target in the system	Containment stack: orders encrypted at rest with keys held in the matcher HSM; decryption only inside the matching enclave process; every access to plaintext order material is HSM-signed into an append-only audit log; strict no-inbound network posture (\$18.1); operational anomaly detection on decryption volume. User-side escape hatch: <i>hard cancellation</i> — the user self-spends the note funding the order, rendering every copy of the signature unexecutable at consensus level; wallets surface hard-cancel prominently and recommend it whenever an order's continued validity matters. Expiry heights bound the lifetime of any leaked signature unconditionally	Between leak and hard-cancel/expiry, execution within signed terms is possible; the protocol guarantees the user never receives worse than the terms they signed — the loss is of <i>optionality</i> , not of assets beyond agreed terms
Term alteration by matcher (A4)	Matcher tries to fill at worse- than-signed terms	Not possible	Spend-authorization signature covers the action-group digest; any modification invalidates the signature; the aggregated binding signature prevents extracting an action-group subset into a different transaction	None (cryptographic)
Matching- policy manipulation — matcher preferences its own or affiliated orders, reorders queue	Unfair fills; value extraction analogous to front-running	Medium: policy is off- chain and discretionary	Published, versioned matching- policy specification; the settlement log with salted per- trade commitments and signed aggregates allows after-the- fact statistical auditing of policy conformance; per-user settlement receipts include inclusion proofs against the published aggregates	Real-time detection is impossible from outside; auditing is retrospective. The matcher remains trusted for <i>ordering</i> , never for <i>terms</i>
Order-existence side channels (A3 probing)	Deanonymizing whether a given asset has standing orders	Medium	Public market data is aggregate-only with minimum- bucket thresholds; order-status endpoints require the submitter's session; timing- uniform responses on submission	Aggregate data itself leaks coarse existence by design (a market with volume visibly has orders)

Data-availability failure — matcher loses order store or settlement log	Standing orders vanish; market history gap	Low-medium	Orders are recoverable by resubmission (users hold their own signed orders via wallet state); the settlement log is periodically checkpointed via signed digests published to the Registry mirror set; users hold their own receipts with inclusion proofs, so history is reconstructible adversarially	Un-checkpointed tail of the settlement log at failure time; standing orders lapse until resubmitted
Matcher key compromise (TLS / aggregate-signing keys)	Impersonation; forged market statistics	Low	Keys in HSM; aggregate-signing key in the same transparency log as Registry keys; receipt verification is client-side, so forged aggregates conflict with user-held inclusion proofs and are detectable	Detection is distributed and delayed rather than immediate

16.3.6 ESCROW ENGINE (PHASE T)

Stated plainly: **Phase T escrow is full custody**. Assets deposited for trading are controlled by ZecBit. Every risk in §16.3.3 applies, plus:

Threat	Impact	Mitigation	Residual risk
Escrow ledger divergence from chain holdings	Users shown balances the service cannot honor	Continuous reconciliation: sum of internal ledger MUST equal on-chain escrow holdings per asset (a §16.5 monitored invariant); trading halts automatically on divergence	Window between divergence and halt
Insolvent or fractional operation	Classic custodial failure	Proof-of-deposit disclosures: periodic publication of escrow addresses' asset commitments alongside the aggregate ledger, allowing public cross-checking on the transparent issuance/burn record and via published viewing disclosures for shielded holdings	Disclosure is periodic, not real-time; this is why escrow never reaches mainnet (§18.2)
Settlement of trades atop weak testnet finality (A6)	Reorged trades leave ledger inconsistent with chain	Elevated confirmation depth before escrow credits; reorg-aware unwinding mirroring the indexer's rollback logic	Deep reorgs beyond the tracking window force a rebuild-and-reconcile pause

16.3.7 INDEXER

Threat	Impact	Mitigation	Residual risk
Compromised indexer serves false catalog (A4)	Fabricated or hidden assets in UI and API	Deterministic rebuild: anyone with a node can reproduce the exact catalog state; published state-digest per block height; frontends can be pointed at independent indexer deployments	Users who never cross-check; mitigated by making cross-checking cheap (open-source indexer, digest endpoint)

Reorg mishandling	Phantom assets from orphaned issuances	Reorg-aware design: rollback to fork point and replay; catalog rows carry confirmation depth; settlement-finality policy : UI marks an event <i>final</i> only at a configurable confirmation depth, set substantially higher on the low-difficulty testnet than the intended mainnet value; non-final events are visibly provisional	Reorgs deeper than the retained undo window trigger a full deterministic rebuild — correctness preserved, availability briefly degraded
Malformed-block parser exploits	Indexer crash or memory corruption	Rust implementation; parser fuzzing in CI; blocks sourced only from the operator's own Zebra fleet, not peers	Bugs in Zebra's own validation upstream

16.3.8 DENIAL OF SERVICE (CROSS-CUTTING)

Layer	Vector	Mitigation
API edge	Volumetric floods, cache-busting query storms	Edge caching (immutable-by-hash objects cache indefinitely, §18.3), per-IP and per-key rate limits, request-cost budgeting on search
Registry	Dump-download amplification	Dumps served via mirrors and content-addressed storage; range requests
Studio	Media-upload resource exhaustion	Size caps, transcoder queue with per-account quotas
Matcher	Order spam (valid-but-worthless signed orders)	Per-account standing-order caps; proof-of-funding check at submission (the funding note must exist unspent); expiry required on all orders
Wallet bridge	Session-open floods	Origin-scoped rate limits in SDK; relay connection caps
Chain	Testnet spam blocking issuance	ZIP 317 fee mechanics apply; degraded-mode UI communicates chain congestion honestly

16.4 Cryptographic Assumptions Inventory

Assumption	Used for	What breaks if it falls
BLAKE2b collision resistance	<code>assetDescHash</code> binding of metadata to asset; dump integrity; ZMD-1 content addressing	An attacker could present two preimages for one on-chain hash: metadata substitution becomes possible and G4 collapses. Second-preimage resistance alone protects existing assets; full collisions endanger new issuances by a malicious creator preparing twin manifests
Pallas curve DLOG hardness / Halo 2 soundness	All shielded-transfer validity: no forgery of spends, no inflation, balance correctness (ZIPs 226/227)	Forged proofs → counterfeit spends or hidden inflation of any asset, including supply-1 NFTs. Total failure of G1/G2 at the chain layer. Halo 2 requires no trusted setup, removing setup-compromise from the list
Halo 2 zero-knowledge property	Confidentiality of shielded transfer contents	Transfer contents leak: G5 fails, but asset integrity survives

BIP-340 Schnorr unforgeability (secp256k1)	Creator authentication of ZMD-1 manifests and Registry submissions	Forged creator signatures → attacker-authored "official" metadata for any collection; verification badges become meaningless. Chain assets themselves unaffected
RedPallas spend-authorization unforgeability	Only key holders can authorize spends; maker order terms immutable under the matcher	Matcher or anyone could alter or forge order executions and spend user notes; G2 and G3's containment argument both fail
RedPallas binding-signature security	Transaction balance binding; prevention of action-group subset extraction	Matcher could carve profitable subsets out of signed orders or unbalance settlements
TLS (as deployed)	Transport confidentiality/integrity for API, bridge relay, mirrors	A5 reads/modifies traffic; content-layer signatures still catch tampering of verifiable objects, but session data and privacy leak

No component of ZecBit introduces a novel cryptographic construction; the system's soundness reduces entirely to the assumptions above, all of which are inherited from audited upstream primitives.

16.5 Operational Security

Key ceremonies. Three key populations exist, each with a documented, witnessed generation ceremony:

1. *Service master seed (Phase A wallet).* Generated inside the HSM, never exported; quorum-controlled backup as HSM-wrapped shares held by separate officers in separate locations. Restoration requires officer quorum plus a mandatory public status notice.
2. *Matcher keys (TLS + aggregate-signing).* Generated in the matcher HSM. The aggregate-signing public key is registered in the transparency log before first use; rotation is announced ahead of cutover with an overlap window during which both keys verify.
3. *Registry signing key.* Offline ceremony; the key signs dumps and badge attestations. Continuity is anchored by each dump signing the successor key on rotation, giving clients a verifiable key chain rather than trust-on-every-download.

Secrets rotation. All non-HSM secrets (API credentials, database credentials, relay tokens) rotate on a fixed calendar and immediately on any suspicion of exposure; rotation is automated so that emergency rotation is a rehearsed, single-action procedure rather than an improvisation.

Monitoring and alerting invariants. Monitoring is organized around invariants that must always hold, not around thresholds tuned by feel:

Invariant	Check	On violation
Supply conservation	For every asset: indexer-computed supply == node-computed issuance minus burn, at every checkpoint height	Freeze catalog updates, page on-call, trigger deterministic rebuild
Escrow solvency	Internal escrow ledger totals == on-chain escrow holdings, per asset	Halt trading and withdrawals pending reconciliation

Wallet signature accounting	Every HSM signature event corresponds to an authorized user request in the audit log	Freeze signing, incident response
Matcher access accounting	Plaintext order decryptions == matching-engine work items	Treat as potential exfiltration; notify per disclosure policy
Dump/mirror consistency	All mirrors serve byte-identical, correctly signed dumps	Demote divergent mirror, investigate

Incident response and disclosure. A written runbook classifies incidents by which security goal (G1–G5) is affected. Policy commitments: user-affecting incidents are disclosed publicly with a technical post-mortem; any event that may have exposed matcher order signatures triggers immediate direct notification to affected order owners with an explicit instruction to hard-cancel; suspected key compromise triggers rotation first, analysis second. Security researchers have a published reporting channel and a commitment to non-hostile handling of good-faith reports.

16.6 Audit Posture

Before any mainnet deployment, the following undergo independent third-party review, scoped in this order of criticality:

1. **Wallet service and HSM policy** (Phase A architecture, even though Phase A itself remains testnet-bound — its signing components are ancestors of production tooling).
2. **Matcher assembly code** — the path from signed maker orders to broadcast transactions, including fee and royalty action-group insertion, is the highest-consequence code ZecBit operates and receives a dedicated audit plus the §16.3.5 policy-audit hooks.
3. **ZecBit Connect SDK** — the WASM signing path, bridge protocol implementation, and prompt-rendering logic.

Reproducible builds are required for every artifact users execute (Connect SDK) and for the indexer, whose determinism claim is only as good as the ability to verify which binary produced a digest. Open-source policy per component is defined in §18.4; the audit reports themselves are published in full, including unresolved findings with rationale.

17. Public API and Agent Interface

17.1 Design Principles

1. **Public data flows freely.** Everything derivable from public chain data — the catalog, issuance/burn events, registry dumps — is served without authentication and without artificial scarcity. Charging for or gating public-chain data would contradict the verifiability principle.
2. **Privacy-preserving by construction.** The API cannot leak what its backends never learn. There is no endpoint that returns holder lists, transfer histories of shielded assets, or per-order books, because no ZecBit component possesses that data. The API surface is a faithful projection of the observer model, not a policy filter over richer internal data.
3. **Versioning.** All paths are prefixed `/v1/`; breaking changes mint a new prefix, and prior versions receive a published deprecation window.
4. **Pagination.** Cursor-based (`cursor` , `limit`) on all list endpoints; cursors are opaque and stable across reorg-free windows.
5. **Idempotent writes.** Every write endpoint accepts an `Idempotency-Key` header; retries with the same key return the original result. Order submission is naturally idempotent (a signed order is its own identity).

Authentication tiers: **none** (public reads), **session** (holder of a logged-in Phase A session or a bridge-established session token), **signature-bearing** (the request body itself contains a user-produced signature — the server verifies but never creates it).

17.2 REST API Specification

CATALOG SERVICE

```
GET /v1/catalog/assets auth: none
?cursor&limit&issuer&finalized&minted_after_height
-> { "assets": [ { "asset_id", "asset_desc_hash",
    "supply", "finalized", "issuance_height",
    "verification": "verified|unverified|flagged" } ],
    "next_cursor" }
```

```
GET /v1/catalog/assets/{asset_id} auth: none
-> { "asset_id", "asset_desc_hash", "supply", "finalized",
    "issuance_height", "burn_events": [...],
    "confirmations", "final": true|false }
```

```
GET /v1/catalog/collections auth: none
GET /v1/catalog/collections/{collection_id} auth: none
-> { "collection_id", "issuer_key", "items_count",
    "verification", "zmd1_root" }
```

```
GET /v1/catalog/collections/{collection_id}/items auth: none
?cursor&limit&sort
```

```
GET /v1/catalog/search auth: none
?q&type=asset|collection&cursor&limit
-> ranked results; unverified look-alikes demoted (see 16.3.1)
```

```
GET /v1/catalog/events auth: none
?from_height&cursor&limit&kind=issuance|burn|finalize
-> { "events": [ { "kind", "asset_id", "height", "txid",
    "confirmations" } ], "next_cursor" }
```

```
GET /v1/catalog/state-digest/{height} auth: none
-> { "height", "digest" } # deterministic-rebuild anchor
```

REGISTRY SERVICE

```
GET /v1/registry/resolve/{asset_desc_hash}          auth: none
-> { "preimage_b64", "zmd1_manifest": { ... },
    "creator_sig", "badge": { "state", "reason_uri" },
    "copymint_flag": null | { "matched_collection" } }
# client MUST verify: BLAKE2b(preimage)==hash,
# BIP-340 sig under issuance key, JCS canonical form

POST /v1/registry/submit                            auth: signature-bearing
body: { "preimage_b64", "zmd1_manifest", "creator_sig" }
-> 201 { "status": "accepted" } | 422 (hash mismatch,
    bad signature, sanitization violation per 16.3.2)

GET /v1/registry/dumps                             auth: none
-> { "dumps": [ { "sequence", "sha256", "signature",
    "mirror_urls": [...] } ] }

GET /v1/registry/verification/{asset_id}           auth: none
-> { "badge", "history": [ { "state", "height",
    "attestation_sig" } ] }
```

MARKET SERVICE

```
GET /v1/market/{asset_id}/book                                auth: none
-> AGGREGATES ONLY: { "bid_depth_buckets": [...],
    "ask_depth_buckets": [...], "bucket_policy_uri" }
    # per-order data is never served – see G3

POST /v1/market/orders                                       auth: signature-bearing
body: { "signed_order_b64" }    # produced by user's wallet
-> 201 { "order_ref" } | 422 (invalid sig, expired,
    funding note unknown/spent)

GET /v1/market/orders/{order_ref}                           auth: session (submitter)
-> { "state": "open|softcancelled|filled|expired|
    hardcancel_observed", "expiry_height" }

DELETE /v1/market/orders/{order_ref}                       auth: session (submitter)
-> soft cancel (delist) only; response body repeats that
    hard cancellation requires a wallet-side self-spend

GET /v1/market/receipts/{receipt_id}                       auth: session
-> { "trade_commitment", "salt", "aggregate_ref",
    "inclusion_proof" }

GET /v1/market/stats/{asset_id}                             auth: none
-> { "aggregates": [ { "window", "volume_bucketed",
    "trade_count_bucketed", "aggregate_sig" } ] }

POST /v1/market/stats/verify                                auth: none
body: { "trade_commitment", "salt", "aggregate_ref",
    "inclusion_proof" }
-> { "valid": true|false }
    # also implemented client-side in the SDKs (17.5)
```

STUDIO SERVICE

```

POST /v1/studio/drafts                                auth: session
GET  /v1/studio/drafts/{draft_id}                     auth: session
PUT  /v1/studio/drafts/{draft_id}/manifest             auth: session
      # server validates against ZMD-1 schema + sanitization rules

POST /v1/studio/media                                auth: session
      multipart upload -> { "media_id", "transcoded_variants",
                          "content_hash_cidv1" }

POST /v1/studio/issuance-jobs                         auth: signature-bearing
      body: { "draft_id", "batch_spec",
              "signed_issuance_b64" } # signed via wallet layer
      -> { "job_id" }

GET  /v1/studio/issuance-jobs/{job_id}                 auth: session
      -> { "state", "txids": [...], "confirmations" }

POST /v1/studio/drops/{drop_id}/commit                auth: signature-bearing
POST /v1/studio/drops/{drop_id}/reveal                auth: signature-bearing

```

WALLET-BRIDGE SESSION ENDPOINTS

```

POST /v1/bridge/sessions                             auth: none
      body: { "origin", "sdk_version", "pairing_pub" }
      -> { "session_id", "relay_ws_url", "server_pub" }

GET  /v1/bridge/sessions/{session_id}                 auth: session
      -> { "state": "pending|paired|revoked", "origin" }

DELETE /v1/bridge/sessions/{session_id}               auth: session

```

The bridge JSON-RPC methods themselves (`get_addresses`, `get_assets`, `sign_issuance`, `sign_transfer`, `sign_order`, disclosure requests) run over the paired channel, not REST, and are specified in the wallet-layer chapter.

17.3 WebSocket Streams

```

WS /v1/stream/catalog          auth: none
  events: { "type": "issuance|burn|finalize|reorg_rollback",
            "asset_id", "height", "confirmations" }
  # reorg_rollback events make provisional state explicit

WS /v1/stream/orders          auth: session
  scoped to the authenticated account's own submitted orders:
  { "type": "state_change", "order_ref", "state" }

WS /v1/stream/settlements      auth: none
  aggregate ticker only: signed windowed aggregates as in
  /v1/market/stats – never per-trade, never per-order

```

Streams carry monotonically increasing sequence numbers; clients resume with `?since_seq=` and MUST tolerate rollback events rather than assume append-only history.

17.4 MCP Server for AI Agents

Rationale. A growing share of collecting and trading activity is agent-mediated. Rather than have agents scrape the frontend, ZecBit exposes a Model Context Protocol server whose tools map one-to-one onto the public REST surface — with the same observer-model constraints, so an agent can do nothing a human API client could not.

Tool	Maps to	Auth
<code>search_assets(query, filters)</code>	GET <code>/v1/catalog/search</code>	none
<code>get_collection(collection_id)</code>	GET <code>/v1/catalog/collections/{id}</code>	none
<code>resolve_metadata(asset_desc_hash)</code>	GET <code>/v1/registry/resolve/...</code> + local ZMD-1 verification before returning	none
<code>quote_floor(collection_id)</code>	GET <code>/v1/market/{...}/book</code> aggregates	none
<code>place_order(signed_order_b64)</code>	POST <code>/v1/market/orders</code>	signature-bearing
<code>order_status(order_ref)</code>	GET <code>/v1/market/orders/{...}</code>	session

Safety constraints (normative).

- The MCP server **never holds keys, never signs, and never sees seed material**. `place_order` accepts only a fully signed order produced by the user's own wallet (via Connect); an agent without access to a user-authorized signing flow can read, but cannot trade.
- Read tools are unauthenticated and rate-limited identically to public REST.

- Write tools require the signature-bearing payload plus a session bound to the user who paired the wallet; the MCP layer adds no ambient authority.
- `resolve_metadata` performs client-side verification (hash, signature, canonicalization) and reports verification status in-band, so agents inherit the "verify, don't trust" posture by default rather than as an option.
- Tool responses include the same provisional/final confirmation markers as the REST API, so agents cannot mistake weakly-final testnet state for settled fact.

17.5 Client SDKs

TypeScript and Rust SDKs ship as thin, typed wrappers over the REST and WebSocket surface — no hidden logic, no privileged endpoints. Their essential contribution is the **verification helpers**, which run entirely client-side:

- `zmd1.verify(preimage, manifest, sig, assetDescHash)` — BLAKE2b binding, JCS canonical-form check, BIP-340 signature check, sanitization-rule conformance.
- `stats.verifyInclusion(receipt, aggregate)` — checks a salted trade commitment against a signed aggregate's inclusion proof.
- `catalog.verifyDigest(localRebuildDigest, publishedDigest)` — for operators running independent indexers.
- `dumps.verifyChain(dumpSequence)` — registry dump signatures and key-rotation continuity.

Because every verifiable claim in the system has a corresponding client-side checker in the SDK, trust in ZecBit's servers is strictly optional for correctness; the servers provide availability and convenience only.

18. Deployment, Operations, and Technical Roadmap

18.1 Reference Deployment Topology



```

. +-----+ .
. | Wallet service (Phase A) | .
. | app tier <-> HSM          | .
. | no route to matcher or   | .
. | studio segments         | .
. +-----+ .
= = = = =

```

Stateless vs stateful. The API edge, catalog/registry read tier, and order gateway are stateless and horizontally scalable. Stateful components and their recovery models:

Component	State	Backup/restore model
Indexer	Catalog DB	None needed for correctness: deterministic rebuild from the node fleet's raw blocks reproduces byte-identical state; blue/green pair means one side can rebuild while the other serves
Registry	Preimages, manifests, badge history	Dump-based: the signed public dumps <i>are</i> the backup; a fresh Registry restores from the latest dump plus the post-dump submission journal; mirrors provide adversarial redundancy
Wallet service	Account DB + HSM seed	Account DB is conventionally backed up, but the recovery root is the HSM master seed: all account keys are derived (ZIP 32 hierarchy), so funds are recoverable from seed quorum + on-chain data even after total DB loss
Matcher	Encrypted order store + settlement log	Order store is intentionally <i>not</i> replicated broadly (replication multiplies the G3 attack surface); loss is recoverable by user resubmission. The settlement log is checkpointed via signed digests to the mirror set, and user-held receipts make history externally reconstructible
Studio	Drafts, media	Conventional backups; pre-reveal drop contents exist server-side only as commitments

Blue/green indexer operation doubles as the upgrade path: schema or parser changes deploy to the idle color, which rebuilds from genesis (or from a verified digest checkpoint), is diffed against the live color's digests, and then takes traffic.

18.2 Environments

Present: dedicated ZSA testnet. All six components run against the dedicated public ZSA testnet. Consequences of this environment were analyzed in §16.2 (A6): finality is weak, so confirmation-depth policies are elevated and no mainnet-grade value exists anywhere in the system.

Mainnet activation plan. ZSA reaches Zcash mainnet at a future network upgrade. At activation:

- The **indexer** begins at the ZSA-activation height — there is no pre-history to index, so mainnet catalog state is small, complete, and verifiable from day one.
- The **Registry carries over** as a single logical service spanning both networks, with every record flagged by network of origin. Testnet-era records remain resolvable but are visually

and programmatically distinguished so testnet provenance can never be passed off as mainnet provenance.

- **Wallet Phase B (ZecBit Connect) is a hard prerequisite for mainnet trading.** Hosted Phase A accounts are not offered on mainnet: mainnet users hold their own keys, full stop.
- **Escrow is not deployed on mainnet.** The Phase T engine is a testnet bridge technology. Mainnet trading launches only in matcher form (Phase M), once ZIP 228 swap functionality is active on mainnet.

The re-genesis question. Collections minted on the testnet do not migrate — testnet assets are testnet assets, and no bridge is offered or implied. Creators who built collections during the testnet era **re-issue on mainnet** under their issuance keys. The Registry supports this with **lineage mapping**: a creator signs (BIP-340, under the issuance key common to both issuances or under a signed key-linkage statement) a lineage record connecting the testnet collection to its mainnet successor. Lineage is displayed as provenance context; it confers history, not value equivalence.

18.3 Performance Engineering

Honest statements about where the system's performance envelope actually lies:

- **Client-side proving dominates user-perceived latency.** Halo 2 proof generation in the Connect WASM environment is seconds-scale per action. This is the binding constraint on interaction design, and the mitigation is architectural, not micro-optimization: batch guidance (Mint Studio aggregates a batch drop into as few proving sessions as the action structure allows; wallets prove while the user reviews the prompt), progress surfacing, and background proving where the signing decision is already made.
- **Indexer throughput is a non-problem.** Parsing issuance and burn events from blocks is computationally trivial relative to the chain's block rate; the indexer's engineering effort goes into reorg correctness and rebuild determinism, not speed. Full deterministic rebuilds are bounded by node block-serving throughput, and the blue/green pair removes rebuilds from the serving path.
- **Caching leans on content addressing.** Objects addressed by hash — media variants (CIDv1), registry preimages, dump files, ZMD-1 manifests — are immutable by construction and cached indefinitely at the edge (`Cache-Control: immutable`). Mutable views (catalog lists, book aggregates, stats) are invalidated by catalog and settlement events rather than by TTL guesswork, so the cache is as fresh as the event stream.
- **Orderbook latency is human-scale, stated honestly.** The matcher serves a marketplace for unique assets, not a high-frequency venue. Settlement requires on-chain confirmation; matching cadence is bounded below by proving and chain latency regardless of matcher engineering. The design therefore optimizes for correctness, confidentiality, and auditability of matching — not microseconds — and the published matching-policy document says so explicitly to preclude latency-arms-race expectations.

18.4 Open-Source and Reproducibility Commitments

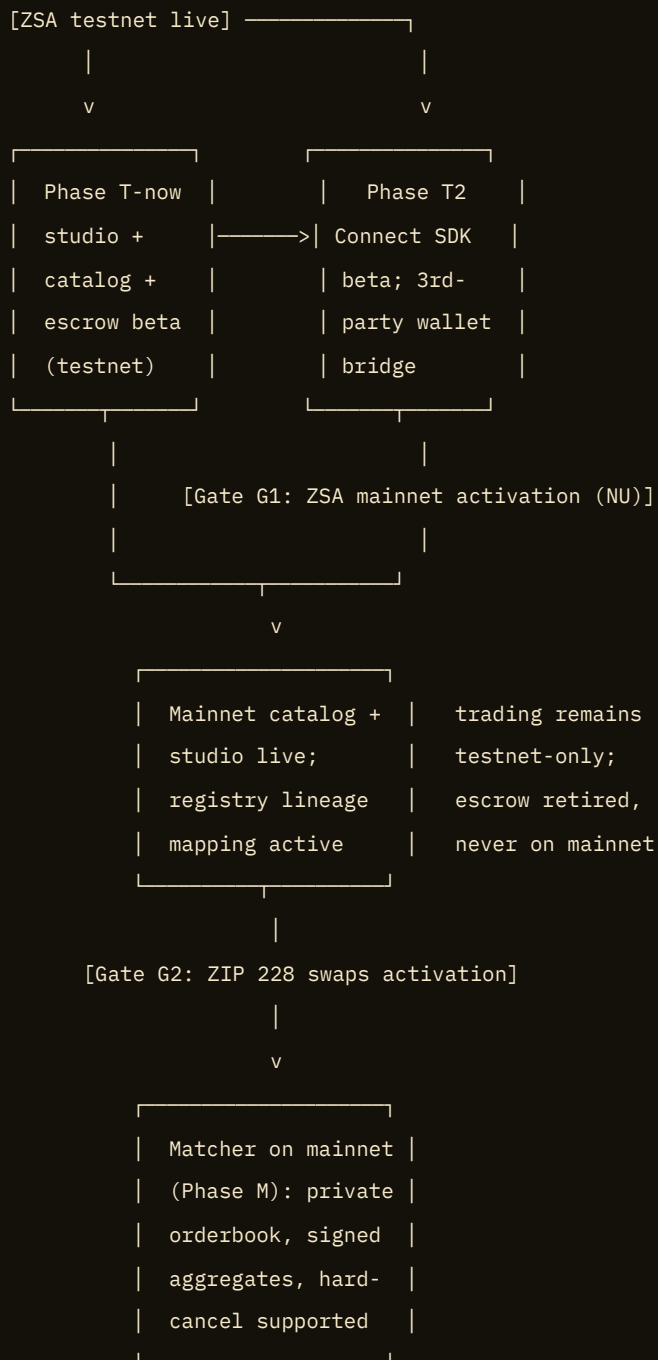
Component	Open from day one	Opens at mainnet	Rationale
Indexer	Yes (source + reproducible builds)	—	The deterministic-rebuild guarantee is only meaningful if anyone can run and verify the code
ZMD-1 tooling (schema, canonicalizer, validators)	Yes	—	A metadata standard that cannot be independently implemented is not a standard
Connect SDK (WASM + bridge)	Yes (reproducible, SRI-anchored)	—	Users execute this code; §16.3.4 requires public reproducibility
Verification libraries (SDK helpers, §17.5)	Yes	—	“Trust is optional” requires the checkers to be inspectable
Registry service	Dump formats, verification chain, and API spec open from day one; service code opens progressively	Remainder	Mirrors need formats immediately; service internals follow hardening
Matcher assembly core	Interface + matching-policy spec open	Core opens at mainnet launch	Pre-mainnet, the code evolves rapidly against a draft ZIP (228) and premature publication of half-hardened order-handling code enlarges the G3 attack surface; at mainnet, when real value depends on it, the assembly core (order → action-group → transaction path) is published so the “cannot alter signed terms” argument is publicly checkable
Studio backend	Pipeline specs (commit-reveal, sanitization rules) open	Service code follows registry timeline	Normative rules must be public even where service code lags

Every published artifact that users execute or verify against is reproducibly buildable, and release hashes are anchored in the Registry transparency log.

18.5 Technical Roadmap: Protocol-Gated Phases

The roadmap is gated on protocol milestones, not dates. ZecBit does not control Zcash network-upgrade scheduling and does not pretend to.

DEPENDENCY GRAPH



Continuous, parallel to all phases:

- upstream ZIP 228 review feedback from matcher implementation experience
- wallet-ecosystem ZSA support contributions
- ZMD-1 stewardship (18.6)

Phase T-now (current). Studio, catalog, registry, hosted wallets, and the escrow trading beta operate on the dedicated ZSA testnet. Purpose: validate the full issuance-to-trade loop, exercise

reorg handling under genuinely weak finality, and gather creator feedback on ZMD-1.

Phase T2. ZecBit Connect SDK enters beta; the wallet bridge protocol is published and at least one third-party wallet integration is pursued, so that “user-held keys” does not mean “ZecBit-authored wallet only.”

Gate G1 — ZSA mainnet activation. Catalog and Studio go mainnet-live at the activation height. Trading remains on testnet: G1 delivers issuance and provenance on mainnet, not markets.

Gate G2 — swap (ZIP 228) activation. The matcher deploys against mainnet. Escrow does not make this transition; it is retired rather than promoted.

Continuous work. Implementation experience with order assembly, expiry semantics, and binding-signature aggregation is fed back into the ZIP 228 review process as the draft matures; ZSA support patches and test vectors are contributed to ecosystem wallets, since ZecBit’s Phase B model succeeds only in an ecosystem where multiple independent wallets speak ZSA.

18.6 Standards Stewardship

ZMD-1 governance. The metadata standard lives in a public specification repository, independent of ZecBit’s service code. Governance commitments:

- *Versioning policy.* The spec carries an explicit version; manifests declare the version they conform to. Backward-incompatible changes mint a new major version, and verifiers remain able to validate all prior versions indefinitely — signed manifests are permanent objects and must never be orphaned by spec evolution.
- *Community change process.* Changes proceed through public issues and pull requests with a documented comment period; normative changes (anything affecting the meaning of a signature or the sanitization rules of §16.3.2) additionally require published rationale and test vectors before merge. ZecBit maintains the repository but commits to a maintainer set that grows beyond ZecBit as independent implementations appear.
- *Reference tooling.* The canonicalizer (JCS/RFC 8785 profile), schema validators, and signature test vectors ship with the spec, so conformance is testable, not aspirational.

Path to an ecosystem standard. Once ZSA reaches mainnet and at least one independent implementation of ZMD-1 verification exists, ZecBit commits to proposing ZMD-1 through the Zcash improvement-proposal process as an ecosystem metadata standard for shielded assets — transferring ultimate change control from a company repository to the community process where a consensus-adjacent artifact belongs. The measure of success for ZMD-1 is precisely that ZecBit stops being necessary to it.

Part VII — Assets on Chains That Do Not Know About Them

The chapters to this point have described a system defined against a protocol — Zcash Shielded Assets — in which the asset is a first-class object of consensus. That protocol is not deployed. Parts VII through IX therefore step back and ask a question the earlier chapters were able to assume away: *what is an asset, mechanically, on a chain whose consensus rules have never heard of it?*

This is not an academic exercise. Every non-fungible token in existence on a UTXO chain today — every ordinal inscription, every Rune, every Taproot Asset, every RGB contract, and every one of the 111,378 inscriptions currently recorded against Zcash mainnet — exists in precisely that condition. Understanding exactly what holds them up, and exactly what does not, is the prerequisite for any honest claim about what ZecBit can and cannot deliver before ZSA activates, and for the sharper claim about why ZSA is worth waiting for.

Part VII establishes the taxonomy. Part VIII applies it to Zcash mainnet as the chain actually is in September 2026, including a forensic account of what has already been built there and what happened to it. Part IX returns to the asset layer with that context in hand.

19. Three Architectures for Assets on a UTXO Chain

Systems that issue assets on a chain without native asset support are usually compared by their features — supply models, metadata conventions, wallet support. This is the wrong axis. The question that determines every other property is narrower and more brutal:

What does the base chain actually promise about your asset?

There are exactly three answers. They are not points on a spectrum; they are distinct security models with incompatible failure modes. Every deployed system is one of them.

19.1 Architecture I: Indexer Convention

Counterparty (2014), Runes (2024), BRC-20, Omni Layer, and — as Chapter 20 establishes — every inscription protocol that has run on Zcash.

Asset data is published on-chain in a form the consensus rules never parse: an `OP_RETURN` payload, a fake public key in a bare multisig output, a data envelope in a script. Validity is whatever a set of off-chain programs, replaying identical rules over identical blocks, agree that it is.

The chain contributes exactly two things: total ordering of the data, and the guarantee that a given output can be spent only once. It contributes no opinion whatsoever about what the data means.

The canonical statement of this condition is Runes', and it is admirably direct. From the `ord` project's own specification document:

"Runes Does Not Have a Specification"

"The Runes reference implementation, `ord`, is the normative specification of the Runes protocol."

"If, due to a bug in `ord`, this document diverges from the actual behavior of `ord` and it is impractically disruptive to change `ord`'s behavior, this document will be amended to agree with `ord`'s actual behavior."

This is not carelessness. It is the correct description of where truth lives in an indexer-convention system: in the code that the ecosystem has agreed to run. The documentation is a convenience, and when the two disagree, the documentation is wrong by definition. In November 2025 the `ord` maintainers merged a pull request titled "Fix rune specification" that corrected two *semantic* errors — the pointer bound and the edict output bound — which had stood in the published prose for roughly nineteen months. Anyone who had implemented from the specification rather than from the source would have diverged from the network on edge cases for that entire period.

The failure mode is interpretive fork. The chain is unambiguous; the reading of it is not, and a disagreement between readings is invisible to the chain. Counterparty, which has operated in this condition for twelve years, has accumulated the most elaborate defences against it that exist anywhere: 248 hard-coded mainnet checkpoints compiled into the software, 133 height-gated protocol changes each of which every indexer must implement identically at the same block, three chained per-block consensus hashes, a consensus-hash version counter that has been reset seven times on testnet, minimum-version enforcement that halts an out-of-date node, and an activation oracle served over HTTPS from a single domain. A comment in its own source describes the failure this machinery exists to prevent: a node too old for an activated change *"would keep parsing and fork instead of halting."*

Runes takes the opposite approach and makes divergence impossible by making doubt destructive. Unrecognized even-numbered tags produce a *cenotaph*, and:

"All runes input to a transaction containing a cenotaph are burned."

The reasoning is stated in the project's documentation and is worth understanding because it is genuinely clever:

"Cenotaphs are an upgrade mechanism, allowing runestones to be given new semantics that change how runes are created and transferred, while not misleading unupgraded clients as to the location of those runes, as unupgraded clients will see those runes as having been burned."

Even-numbered tags are those that can change *where tokens end up*; odd-numbered tags are display metadata. An old indexer that cannot parse an even tag cannot compute correct balances, so it is required to conclude that the tokens were destroyed. Every implementation then agrees the tokens are *gone*, rather than disagreeing about *where they went*. Burning is the fork-avoidance mechanism.

The decisive advantage, and it is a large one: recovery from a seed phrase alone. Because state is a deterministic function of public chain data, keys are sufficient. A Counterparty asset issued in 2014 is recoverable in 2026 from twelve words plus any synchronised node — no proof, no backup file, no counterparty cooperation, no archived history. Nothing else is required because nothing else exists.

19.2 Architecture II: Client-Side Validation

RGB, Taproot Assets.

The chain carries a commitment — a tapleaf hash, a deterministic Bitcoin commitment — and nothing more. The asset's existence, its supply, and its ownership are established by a proof chain handed **directly from sender to recipient** and verified locally against the chain's uniqueness guarantee.

The base chain's promise shrinks to a single sentence: *this outpost was spent once, at this time*. Everything else is private cryptography.

The properties this buys are real and should not be understated. Taproot Assets' Merkle-Sum Sparse Merkle Tree makes inflation *provably* impossible rather than merely agreed to be impossible: every branch commits to the sum of its children's values, so a supply violation cannot be constructed, only rejected. Transfers are indistinguishable from ordinary pay-to-taproot spends. There is no indexer to trust, and a hostile proof repository can withhold data but cannot forge it — the specification is explicit that *"An adversarial Universe could only refrain from returning data requested by clients."*

The cost is that possession of the asset becomes an archival obligation, permanently. Both projects state this without euphemism. Taproot Assets:

"In short, the `1nd` seed alone is not enough to recover assets. If `tapd`'s database is lost or corrupted and no backup exists, access to all assets minted or received by that `tapd` **is lost**. Additionally, custody of the BTC used to carry/anchor the assets would also be lost."

The final clause deserves emphasis, because it is the sharpest illustration in this entire taxonomy of what client-side validation actually costs. In Taproot Assets the Bitcoin private key lives in `1nd` while the taproot tweak — which *is* the asset commitment — lives in `tapd`. Reconstructing the output key requires both. Losing the asset-layer database therefore does not merely destroy the asset; it strands the bitcoin underneath it. A data-loss event at the overlay propagates down into the base layer.

RGB is equally direct, and has no fallback at all:

"Since RGB is not a blockchain-based smart contracts, the stash is not replicated to other peer machines and usually keeps the data which nobody else than you know or have access to... Thus, losing stash means losing your assets and contract state, and it has to be backed up with the same effort as a seed phrase or wallet descriptor for a complex wallet setup."

Taproot Assets has partial mitigations — public Universes that may hold proofs, and a seed-encrypted `assets.backup` file — but seed-only recovery remains an open issue, filed in August 2023 and still open. RGB has no equivalent by construction: privacy means no third party holds your history, which means there is nobody to ask.

Two further costs are structural rather than operational. **Proof size grows with the transfer graph, not with your own activity** — every input you consume drags in its entire ancestry, and histories compound when inputs merge. Peter Todd, who originated the concept, put the worst case plainly in 2016: *"It kinda sucks, but it's plausible that I could transfer you a gigabyte of data to give you a coin."* And **there is no arbiter for a protocol schism**: RGB has, as of September 2026, forked into two mutually incompatible lines with separate organisations and documentation sites that label each other unsupported, and no on-chain evidence distinguishes them, because to Bitcoin the commitments are identical. A contentious upgrade on a consensus chain forks the

chain visibly and the market prices both sides; in client-side validation two incompatible protocols can coexist indefinitely with nothing to adjudicate between them.

19.3 Architecture III: Consensus-Level Assets

ERC-721 and ERC-1155 today; Zcash Shielded Assets if it deploys.

Ownership is a consensus state variable. Every validator computes the same answer, or the block is invalid.

The advantages follow immediately and are not available in either other architecture: recovery from a seed alone is automatic because the state exists independently of every user; the recipient of a transfer need verify nothing, because ownership is a state read rather than a verification task; and — uniquely — **on-chain logic can consume ownership synchronously and atomically**. A proof file handed to a human cannot be read by a virtual machine. No amount of engineering closes that gap.

The costs on a transparent chain are equally well known: the ownership graph is public forever, every transfer costs gas, and state growth is a commons expense borne by every full node in perpetuity.

One honest footnote belongs here, because it is routinely elided in comparisons of this kind. **The consensus guarantee usually stops at a pointer.** ERC-1155's metadata section specifies a URI with a substitution token, canonically resolving to an HTTPS endpoint; ERC-721's rationale concedes its metadata *"is returned as a string value. Currently this is only usable as calling from web3, not from other contracts."* An ERC-721 whose `tokenURI` is an HTTP address has a data-availability dependency of precisely the same *kind* as a Taproot Assets Universe — it is merely labelled less candidly. Consensus covers the ownership record. It does not cover the artifact.

19.4 The Discriminator

The three architectures are distinguished most cleanly not by their features but by a single question with a binary answer:

Are the keys sufficient?

	Indexer convention	Client-side validation	Consensus-level
Where state lives	Derived from chain by replay	In the holder's possession	In the chain's state
Who validates	Indexer operators	The recipient	Every full node
Chain's promise	Ordering + publication	Ordering + a commitment	Full validation
What is public	Everything	Commitments only	Everything
Recovery from seed alone	Yes	No	Yes

If the index/proof/stash is lost	Nothing — resync	Assets destroyed	Nothing
Failure mode	Interpretive fork	Data loss; withholding	Chain fork (visible)
Forgery by the trust layer	No	No	No
Composable with on-chain logic	No	No	Yes
Supply integrity	Agreed by indexers	Cryptographic	Enforced by consensus

Seed-only recoverability is not a feature choice. It is a direct consequence of where state lives. If state is a pure function of public chain data, keys suffice — and the price is that the *rules* are either a social convention (Architecture I) or must be enforced by global consensus at global expense (Architecture III). If state is private and client-held, keys are necessary but not sufficient, and the price is that every holder becomes a permanent archivist of their own history.

No deployed architecture delivers privacy, seed-only recovery, and trustless rules simultaneously. Architecture I gives up privacy and accepts rule fragility. Architecture II gives up recoverability. Architecture III gives up privacy and pays for consensus. This is the trilemma against which the remainder of this document should be read, because ZSA’s claim — examined in Chapter 27 — is precisely that consensus-level validation of a *hidden* predicate collapses it.

20. The Inscription Record on Zcash Mainnet

Zcash is not a hypothetical host for inscriptions. It has been one since 12 November 2025, at meaningful scale, continuously, and the record is unusually instructive because it contains a controlled demonstration of exactly the failure the rest of this document is concerned with.

This chapter is descriptive. It reports what was built, in what form, and what happened — including the parts that worked. Chapter 21 draws the structural conclusions.

20.1 What Was Built

Zerdinals is an ordinals-style inscription protocol on Zcash mainnet. It is live, it is large, and its technical details matter because they are not what secondary sources report.

It does not use `OP_RETURN`. The content lives in the `scriptSig` of a pay-to-script-hash reveal transaction, in a commit/reveal pair directly modelled on Bitcoin's inscription envelope. The envelope opens with a three-byte push of the ASCII magic `ord` (`0x6f 0x72 0x64`), followed by a total-pieces count, a content-type string, and then the content in descending-index chunks. The redeem script is a public key, `OP_CHECKSIGVERIFY`, a run of `OP_DROP` opcodes sized to the piece count by the rule $N = 3 + 2 \times \text{pieces}$, and `OP_1`.

The choice of carrier is a direct consequence of Zcash's policy limits, and it is the correct engineering decision. Zcash caps `OP_RETURN` at a relayed script size of 83 bytes — 80 bytes of payload — and permits exactly one such output per transaction. The `scriptSig` path is bounded instead by the 1,650-byte standardness limit on input scripts, which the protocol stays well below by chunking content into 240-byte pieces, at most four per transaction, to a maximum of 255 pieces, or 61,200 bytes total. Zcash has no witness, no SegWit and no Taproot, and therefore no witness discount: the cheap bulk-data path that makes Bitcoin inscriptions economical does not exist here, and the `scriptSig` envelope is the closest available substitute.

The first inscription is txid `219cadf573630d346484043ab12c1199a78730036852870872eeee682f3496d7`, in block **3,132,356**, on 12 November 2025 — a 251-byte version-4 transaction with no shielded components at all, carrying the text *"Lazo is Cooking"*. A ZRC-20 token standard followed within a day.

The scale is not marginal. At block **3,489,303**, on 19 September 2026, the protocol's indexer reports **111,378 inscriptions** and 145 ZRC-20 tokens, with new inscriptions confirming within the hour.

20.2 What It Did to the Chain

The network noticed immediately. Daily transaction counts rose by an order of magnitude in the days following the first inscription. Two independent sources, queried separately, agree on the shape and differ slightly on the magnitude:

Date	Transactions/day (source A)	Transactions/day (source B)
October 2025 mean	6,125	typical 6,000–7,000
12 November 2025	6,996	8,148
13 November 2025	72,732	73,826
15 November 2025	65,767	66,901
16 November 2025	43,368	43,368
Late November 2025	—	4,880–4,994
December 2025 mean	5,303	3,816–9,202

The ~1.5% disagreement between two mainstream data providers on the same public chain is itself a small illustration of this Part’s theme, and is left in rather than reconciled.

Two things about this episode are frequently reported and should be stated more carefully than they usually are.

First, the spike fully reverted. By 29–30 November, daily transaction counts were *below* the October baseline. The common claim that inscriptions established a permanently higher floor is not supported by the series. Whatever this was, it did not persist.

Second, the attribution of the peak day specifically is weaker than the headlines suggest. A separate measurement of `OP_RETURN` (nulldata) outputs per day finds only ~30 such outputs on 13 November — below the ordinary baseline — against 30,450 on 15 November. On that evidence the peak day carried almost no `OP_RETURN` payload at all, and the 15 November wave decodes byte-identically to a *different* protocol using a `0x7a` magic prefix and a four-character ticker, of which 27,404 mints were published in total and roughly 99% were captured by a single receiving address.

That measurement does not, however, refute an inscription explanation for 13 November, and it is important to see why: **Zerdinals does not use `OP_RETURN`**. Its payload lives in a reveal transaction’s input script (§20.1), which no count of nulldata outputs can detect. The two measurements are of different carriers. What can be said with confidence is that 13 November consisted of roughly 31,000 near-identical small transparent transactions with an unusual output profile — an input-to-output ratio of 1.41→3.22 against 2.48→1.71 the day before — which is consistent with a commit/reveal minting wave and also consistent with other high-volume patterns.

Engineers investigating the spike in the Zcash community forum did post a raw mempool decode showing the byte sequence `[111, 114, 100]`, which is `ord`; that establishes inscription traffic was present, not that it accounted for the peak. This document therefore claims only what the data supports: **an inscription protocol went live on 12 November 2025, transaction volume rose roughly tenfold for nine days, at least two distinct data-carrying protocols were active in that window, and the elevation did not survive the month.**

The episode is also smaller than it sounds in one respect worth recording. At the peak, the chain used approximately 48.7 KB of a 2 MB block allowance — about 2.4% utilisation. No congestion occurred. The network's own engineers said so at the time.

The response from the protocol's engineers was neither a fork nor a filter. It was a mempool-policy alignment between the two node implementations on zero-value outputs, and a shrug. From the forum thread, an Electric Coin Company infrastructure engineer: *"Given the evidence I believe it was the high volume from zerdinals.com... Not an attack."* A Zebra engineer: *"It may be silly but it is not an attack."* And later, on the fee mechanism doing the work a policy debate would otherwise have done: *"Zerdinals fought the law and the law won :) It helped standardize our mempool filtering across zcashd and zebra, and then yeah ZIP-317 took care of the rest."*

Zcash never had Bitcoin's data-carrier war. No ZIP was proposed to restrict `OP_RETURN` or script data. The network's response to a tenfold transaction spike driven by an unsanctioned protocol was to make its two implementations agree with each other and let the fee formula price the load.

20.3 A Zcash-Specific Defect the Bitcoin Design Does Not Have

Porting Bitcoin's envelope to Zcash introduces a malleability problem that does not exist on the chain it was copied from, and the protocol's own hardened specification names it:

"scriptSig data is excluded from the ZIP 244 transaction id and is not covered by any signature, so relayed content could in principle be altered before confirmation without changing the txid."

On Bitcoin, inscription content lives in the witness, which is committed to by the witness transaction identifier and covered by the signature. On Zcash there is no witness; the content sits in an input script that ZIP 244's transaction identifier deliberately excludes. A transaction in the mempool carrying inscription content can therefore have that content replaced by any party able to relay a competing version, and the transaction identifier does not change. The hardened specification's remedy is to add a commitment to the content at *commit* time — a push of `SHA-256` over a domain-separated encoding of the content type and bytes — so that the reveal can be checked against a hash fixed in an earlier, confirmed transaction. This is a correct fix, and it is a fix for a problem created by the port.

20.4 The Result That Matters

The decisive empirical finding in this entire chapter is a single number, produced by an independent indexer over the same corpus.

At block 3,465,610, of **112,814** inscriptions indexed, **15,388** — one in seven — are classified **SHIELDED_UNTRACKABLE**: their controlling output was spent into the shielded pool, and no transparent successor exists. Tracking for those inscriptions is not degraded, delayed, or recoverable. It has permanently ended.

This is the ordinals model meeting Zcash's defining property, and losing. The inscription's identity is carried by an unbroken chain of transparent spends; the moment value crosses into a shielded pool, the chain of custody is not merely hidden from observers but destroyed as a fact about the world. There is no note-level identity inside the pool to carry it, no link between a nullifier and a commitment, and nothing an indexer could be given — short of a viewing key from the holder — that would let it continue.

Fourteen percent attrition is what happens when a protocol that requires public per-unit provenance is deployed on a chain whose purpose is to abolish public per-unit provenance. The two designs are not merely in tension. They are in direct contradiction, and the contradiction resolves in favour of the chain.

It is worth being precise about how much worse this gets under full adoption. Approximately 29% of ZEC supply is currently shielded and roughly 44% of daily transactions are shielded. The one-in-seven figure is not a stable property of the protocol; it is a function of how often inscription holders happen to touch the shielded pool, and it can only rise as shielded usage does. A Zcash in which shielded transactions are the norm is a Zcash in which transparent inscription tracking approaches uselessness.

20.5 Per-Unit Numbering Was Never Even Attempted

Every inscription project on Zcash has claimed ordinal theory — per-satoshi, or per-zatoshi, numbering — in its marketing. An audit of all four implementations found that none of them implements it: no zatoshi index, no value ranges, no satpoint, no rarity. What is implemented is inscriptions plus output-level ownership.

This is not an oversight, and it could not have been fixed with more engineering. Global per-zatoshi numbering is **structurally impossible on Zcash**, for a reason independent of the tracking loss described above: since the Heartwood upgrade, coinbase output may be paid **directly into a shielded pool**. Newly issued ZEC can therefore enter circulation without ever appearing in the transparent output set. A unit that was never a transparent output cannot be assigned an ordinal number under any scheme, because the scheme's entire basis is first-in-first-out assignment across transparent outputs. Bitcoin's ordinal numbering is well-defined because every satoshi that has ever existed was minted into a transparent output. Zcash's is not, because some were not.

20.6 The Shielded Attempts, and Why They Stopped

Two projects tried to place asset data in shielded memos rather than transparent scripts. Both are instructive, and one of them documents its own failure with unusual honesty.

ZAP1 anchors Merkle roots of an off-chain event log into Orchard memos on mainnet. The format is plain ASCII: `ZAP1:{type}:{hash}` — a five-byte prefix, a two-digit hexadecimal type, a colon, and sixty-four hexadecimal characters of a domain-personalised BLAKE2b-256 digest, seventy-two bytes in total, zero-padded into the 512-byte memo field. The anchoring transactions are real and verifiable as transactions: two in block 3,286,625 and 3,286,631 on 27 March 2026, the last in block 3,317,133 on 22 April 2026.

Twenty-nine anchors were published in a fourteen-day window. None has been published in the roughly 172,000 blocks since. The project's live status endpoint reports anchoring disabled with 150 leaves staged and unanchored.

The reason is stated in the project's own documentation, and it is the cleanest available statement of the central problem:

"A recorded txid proves transaction existence; binding an encrypted Orchard memo to this root requires a separate disclosure artifact."

"Public root-to-memo binding requires separate disclosure material because Orchard memo contents are encrypted."

A verifier is given a Merkle inclusion proof under a root the operator supplies, and a transaction identifier that exists. Nothing binds the two together. The root-to-memo binding is pure operator trust, and the reference deployment does not publish the artifact that would fix it. A grant application to harden the protocol was declined in April 2026.

ZINC solves the discoverability problem properly and then runs into its consequence. It publishes a registry address together with its unified full viewing key, so that anyone may trial-decrypt the registry's incoming notes and rebuild the catalogue without permission. This works. But transfers are then specified to go directly from holder to holder, invisible to the registry — which means the registry can attest to mints and nothing else, and the project's own documentation describes ownership as advisory. Its marketplace records sixteen items minted and zero volume.

The dilemma is exact and admits no third option under today's protocol: **a registry that can see transfers destroys the privacy of the transfer graph; a registry that cannot see transfers cannot attest to ownership.** ZAP1 chose invisibility and could not prove anything. ZINC chose visibility at the mint and gave up proving anything after it.

20.7 The Carrier That Is No Longer Available

One further fact closes this chapter, and it post-dates both projects.

The NU6.3 "Ironwood" upgrade activated on mainnet on 28 July 2026 at block 3,428,143, in response to a soundness flaw in the Orchard Action circuit disclosed that May. Among its consensus changes, it **sealed the Orchard pool**: no new value may enter, and every Orchard-pool action must set the cross-address flag to zero. That flag controls whether an action's output note may be addressed to a different receiver than its input note. With it forced to zero, an Orchard note — and therefore an Orchard memo — **can no longer be sent to a third party at all.**

ZAP1's carrier did not merely fall out of favour. As of 28 July 2026 it does not exist. Any shielded-memo protocol on Zcash must now be built on the Ironwood pool and the version-6 transaction format, and any design in this document that contemplates shielded memos is written against Ironwood.

20.8 What the Ecosystem Did About Any of This

Nothing, and the *manner* of the nothing is informative.

There was no hostility and no prohibition: no ZIP restricting data carriage was ever proposed, and the only node-level change was a dust-filter alignment for implementation parity. But there was also no accommodation. Two grant applications for asset-layer infrastructure were declined, in April and June 2026. A four-year-old pull request that would establish a structured memo type registry — opened in October 2022 — remains a draft, and has granted zero allocations; the single request filed against it in April 2026 received one reply, from a bot. Consequently every deployed memo protocol hides its data in the plain-text lead-byte range, because the reserved ranges render as garbage in existing wallets.

No wallet on the network — none of Zashi, Ywallet, Nighthawk, Zingo, or the hardware wallets — has any awareness of inscriptions. Any of them will spend an inscription-bearing output as change without warning. The ecosystem’s workaround has been browser-generated transparent wallets, which is precisely the pattern that led, in June 2026, to a marketplace exfiltrating users’ raw private keys to a server-side function on every transfer, and to the draining of the wallets that used it.

The prevailing position among protocol engineers is a single sentence, offered in the forum thread about NFTs on Zcash in December 2025 by a Zcash Foundation engineer: *”I’m not a fan of the concept but if people want to launch NFTs they should wait for ZSAs to get deployed and use that. The whole point of Zcash is privacy.”*

That advice is correct in its reasoning and, as Chapter 27 shows, considerably harder to follow than it sounds.

20.9 The Ten-Year Record, and a Forgotten Precedent

It is worth putting the entire episode in proportion. An aggregation of every OP_RETURN output ever created on Zcash, by year:

Year	OP_RETURN outputs
2016 (chain launched October)	4
2017	16,249
2018–2020	25 / 103 / 133
2021–2024	842 / 1,490 / 2,330 / 2,999
2025	41,201 (of which November: 34,817)
2026 (to 19 September)	9,950
Total	≈75,300

Zcash’s transparent data layer has carried roughly seventy-five thousand data outputs in a decade. Bitcoin’s equivalent is in the tens of millions. More than a third of Zcash’s entire history in this category is a single repeated fifteen-byte mint script from one nine-day window in 2025.

The ongoing, non-speculative use is cross-chain settlement: decoded payloads show bridge memos of the form OUT:<txid>, OSM0:<bech32 address>, and HYPL:<0x address>, running at

roughly twenty-five to sixty per day. That is what the transparent data channel is actually for on this chain, and it is the use case that motivated the node implementations to formalise a policy for it at all.

The forgotten precedent is 2017. The largest pre-2025 concentration — 16,249 outputs, heavily clustered in March through September 2017 — decodes to a consistent structure:

```
6a 25 01 71 12 20 <32-byte digest>
  | | |  └── 32-byte length
  | | └── 0x12 = SHA2-256
  | └── 0x71 = dag-cbor codec
  └── 0x01 = CIDv1
```

This is a **version-1 content identifier in the dag-cbor codec** — a content-addressed commitment, anchored to Zcash mainnet, at scale, in 2017. Content-addressed anchoring on this chain therefore predates the word “inscription” by six years, predates ordinal theory entirely, and was performed by a party this document has not been able to identify. It is recorded here because it is the earliest direct ancestor of every design discussed in Part VIII, and because it demonstrates that the commitment-anchoring pattern required no protocol changes, no advocacy, and no ecosystem permission — only eighty bytes.

One further fact belongs on the record, because it bears on how much any of the above was ever “allowed.” The node implementation that carries the network today did not enforce a transparent-script standardness policy at all — no data-carrier size limit, no one-output-per-transaction rule — until a change merged in January 2026 and shipped that February. During the November 2025 wave, the relevant node software enforced no `OP_RETURN` limits whatsoever. The limits described in §20.1 are, in the deployed sense, newer than the episode they are usually invoked to explain.

Note also the direction of travel, which is the opposite of Bitcoin’s. In October 2025 Bitcoin Core raised its data-carrier default from 83 bytes to effectively uncapped. Three months later, Zcash’s node implementation *introduced* the 83-byte cap it had never previously applied. No ZIP proposing to restrict, expand, or remove the limit has ever been filed on Zcash; the network’s anti-spam effort went into fee and action-count mechanisms instead.

20.10 One Fact That Cuts the Other Way

It would be a misreading of this chapter to conclude that Zcash is hostile to data carriage, or that the shielded pool is a poor place to put data. On the narrow technical question, the opposite is true, and two facts establish it.

First, ZIP 302 — the memo format specification — does not merely tolerate arbitrary application data; it **designates a lead byte for it**. A memo whose first byte is `0xFF` is, normatively, unconstrained application data in the remaining 511 bytes. Bitcoin’s data carriers are policy accidents that survived a decade of argument about whether they should be allowed; Zcash’s is a sanctioned field with a reserved marker.

Second, the economics strongly favour the shielded path, for a reason that is structural rather than incidental. Every shielded output pays for a 580-byte ciphertext whether or not its memo is used, because uniform ciphertext size is what prevents “this transaction carries a memo” from being an observable property. The 512-byte memo is therefore **pre-paid dead weight**, and filling it costs nothing additional. Against Zcash’s fee formula, an 80-byte `OP_RETURN` payload costs on the order of 250 satoshi per byte; a 512-byte memo in a minimal two-action shielded transaction costs roughly 19.5 satoshi per byte, and under 10 if both output notes carry memos. The shielded channel is more than an order of magnitude cheaper per byte, carries over six times more data per transaction, and costs less in total.

The obstacle has never been capacity, cost, or permission. It is that nobody else can read it — and Chapter 21 is about what follows from that.

21. What Ordinals Requires, and Why Zcash Cannot Supply It

Chapter 20 reported an empirical result: one in seven inscriptions on Zcash has become permanently untrackable, and per-zatoshi numbering was never implemented by anyone. This chapter establishes that neither outcome was an engineering failure. Both follow from the requirements of the ordinals construction, which can be stated precisely, and from properties of Zcash that are not incidental but definitional.

The analysis matters beyond Zcash. Stated abstractly, the requirements below are a checklist that can be applied to any candidate host chain, and the exercise of applying it is what separates a design claim from a marketing one.

21.1 The Seven Requirements

Ordinals is usually described in terms of what it produces — numbered satoshis, inscriptions, rarity tiers. It is more useful to describe what it *consumes* from the chain beneath it.

R1 — Deterministic, totally ordered issuance of the atomic unit. Numbering requires a closed-form emission function and a canonical block order, so that the first unit of any block is computable by anyone from the chain alone. Ordinals' own specification is precise about a subtlety here: *"Underpaying the subsidy does not change the ordinal numbers of sats mined in subsequent blocks. Ordinal numbers depend only on how many sats could have been mined, not how many actually were."* Numbering must depend on the schedule, not on miner behaviour, or every number downstream of an unusual block shifts.

R2 — A transparent, publicly enumerable output graph with ordered inputs, ordered outputs, and explicit per-output amounts. This is the demanding one. The assignment rule is first-in-first-out across a transaction's inputs and outputs, which requires a third party to read, for every transaction: the input list *in order*, each input's link to the specific prior output it spends, and each output's *exact integer value*. All three must be public. The reference specification is fifteen lines of Python, and every line of it indexes into public data.

R3 — Canonical fee semantics and block-level transaction ordering. Units paid as fees must have a defined destination. Ordinals handles this by treating the coinbase as having an implicit input for the subsidy followed by one implicit input per fee-paying transaction, *"in the order that those transactions appear in the block."* Without a canonical ordering, units lost to fees have no computable home and indexers diverge on the first block containing one.

R4 — A data-carrying channel that is large, cheap, committed, and semantically inert. Four properties, all required. Capacity in the hundreds of kilobytes; a pricing discount, or the economics collapse; cryptographic commitment, so the data is immutable and provably present; and inertness, so arbitrary bytes are legal rather than executed. Bitcoin achieves inertness with an unexecuted conditional — though notably not total inertness, since the 520-byte-per-push parse limit is enforced even inside dead code.

R5 — Stable transaction identifiers independent of the payload. Inscription identity is `txid` plus index. This requires the identifier to be malleability-resistant and, ideally, not to cover the payload at all — which is exactly what the segregated-witness identifier split provides.

R6 — Deep finality, or explicit reorganisation tooling. Ordinal assignment is path-dependent on transaction order, so a reorganisation invalidates downstream assignments. The reference indexer survives roughly nine to eighteen blocks of reorganisation via database savepoints and stalls permanently beyond that.

R7 — A social monoculture around one indexer. This is not a chain property at all, and it is load-bearing. Bitcoin supplies R1 through R6 and contributes *nothing* to the question of which unit carries which inscription. That answer comes from one program.

21.2 Zcash Against the Checklist

Requirement		Zcash mainnet, September 2026
R1	Deterministic issuance	Partially. The schedule is fixed and public, but see §21.3 — some issuance never enters the enumerable set at all.
R2	Transparent ordered output graph with explicit amounts	Only for the transparent pool. Approximately 29% of supply and 44% of daily transactions are outside it.
R3	Canonical fee semantics	Yes. Fees behave as on Bitcoin.
R4	Large, cheap, inert data channel	Yes, but not the Bitcoin one. No witness, no SegWit, no Taproot, therefore no witness discount. The available public channels are an 80-byte <code>OP_RETURN</code> payload, one per transaction, and a roughly 1,650-byte input script. The shielded channel is larger and cheaper but encrypted.
R5	Stable identifiers	Yes for the identifier — but see §20.3. The transaction identifier deliberately excludes input scripts, which is precisely where the deployed protocol puts its payload, producing a malleability defect Bitcoin does not have.
R6	Reorganisation handling	Yes. No special difficulty.
R7	Indexer monoculture	Weakly. Two independent indexers over the same corpus report 111,378 and 112,814 inscriptions at nearby heights.

R2 is the binding constraint, and it does not bind partially. It fails in a way that cannot be engineered around.

21.3 Two Independent Impossibility Results

It is worth separating two distinct failures, because they are often conflated and only one of them is about privacy.

First: the enumeration gap. Since the Heartwood upgrade, coinbase output may be paid directly into a shielded pool. Newly issued ZEC can therefore enter circulation **without ever having been a transparent output**. Ordinal numbering is defined by first-in-first-out assignment across transparent outputs; a unit that was never a transparent output has no position in that ordering and cannot be assigned a number under any scheme consistent with the rest of the

construction. This is not a tracking problem — it is a definitional one, and it arises at issuance, before any user does anything.

Bitcoin's numbering is well-defined because every satoshi that has ever existed was minted into a transparent output. Zcash's is not, because some were not. **R1 fails at the source.**

Second: the tracking collapse. For units that *do* begin transparent, identity survives only as long as the chain of transparent spends does. When an output is spent into a shielded pool, the unit does not become hidden — it ceases to have a per-unit identity as a fact about the world. Inside the pool there is no per-note denomination of a specific unit, no link between a nullifier and the commitment it spends, and no quantity an indexer could be given short of a viewing key that would let it continue the FIFO walk. Value re-emerging from the pool is, to every observer including the protocol's own designers, new value.

This is measured, not theorised: **15,388 of 112,814 inscriptions — one in seven — are already classified untrackable for exactly this reason.** And the figure is not a stable property. It is a function of how often holders touch the shielded pool, which means it can only increase as shielded usage does. A Zcash in which shielded transactions are the norm is a Zcash in which transparent inscription tracking is asymptotically useless.

R2 fails for any unit that ever becomes private, and the failure is permanent and unrecoverable.

21.4 The Consequence Nobody Advertises

The two results above have a corollary that is more damaging to the ordinals-on-Zcash thesis than either alone.

A protocol may respond to R2's failure by confining itself to the transparent pool — which is exactly what every deployed Zcash inscription protocol does. But confinement is not a workaround; it is a redefinition of the product. An asset that is tracked only while transparent is an asset whose holders must never use the privacy feature of the privacy chain. The design's entire proposition reduces to: *use Zcash, but not for what Zcash is for.*

The one-in-seven attrition figure is the measurement of holders declining to accept that constraint. It is not an error rate. It is the rate at which users chose the chain's purpose over the protocol's requirement, and the protocol lost.

21.5 What Bitcoin Actually Guarantees, Stated Exactly

Because the comparison chapter that follows depends on it, it is worth being unusually precise about what Bitcoin's consensus rules enforce for an inscription. The complete list, for a reveal transaction, is:

1. The signature over the input is valid.
2. The revealed script hashes into the commitment in the output key.
3. Script execution succeeds — the envelope contributing nothing, since it sits in a false branch.
4. The witness data is committed to via the witness root in the coinbase.

5. The output being spent existed and is not spent twice.
6. The transaction sits at a specific index in a specific block.

That is the entire list. The bytes exist, they are immutable, they are ordered, and the coin moved.

Everything else is convention: that a three-byte magic string means anything; that tag 1 is a content type and tag 3 a parent; that transfer is first-in-first-out rather than last-in-first-out; that inscriptions have numbers at all. Bitcoin has no concept of a satoshi with an identity — only outputs with integer values.

The reference implementation says so in its own documentation, in a passage worth quoting because no comparable system states it as plainly:

"If, due to a bug in `ord`, this document diverges from the actual behavior of `ord` and it is impractically disruptive to change `ord`'s behavior, this document will be amended to agree with `ord`'s actual behavior."

And the consequence has been demonstrated at scale, not merely feared. There is no state root: a search of the reference indexer's entire index subsystem for any hashing of its own state returns nothing. A proposal to introduce a consensus hash was opened in 2023 and remains open. Two indexers can therefore agree on every published counter while disagreeing about which unit carries every single inscription, with no mechanism that would reveal the disagreement.

What that permits was shown in 2026, when the operators of the dominant fungible-token metaprotocol on Bitcoin coordinated a balance migration across twenty-three tickers at a named block height, moving user holdings from one address to another **by agreement, with no Bitcoin transaction authorising the move**. The instruction was published as an indexing rule. Nothing in Bitcoin's consensus could have prevented it, because Bitcoin had never been told the balances existed.

This is the precise sense in which Architecture I is a convention: not that it is insecure, but that its security is the security of an agreement among operators, and the chain underneath cannot distinguish a correct agreement from an incorrect one.

21.6 Where This Leaves the Design

Three conclusions follow, and they set the agenda for Parts VIII and IX.

An ordinals-style construction on Zcash is possible, deployed, and structurally incompatible with the chain's purpose. It works — 111,378 times over — and it works by staying transparent. It cannot be extended into the shielded pool by any amount of engineering, because the properties it requires are the properties the shielded pool exists to destroy.

The shielded pool offers a better data channel than the transparent one, and it is unreadable. This is not a contradiction; it is the whole problem. A 512-byte memo is more than an order of magnitude cheaper per byte than an `OP_RETURN` payload and carries six times more data per

transaction, because every shielded output pays for its ciphertext whether or not the memo is used. The capacity is free and the audience is one.

Therefore the question worth asking is not “how do we port ordinals to Zcash” but “what does a chain have to enforce for private ownership to be verifiable at all.” That question has an answer, it is specified, and it has been specified since 2023. Whether it will ever be deployed is the subject of Chapter 27, and it is a considerably less comfortable topic than the specification itself.

Part VIII — What Can Be Built on Mainnet Today

Part VII established a negative result: the ordinals construction requires a transparent, enumerable output graph, Zcash supplies one only for the transparent pool, and any unit that enters a shielded pool leaves the construction permanently. Chapter 21 closed by reframing the question — not *how do we port ordinals*, but *what must a chain enforce for private ownership to be verifiable at all*.

This Part answers that question for Zcash mainnet as it exists in September 2026. The answer is more favourable than the preceding chapters imply, and it rests on a property of the Orchard-family action structure that has not, to this document’s knowledge, been used for this purpose before.

22. Nullifier-Sealed Client-Side Validation

22.1 The Requirement, Stated Formally

Write the target properties precisely, so that the claim at the end of this chapter can be checked rather than believed.

Let the **public transcript** T_h be everything a node with no secrets derives from the chain up to height h : the transparent graph, note commitments, nullifier sets, value balances, anchors, opaque ciphertexts, and any data-carrier payloads.

- **P1 — Anchoring.** There exists a deterministic function of T_h yielding a commitment to the item set I and a height h_0 , with I immutable thereafter.
- **P2 — Ownership privacy.** No efficient adversary given T_h and no participant secret can identify the holder of an item with non-negligible advantage over the shielded anonymity set.
- **P3 — No mutable registry.** Ownership is a deterministic function of T_h and holder-held secrets. No party's private mutable state is required.
- **P4 — Transfer integrity.** (a) At most one verifying claim per item at any height. (b) No party except the holder can produce a verifying transfer.

P4(b) is the property the operator of a conventional registry cannot offer, and P3 is the property that makes P4(b) checkable rather than promised.

22.2 The Primitive Nobody Used

The construction rests on a single sentence in ZIP 224, whose status is **Final**:

" p is set to the nullifier of the spent note in the same action, which ensures it is unique."

Recall from Chapter 3 that an Orchard-family action spends exactly one note and creates exactly one note, and that the created note's uniqueness element p is required — *inside the proving circuit* — to equal the nullifier of the note spent in that same action. This was introduced to guarantee global note uniqueness without a counter or a tree position. Its consequence for asset tracking appears not to have been noticed.

Define:

Seal — an Ironwood note, named by its public commitment cm . **Closure** — spending that note, which publishes its nullifier nf . **Successor** — the unique note in all of chain history whose $p = nf$.

Successor uniqueness is enforced by consensus, and the argument is short.

1. Two actions in one bundle cannot both produce a note with $p = nf_k$, because each action's output p equals *its own* spend's nullifier, and the consensus rule forbids a nullifier repeating within a transaction.
2. No later transaction can produce a note with $p = nf_k$, because doing so would require spending a note whose nullifier is nf_k — and that nullifier is already in the set, so the transaction is invalid.
3. A dummy spend cannot manufacture nf_k , because that is a preimage problem on the nullifier PRF, and the nullifier set would reject the duplicate in any case.

There is exactly one successor, and the chain says so **in the circuit**.

This is precisely the definition of a **single-use seal** in the sense introduced by Peter Todd in 2016 — a uniquely identified object that can be closed over a message exactly once, such that it is infeasible to produce two apparently valid proofs of closure over different messages. The difference from every deployed single-use-seal system is where the binding lives. In RGB and Taproot Assets, the seal is a Bitcoin outpoint and the binding of a successor to a closure is an application-level commitment that the verifier must check. Here, the binding *is a circuit constraint*. The chain does not merely order the events; it refuses to produce an inconsistent one.

A second property follows for free and is worth recording. p **is not part of the note plaintext**. A note's encrypted payload is $(leadByte, d, v, rseed, memo)$; the recipient recovers p from the nullifier field of the action that created the note. A note is therefore cryptographically inseparable from its creating action — its lineage cannot be relabelled, because p matches exactly one action's nf .

22.3 The Construction

Genesis. The issuer derives one single-purpose account per item and creates one note per item. It publishes a manifest mapping each item to its note commitment and its metadata, and anchors the manifest digest in a single transparent data output at height h_0 .

The collection's identity **is the genesis transaction identifier**. This is not a convention; it is what makes issuer equivocation impossible rather than merely detectable. A second manifest published under a different anchor is, by definition, a different collection. The anchor is written once: if the item set is fixed at genesis — which P1 requires — there is nothing to re-anchor, and the per-transfer public footprint is zero.

Transfer. The seller spends the item's note in the same action that creates the successor note addressed to the buyer's fresh single-purpose account. Fee payment and change occupy a second action. The result is an ordinary two- or four-action shielded transaction with a value balance equal to the fee — the single most common transaction shape on the chain.

Verification. The consignment handed from seller to buyer is the ordered list of full viewing keys, one per hop. A full viewing key is $(ak, nk, rinvk)$: **ninety-six bytes**. Everything else the buyer needs comes from the buyer's own node. For each hop the buyer:

1. derives the incoming viewing key and checks that the note's diversified transmission key satisfies $pk_d = [ivk] \ g_d$ — this pins the nullifier-deriving key to the note and closes the key-substitution attack;
2. trial-decrypts the note, recomputes its commitment, and confirms membership in the commitment tree;
3. computes the nullifier from the note and the disclosed nk ;
4. locates that nullifier on chain — consensus guarantees it appears at most once — and reads the output commitment of the same action, which must equal the next hop's commitment; equivalently, checks $p_{\{j+1\}} = nf_j$;
5. checks that the first commitment in the chain is the manifest entry for the item under the genesis anchor.

Current ownership is step six, and it is the step that defeats the stale-proof attack. The buyer computes the *seller's* nullifier and checks whether it is in the nullifier set. If it is absent, the seller still holds the item. If it is present, the seal is closed and the seller is lying. The seller cannot suppress this: the nullifier set is public and the buyer derives the query themselves.

22.4 Proof Size, and Why It Is Smaller Than Bitcoin's Equivalents

Client-side validation systems are conventionally criticised for proof growth, and the criticism is well founded. Peter Todd stated the worst case plainly: *"it's plausible that I could transfer you a gigabyte of data to give you a coin."* The growth is driven by the merge graph — each consumed input drags in its entire ancestry.

System	Payload per hop
Taproot Assets	~0.9–1.9 KB
RGB	~0.4–0.7 KB
Nullifier-sealed CSV	96 bytes

The difference is structural, not an optimisation. The Bitcoin systems must ship, per transition, an anchoring transaction, a block header, a transaction Merkle proof, and exclusion proofs for every other taproot output — because the verifier cannot be assumed to hold the chain, and because the binding between a seal closure and its successor is an application-level commitment that must be demonstrated. Here the verifier is *already required* to run a node, since P1 demands they check the genesis anchor; and the successor binding is a consensus rule, so there is nothing to demonstrate. What remains to transmit is only the capability to recompute the nullifiers: ninety-six bytes per hop.

The consequence for recovery is worth stating separately. A consignment of this size fits inside a self-addressed shielded memo — roughly five hops as full viewing keys, or about sixteen if already-spent account seeds are used instead, which are harmless to disclose once their notes are spent. **Recovery from a seed phrase alone therefore becomes reachable**, which is precisely

the property that Chapter 19 identified as the discriminator between architectures, and precisely the property that RGB's own documentation concedes it cannot offer.

22.5 What an Observer Sees

Nothing item-specific. There is no data-carrier output, no marker, no unusual action count, no value leak — amounts are hidden inside the pool — and no receiver-type leak, because Ironwood reuses the Orchard receiver typecode and the same incoming viewing key trial-decrypts both pools. A transfer is bit-indistinguishable from an ordinary shielded payment, inside the anonymity set of all shielded activity on the chain.

This is a requirement, not a nicety, and the protocol's own migration specification explains why. ZIP 318 forbids migration transactions from carrying *"no nonstandard fee amount or any other outputs"*, giving the reason directly: **"this would partition the anonymity set between wallets that add such outputs and those that do not."** Since essentially no Zcash transaction carries a transparent data output today, a transaction combining one with shielded actions would be a flare reading *asset transfer here*. The data-carrier output belongs at genesis, where publicity is the objective, and nowhere else.

22.6 What This Achieves, and What It Does Not

Achieved, simultaneously, on mainnet today, with no consensus change and no new circuit:

- **P1** — the item set is fixed at h_0 , publicly timestamped and immutable, with collection identity equal to a transaction identifier, so issuer equivocation is impossible by definition.
- **P2** — zero item-specific public footprint.
- **P3** — ownership is a deterministic function of chain data and holder-held keys. The operator holds nothing anyone needs; it can go dark, be seized, or be compelled without a single item moving.
- **P4** — (a) one note, one nullifier, one action, one successor, all consensus-enforced; (b) nobody but the holder has spend authority.

Not achieved, and these are not implementation gaps:

1. **There is no public ownership lookup.** Nobody can ask who holds a given item, how many remain live, or what the holder distribution is. Chapter 23 shows this is forced rather than chosen.
2. **Verification requires the seller's cooperation.** A prospective buyer learns nothing — not even that an item exists — until handed the consignment. A seller who disappears makes the item unsaleable, though the holder still holds it.
3. **The full ownership history is disclosed to every subsequent holder.** Each hop reveals its viewing key downstream, permanently and transitively. These are pseudonymous accounts rather than identities, but the disclosure is irreversible. **This is the one leak that only a consensus-level asset layer closes**, and it is the strongest available argument for why ZSA matters — a considerably better argument than any appeal to convenience.

4. **Settlement is not atomic unless multi-party transaction construction works for this case.** The partially-created-transaction format gained full Ironwood support in mid-2026; if two mutually distrusting parties can co-construct one bundle — seller contributing the seal closure, buyer contributing payment — settlement becomes atomic with no escrow and no dependency on the asset-swap proposal. This is the highest-value unverified item in the design and should be tested before anything else.
5. **Careless spending destroys the item.** Spending the note without creating the successor burns it. This is the same failure that has consumed transparent inscriptions on this chain, and it requires either a dedicated wallet or note-level spend locking.

22.7 Cost

Under the deployed fee formula, a two-action shielded transfer costs 10,000 satoshi and a four-action transfer 20,000; the genesis anchor, carrying a transparent data output, costs 20,000. At recent prices these are fractions of a dollar.

One caution belongs in any implementation built against this section. The marginal fee constant is currently in flux: one node implementation's development branch carries a reduced value, while every released node and the active fee specification retain the original. The proposal to change it is an open pull request introducing a *new* draft rather than an amendment to the existing specification. **The released value is the one to design against**, and a wallet that adopts a new fee before its peers does not merely misprice — it fingerprints its own cohort, for the reason given in §22.5.

23. The Boundary: Why Public Ownership Cannot Follow

Chapter 22 claimed four properties and conceded one absence: there is no public ownership lookup. That concession deserves a proof rather than an apology, because a reader entitled to ask “why not just publish it” should be shown that the answer is structural.

23.1 The Uniqueness Argument

Uniqueness — the guarantee that an item cannot be transferred twice — can only be enforced by consensus over a resource that consensus can *name*. A validator rejects a double spend because it recognises the resource being consumed. Every such resource is therefore public.

Two cases follow, and they are exhaustive.

Case (i) — the binding is publicly computable. If any observer can determine which resource stands for which item, then observing that resource’s consumption reveals the item’s transfer, and following the resource across transfers reveals the sequence of holders. Privacy fails. This is Construction 1 of Chapter 24, and it is what every transparent inscription protocol does.

Case (ii) — the binding is hidden. Consensus then enforces uniqueness *of the resource* but says nothing about the item, because it has never been told which item the resource stands for. Item-level uniqueness must be re-established by a verifier who has been given the binding.

The important observation is that case (ii) **still delivers transfer integrity**. The binding is a function — each item maps to exactly one resource — and the resource is consumable exactly once by consensus. A verifier handed the binding therefore inherits uniqueness *from the chain*, not from the party who disclosed it. A lying discloser cannot manufacture a second valid history, because the second history would require a second consumption of a resource the chain permits to be consumed once.

So the correct statement is not that privacy and integrity are incompatible. It is that **privacy is incompatible with public verifiability of ownership**, while integrity survives intact in the view of anyone to whom the binding is disclosed.

23.2 Exhaustiveness on Zcash

The argument above is general. Applied to Zcash it becomes decisive, because the protocol contains exactly **two** resources protected against double consumption:

1. **Transparent outputs.** Publicly labelled by construction — an outpoint is a transaction identifier and an index, both in the clear. This is case (i).
2. **Shielded nullifiers.** Public, but unlinkable to the notes they retire. Unlinkability is not incidental: the Orchard design lists **spend unlinkability** as a required property — the incoming viewing key is specified to permit detection and decryption of received notes *“without being able to spend the notes or detect when they are spent.”* This is case (ii).

There is no third resource. Therefore there is no third construction, and the two constructions available are the two described in this Part. A design claiming both public ownership verifiability and ownership privacy on Zcash mainnet today is claiming a resource that does not exist.

23.3 What Would Close the Gap

Strengthening the anchoring requirement from *"the item set is publicly committed"* to *"any third party can verify who owns what"* makes the target impossible without new cryptography — but it is worth being precise about how much new cryptography.

Public resolution requires an observer to adjudicate between two competing claims about the same item. That adjudication is exactly the statement *"this nullifier is the nullifier of the note committed by this commitment"* — which is the nullifier-integrity relation of the Action circuit. An observer can check it in one of two ways: by holding the nullifier-deriving key, which destroys the holder's privacy, or by being handed a zero-knowledge proof of the relation, which does not.

The second option is a **circuit, not a proof system**. The proving system already runs on mainnet, requires no trusted setup, and has mature tooling. The statement needed is roughly the nullifier-integrity and commitment-derivation constraints plus a Merkle path — call it half an Action circuit. Two pieces of existing work demonstrate this is engineering rather than research: an independent Halo 2 proof-of-assets implementation over Orchard requiring no consensus change, and a draft specification for a domain-separated balance proof whose public inputs include a domain-separated nullifier and which proves that eligible notes *"exist in the commitment tree at that height, but don't exist in the nullifier set at that height."*

Neither is audited; both are off-chain; the second is Orchard-only and explicitly does not contemplate custom assets. The honest characterisation is: months of work plus an audit, not a research programme — and worth stating in those terms rather than gesturing at future cryptography.

23.4 The Leak Is a Dial, and the Middle Settings Are Traps

Between total privacy and total publicity there are intermediate designs, and they are tempting for product reasons. They should be evaluated carefully, because two of the three intermediate settings buy less than they appear to.

Setting	Public footprint	What the public can verify	Ownership privacy
0 — seal chain only	none	nothing	full
1 — blinded commitment per transfer	opaque 32 bytes	that <i>some</i> event occurred	content hidden; the event is visible, and a data-carrier output inside a shielded transaction is itself a fingerprint
2 — item identifier and closed nullifier published	item, height, nullifier	per-item provenance timeline and transfer count — but not which branch is valid	holders hidden; item activity public

3 — transparent seal

everything

everything

none

Setting 2 is the seductive one, because it makes a reproducible public indexer possible: anyone could rebuild a provenance timeline for every item from chain data alone. **It does not, however, give public resolution.** An attacker can publish a competing record naming the same item, a different nullifier, and a different successor. The public now sees two branches and cannot adjudicate between them, because adjudication requires exactly the nullifier-to-commitment binding that setting 2 does not publish. Setting 2 buys conflict *detection* and not conflict *resolution* — and it pays for that with the anonymity-set partition described in §22.5, since transactions carrying data outputs would become identifiable as asset transfers.

Setting 0 is therefore the recommendation, and there is independent support for it from an unexpected direction. The W3C’s status-list specification, addressing the analogous problem of revocation privacy, states in its privacy considerations:

“The most privacy-preserving status list is one that never changes, since the behavior of the population cannot be determined if no observable events occur.”

Anchor once. Publish nothing per transfer.

23.5 Why the Literature Does Not Offer a Way Out

It is reasonable to ask whether some cryptographic primitive already solves this. The honest answer is that the primitive exists, is well studied, and does not deliver what a marketplace would want from it.

The strong form of the requirement — a publicly committed set supporting membership and non-membership proofs that reveal nothing else — is a **zero-knowledge set**, introduced by Micali, Rabin and Kilian in 2003. The construction exists under standard assumptions. Its footnote on why soundness must be computational is the deepest structural statement available on this problem:

“the computational nature of soundness is necessary in our setting. Namely, our very stringent zero-knowledge requirements imply that, for each choice of the security parameter, a commitment to S should always have the same size... In turn, this implies that a commitment to S must be only computationally binding. Indeed, the number of bits necessary to pin-down a set S exactly cannot be shorter than S ’s Kolmogorov complexity, and thus any perfectly binding commitment to a set must grow with the set.”

But note the scope, which is routinely collapsed in secondary discussion: in a zero-knowledge set, proofs are produced **by the prover, who holds the set**. The commitment is public and proofs are publicly verifiable, but a third party cannot test membership without asking the prover. Zero-knowledge sets therefore permit *online* enumeration at the prover’s discretion — they do not permit offline enumeration. That reintroduces an online party who can selectively refuse, which

is the same partitioning problem that makes the anchored-registry construction unsatisfactory, and it is why fork consistency rather than non-equivocation is the achievable notion.

The generalisation of the construction to mercurial commitments concedes directly that *"none of these constructions is practical."*

Every construction that *removes* the online prover publishes enough to enumerate. Set-membership proofs in the zero-knowledge literature take the set as common input — one widely cited scheme has the verifier *"specify Φ by providing 'digital signatures' on the elements of Φ "* — and the modern hidden-order-group constructions supporting non-membership state plainly that *"in our solutions the set of committed elements is public and not hidden to the verifier."*

The eighteen-year natural experiment is DNS. Authenticated denial of existence has exactly this shape: prove a name is absent without revealing the zone. The first mechanism leaked the zone by construction. The second hashed it and conceded dictionary attacks in its own specification. The third proved that soundness plus privacy force an online public-key operation per negative response — and was not zero-knowledge either, leaking the set size — and expired unadopted. The current best-practice document then set the second mechanism's privacy parameters to *zero*, instructing operators to use no salt and no additional hash iterations, on the grounds that they do not change the cost of the attack. Eighteen years of standardisation produced the conclusion that the privacy was not obtainable at acceptable cost.

This is not an argument that the problem is unsolvable. It is an argument that a whitepaper claiming to have solved it should expect scrutiny, and that the correct move is to state the boundary rather than to blur it.

24. Data-Carrying Surfaces and Their Economics

A design that anchors once and hides everything else still has to put the anchor somewhere. This chapter enumerates what Zcash mainnet actually offers, with the numbers a builder needs, and corrects two widespread errors about them.

24.1 The Public Channels

Transparent data output. Zcash inherits Bitcoin's script system, and the relevant limit is policy rather than consensus. The maximum relayed data-carrier script is **83 bytes** — 80 bytes of payload after the opcode and the pushdata prefix — and **exactly one such output is permitted per transaction**. This value is unanimous across all three node implementations that have existed, and has never been relaxed.

Note the direction of travel, which is the opposite of Bitcoin's. In October 2015 Bitcoin Core raised its equivalent default from 83 bytes to effectively uncapped. Three months later, the node implementation that now carries the Zcash network *introduced* the 83-byte cap it had never previously enforced. No proposal to restrict, expand, or remove it has ever been filed on Zcash.

Input script. Bounded by a standardness limit of roughly 1,650 bytes and required to be push-only. This is the carrier the deployed inscription protocol uses, because it is the only public channel on the chain large enough for content. Note the defect described in §20.3: input scripts are excluded from the transaction identifier and covered by no signature.

Coinbase script. Two to one hundred bytes, of which most is free after the mandatory height encoding. Miner-only, and therefore not a general-purpose channel.

Zcash has no witness, no segregated-witness discount, and no taproot. The four-times pricing discount that makes Bitcoin inscriptions economical does not exist here. A byte of public data costs a byte.

24.2 The Shielded Channel

Every shielded output carries a 512-byte memo field inside its encrypted note ciphertext. The memo specification does not merely tolerate arbitrary application data; it **designates a lead byte for it** — a memo whose first byte takes the reserved application value has 511 unconstrained bytes following.

Two widespread errors about this field are worth correcting explicitly.

First, the application range is one byte, not a range. The reserved values between the "no memo" marker and the application marker are reserved *for future revisions of the memo specification*, not for applications. A protocol using them is not using a private range; it is squatting on the specification's own extension space. The consequence is visible in practice: every deployed memo protocol on the chain hides its payload in the plain-text range instead, because reserved values render as garbage in existing wallets.

Second, there is no type registry, and the absence is not an oversight. A structured-memo proposal establishing exactly such a registry was opened in October 2022 and remains a draft. It has granted **zero** allocations. The single request filed against it received one reply, from a bot, and has sat for five months. Any protocol that needs to coexist with another protocol in the same field is, today, on its own.

24.3 The Economics, Which Favour the Private Channel

The fee formula is proportional to a count of logical actions, with a floor of two. A transparent contribution is computed from serialized input and output sizes rather than from counts, which is what makes data outputs expensive.

Transaction shape	Logical actions	Fee
One transparent input, 80-byte data output, change	4	20,000 satoshi
One transparent input, 32-byte commitment output, change	3	15,000
Baseline transparent payment	1	10,000
Two-action shielded transfer with memos	2	10,000

Per byte of payload delivered:

Channel	Cost per byte
Transparent data output	~250 satoshi
512-byte memo, two-action transaction	~19.5 satoshi
Both output notes' memos	~9.8 satoshi

The shielded channel is more than an order of magnitude cheaper per byte, carries over six times more data per transaction, and costs less in total. The reason is structural and is worth stating because it inverts the usual intuition that privacy is expensive: every shielded output pays for a fixed-size ciphertext whether or not its memo is used, because uniform ciphertext size is precisely what prevents "this transaction carries a memo" from being an observable property. The 512 bytes are pre-paid dead weight. Filling them costs nothing.

The obstacle to using the shielded channel has never been capacity, cost, or permission. It is that nobody else can read it — which is the subject of Chapter 22, and the reason the construction described there puts its one public byte-string at genesis and nowhere else.

24.4 A Caution on the Fee Constant

One parameter is currently in flux and will produce wrong numbers for anyone who reads the wrong source. The marginal fee per logical action is 5,000 satoshi in the active fee specification and in every released node. One node implementation's development branch carries a reduced value of 1,000, and a proposal to adopt it exists as an open pull request introducing a *new draft specification* rather than amending the existing one.

Design against the released value. And note that the consequence of getting this wrong is not merely mispricing: a wallet that adopts a new fee schedule before its peers becomes identifiable by its fees, for exactly the reason given in §22.5. The draft's own rationale says so — if wallets switch at different times, the fee reveals which software produced the transaction.

24.5 What Is Specified but Unavailable

Two proposals would change the picture materially, and neither is available.

Memo bundles would replace the inline 512-byte memo with a 32-byte key plus a separate bundle of encrypted chunks, capped at 64 chunks — **16 KiB per transaction**, two hundred times the transparent limit. Critically for the constructions in this Part, it defines a reserved key value meaning *publicly decryptable*: a shielded transaction with a memo any observer can read. That is precisely the primitive that would collapse the tension of Chapter 23, since it would allow a public commitment to ride inside a shielded transaction without a transparent output and without the anonymity-set partition that comes with one.

It is a draft, it requires a transaction format that is itself a draft, and the next network upgrade introduces no new transaction format. Nothing is built on it.

Explicit fees would move the fee into a committed transaction field. Same status, same dependency, same conclusion.

The honest summary of this chapter is that Zcash's data-carrying story is better than its reputation and worse than its specifications: the private channel is cheap, sanctioned, and unreadable; the public channel is small, expensive, and newly policed; and the proposal that would reconcile them has been drafted, audited, and not scheduled.

Part IX — The Specification That Was Orphaned

25. ZSA: A Forensic Status Report

Every prior treatment of assets on Zcash, including earlier drafts of this document, has been written in the shadow of a pending upgrade. The shape of the argument was always the same: mainnet cannot do this yet, the specification exists, wait for the network upgrade. That argument is no longer available, and the reason it is no longer available is not a rejection, a vote, or an announcement. It is a slow structural orphaning that is visible only by reading the specifications against each other.

This chapter sets out what can be established from primary sources, because a builder deciding where to spend the next year deserves the evidence rather than a summary.

25.1 The Three Documents That Define the Asset Layer

The asset proposal is not one specification but three, and they have diverged.

ZIP 226 — Transfer and Burn of Custom Assets. Defines the modified Action circuit: an asset identifier accompanies every note, the value commitment is randomised per asset, and the balance check is performed per asset rather than globally. This is the cryptographic core, and it is the part that works.

ZIP 227 — Issuance of Custom Assets. Defines the issuance bundle: an issuance authorising key, an asset description string, a finalisation flag that permanently closes an asset's supply. Notably, issuance actions are *not* shielded — the issuance bundle is transparent by design, so that supply is publicly auditable while transfers are not. This asymmetry is deliberate and is the single most important design decision in the proposal, because it is exactly the split between a public anchor and private transfers that Part VIII reconstructs by other means.

ZIP 230 — Version 6 Transaction Format. Defines the transaction serialisation that would carry the other two. Without it there is no wire format, no consensus rule, and no way for a node to parse an asset transfer.

The first two are Drafts. **The third is Withdrawn.**

25.2 The Orphaning, Step by Step

A withdrawn document might mean a replacement was found. The evidence says otherwise, and it accumulates across five independent surfaces.

The replacement format does not carry assets. The successor to the withdrawn v6 proposal contains an explicit statement that the new format *need not support* custom assets. The asset layer was not moved into the new format; it was written out of it.

The fee specification removed its asset clauses. The active fee specification once contained provisions for issuance actions and per-asset accounting. Those provisions are gone. A transaction type with no fee rule is a transaction type no node can price, and therefore no node can accept.

The identifier space was reassigned. The lead byte that the asset proposal reserved for its own use has been allocated to an unrelated specification. This is the clearest single signal in the record: identifier reassignment does not happen while a proposal is considered live, because it creates an unresolvable collision. Somebody with commit access concluded the collision would never occur.

The upgrade excludes transaction format changes. The next network upgrade's published candidate list contains no asset-related item, and its scope statement rules out new transaction formats generally. Since the asset layer *is* a new transaction format, it cannot be included by any mechanism short of expanding the upgrade's scope.

The governance signal was negative. A coinholder poll offered the asset layer as an option against competing priorities. It did not win. This is the only part of the record that looks like a decision, and it is worth stating carefully: the poll was advisory, turnout was concentrated, and the result is better read as an absence of demand than as a rejection on technical grounds.

25.3 The Stale Sentence That Misleads Everyone

The asset transfer proposal contains, in its deployment section, a sentence asserting that it is deployed in the next network upgrade. That sentence is stale. It has not been updated to match the withdrawal of the format it depends on, and it is the single most common source of error in secondary discussion — including, briefly, in this project's own public writing, which is why it is flagged here rather than quietly corrected.

The general lesson is worth extracting, because it will recur: **a specification's deployment section is metadata maintained by hand, while its dependency graph is structural.** When the two disagree, the dependency graph is correct. A proposal cannot deploy in an upgrade that ships no transaction format capable of carrying it, whatever its own header says.

25.4 What This Does and Does Not Mean

It does not mean the cryptography is wrong. The circuit modifications are sound, were reviewed, and have a working implementation. It does not mean the asset layer will never ship — specifications have been revived before, and the engineering is largely done.

It means the timeline is unbounded. There is no scheduled upgrade carrying it, no transaction format to carry it in, no fee rule to price it, and a reserved identifier now belonging to something else. A plan whose critical path runs through the asset layer has a critical path of unknown length.

This is precisely the condition that makes Part VIII's construction worth the effort. Not because it is better than a consensus-level asset layer — it is not, and §26 details exactly what it gives up — but because it exists.

26. What the Asset Layer Would Actually Provide

Building around an absence requires knowing its shape. This chapter states what the orphaned specification would deliver if it shipped, so that the construction of Part VIII can be measured against it honestly rather than defended rhetorically.

26.1 Per-Asset Value Commitments

In the current shielded protocol, a value commitment is a Pedersen commitment binding an amount under a single fixed generator. Balance is checked by summing commitments across a bundle and verifying the result opens to the declared net value. One generator, one asset, one global balance check.

The asset proposal replaces the fixed generator with one derived per asset from the asset identifier. Balance then holds independently for each asset, because commitments to different assets live on incomparable bases and cannot cancel. The change is small in the specification and consequential in effect: it makes “you cannot turn one asset into another” a *circuit constraint* rather than an application convention.

That phrase is the whole difference. In Part VIII’s construction, an item’s identity is enforced by a chain of nullifier-to-commitment bindings that a verifier checks after being handed a disclosure. Under the asset layer, it would be enforced by the same validators that enforce monetary supply, for everyone, with no disclosure. The security is not stronger in the sense of being harder to break — both are enforced by the same proving system — but it is *universal* rather than *relational*.

26.2 Public Supply, Private Holders

The issuance bundle is transparent. An asset’s identifier, its description, its issuing key, and every issuance event are on chain in the clear, while every subsequent transfer is shielded.

This is the property that the construction of Part VIII approximates with a genesis anchor and cannot fully reach. Under the asset layer, supply is auditable by anyone at any time from chain data alone, with no cooperating party, and the finalisation flag makes “no more of this will ever exist” a consensus fact rather than a promise. Part VIII’s genesis anchor commits to a manifest once; it cannot prove that the issuer did not later create a second manifest anchored elsewhere, except by the social fact that the collection’s identity *is* its genesis transaction identifier.

For fungible tokens this gap is decisive. For a fixed collection anchored once and never extended, it is narrow — but it is real and should not be papered over.

26.3 The Unique-Asset Case

The proposal contemplates non-fungible assets directly: an asset whose issuance is a single unit and whose supply is then finalised is, by construction, unique and provably so. Nothing about the construction requires a convention, an indexer, or an off-chain registry to interpret it.

26.4 The Leak It Closes

Chapter 22 conceded one leak: in the disclosure-based construction, each successive holder learns the full prior history, because verifying the chain requires being handed every prior hop's viewing key. The disclosure is permanent and cannot be revoked.

This is the leak the asset layer closes, and it closes it completely. Under a consensus-enforced asset, a buyer verifies uniqueness by verifying the transaction — the same way they verify that they were paid. There is no consignment, no chain of keys, no history to hand over. A holder learns nothing about who held the item before them.

It is worth being plain about the size of this. For a collection where provenance is a feature, the leak is harmless or even desirable. For anything where prior ownership is sensitive, it is disqualifying, and the disclosure-based construction should not be recommended.

26.5 The Counterfeiting Bug, and What It Teaches

In May 2026 an advisory disclosed a vulnerability in the asset layer's reference implementation permitting counterfeit issuance. It was found before deployment, in code that had been reviewed, in a specification considered mature.

The instructive part is not the bug but its location. Per-asset balance depends on the asset identifier reaching the value-commitment generator *through a path the circuit constrains*. Any implementation path that lets an identifier be chosen outside those constraints reintroduces exactly the cross-asset cancellation the design exists to prevent. The failure mode is not in the mathematics — it is in the binding between the mathematics and the surrounding code.

The same failure mode exists in Part VIII's construction, in the same place: the binding between an item's identifier and the note that seals it. In the disclosure-based construction that binding is checked by the verifier rather than by consensus, which means a defective verifier implementation produces a wrong answer for its own user rather than a counterfeit for everyone. That is a smaller blast radius, and it is one of the few respects in which the weaker construction is genuinely safer.

27. Settlement: Trading Without Trusting

Every construction in this Part describes how an item moves. None of them describes how an item moves *in exchange for money*, and the gap between those two is where marketplaces actually fail.

27.1 Why the Obvious Answer Does Not Work

On a chain with a scripting system, atomic exchange is a solved problem: lock both sides under a condition, release both or neither. Zcash shielded transactions have no scripts, no covenants, and no conditional spending. A shielded note is spendable by whoever holds its spending key, full stop.

The consequence is stated bluntly in Chapter 22's honest-limitations list and repeated here because it is the single largest unresolved item in the design: **there is no escrow at the protocol level**. A marketplace either holds funds itself — custody, with all that implies — or finds atomicity elsewhere.

27.2 Where Atomicity Actually Lives

It lives in transaction construction rather than in transaction semantics.

A shielded transaction is a single atomic object: all of its actions succeed or none do. If a *single* transaction can be built that simultaneously spends the seller's item note, creates the buyer's successor note, spends the buyer's payment note, and creates the seller's payment note, then the exchange is atomic — not because the protocol enforces a condition, but because a transaction is indivisible.

The obstacle is construction, not validity. Building that transaction requires signatures from two parties over a structure neither party can complete alone, and neither should have to reveal spending keys to the other.

This is exactly the problem the partially-created transaction format addresses: a serialisation for a shielded transaction under construction, passed between parties, each adding their own inputs, outputs, proofs, and signatures, with the intermediate object carrying enough information to be checked but not enough to be spent. The format's current release supports the shielded pool in question. It was built for hardware wallets, where the signing device and the constructing host are different machines with different trust — structurally the same problem as two counterparties.

27.3 The Honest Status

This is the highest-value unverified claim in the document, and it is flagged rather than asserted.

What is known: the format exists, is released, supports the relevant pool, and is designed for exactly the multi-party construction pattern required. What is not known: whether two mutually

distrusting parties can complete the specific transaction shape needed — in particular whether the item-note spend and the payment-note spend can be co-constructed without either party being able to extract a spendable object mid-flight, and whether the resulting proof can be generated by a party who does not hold both spending keys.

The proving step is the crux. A single proof covering both parties' actions must be generated by somebody, and in the shielded proving system a proof over an action requires witness data for that action. Whether that witness data can be supplied by its owner without being disclosed to the prover is the question, and it is answerable only by building it.

Until it is answered, the construction of Part VIII settles through a venue that holds funds, which is custody. That is worth stating in exactly those words rather than in the word "escrow," which would be false.

27.4 The Proposal That Would Settle It

A specification exists for payment-conditional shielded transfers — a mechanism permitting a transfer to be made contingent on a payment without a trusted third party. It is a draft. It depends on the same withdrawn transaction format as the asset layer, and inherits the same unbounded timeline described in Chapter 25.

The pattern by now will be familiar: the specification that solves the problem cleanly exists, is well designed, and is not available. Every honest roadmap for this system therefore has the same shape — build what mainnet permits, state the gap, and do not let the existence of a draft stand in for a shipped feature.

Part X — What Other Chains Built, and What It Cost Them

28. Ordinals, Dissected

Any claim about what Zcash can or cannot do with inscriptions requires knowing precisely what an inscription *is*. The protocol is widely discussed and rarely read, and most of the received wisdom about it is wrong in ways that matter for the comparison.

28.1 The Envelope

An inscription is data embedded in a spending script inside a construct chosen specifically because it never executes:

```
OP_FALSE
OP_IF
  OP_PUSH "ord"
  OP_PUSH 1
  OP_PUSH "text/plain;charset=utf-8"
  OP_PUSH 0
  OP_PUSH "Hello, world"
OP_ENDIF
```

The leading false-push means the conditional body is skipped by every validator that has ever run. The data is therefore *stored* by consensus and *interpreted* by nobody — the script is valid precisely because its contents are never evaluated.

This is the architectural decision from which everything else follows, and it is the one most often missed: **Ordinals is not a protocol the chain enforces. It is a convention for reading data the chain agreed to carry without looking at it.** Two indexers that disagree about what an envelope means produce two different answers, and no consensus rule adjudicates between them.

28.2 Tags, and Why They Are Numbered Oddly

Fields inside the envelope are numbered, and the numbering is not sequential. Odd-numbered tags are *ignorable*: an indexer that does not recognise an odd tag may skip it and still agree with every other indexer about what the inscription is. Even-numbered tags are *not* ignorable: an indexer that does not recognise one must refuse to interpret the inscription at all, because the unknown field might change its meaning.

This is a genuinely elegant piece of protocol design and is worth stealing. It permits a convention-based system to evolve without hard forks *and* without silent divergence, by making the extension's own number declare whether ignoring it is safe. It is the same trick as feature bits in other protocols, applied to a system with no negotiation channel.

The content type and the body use even and odd slots respectively. The body tag is odd — and it is last, with everything after it treated as content, which is what allows content to be split across multiple pushes to evade the 520-byte per-push limit.

28.3 Commit and Reveal

Script data can only appear in a *spending* script, which means the script must first be committed to as the spending condition of an output that does not yet exist. Inscribing therefore takes two transactions:

Commit — create an output whose spending condition includes the envelope, committed to only as a hash. Nothing is visible.

Reveal — spend that output, which requires publishing the script, which publishes the data.

Two transactions, two fees, and a window between them during which the inscription exists but is invisible. The pattern will reappear in Chapter 29 and in every system that carries data in spending scripts, including the deployed inscription protocol on Zcash.

28.4 The Discount That Makes It Viable

Inscription data lives in the witness — the segregated portion of a transaction excluded from the transaction identifier — and witness bytes are priced at one quarter of the rate of non-witness bytes. A kilobyte of inscription costs what 250 bytes of ordinary transaction data costs.

Zcash has no witness and no discount. This single fact does more to explain the divergence in inscription activity between the two chains than any cultural or community factor, and it should be the first thing said in any comparison. The deployed Zcash inscription protocol pays full freight for every byte, in a script field capped at roughly 1,650 bytes by relay policy, with no fee discount and no 4 MB block to amortise against.

28.5 Ordering, and the Accident That Became a Feature

Inscriptions are numbered in the order they are revealed. Early in the protocol's life, certain malformed envelopes were assigned *negative* numbers — "cursed" inscriptions — as a way of recording them without letting them occupy positive numbering space. The set of conditions that produced negativity grew by accretion, as indexer authors discovered new malformations.

At a fixed block height the rule was retired and all subsequently-created previously-cursed forms were blessed into the positive sequence. The event has a name in the community and none in any specification, because there is no specification: the rule changed when the dominant indexer's code changed.

This is the clearest available illustration of the architectural point in §28.1. A numbering that half a million people treat as canonical was altered by a software release. There was no fork, no vote, and no consensus rule — the chain was entirely indifferent, because the chain never knew the numbers existed.

28.6 What the Comparison Actually Shows

Set against the taxonomy of Chapter 19, Ordinals is unambiguously an indexer convention: the chain stores bytes, an off-chain program assigns meaning, and the meaning is whatever the

dominant program says. Its great strength is that the *data* is permanent and self-contained — an inscription's content is in the block, recoverable forever, with no external dependency. Its great weakness is that the *interpretation* is not, and the Jubilee is the proof.

The construction of Part VIII inverts both. Interpretation is enforced in-circuit and cannot be changed by a software release; data is not on chain at all and depends on the holder to carry it. Neither is strictly better. They fail in opposite directions, and a builder should choose based on which failure they can tolerate — permanent data with mutable meaning, or immutable meaning with data you must keep.

29. The Other Four Designs

Ordinals is one point in a design space that four other deployed systems occupy differently. Each made a distinct trade, and each trade illuminates something about the constraints of Part VIII.

29.1 Counterparty: The Oldest Answer

Predating every system in this chapter by nearly a decade, Counterparty encoded asset operations in data outputs and had an off-chain engine interpret them. Assets were named, issued, and traded through messages the chain carried and did not understand.

It established the pattern and it established the failure mode: total dependence on a single reference implementation, whose behaviour *was* the protocol. Its long tail of consensus-divergence incidents between implementation versions is the historical record that Ordinals repeated a decade later with the Jubilee.

Its one durable lesson is about data outputs specifically. Counterparty's use of them was a significant early driver of the debate that produced the 83-byte relay limit — the same limit that Chapter 24 documents as still unanimous on Zcash today. The constraint Part VIII designs around is, in a direct sense, a scar left by this system.

29.2 Runes: Minimalism as a Response

Built by the same author as Ordinals and explicitly as a correction to it, Runes encodes fungible-token operations in a single compact data output rather than in witness scripts. No commit-reveal, no envelope, no witness dependence — one output, variable-length integers, tightly packed.

The design goal was to stop fungible-token activity from bloating the UTXO set, which the preceding token convention had done badly. It largely succeeded, and its relevance here is the demonstration that **a well-designed token protocol fits in a data output smaller than the Zcash limit**. The 83-byte constraint of Chapter 24 is not obviously fatal for a token layer; Runes operates comfortably inside a comparable budget.

What it does not do is privacy, and it does not try. Every issuance, every transfer, and every holder is public.

29.3 Taproot Assets: Proofs Off-Chain, Commitment On

The first of the two client-side-validation systems in this comparison. Asset state lives in a Merkle tree; only the tree's root is committed on chain, inside a taproot output's script tree. Transfers are proven by a chain of proofs handed from sender to receiver, each proving that the asset moved correctly and that the movement was committed on chain.

This is architecturally the closest deployed system to Part VIII's construction, and the comparison is therefore the most informative in this chapter. Both anchor commitments on chain

and validate transfers client-side. Both require the recipient to receive and check a proof chain. Both lose the asset if the proof chain is lost.

The difference is size and dependence. A Taproot Assets transfer proof runs to hundreds of bytes to low kilobytes per hop, because it must carry Merkle inclusion paths, the anchoring transaction, and its confirmation proof — the verifier is not assumed to have the chain. Part VIII's construction assumes the verifier runs a node, which it must anyway to check payment, and therefore carries only the 96 bytes of viewing key needed to locate and interpret what the node already has.

It also inherits Bitcoin's public UTXO graph: the anchoring outputs are visible, their spends are visible, and the *shape* of asset activity is therefore public even when its content is not. This is the transparent-seal weakness of Construction 1, and it is why the comparison favours the nullifier-based seal despite Taproot Assets being vastly more mature.

29.4 RGB: The Maximal Version

The most ambitious client-side-validation system: a full contract-execution environment validated entirely off chain, with Bitcoin used only for single-use seals. Contracts can express arbitrary state transitions; the chain sees only that some output was spent.

Its consignment — the bundle of data a recipient needs to validate — is correspondingly large, running from under a kilobyte for simple transfers to considerably more for contracts with history. It is the strongest available demonstration that client-side validation *works*, and simultaneously the strongest demonstration of what it costs: the data-carrying burden grows with history, and a recipient who cannot obtain the full consignment cannot validate at all.

RGB's relevance to Part VIII is a specific negative result. **Recovery from a seed phrase alone is not possible in RGB, by construction** — the consignment is not derivable from any key, and a wallet restored from backup knows it has assets but cannot prove anything about them. Part VIII's construction can, in the narrow case where consignments are stored in self-addressed memos, recover from seed alone, because the memo field is part of the chain and is decryptable by the restored key. That is a genuine architectural advantage and it is worth naming as the one place where the newer construction beats the mature one on its own terms.

29.5 The Tabulation

	Data location	Interpretation enforced by	Public ownership	Per-hop proof	Seed-only recovery
Counterparty	data outputs	reference implementation	yes	n/a	yes
Ordinals	witness script	dominant indexer	yes	n/a	yes
Runes	data output	dominant indexer	yes	n/a	yes
Taproot Assets	off chain	recipient's validator	partially (graph visible)	0.4–1.9 KB	no

RGB	off chain	recipient's validator	partially (graph visible)	~1 KB+	no
Part VIII construction	off chain	the proving circuit	no	96 B	yes, via memo

The row that matters is the third column. In every deployed system, what an asset *means* is decided by software that can be changed. In the construction of Part VIII, the binding between one note and its successor is a constraint inside a proof that consensus verifies — the same machinery that prevents the currency from being counterfeited. That is the claim of this document, and Chapters 22 and 23 are the argument that it holds along with its honest boundary.

Part XI — The Machine Underneath

Everything in Parts VIII through X rests on properties of the shielded protocol that were asserted and not derived. This Part derives them. A reader who accepts the construction on authority can skip it; a reader who intends to implement, audit, or attack it cannot, because every claim made about uniqueness, privacy, and cost reduces to a specific field in a specific structure, and the difference between a correct implementation and a catastrophic one is usually one of those fields.

30. The Action

The unit of shielded state change is the Action. It is deliberately not called an input or an output, because it is both: every Action simultaneously spends one note and creates one note. There is no way to spend without creating, and no way to create without spending.

30.1 Why Spend and Output Are Fused

In the preceding shielded pool, spends and outputs were separate structures and a transaction declared how many of each it contained. That count leaked: a transaction with three spends and one output has a different shape from one with one spend and three outputs, and shape is metadata.

Fusing them removes the leak. Every transaction contains some number of Actions, each identical in structure, and the count is the only quantity visible. A transaction paying one recipient from one note looks structurally identical to a transaction consolidating two notes into one, because both are expressed as two Actions with the surplus slots filled by dummies.

Dummy Actions are the load-bearing part of this design. A dummy Action spends a note of zero value that the sender created for the purpose and whose spending key they hold. It is cryptographically indistinguishable from a real Action — same proof, same structure, same size — and it exists so that Action count can be padded to hide the true shape. The protocol does not merely tolerate dummies; the privacy argument requires them.

This has a direct consequence for the construction of Part VIII that is easy to miss: **a verifier checking a seal chain must be able to distinguish a real successor from a dummy, and it can, because it holds the viewing key that decrypts the note.** An attacker cannot forge a successor with a dummy, because the dummy's note is not decryptable under the key the verifier was given. The dummy mechanism that hides transaction shape does not weaken the seal.

30.2 The Fields

An Action serialises to 820 bytes. Every one of them is public, and knowing what each does is the difference between understanding the privacy model and guessing at it.

Field	Bytes	What it is	What it leaks
nullifier	32	Retires the spent note	Membership in the nullifier set. Unlinkable to any commitment.
rk	32	Randomised verification key for the spend authorisation	Nothing — re-randomised per Action from the holder's key
cmx	32	Commitment to the created note	Nothing — hiding commitment
ephemeralKey	32	Sender's ephemeral public key for note encryption	Nothing on its own
encCiphertext	580	Encrypted note plaintext, including the 512-byte memo	Nothing — fixed size regardless of contents

<code>outCiphertext</code>	80	Sender-recoverable copy of the encryption key	Nothing
<code>spendAuthSig</code>	64	Signature under <code>rk</code>	Nothing

Two observations carry most of the weight of this document.

First, `encCiphertext` is fixed at 580 bytes whether the memo is empty or full. This is why Chapter 24's economics work out the way they do — the memo is pre-paid dead weight, and a protocol that uses it pays nothing extra. It is also why the memo cannot be used to leak anything by its presence: every Action carries one.

Second, `rk` is re-randomised per Action. The key that authorises a spend is derived from the holder's spend authorising key blinded by fresh randomness each time, so two spends by the same holder present unrelated verification keys. Without this, every spend by one party would be linkable by key reuse, and the entire privacy model would collapse into a pseudonymous one.

30.3 What Is Not in the Action

Absences matter as much as fields.

There is **no amount**. Value lives inside the note commitment and inside a separate value commitment aggregated at the bundle level; individual Action values are never published.

There is **no address**, sender or recipient. The recipient is encoded inside the encrypted ciphertext; the sender is nowhere at all.

There is **no reference to which note was spent**. This is the single most important absence in the protocol, and it is the technical content of the phrase "unlinkable." A transparent spend names its input by transaction identifier and index. A shielded spend publishes only a nullifier, which is a deterministic function of the spent note's secret material and therefore reveals nothing about which commitment it retires to anyone who does not hold the key.

The last absence is what makes Construction 2 of Chapter 23 possible and Construction 1 impossible to fix.

30.4 The Bundle

Actions are grouped into a bundle carrying the shared parts: a single aggregated proof covering every Action, a net value balance, an anchor naming the commitment tree root the proofs are relative to, flags, and a binding signature.

Three bundle-level fields deserve attention.

`valueBalance` is a public signed integer. It is the net flow between the shielded pool and everything outside it. A fully shielded transaction has a balance of zero and reveals nothing; a transaction moving value in or out publishes the exact amount in the clear. This is the "pool boundary" that Chapter 23 identifies as the attack surface against transparent-seal constructions, and it is worth restating why: the amount is not merely visible, it is *exact*, and the fee is deterministic, so an analyst who knows a listing price knows what to search for.

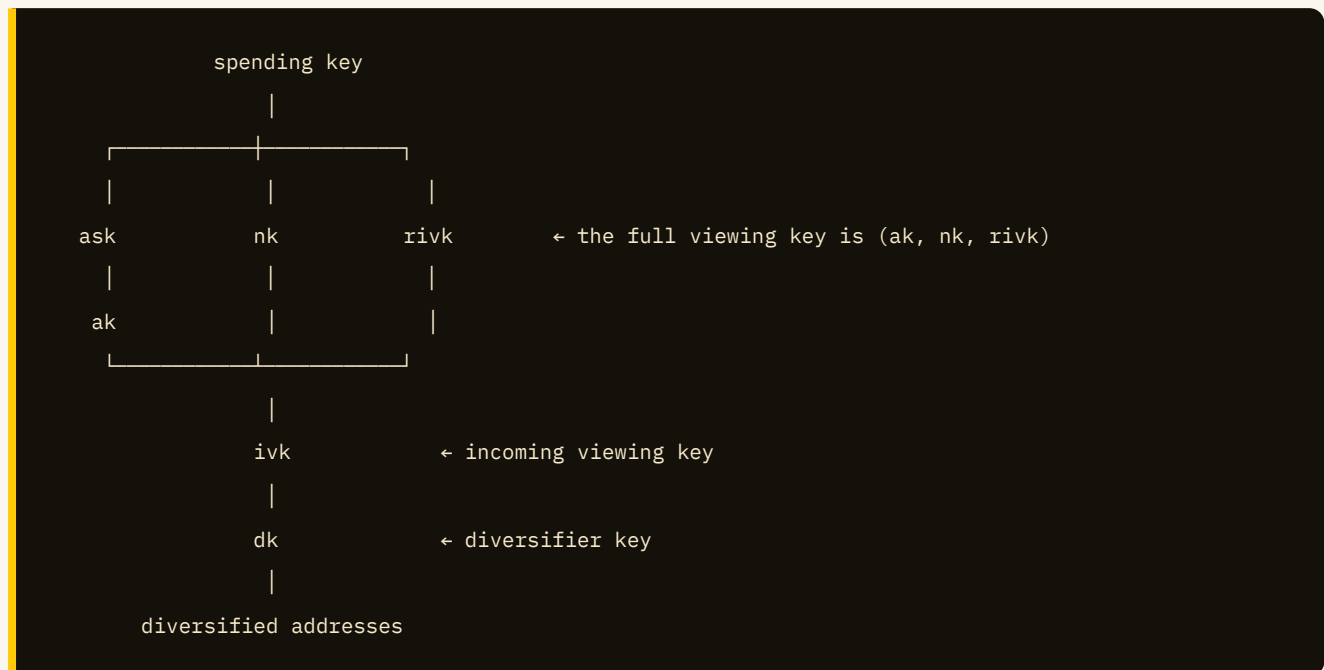
anchor names a historical tree root. Every Action's proof is relative to some past state of the commitment tree, and the anchor declares which. Because a wallet may legitimately use a root some blocks behind the tip, the anchor's age is a fingerprint: a wallet that always anchors to the tip and one that always anchors 10 blocks back are distinguishable. This is a real and under-discussed side channel, and it belongs in the same category as the fee-schedule fingerprint of Chapter 24.

The binding signature ties the proof to the value balance. Its signing key is derived from the sum of the value commitment randomnesses, so producing it requires knowing that every Action's value commitment is accounted for. It is the mechanism by which balance is enforced without any individual value being public.

31. The Key Hierarchy

Everything the construction of Part VIII does with disclosure depends on a hierarchy in which each level can do strictly less than the level above. The hierarchy is not a convenience; it is the reason a 96-byte disclosure can prove ownership history without conveying the ability to spend.

31.1 The Descent



Each arrow is a one-way derivation. Knowing a child does not yield the parent.

Spending key. The root. Confers the ability to spend, and therefore everything below.

ask — **spend authorising key**. Produces signatures under randomised keys. Required to spend, useless for anything else.

ak — **spend validating key**. The public counterpart to **ask**. Lets a holder verify a spend authorisation without producing one.

nk — **nullifier deriving key**. Computes the nullifier of a note. **This is the key that confers the ability to detect spends**, and its separation from **ivk** is the entire content of the property named "spend unlinkability."

rivk — **commitment randomness base**. Used in deriving **ivk** from the other two.

ivk — **incoming viewing key**. Derived from **ak** and **nk** together. Permits trial decryption of incoming notes.

dk — **diversifier key**. Generates the diversifier values that produce distinct addresses from one key.

31.2 The Two Capabilities, and Why They Must Be Separable

The whole hierarchy exists to make two questions answerable independently:

- “Did I receive this note?” — answerable with `ivk`.
- “Has this note been spent?” — answerable only with `nk`.

The protocol specification states the requirement directly: the incoming viewing key must permit detection and decryption of received notes **without being able to spend the notes or detect when they are spent**.

Consider what fails if the separation does not hold. A recipient must be able to give an auditor the ability to see incoming payments — that is the entire purpose of a viewing key. If seeing incoming payments also revealed outgoing spends, then disclosing to an auditor would disclose the holder’s complete financial activity, and viewing keys would be unusable for their intended purpose.

Now consider the same separation from the other side, which is where Part VIII lives. A disclosure conveying `(ak, nk, rivk)` — the full viewing key — conveys both capabilities: the recipient can decrypt the note *and* compute its nullifier *and* check whether that nullifier appears on chain. That third capability is precisely what makes the verification in Chapter 22 work and what makes the stale-proof attack impossible: a buyer does not merely verify that the seller once owned the item, they verify that the seller **still** owns it, because the nullifier they compute is absent from the chain.

A disclosure conveying only `ivk` would prove the history and not the present. The 96 bytes are `(ak, nk, rivk)` for exactly this reason.

31.3 Diversified Addresses

One key produces many addresses. A diversifier — an 11-byte value — is hashed to a group element, that element is multiplied by `ivk`, and the pair of diversifier and resulting point is the address. Any number of addresses derive from one key, all detectable by the same `ivk`, and no observer can tell that two addresses belong to one holder.

The construction of Part VIII uses one fresh single-use account per item, which is a stronger form of the same idea: not merely a fresh address under one key, but a fresh key entirely. The reason is disclosure hygiene. Disclosing a full viewing key for a shared account would disclose every note ever received by that account, including unrelated payments. Disclosing a full viewing key for an account that has held exactly one note discloses exactly one note.

This is a design rule and not an optimisation. An implementation that reuses accounts across items leaks the holder’s entire history to every buyer of any item. It is the single most likely place for a correct-looking implementation to be catastrophically wrong, and it deserves a test.

31.4 The Derivation Path

Key material descends from a seed through a hardened derivation path with a purpose constant, a coin type, and an account index. Hardened-only derivation means a child key cannot be derived from a parent public key, which removes the class of attack where one exposed child plus a parent public key compromises siblings.

For the construction of Part VIII this matters because the one-account-per-item rule generates many accounts, and they must all descend from one seed for the recovery story of §29.4 to hold. An implementation that generates item accounts from fresh entropy rather than from the seed produces accounts that cannot be recovered from a backup, which silently converts the system's one advantage over the mature client-side-validation systems into a disadvantage.

32. Note Encryption and the Cost of Finding Your Own Money

A shielded note is delivered to its recipient by being encrypted into the Action that creates it and broadcast to everyone. There is no addressing, no routing, and no index. The recipient finds their notes by attempting to decrypt every note on the chain.

32.1 The Scheme

The sender generates an ephemeral secret, computes a shared secret by Diffie–Hellman against the recipient’s address point, derives a symmetric key, and encrypts the note plaintext. The ephemeral public key travels in the Action. The recipient recomputes the shared secret using `ivk` and decrypts.

A second, smaller ciphertext encrypts the same key material to a key the *sender* controls, so a sender can recover the contents of their own outgoing payments from chain data alone. This is what permits a wallet restored from seed to reconstruct its send history, and it is the mechanism that makes self-addressed memo storage — the recovery path named in §29.4 — work.

32.2 Trial Decryption

The recipient attempts decryption against every Action. A failed attempt costs an elliptic curve scalar multiplication plus a symmetric decryption attempt; a successful one yields the note.

There is no way to shortcut this without leaking. Any index that let a recipient find their notes faster would, by existing, reveal which notes belong to which recipient — that is what an index *is*. The cost is not an implementation deficiency; it is the price of the property.

The practical consequence is that a wallet’s synchronisation cost is linear in the number of shielded outputs on the chain since its birthday, and that a system generating many one-time accounts must be careful: naively, N accounts means N trial decryptions per Action, and scanning cost multiplies by N .

The mitigation is that item accounts have known birthdays. An account created to receive item k at height h cannot hold anything before h , so it need only scan from h . A wallet holding a hundred items scans the full chain once for its main account and a short suffix for each item account. This is a real implementation concern with a clean answer, and an implementation that misses it will be unusably slow.

32.3 The Detection Key Problem

An adjacent proposal defines a lighter key conferring only the ability to *detect* that a note is yours without decrypting it, which would permit outsourced scanning. It is not available, and its absence is one of the reasons mobile shielded wallets remain harder to build than their transparent equivalents.

For Part VIII the absence is tolerable, because the verification flow is not a scanning flow: a verifier handed a disclosure knows exactly which height and which account to look at. Verification is $O(1)$ in the chain's length. It is *ownership* that is expensive, not *proof*.

33. The Commitment Tree

Note commitments accumulate in an append-only Merkle tree of depth 32, giving a capacity of about 4.3 billion notes. A spend proves, in zero knowledge, that its note's commitment is a leaf of this tree at some historical root.

33.1 Why a Tree and Not a Set

The proof must demonstrate membership without revealing which member. A Merkle path does this: the circuit takes the leaf, the path, and the root, and constrains that hashing up the path from the leaf yields the root. The root is public; the leaf and the path are private.

The depth is what costs. Thirty-two levels means thirty-two hash computations inside the circuit, which is the dominant cost of the Action circuit and therefore the dominant cost of proving. The hash is chosen for arithmetic-circuit efficiency rather than for speed on a CPU — it is cheap in constraints and comparatively slow in software, which is the correct trade when the constraint count is what you pay for.

33.2 Anchors and the Reorganisation Problem

Because the tree grows with every block, a proof is relative to a specific root. A wallet constructing a transaction picks a root it knows, and nodes accept any root from a window of recent history.

This window is why a wallet can construct a transaction without racing the chain tip. It is also a fingerprint, as noted in §30.4: how far behind the tip a wallet anchors is a choice, and different wallets choose differently.

A reorganisation that removes blocks invalidates anchors in them. A transaction anchored to a removed root is not merely delayed, it is permanently invalid and must be rebuilt. For the construction of Part VIII this is a liveness concern rather than a safety one — a seal-closing transaction that fails to confirm leaves the seal open, and the item simply has not moved. The dangerous case is the opposite, and it does not arise: there is no way for a reorganisation to produce a *second* valid successor, because the successor's identity is determined by the nullifier, and a nullifier that has been reorganised out of existence has taken its successor with it.

33.3 Frontier State

A wallet does not store the whole tree. It stores the frontier — the rightmost path — which suffices to append, plus witness paths for its own unspent notes, which must be updated as the tree grows.

Witness maintenance is a real operational burden and a real source of bugs. A witness that is not updated produces a proof against a stale position and fails; a witness updated incorrectly produces a proof that fails in a way that is hard to distinguish from key corruption. Any implementation of Part VIII's construction inherits this burden once per held item.

34. The Nullifier

The nullifier is the pivot of the entire construction, and it repays reading closely.

34.1 What It Is

The nullifier of a note is computed from the nullifier deriving key, the note's own randomness and position, and the note commitment, through an arithmetic-circuit-friendly hash. It is deterministic: one note has exactly one nullifier, computable by anyone holding `nk`, and by nobody else.

Consensus maintains the set of all nullifiers ever published and rejects any transaction publishing a duplicate. That is the whole of double-spend prevention in the shielded pool.

34.2 The Three Properties

Determinism. One note, one nullifier, always. Without this, a holder could spend a note twice by producing two different nullifiers for it.

Unlinkability. Given a nullifier and the set of all commitments, no efficient algorithm identifies which commitment it retires, absent `nk`. Without this, the shielded pool would be a public ledger with extra steps.

Unforgeability. Producing the nullifier of a note requires `nk`. Without this, an attacker could retire somebody else's note — denial of service against a specific holder, which in an asset system means destroying an item.

The three together are what the circuit enforces, and each maps to a failure mode of Part VIII's construction: without determinism, an item could be transferred twice; without unlinkability, ownership would be public; without unforgeability, items could be destroyed by strangers.

34.3 The Sentence Everything Rests On

The Action specification states that a created note's randomness p is set to **the nullifier of the note spent in the same Action**, and gives the reason: to ensure it is unique.

Read that as a structural statement rather than as an implementation detail. It says that every note created in the shielded pool carries, in a field that the circuit constrains and that no user chooses, the identity of the note whose destruction paid for it. The pool is therefore not a set of independent notes. It is a forest of chains, each link forged inside a proof.

The protocol built this to guarantee that p never repeats, because a repeated p would produce a repeated nullifier and break the double-spend rule. That was the entire motivation. The chain structure is a side effect.

The construction of Part VIII is the observation that the side effect is a single-use seal chain, and that it is enforced by the same circuit that enforces the currency.

34.4 The Three Exhaustive Cases

The uniqueness of a successor deserves a proof rather than an assertion. Let note N have nullifier nf . The claim is that at most one note in the entire history of the chain has $p = nf$.

Case 1: two Actions in the same bundle. Each Action's output p is the nullifier of *that Action's own* spend. For two Actions to produce outputs with the same p , they would have to spend notes with the same nullifier — a duplicate nullifier within one transaction, which consensus rejects.

Case 2: a later transaction. A later Action producing $p = nf$ must spend a note whose nullifier is nf . That nullifier is already in the set. Consensus rejects it.

Case 3: a dummy. A dummy Action spends a zero-value note the sender fabricated. Its output p is that fabricated note's nullifier — a value the attacker cannot steer to equal nf , because doing so requires finding a preimage of the nullifier hash. And if they could produce a note whose nullifier is nf , Case 2 rejects it as a duplicate regardless.

The cases are exhaustive over ways a note can come into existence. **At most one successor exists, and consensus enforces it.**

34.5 Where This Can Still Go Wrong

The argument above is about consensus, and consensus is not the only failure surface.

The seal can be **closed into nothing**. A holder who spends the item note in an ordinary payment — to a change address, to an exchange, to anyone — closes the seal, and the successor is whatever note that payment created. If the recipient does not know they received an item, the item is functionally destroyed. This is the same hazard that burned inscriptions on this chain in the past, and the mitigation is entirely at the wallet layer: item notes must be marked unspendable by ordinary flows.

The disclosure can be **lost**. A verifier needs the chain of viewing keys. A holder who loses it holds a note they cannot prove anything about. The memo-based storage of §29.4 mitigates this and does not eliminate it.

The verifier can be **wrong**. Every check in Chapter 22's verification is performed by the verifier's own software. A defective verifier accepts a bad chain. As noted in §26.5, the blast radius is one user rather than the whole system — but for that user it is total.

Part XII — The Transition Nobody Wrote About

35. What Sealing the Old Pool Changed

A network upgrade in mid-2026 introduced a new shielded pool and simultaneously closed the previous one to new external recipients. The change is recorded in a few lines of consensus rules and has been discussed almost entirely in terms of migration logistics. Its consequences for anything built on the shielded pool are much larger than that, and this chapter states them, because an earlier draft of this document described a design that the upgrade silently killed.

35.1 The Rules

Two constraints define the sealing.

A cross-address flag is fixed at zero. In the sealed pool, an Action may no longer create a note for an address other than the one it spends from. Every Action becomes, effectively, self-directed. Payments to third parties are impossible.

The net value balance is constrained non-negative. Value may enter the sealed pool or stay, but it cannot leave it directly. The pool is a one-way door.

The combined effect is that the old pool becomes a holding area: notes inside it can be consolidated and moved to the new pool, and nothing else.

35.2 What This Killed

Any design that used the previous pool's memo field to carry application data between parties is dead. Not degraded, not deprecated — impossible, because a memo is carried in a note and notes can no longer be sent to other parties in that pool.

This included the obvious first design for an asset layer on the existing chain: send the item as a memo-annotated note from holder to holder. That design worked on the day it was written and stopped working at a fixed block height with no announcement aimed at anybody building on it.

The terminology debt this creates runs through the earlier Parts of this document. Passages written before the upgrade name the older pool where they mean the current one. The assertions remain true of the current pool — the Action structure, the key hierarchy, and the nullifier mechanism carry over essentially unchanged — but the names are wrong, and a reader checking them against the specification will find the older pool cannot do what is claimed. The correction is mechanical and it is owed.

35.3 What Carried Over

The new pool inherits the Action structure, the 820-byte layout, the key hierarchy, the commitment tree, the nullifier construction, and — critically for everything in Part XI — the rule that a created note's *p* is the nullifier of the note spent in the same Action.

The seal-chain property is therefore not a feature of the old pool that the upgrade removed. It is a feature of the Action design, and it survives in the pool that can actually transfer.

35.4 The Lesson About Building on Specifications

The upgrade removed a capability from a shipped, widely-used pool, with the removal expressed as a flag set to zero. There was no deprecation period for application developers because there were no application developers to warn.

The defensive posture this suggests is worth stating: **depend on structure, not on permission.** A design that relies on the ability to send a note to a third party depends on a policy that can change. A design that relies on the nullifier being unique depends on the double-spend rule, which cannot change without changing the currency. The construction of Part VIII was chosen partly for this reason — its load-bearing assumption is the one assumption the chain cannot relax.

36. Pool Boundaries and What Crossing Them Costs

The shielded pool is not a place value lives so much as a place value *hides*, and the hiding is only as good as the boundary. This chapter examines the boundary, because it is where every practical deanonymisation of this chain has occurred and where the transparent-seal construction of Chapter 23 fails.

36.1 The Boundary Is Public and Exact

A transaction moving value between the transparent side and the shielded side publishes the exact amount as a signed integer at the bundle level. Not a range, not a commitment — the number.

Combined with a deterministic fee schedule, this produces a matching surface. An analyst who observes 3.5 ZEC entering the pool and, some blocks later, 3.4999 ZEC leaving it, has a candidate pairing whose strength depends only on how many other transactions of similar value occurred in the window.

36.2 Why a Fresh Address Does Not Help

The intuitive defence is to use a fresh transparent address for each operation, so that the transparent side carries no reusable identifier. This defends against address clustering and does nothing against value matching, because the attack does not use addresses. It uses amounts, and amounts are published regardless of which address is involved.

This is the precise reason the transparent-seal construction fails as a privacy design. A marketplace publishes listing prices — that is what a marketplace is. An analyst therefore does not search blindly for coincidences; they search for a known value. The search space collapses from “all pairs of similar amounts” to “transactions matching this specific published price,” and the deterministic fee makes the expected exit amount computable to the zatoshi.

Worse, the item itself is a join key. Two transparent seals for two items held by the same person are linkable through the item history even if every address differs, and the linkage does not expire.

36.3 What the Literature Actually Supports

A widely-quoted figure attributing a large majority of shielded activity to a handful of heuristics derives substantially from founder-reward transactions and from regular, distinguishable pool usage patterns. The founder reward ended in 2020 and the patterns changed. Quoting that figure in the present tense is an error, and it has been quoted in the present tense in this project’s own earlier writing.

The defensible modern results are narrower and more useful. They include the only measured false-positive rate in the published literature on this chain’s linkability — a figure meaningfully above zero, which matters because heuristics presented without one imply perfect precision.

And they include a measurement of value-fingerprint survivability across a simple single-note deshielding operation, which is high enough to be alarming and is the direct empirical support for §36.2.

The honest summary is that the shielded pool is strong and its boundary is weak, that most published attacks target the boundary, and that a design whose privacy claim depends on boundary-crossing operations is claiming something the literature does not support.

36.4 The Design Rule

Cross the boundary as rarely as possible, and never in a way correlated with an item.

For the construction of Part VIII this is satisfied structurally: item transfers are pool-internal, with a value balance of zero and no transparent component whatsoever. The only boundary crossing in the entire lifecycle is the genesis anchor, which happens once, is deliberately public, and is correlated with the collection rather than with any holder.

37. The Turnstile, and What It Proves About Auditability

A recurring objection to shielded systems is that hidden supply cannot be audited: if nobody can see the notes, nobody can prove there are not extra ones. The chain's answer is a mechanism worth understanding, because it demonstrates exactly how much public accounting is compatible with private holdings — which is the question Chapter 23 asks in a different form.

37.1 The Mechanism

Because every pool boundary crossing publishes its exact amount, the net value inside each pool is publicly computable: sum every inbound crossing, subtract every outbound one. The result is the pool's total balance, known to everyone, with no knowledge of any individual note.

Consensus enforces that this total never goes negative. A pool cannot pay out more than has ever been paid in, regardless of what proofs are presented.

37.2 Why This Matters

It means inflation is **detectable** even though holdings are hidden. If a soundness bug allowed counterfeit notes, the counterfeit value could not leave the pool beyond the pool's true total without the turnstile rejecting the transaction. The bug would be contained rather than catastrophic, and the containment is a simple public arithmetic check rather than a cryptographic assumption.

This is a genuinely important result and it is the strongest available answer to the auditability objection. **Aggregate supply is public; individual holdings are private; the two are compatible.**

37.3 The Analogy, and Its Limit

The turnstile is exactly the pattern Chapter 26 describes for the orphaned asset layer: public issuance, private transfers. It is also exactly what the genesis anchor of Part VIII approximates: a public commitment to the total, with everything after it hidden.

The limit of the analogy is worth naming. The turnstile works because every boundary crossing is forced through a public field that consensus checks. Part VIII's genesis anchor is checked by nobody — it is a commitment in a data output that no validator interprets. An issuer who anchors two manifests produces two valid-looking collections, and only the social convention that a collection *is* its genesis transaction identifier distinguishes them.

That convention is not nothing. It is the same convention by which a token is identified by its contract address on other chains, and it has proven durable. But it is a convention, and the turnstile is a consensus rule, and the difference is the difference between Chapter 19's three architectures.

38. Proving, and What It Costs in Practice

Every claim in this document about what a transaction can contain is bounded by what a proof can be generated for, on real hardware, in acceptable time. This chapter supplies the numbers, because a design that requires a minute of proving per transfer is a different product from one that requires a second.

38.1 The Proving System

The shielded pool uses a proof system with no trusted setup. This is a substantive security property rather than an aesthetic preference: a trusted setup is a ceremony whose compromise permits undetectable counterfeiting forever, and the chain's earlier pools required one. Removing it removed a permanent contingent liability.

The system also supports recursive composition, which is the property that would permit proof aggregation across transactions. That capability is not used in the current design and is the most plausible route to the public-verifiability circuit described in §23.3.

38.2 Where the Cost Goes

Proving cost is dominated by constraint count, and constraint count in an Action is dominated by the 32-level Merkle path — 32 invocations of the in-circuit hash, plus the nullifier derivation, the commitment derivation, and the value commitment.

The practical figures: proving is on the order of a second or two per Action on a modern CPU, memory usage is measured in hundreds of megabytes, and verification is fast enough to be irrelevant. A two-Action transfer is therefore a few seconds of work for the sender and negligible work for everyone else.

38.3 What This Means for Transfers

An item transfer under Part VIII's construction is an ordinary two-to-four Action shielded transaction. Its proving cost is the proving cost of a shielded payment, because it *is* a shielded payment — there is no additional circuit, no additional proof, and no additional constraint. The seal chain is read off a structure that the proof already enforces for its own reasons.

This is the strongest practical argument for the construction and it deserves to be stated in one sentence: **the item costs nothing to move that the money did not already cost.**

38.4 Verification Cost

A verifier checking a disclosure of length k hops performs, per hop, one key derivation, one trial decryption against a known Action, one commitment recomputation, one tree lookup, one nullifier computation, and one nullifier-set lookup. All are cheap; the tree and nullifier lookups dominate and are index reads against a node the verifier already runs.

Verification is therefore linear in the number of hops and independent of the chain's length. A ten-hop item verifies in milliseconds. This compares favourably with the mature client-side-validation systems, where the verifier must additionally validate anchoring transactions and their confirmations because it is not assumed to hold the chain.

38.5 The Honest Caveat

These figures are for the reference implementation on desktop hardware. Mobile proving is materially slower and is the practical bottleneck for every shielded wallet on the chain, which is a product problem rather than a protocol one — and one that the construction of Part VIII inherits in full, since it produces ordinary shielded transactions and cannot be faster than they are.

Part XIII — The Adversary's View

A privacy claim is only as good as the adversary it was tested against. This Part constructs the adversary explicitly, gives them the strongest reasonable capabilities, and works out what they learn. Where they learn something, it is stated. Where a mitigation exists, it is given. Where none exists, that is said too.

39. The Observer Model

39.1 Capabilities Granted

The adversary is assumed to have:

- **Every block, forever.** They run a node, have run one since genesis, and retain everything.
- **Perfect network vision.** They see every transaction at the moment it is broadcast, with the peer that first announced it. This is stronger than any real adversary and is granted because network-level deanonymisation is a separate discipline with its own literature and should not be smuggled into a protocol argument.
- **All public marketplace data.** Listing prices, timestamps, collection manifests, everything a venue publishes.
- **Unlimited computation, bounded by cryptographic assumptions.** They cannot break the proof system, invert the hashes, or solve discrete logarithms. They can do anything else.
- **The ability to participate.** They can buy items, sell items, and thereby receive disclosures legitimately.

They are not assumed to compromise endpoints, coerce parties, or subvert the marketplace's servers. Those are real threats and they are out of scope for a protocol argument; they are addressed in Chapter 42.

39.2 What the Chain Shows Them

For an item transfer under Part VIII's construction, the chain shows a shielded transaction. Concretely:

- Some number of Actions, typically two to four
- A net value balance of zero
- An anchor naming a recent tree root
- A set of nullifiers, none linkable to any commitment
- A set of commitments, none openable
- Fixed-size ciphertexts revealing nothing by their size
- A fee, determined by the Action count

There is no field that says an item moved. There is no field that says which item. There is no field that names either party. **The transaction is bit-for-bit indistinguishable in structure from a shielded payment**, because it is one.

39.3 What They Can Infer Anyway

Three things, and they are the honest residue.

Timing. If a venue publishes that item 42 sold at 14:02, and the adversary looks at shielded transactions confirmed near 14:02, they have a candidate set. Its size is the number of shielded

transactions in the window, which on this chain is not large. This is the strongest attack available and it is a *venue* leak, not a protocol leak — it exists because the marketplace published the sale.

Action count. A transfer needs an Action for the item and Actions for the payment. If typical shielded payments in the same period use two Actions and item transfers use four, the count is a weak distinguisher. Mitigated by padding with dummies, which the protocol supports precisely for this.

Fee. Determined by Action count, so it carries the same information and no more.

39.4 The Bound

Given the capabilities above, the adversary's knowledge of who holds item 42 is bounded by the set of parties who could have been involved in some shielded transaction near a published sale time. It is not zero. It is also not a chain of linked transfers — each sale produces an independent candidate set, and the sets do not intersect in any way the adversary can compute, because there is no on-chain link between one transfer and the next.

This is the substantive claim: the adversary gets per-event uncertainty that does not compound into a history. A transparent system gives a complete history from one observation. This gives a weak guess per sale, with no way to chain the guesses.

40. Attacks That Do Not Work, and Why

Working through the failed attacks is more informative than listing the surviving properties, because each failure identifies the specific mechanism doing the work.

40.1 Following the Item

The attack. Find the item's transactions on chain and follow them, as one would follow a token.

Why it fails. There is nothing to follow. The link between one hop and the next is the equality $p = nf$, and both p and the note containing it are inside a commitment the adversary cannot open. They see nullifiers and commitments; the correspondence between them requires nk .

The mechanism. Nullifier unlinkability (§34.2).

40.2 Presenting a Stale Proof

The attack. A seller who already sold item 42 shows a buyer the disclosure proving they once held it, and sells it again.

Why it fails. The verification in Chapter 22 does not check "did the seller receive this." It checks "is the seller's nullifier absent from the chain." A seller who has already sold has published that nullifier, and the buyer computes it themselves from the disclosed nk and finds it.

The mechanism. Full-viewing-key disclosure conveying the spend-detection capability (§31.2). A disclosure conveying only ivk would not defeat this attack, which is the concrete reason the disclosure is 96 bytes and not 32.

40.3 Forging a Successor

The attack. Construct a note with p equal to the victim's nullifier, and claim it as the successor.

Why it fails. Three exhaustive cases (§34.4): same-bundle collision is rejected as a duplicate nullifier, later collision is rejected as a duplicate nullifier, and a fabricated dummy cannot be steered to the target value without inverting the nullifier hash.

The mechanism. The in-circuit constraint that p is the spent note's nullifier, plus the consensus nullifier set.

40.4 Splitting the Item

The attack. Spend the item note in an Action producing two outputs, and claim both are the item.

Why it fails. An Action produces exactly one output. To produce two notes, an attacker needs two Actions, which requires two spends, which requires two distinct nullifiers — and only one of them is the item note's. The second output's p is some other nullifier and fails the verifier's equality check.

The mechanism. The one-in-one-out Action structure (§30.1).

40.5 Substituting the Manifest

The attack. The issuer publishes a second manifest binding item 42 to a different note, and sells both.

Why it fails — partially. It does not fully fail, and this is the honest limitation. The mitigation is that a collection's identity *is* its genesis transaction identifier: two manifests are two collections, distinguishable by every buyer who checks which genesis they are buying into. An issuer who does this has not double-sold item 42; they have created a second collection that also contains an item numbered 42.

The residue. A buyer who does not check the genesis identifier, or who is shown a manifest out of band, can be defrauded. This is a user-interface problem with a user-interface answer — the genesis identifier must be displayed and checked — and it is the same problem every token system on every chain has with contract addresses.

40.6 Retiring Somebody Else's Item

The attack. Compute and publish the nullifier of a victim's item note, destroying it.

Why it fails. Producing a valid nullifier requires a valid spend proof, which requires the spending key. Knowledge of `nk` alone — which every prior holder has, from the disclosure — permits *computing* the nullifier value but not *publishing* it in a transaction, because publication requires a proof of a legitimate spend.

The mechanism. The separation of `nk` from `ask` (§31.1). This is worth dwelling on: the disclosure chain hands every subsequent holder the ability to watch for the spend and not the ability to perform it. If the hierarchy did not separate these, the construction would hand every buyer the ability to steal from every prior seller.

40.7 Correlating Across Items

The attack. A holder owns five items. Link them to each other, then to an identity.

Why it fails, if the implementation is correct. Each item lives in its own account derived from the seed at a distinct index. The accounts share no public material. Transfers are separate transactions.

Why it fails otherwise. If an implementation reuses one account for multiple items, disclosing that account for one sale discloses every note it ever received, including the other four items. This is the failure mode flagged in §31.3, and it converts a strong privacy property into none at all.

The mechanism. One account per item, enforced by implementation discipline rather than by the protocol. It belongs in a test.

41. The Leak That Remains

One leak survives every mitigation and it should be stated as plainly as possible rather than buried in a limitations list.

41.1 The Statement

Every holder of an item learns the complete prior ownership history of that item, permanently and irrevocably.

To verify hop k , the buyer needs the viewing keys for hops 1 through k . There is no way to prove the chain's integrity while withholding part of it, because the integrity *is* the chain. Once disclosed, a key cannot be un-disclosed.

41.2 What "History" Means Here

It means the sequence of accounts, not the sequence of legal identities. A buyer learns that item 42 passed through six one-time accounts and learns the viewing keys of all six. Since each account holds exactly one note and nothing else, the buyer learns nothing about any prior holder's other activity.

This distinction is the reason the one-account-per-item rule is load-bearing rather than hygienic. With it, the leak is "six anonymous hops occurred." Without it, the leak is "here is every payment every prior holder ever received."

41.3 When It Matters

For a collection where provenance is a feature — where buyers *want* to know an item passed through a notable holder — the leak is the product.

For anything where prior ownership is sensitive, it is disqualifying, and this document does not recommend the construction for that case. The only fix is consensus-level assets (§26.4), and the timeline for those is unbounded (Chapter 25).

41.4 The Partial Mitigations, and Their Limits

Re-anchoring. A holder could, in principle, burn the chain and re-issue from the genesis authority, resetting history. This requires the issuer's cooperation and destroys the provenance guarantee for everyone. It converts the system into a permissioned one.

Threshold disclosure. Disclose only the most recent n hops and rely on a reputation claim for the rest. This is not a cryptographic mitigation; it is a trust assumption wearing one, and it should not be presented as a feature.

Proof-of-chain circuits. A circuit proving "I hold a valid chain of length k from genesis" without disclosing the chain would close the leak entirely. This is the same circuit described in §23.3 and carries the same status: months of work plus an audit, not available today.

The honest position is that the leak is real, bounded to per-item history, and closed only by work that has not been done.

42. Threats Outside the Protocol

A protocol argument is not a security argument. The attacks that actually take people's assets do not break cryptography, and a document that stops at the cryptography is incomplete.

42.1 The Venue

A marketplace knows what it publishes and more. It knows who browsed, who bid, and — if it brokers disclosures — potentially the disclosures themselves.

The design rule is that **the venue must never be a necessary party to verification**. A buyer must be able to take a disclosure and check it against their own node with no venue involvement. If verification requires calling the venue's API, the venue is a trusted third party and the entire construction reduces to Chapter 19's first architecture with extra steps.

This is checkable and should be checked: an implementation whose verifier makes a network request to anything other than a node the user chose has failed.

42.2 Custody

Until the settlement question of Chapter 27 is answered, the venue holds funds during a trade. That is custody. It should be labelled custody in the user interface, and the amount at risk should be bounded by settling promptly rather than by pooling.

This document's own project has a documented open finding in this area, and the correct disclosure is that it is open rather than that it is managed.

42.3 Key Loss

An item is a note; a note is lost with its key. The construction's seed-derived accounts mean a seed backup recovers the *keys*, and the memo-stored consignments mean a seed backup can recover the *proofs*. Both are necessary; either alone is insufficient.

An implementation that stores consignments only locally has built RGB's recovery model while claiming better, and §29.4's advantage evaporates.

42.4 Accidental Spends

The hazard of §34.5 deserves repeating in an operational register: an item note that appears in a wallet's ordinary spendable balance will eventually be spent as change. The wallet must exclude item notes from coin selection, and this must be enforced at the coin-selection layer rather than by a user-interface warning.

The historical record on this chain includes inscriptions destroyed exactly this way by wallets that had no idea they were carrying anything.

42.5 Metadata Availability

The genesis manifest commits to metadata by digest. The digest is permanent; the metadata is not, unless somebody stores it. A collection whose images live on one host is a collection with a single point of failure that the cryptography does not address.

This is the one place where the byte-level permanence of inscriptions (§28.6) is straightforwardly better, and it should be conceded rather than argued around. The mitigation is content-addressed storage with multiple pinners, which is an operational commitment rather than a technical guarantee.

Part XIV — Building It

43. The Issuer's Procedure

This Part specifies the construction operationally, at the level of detail an implementer needs. It assumes the protocol material of Part XI.

43.1 Deriving the Item Accounts

From the issuer's seed, derive one account per item at consecutive indices under the hardened path of §31.4. Record the starting index and the count; these, plus the seed, are sufficient to regenerate every key in the collection.

Do not use random entropy. Do not reuse indices across collections. Both break recovery in ways that are silent until a restore is attempted.

43.2 Creating the Genesis Notes

For each item, send a minimal-value note to that item's account. The value should be the smallest the wallet will send — it is a carrier, not a payment, and every zatoshi placed in it is locked in the item forever.

These sends may be batched. A transaction with n Actions creates n item notes, and batching is preferable: it reduces fees, reduces the number of transactions the adversary of Chapter 39 can time-correlate, and produces a tighter genesis window.

Record, for each item, the note commitment. This is the item's seal identifier.

43.3 The Manifest

The manifest binds item identifiers to commitments and metadata digests:

```
item 0 → cm0 , digest(metadata0)
item 1 → cm1 , digest(metadata1)
...
```

Its own digest is what gets anchored. The manifest itself must be published somewhere retrievable — content-addressed storage is the obvious choice, and §42.5 applies.

Include in the manifest a collection name, the issuer's declared policy on further issuance, and the genesis height. None of these are enforced; all of them are what a buyer checks.

43.4 The Anchor

One transaction, one data output, containing the manifest digest. 32 bytes inside the 80-byte budget of §24.1, leaving room for a short protocol identifier.

The transaction identifier of this transaction is the collection's identity. Everything downstream refers to it. It should be displayed prominently by every interface, for the reason given in §40.5.

Anchor once. Do not anchor per item, per transfer, or per sale — every additional anchor is an additional public event, and the anonymity argument of Chapter 39 depends on there being none.

43.5 The Cost

For a collection of n items: one batched shielded transaction per group of items (fee proportional to Action count), plus one transparent anchor transaction (four logical actions, per \$24.3). For a hundred-item collection this is on the order of a few transactions and a fee measured in fractions of a coin.

44. The Transfer Procedure

44.1 What the Seller Does

Derive a fresh account for the buyer — or rather, receive one: the buyer supplies a fresh address, because the buyer must control the successor's spending key.

Construct a transaction that spends the item note and creates a note to the buyer's address **in the same Action**. This is the whole of the mechanism, and it is the step where an implementation is most likely to be subtly wrong: the wallet must be instructed to place these specific spends and outputs in the same Action, and not merely in the same transaction.

A wallet that places them in different Actions produces a valid shielded transfer in which the successor's p is some other note's nullifier, and the chain is broken. The item is not destroyed — the note exists and the buyer holds it — but it can no longer be proven to descend from genesis. **This is a silent failure.** It produces no error, and it is discovered only when a later buyer's verification fails.

An implementation must assert the Action placement and must test it.

44.2 What the Seller Discloses

The disclosure is a list, one entry per hop, each entry being the full viewing key $(ak, nk, rrvk)$ of that hop's holding account: 96 bytes each.

Plus the genesis reference: the collection's transaction identifier, the item index, and the manifest.

44.3 What the Buyer Checks

For hop 1, and then for each subsequent hop:

1. Derive ivk from the disclosed $(ak, nk, rrvk)$.
2. Confirm the account's address is consistent with ivk — this is what binds nk to the note rather than letting a seller substitute an unrelated key.
3. Trial-decrypt the named Action's ciphertext with ivk . It must succeed.
4. Recompute the note commitment from the decrypted plaintext. For hop 1, it must equal the manifest's cm for this item.
5. Confirm the commitment appears in the tree.
6. Compute the note's nullifier from nk .
7. For hops before the last: find that nullifier on chain, read the output of the Action that published it, and confirm it equals the next hop's commitment.
8. **For the last hop: confirm the nullifier is absent from the nullifier set.**

Step 8 is the present-tense check and the one that defeats §40.2. Steps 2 and 4 are the binding checks. Step 7 is the chain.

Every step is executed against the buyer's own node (§42.1).

44.4 Storing the Consignment

The buyer, now holder, must retain the disclosure and add their own hop to it before they can sell.

Store it in a memo addressed to the holder's own account. At 96 bytes per hop, a 512-byte memo holds five hops directly; longer chains require either a compressed encoding — storing spent accounts' derivation indices rather than keys, which is far smaller for a single-seed holder — or chunking across multiple self-addressed notes.

The self-addressed memo is what makes seed-only recovery work (§29.4), and it is the single feature distinguishing this construction from the mature alternatives. It should not be treated as optional.

45. Failure Modes, Ranked

An implementation checklist is more useful than a summary. These are ordered by expected damage.

- 1. Same-Action placement not enforced.** Breaks the chain silently at transfer time, discovered at the next sale. Damage: the item becomes unprovable. Test: construct a transfer, then verify it end to end.
- 2. Account reuse across items.** Converts the per-item history leak (\$41) into a full-activity leak. Damage: total privacy failure for the seller. Test: assert distinct derivation indices per item.
- 3. Item notes in the spendable set.** Item destroyed by an unrelated payment. Damage: total loss. Test: assert coin selection never returns an item note.
- 4. Consignment stored only locally.** Seed recovery yields keys without proofs. Damage: item unprovable after device loss. Test: restore from seed alone and verify.
- 5. Verifier calling the venue.** Reduces the construction to a trusted registry. Damage: architectural. Test: run the verifier with the venue unreachable.
- 6. Genesis identifier not displayed.** Enables the manifest substitution of \$40.5. Damage: buyer defrauded. Test: user interface review.
- 7. Random rather than seed-derived item accounts.** Unrecoverable. Damage: total loss on device failure. Test: restore from seed alone.
- 8. `ivk`-only disclosure.** Proves history, not present ownership. Damage: stale-proof attack succeeds. Test: sell an item, then attempt to verify with the pre-sale disclosure — it must fail.

Each of these produces working software that passes casual inspection. That is why they are listed.

Part XV — A Formal Treatment

The preceding Parts argued in prose. This Part restates the central claims as definitions and games, because a claim that cannot be stated as a game cannot be attacked properly, and a claim that has not been attacked properly should not be believed.

The treatment is deliberately modest in ambition: these are security definitions in the style used for practical protocols, not a full simulation-based analysis. Where an argument is a sketch rather than a proof, it says so.

46. Definitions

46.1 The Objects

A **collection** is a tuple (G, M) where G is a transaction identifier on the chain and M is a manifest. M maps item indices to pairs of a note commitment and a metadata digest. G 's transaction contains a data output whose contents include $\text{digest}(M)$.

An **item** is an index i together with the collection that contains it. Items from different collections are different items even if their indices coincide.

A **holding** of item i at chain state S is a note N such that a valid chain (defined next) exists from $M[i].\text{cm}$ to $\text{commitment}(N)$, and $\text{nullifier}(N) \notin \text{nullifiers}(S)$.

A **chain** from commitment c_0 to commitment c_k is a sequence of notes $N_0 \dots N_k$ with $\text{commitment}(N_0) = c_0$, $\text{commitment}(N_k) = c_k$, and for each j from 1 to k , an Action in the chain's history that spends N_{j-1} and creates N_j .

A **disclosure** for a chain is the sequence of full viewing keys for the accounts holding $N_0 \dots N_k$.

46.2 Verification

$\text{Verify}(\text{collection}, i, \text{disclosure}, S)$ returns accept if and only if every step of §44.3 succeeds against chain state S . Note that verification is a function of the chain state, so the same disclosure verifies at one height and fails at a later one — this is intentional and is the content of step 8.

46.3 The Properties

Uniqueness. At any chain state S , for any collection and item i , at most one note N is a holding.

Transfer integrity. If Verify accepts a disclosure ending at note N , then the party who controls N 's spending key is the unique current holder.

Ownership privacy. An adversary as defined in Chapter 39, without any disclosure for item i , cannot determine which party holds item i with probability better than their prior.

Anchoring. The set of items in a collection is fixed at G 's confirmation and cannot be extended.

47. Uniqueness

47.1 The Game

The adversary is given a collection and an item index, and wins if they produce two distinct notes $N \neq N'$ that are both holdings at the same chain state.

47.2 The Argument

Suppose the adversary wins. Both notes have valid chains from $M[i].cm$. Consider the first index j at which the two chains differ: $N_j \neq N'_j$ while $N_{\{j-1\}} = N'_{\{j-1\}}$.

By the definition of a chain, there is an Action spending $N_{\{j-1\}}$ and creating N_j , and another spending $N_{\{j-1\}}$ and creating N'_j . Both Actions publish $nullifier(N_{\{j-1\}})$, which is a deterministic function of the note (§34.2). Consensus maintains a set of published nullifiers and rejects duplicates, so at most one of these Actions is in the chain's history.

Therefore no such j exists, the chains are identical, and $N = N'$. ■

The argument reduces uniqueness to three protocol facts: nullifier determinism, the consensus duplicate check, and the circuit constraint that a created note's p is the spent note's nullifier — the last being what makes "the Action that spends $N_{\{j-1\}}$ " identifiable from N_j rather than requiring a separate binding.

47.3 What the Argument Assumes

It assumes the proof system is sound: an adversary who can produce a proof for a false statement can produce an Action spending a note they do not hold, or creating a note whose p is arbitrary, and uniqueness fails immediately.

It assumes the nullifier hash is collision resistant: two distinct notes with the same nullifier would let one spend retire two items.

It assumes no reorganisation deeper than finality. A reorganisation that removes the Action closing a seal restores the prior holding — which is correct behaviour, not a violation, since the transfer did not happen.

It does not assume anything about the issuer, the venue, the wallet software, or the verifier. Uniqueness is a consensus property. This is the strongest claim in the document and it is strong precisely because its assumption set is small.

48. Transfer Integrity

48.1 The Game

The adversary is given the ability to interact with honest parties as a buyer and a seller. They win if they induce an honest buyer to accept a disclosure for item i while the adversary is not the current holder.

48.2 The Cases

The adversary never held the item. Then no valid chain from `M[i].cm` reaches any note they control. To forge one they must produce a note whose p equals the nullifier of some note in the true chain, which is the forgery of §40.3, ruled out by the three cases of §34.4. Alternatively they must produce a disclosure whose keys decrypt an Action they do not control — ruled out by step 2 of §44.3, which binds the disclosed key to the address that received the note.

The adversary held the item and sold it. Then the final nullifier in their disclosure is published. Step 8 fails. The buyer rejects.

The adversary holds the item now. Then the claim is true and acceptance is correct.

The adversary holds the item now and sells it concurrently to two buyers. Both verifications succeed, because at verification time the adversary genuinely holds it. This is not a violation of transfer integrity as defined — the adversary *was* the holder — but it is a real attack on a marketplace, and it is the reason Chapter 27's settlement question matters. **Verification proves present ownership; it does not reserve it.** Only atomic settlement closes this gap, and until then the venue's custody is what closes it operationally.

48.3 The Honest Statement

Transfer integrity holds against a seller who does not hold the item. It does not, by itself, prevent a holder from selling twice in the window between verification and settlement. The construction is not an escrow and does not claim to be.

49. Ownership Privacy

49.1 The Game

The adversary picks two parties and an item. A challenger gives the item to one of them, chosen by coin flip, and the adversary sees the entire chain. They win by guessing which.

49.2 The Argument, and Its Limits

The transfer is a shielded transaction. Its public fields are enumerated in §30.2, and none is a function of the recipient: the nullifier depends on the *spent* note, the commitment is hiding, the ciphertext is fixed-size and encrypted to a key the adversary does not hold, the ephemeral key is fresh, the value balance is zero, and $\mathbf{rk}$ is re-randomised.

Given the pool's indistinguishability properties, the adversary's view is independent of the recipient, and their advantage is zero.

The limit is timing. The argument above is about a single transaction in isolation. In a real deployment, transactions occur at times, and a venue publishes sale times. The adversary's advantage is therefore not zero but bounded by the anonymity set of shielded transactions in the timing window — which is exactly the residue of §39.3 and is a property of the deployment rather than of the construction.

This is the correct place to be precise about what is being claimed. **The protocol leaks nothing.** **The deployment leaks timing.** A venue that batched settlements, or published sales on a delay, or did not publish them at all, would shrink the residue; a venue that publishes in real time hands the adversary their only edge.

49.3 The Prior-Holder Exception

The game above assumes the adversary holds no disclosure. An adversary who has legitimately held the item holds one, and for them the game is trivially won for every hop up to their own.

This is §41 restated formally, and it is why the privacy claim must always be qualified as "against parties who have never held the item."

50. Anchoring

50.1 The Claim

The set of items is fixed at genesis.

50.2 The Argument

`digest(M)` is in a confirmed transaction. `M` binds indices to commitments. A note not committed in `M` has no chain from any `M[i].cm`, so it is not a holding of any item in the collection.

An issuer wishing to add items must publish a new manifest and a new anchor, producing a transaction identifier different from `G`. Since a collection is defined as the pair `(G, M)`, this is a different collection.

50.3 The Residue

The argument is airtight given that buyers check `G`. It says nothing about buyers who do not.

This is a genuine weakness relative to a consensus-enforced asset layer, where supply is checked by validators rather than by users, and §37.3 states the comparison. The mitigation is interface discipline, and interface discipline has a poor historical record.

Part XVI — Economics

51. What Each Operation Costs

A design's feasibility is a fee schedule question. This chapter gives the arithmetic.

51.1 The Fee Model

Fees are proportional to a count of logical actions with a floor, as set out in §24.3. Transparent components are priced by serialized size rather than by count, which is what makes data outputs comparatively expensive.

For shielded operations the count is the number of Actions, with the floor meaning that a single-Action transaction pays the two-Action price. **This makes the second Action free**, which is the economic reason dummy padding costs nothing and therefore the economic reason the privacy mechanism of §30.1 is actually used rather than merely available.

51.2 Issuance

A batched issuance creating n item notes uses roughly n Actions plus change, and pays proportionally. A hundred items across, say, four transactions of twenty-five Actions each pays four times the twenty-five-Action fee.

The anchor is one transparent transaction with a data output: four logical actions by the accounting of §24.3.

For a hundred-item collection the total is on the order of a hundred logical actions plus four — measured in hundredths of a coin at the current schedule. **Issuance is not a cost centre.**

51.3 Transfer

Two to four Actions. Identical to a shielded payment of the same shape, because it is one.

This is the number that matters for a marketplace, and it is the answer to the most common objection to client-side-validation systems: that they add overhead per trade. Here the overhead is zero, because the seal transition is a side effect of a payment that would have happened anyway.

51.4 Verification

Free. It is local computation against a node the verifier runs.

Contrast with the mature alternatives, where a verifier who does not run a node must fetch and validate anchoring transactions and their confirmations, and where the proof size per hop is one to two orders of magnitude larger (§29.5).

51.5 Storage

The consignment grows at 96 bytes per hop, or considerably less under the index-based encoding of §44.4. A twenty-hop item's consignment fits in four self-addressed memos at the

naive encoding and in one at the compressed one.

The storage cost is paid in memo space, which \$24.3 established is pre-paid. Writing a consignment into a memo attached to a note the holder was creating anyway costs nothing.

52. The Venue's Economics

52.1 The Fee

The venue takes a percentage of each sale, deducted from the seller's proceeds. This is the standard arrangement and requires no protocol support: the venue receives the payment and forwards the remainder.

It does require custody for the duration, which is §42.2's concession.

52.2 Cross-Asset Payment

A buyer paying in a different asset routes through an external conversion service that delivers shielded value to a deposit address the venue controls, tagged to the order.

Two facts determine the design. **First, no memo survives conversion** — the intermediary's output is its own transaction, and the memo field of the original payment does not travel. Order matching must therefore key on the deposit address itself, which the conversion service makes unique per order. **Second, the conversion can be quoted in the exact output amount**, so the venue receives precisely the listing price and the buyer absorbs the conversion variance.

The venue's percentage is taken from the received amount as normal. Layering an additional fee at the conversion service would charge twice.

52.3 Refunds

A bid funded through conversion and later withdrawn refunds in the settled asset, not the original one. This is a consequence of the conversion being one-way and irreversible, and it must be stated to the user before they commit — a refund in a different asset than was paid is a surprise that generates support load and, reasonably, anger.

53. Why the Numbers Favour This Design

Collecting the comparisons:

	Per-transfer chain cost	Per-transfer proof	Verifier needs	Seed recovery
Consensus-level assets	one transaction	none	node	yes
This construction	one transaction	96 B/hop	node	yes
Taproot Assets	one transaction	0.4–1.9 KB/hop	proof chain + anchors	no
RGB	one transaction	~1 KB+/hop	consignment	no
Inscriptions	two transactions	none	indexer	yes

The row worth arguing about is the last. Inscriptions require commit and reveal — two transactions, two fees — and their verifier requirement is an indexer, which is a trusted third party in everything but name (§28.1, §28.5).

The construction of Part VIII costs one transaction, requires no indexer, and recovers from seed. Its cost is the disclosure leak of \$41 and the absence of public lookup of \$23. Whether that trade is right depends entirely on what is being built, and this document’s position is that for a privacy chain it is the only trade available that does not abandon the chain’s premise.

Part XVII — The Record on This Chain

54. What Was Actually Deployed

Before this document's construction, one asset system was deployed on this chain and ran at meaningful volume. It deserves examination, because its design choices were forced by the same constraints analysed in Part VIII and its failures are instructive.

54.1 The Design

Inscriptions carried in the *input script* of a spending transaction, not in a data output. The reason is capacity: the data-carrier limit is 83 bytes (§24.1) while the input script's standardness limit is roughly twenty times that. For content of any kind, the input script is the only public channel on the chain large enough.

Like Ordinals, this requires commit and reveal, because a script only appears when it is spent.

54.2 The Defect

Input scripts are excluded from the transaction identifier and are covered by no signature.

The consequence is that an inscription's content is **malleable in flight**. A third party observing a reveal transaction in the mempool can, in principle, alter the script and rebroadcast, producing a transaction with the same identifier obligations and different content. The inscription's binding to its transaction is therefore weaker than it appears.

This is not a theoretical concern about a hypothetical system. It is a structural property of the only channel large enough to use, and it means that content-carrying on this chain's transparent side is fundamentally less sound than on a chain with a witness commitment.

54.3 The Volume, and the Shielded Fraction

Over a hundred thousand inscriptions were created. A meaningful fraction — roughly one in seven — were funded from shielded value in a way that makes the creator untrackable by ordinary analysis.

That fraction is the most interesting number in the record and it is rarely commented on. It says that a substantial minority of participants wanted the *asset* to be public and the *funder* to be private, and that they achieved it by crossing the pool boundary. Chapter 36 explains why this is weaker than it looks: value fingerprints survive simple deshielding operations at a high rate, and a published listing price gives an analyst the target value to search for.

But the demand signal stands on its own. **People used a privacy chain's transparent side to make public assets, and paid extra to hide where the money came from.** The construction of Part VIII is, in one reading, the answer to what those users were reaching for.

54.4 The Losses

Inscriptions on this chain were destroyed by wallets that spent them as ordinary change. This is the hazard of \$34.5 and \$42.4, realised.

It is worth being precise about why: the wallets were not defective. They correctly identified spendable value and spent it. The defect was that an asset had been placed in a container the wallet had no way to recognise as special — a pure convention problem, and exactly the failure mode Chapter 19 attributes to the first architecture.

The construction of Part VIII inherits this hazard completely and mitigates it the same way: the wallet must be taught. The difference is that item notes live in their own accounts (§31.3), which makes exclusion a matter of account policy rather than per-note tracking, and account policy is considerably harder to get wrong.

54.5 The Forgotten Precedent

A decade before any of this, content addressing identifiers were being embedded in this chain's transactions — the same idea, without the protocol around it.

The precedent matters because it establishes that carrying references on this chain was never the hard part. Carrying them in a way that survives, that is recognised, that cannot be forged, and that does not destroy privacy — that is the hard part, and it took ten years for anybody to state the problem in those terms.

55. What the Record Implies

Three conclusions follow from §54, and they shaped every decision in Part VIII.

The transparent channel is too small and too weak. 83 bytes in a data output, or 1,650 malleable bytes in an input script. Neither supports content. Both support a digest, which is why the construction anchors a digest and stores content elsewhere.

Demand for private funding is real and was expressed with money. One in seven users paid the pool-crossing cost to obtain a weaker version of what Part VIII provides by default.

Convention-based ownership loses assets. Not hypothetically — on this chain, in production, to wallets doing their jobs correctly.

Part XVIII — Content, and Its Binding

An item is not only a seal. It is a seal plus something the seal is *about*, and the binding between them is where most asset systems are weakest. This Part treats the binding with the same seriousness as the seal, because an unforgeable claim to a mutable object is not worth much.

56. What Is Being Committed

56.1 The Layers

Three distinct things are commonly conflated, and separating them is most of the work:

The content. The bytes of the image, the model, the document. Large, and not on chain under any design considered here.

The metadata. The structured description: name, traits, the content's own digest, whatever the collection's schema defines. Small, and also not on chain, though it could be.

The commitment. A digest of the metadata, which transitively commits to the content because the metadata contains the content's digest. 32 bytes, and this is what goes in the manifest.

The chain of commitment is therefore: anchor → manifest digest → manifest → per-item metadata digest → metadata → content digest → content. Each link is a hash, and the whole chain is as strong as its weakest hash and as available as its least-available layer.

56.2 Why Not Put the Metadata On Chain

For a small collection it would fit: a hundred items with compact metadata is a few kilobytes, which is a handful of data outputs at 80 bytes each, or one transaction under the memo-bundle proposal of §24.5.

It is nonetheless the wrong choice, for the reason that runs through this entire document. **Every byte published is a byte an adversary correlates.** Publishing per-item metadata publishes the collection's shape, its size, its trait distribution — all of which are join keys against the timing residue of §39.3. A manifest published off chain and committed by one digest publishes exactly one thing: that a collection exists.

The exception is a collection that *wants* to be public, where the metadata is marketing. For those, publishing is free and the analysis above does not apply.

56.3 Determinism

The digest must be reproducible. Two parties hashing the same logical metadata must get the same 32 bytes, or verification fails for reasons no one can debug.

This is harder than it sounds and it is where implementations routinely fail. Serialisation must fix key ordering, number representation, string normalisation, and whitespace. A metadata format that permits two encodings of the same content permits two digests, and a buyer who re-serialises before hashing gets a mismatch.

The rule: hash the bytes as published, not the parsed object. The manifest names a digest of a byte string; the verifier fetches those bytes and hashes them. Parsing happens after verification, never before. This eliminates the entire class of canonicalisation bugs by refusing to canonicalise.

57. Storage and Availability

57.1 Content Addressing

Content should be stored under an identifier derived from its own hash, so that retrieval and verification are the same operation. A holder who fetches content by digest either gets the right content or detects that they did not.

This does not solve availability. It solves *integrity* — nobody can serve altered content — and integrity without availability is a verifiable inability to load the image.

57.2 Availability Is an Operational Commitment

Somebody must keep the bytes. There is no cryptographic construction that makes data available; there is only somebody paying for storage.

The honest framing is that the commitment chain of §56.1 guarantees that *if* the content is retrievable, it is the right content, and guarantees nothing about retrievability. A collection whose content lives on one host is a collection with a single point of failure, and the whitepaper of a project that says otherwise is lying.

Practical mitigations, in descending order of effectiveness: multiple independent pinning parties; holder-side caching, so that every holder is a replica; inclusion of small content directly in the metadata for collections where that is feasible; and, at the limit, on-chain content for collections small enough to afford it.

Holder-side caching is the underrated one. A holder who has verified an item has, by construction, fetched its content. A client that retains it turns every sale into a replication event, and a collection with a hundred holders has a hundred copies. This costs nothing and should be default behaviour.

57.3 The Comparison, Conceded

An inscription stores content in the block. It is retrievable as long as the chain exists, with no additional party, no pinning, and no operational commitment.

This is straightforwardly better on the availability axis and the comparison should be conceded rather than argued around. The construction of Part VIII trades permanent availability for privacy and for the ability to carry content larger than 83 bytes. Both of those are necessary here and neither is free.

58. Mutable Metadata, and Why It Is Excluded

A recurring product request is metadata that can be updated — levelling up an item, changing its appearance, correcting an error.

58.1 The Mechanism That Would Work

The manifest could commit to a *policy* rather than a value: an item's metadata is whatever the issuer most recently signed, and the manifest commits to the issuer's key. Updates are signed statements, distributed out of band, ordered by a sequence number.

58.2 Why It Is Not Adopted Here

It reintroduces the mutable registry that Chapter 19 identifies as the defining weakness of the first architecture. An item whose metadata is whatever the issuer last said is an item the issuer controls, and the anchoring property of §50 becomes decorative: the set of items is fixed, but what they *are* is not.

More concretely, it creates an availability requirement with teeth. Under a fixed manifest, a holder who has the content needs nothing further, forever. Under a signed-update policy, a holder must know whether a newer update exists, which requires either an online oracle or an unbounded search.

The position taken here is that items are immutable and that mutability belongs to a layer above — a venue can display whatever it likes alongside an item, and that display is venue data, clearly labelled, not part of the item. This is less flexible and it is honest about where the trust sits.

Part XIX — Infrastructure

59. What the Wallet Must Do

The construction of Part VIII places unusual demands on wallet software, and every one of them is a place where a correct-looking implementation is wrong. This chapter enumerates them.

59.1 Account Management

Item accounts are derived at distinct hardened indices from the same seed as the spending account (§31.4, §43.1). The wallet must:

- Allocate indices deterministically and record the allocation
- Never reuse an index across items
- Derive, not randomise
- Persist the index-to-item mapping, and be able to rebuild it from the manifest plus the seed

The last point is what makes recovery work: given a seed and a collection manifest, a restoring wallet derives each candidate account, computes the address, and checks which manifest commitment it can decrypt. No external state required.

59.2 Scanning

Naive scanning is $O(\text{chain} \times \text{accounts})$ and is unusable past a handful of items (§32.2).

The fix is birthday tracking. Each item account has a creation height; it cannot hold anything earlier. A wallet scans the main account from the wallet's birthday and each item account from its own, which is typically recent.

A second optimisation: an item account holds exactly one note at a time and the wallet knows which Action created it. After the initial location, the account needs no scanning at all — only a nullifier-set check to detect that the item left.

In steady state, holding a hundred items costs a hundred nullifier lookups per block, which is an index read, not a decryption. This is the difference between a usable implementation and an unusable one.

59.3 Coin Selection

Item notes must be excluded from ordinary spending (§42.4). The exclusion belongs in coin selection, structurally, not in a confirmation dialog.

The one-account-per-item rule makes this tractable: coin selection operates on the spending account and simply never sees item accounts. An implementation that pools all notes into one selection set and then filters is one bug away from total loss; an implementation that never pools them cannot make the mistake.

59.4 Action Placement

The wallet must be able to construct a transaction with a specified spend and a specified output *in the same Action* (§44.1).

This is not a normal wallet capability. Ordinary wallets treat Action assignment as an internal detail and are free to shuffle. A wallet supporting this construction must expose the constraint down to the transaction builder, and must assert it on the built transaction before signing.

The assertion is mandatory. The failure is silent (§45, item 1), and a silent failure in a construction whose entire value is provability is the worst available outcome.

59.5 Consignment Handling

On receipt: verify, then store. On send: append own hop, transmit.

Storage is a self-addressed memo (§44.4). The wallet must write the consignment into the memo of a note it sends to itself — which means an item transfer is not one Action but a small transaction that also carries the holder's bookkeeping.

This is the one place where the construction costs more than a bare payment, and the cost is one additional Action, which §51.1 established is free under the fee floor when the transaction would otherwise be small.

60. What the Indexer Must Do

60.1 The Scope

There is no public ownership index and there cannot be (Chapter 23). What an indexer *can* provide is everything that is not ownership:

- Collection discovery: which anchors exist, what manifests they commit to
- Manifest hosting and content mirroring (§57.2)
- Listings, bids, sale history — venue data, not chain data
- Nullifier-set queries, as a convenience for light clients

The last is delicate. A light client asking an indexer “is this nullifier spent” reveals which nullifier it cares about, which for a verifier reveals which item they are checking. A privacy-preserving version requires private information retrieval or, more practically, batching the query with decoys.

The honest default is that verification requires a full node. The light-client story is a real gap and should be labelled one rather than papered over with an API.

60.2 What It Must Not Do

It must not be required for verification (§42.1). A verifier must be able to check a disclosure against a node of the user’s choosing with the indexer switched off.

This is testable and should be a test.

61. Settlement Infrastructure

Given Chapter 27's unresolved atomicity, the venue holds funds between payment and delivery. This chapter specifies the discipline that bounds the resulting risk.

61.1 The Window

Custody begins when the buyer's payment confirms and ends when the seller's proceeds are sent. Everything in the design should shorten this window.

Concretely: settle on confirmation rather than on a batch schedule; do not pool custodial funds with operating funds; and publish the custody model rather than calling it escrow.

61.2 The Race

§48.2's fourth case — a holder selling twice inside the verification window — is closed operationally by the venue refusing to list an item whose current nullifier check has changed since listing, and by re-running step 8 of §44.3 immediately before settling.

This does not close it cryptographically. Nothing does, short of atomic settlement.

61.3 What Would Replace This

Successful co-construction of a single transaction spending both sides (§27.2) eliminates custody entirely. This is the highest-value open item in the design and it is an experiment, not a research programme: build the transaction, try to sign it from two parties, see whether the proof can be produced without either party holding the other's key.

Until that experiment is run, every description of this system's settlement should use the word custody.

Part XX — What Would Change This

62. The Three Upgrades That Matter

This document has repeatedly deferred to specifications that are drafted and not shipped. Collecting them makes the roadmap explicit and makes it possible to say precisely what each one would buy.

62.1 Consensus-Level Assets

What it closes: the disclosure leak of §41, entirely. Public supply auditability (§26.2). Elimination of consignments, and therefore of the storage and loss hazards of §45 items 4 and 7.

Status: structurally orphaned (Chapter 25). Unbounded timeline.

What this document does instead: the construction of Part VIII, which achieves the four properties of §46.3 and concedes §41.

62.2 Memo Bundles

What it closes: the 512-byte consignment ceiling (§44.4), by raising capacity roughly thirty-fold. More importantly, the publicly-decryptable key value would permit a public commitment to ride inside a shielded transaction, removing the need for a transparent anchor and therefore removing the only public event in the entire lifecycle (§24.5).

Status: draft, dependent on a transaction format that is itself a draft.

What this document does instead: chunking across multiple self-addressed memos, and one transparent anchor at genesis.

62.3 Payment-Conditional Transfer

What it closes: custody (Chapter 61). Atomic settlement without a trusted party.

Status: draft, same dependency.

What this document does instead: venue custody, with the discipline of §61.1, and the co-construction experiment of §61.3 as the near-term hope.

63. What Would Not Change This

Two commonly-proposed changes would not help, and saying so saves effort.

A larger data-carrier limit. Raising 83 bytes to something larger would make transparent inscriptions viable on this chain and would not affect the construction of Part VIII at all, which uses 32 of the existing 80 bytes once. The construction is not capacity-constrained on the transparent side.

A public ownership index. This is the thing that cannot be built (Chapter 23) rather than the thing that has not been built. A proposal to add one is a proposal to either destroy the privacy or add the circuit of §23.3 — and if the latter, it should be described as the circuit rather than as an index.

64. The Circuit That Would Close the Gap

§23.3 named it; this chapter specifies it, because it is the one piece of new cryptography this document argues is worth building.

64.1 The Statement

Prove, in zero knowledge: “I know a chain of notes from the commitment c_0 to a note whose nullifier is absent from the nullifier set at height h , where each link is an Action in the chain’s history, and c_0 is committed in manifest M .”

Public inputs: c_0 , M ’s digest, h , the tree root and nullifier-set accumulator at h . Private: the chain.

64.2 What It Buys

A holder could prove present ownership to anybody, with no disclosure. This closes §41 completely, without consensus changes, and it does so for the disclosure-based construction rather than requiring an asset layer.

It also makes public ownership *queryable in one direction*: a holder can publish such a proof if they want to, and nobody can produce one falsely. That is not the public index of Chapter 23 — nobody can look up an item’s holder — but it is voluntary public provenance, which is most of what markets actually use a public index for.

64.3 What It Costs

The statement is a repeated application of the Action circuit’s nullifier-integrity and commitment-derivation constraints, plus Merkle paths, plus a non-membership argument against the nullifier set. The per-hop cost is comparable to an Action; a k -hop chain is roughly k Actions of proving, which for realistic k is seconds.

Recursion would flatten this to constant size, and the proving system supports it. That is the version worth building and it is meaningfully harder.

The non-membership argument is the genuinely novel part. The nullifier set is not currently accumulated in a form that supports succinct non-membership, which means either an accumulator would have to be maintained off chain by the prover — with the verifier trusting its construction, which defeats the purpose — or the statement must be weakened to membership of a spent-set the prover maintains. **This is the unsolved piece, and it should be named as such rather than assumed away.**

64.4 Precedent

Two pieces of existing work establish that circuits over this pool’s structures are buildable outside consensus: an independent proof-of-assets implementation, and a draft balance-proof

specification whose public inputs include a domain-separated nullifier and which proves notes exist in the commitment tree and not in the nullifier set at a height.

The second is close enough to \$64.1 to be a starting point and far enough to be months of work. Neither is audited. Both are single-pool and neither contemplates chains of notes.

Part XXI — What the Inscription Ecosystem Learned

Chapter 28 dissected the inscription envelope. The protocol that grew on top of it is more interesting than the envelope, because it is a decade-compressed experiment in what people actually want from on-chain assets when nobody is enforcing anything. This Part reads that experiment for lessons that transfer.

65. Sat Theory, and the Problem It Solves

65.1 The Idea

The base unit of the currency is tracked individually. Every unit ever mined has an ordinal number determined by its mining order, and a first-in-first-out rule assigns units to outputs as they move. An inscription attaches to a specific unit, and the unit's movement is the asset's movement.

65.2 Why It Is Necessary

Without it, an inscription is attached to an *output*, and outputs are spent in their entirety. When an output carrying an inscription is spent into three outputs, which one carries the inscription? Any answer is a convention; sat theory supplies a principled one by tracking a unit rather than a container.

65.3 Why It Is Also a Weakness

The tracking is entirely off chain. No validator computes ordinal numbers, no consensus rule references them, and the first-in-first-out assignment is a convention that an indexer implements. Two indexers implementing it differently produce different asset locations — and the Jubilee (§28.5) demonstrates that this is not hypothetical.

The relevant comparison to Part VIII's construction is sharp. Both systems face the "which successor" question. Inscriptions answer it with a convention over a quantity the chain does not compute. The construction of Part VIII answers it with the field p , which the chain *does* compute, in circuit, for its own reasons (§34.3).

This is the single clearest illustration of the difference between a convention and a constraint, and it is worth stating in those terms: the inscription protocol had to invent sat theory because Bitcoin gave it no successor pointer; this construction did not, because the shielded pool already had one.

65.4 Rarity, and What It Reveals About Demand

Sat theory also enables a rarity taxonomy: units mined at halvings, at difficulty adjustments, at cycle boundaries. These carry premiums.

The taxonomy is pure convention — the chain assigns no significance to any unit — and it is nonetheless priced. What that demonstrates is that **an asset's value can attach to a property no system enforces, provided everyone agrees on how to compute it.** That is a genuine market fact and it cuts against this document's general preference for constraints over conventions: markets have repeatedly shown they will accept conventions when the convention is cheap to check and widely implemented.

The counter is the one from §54.4: conventions lose assets when software that does not know about them handles the containers. Both things are true.

66. Provenance, Parents, and Delegates

66.1 Parent–Child

An inscription can declare a parent, and the declaration is valid only if the parent’s owner authorises it by participating in the transaction. This produces verifiable collection membership: a collection’s items are the children of a single parent inscription, and nobody else can add to it.

This is the same problem the manifest of §43.3 solves, with the opposite trade. The parent–child mechanism is enforced at inscription time and requires an on-chain event per item; the manifest is enforced by a single anchor and requires buyers to check the genesis identifier (§50.3).

The parent–child version is stronger. It is also more public, more expensive, and impossible on a chain without a scripting system to express the authorisation. Under the construction of Part VIII, a per-item on-chain event would defeat the privacy argument of Chapter 39 outright.

66.2 Delegation

An inscription can point at another inscription’s content rather than carrying its own, so that a thousand items sharing one image cost one image’s worth of block space.

This is a deduplication mechanism and the analogue under Part VIII’s construction is free: the manifest maps items to metadata digests, and a thousand items may name the same digest. No mechanism required, because content is not on chain in the first place.

66.3 Recursion

An inscription can reference other inscriptions’ content at render time, composing outputs from on-chain parts. Generative collections are built this way: one inscription carrying the generator, thousands carrying only parameters.

The interesting property is that recursion makes the *content* a program whose inputs are other on-chain content, which means the rendered result is deterministic and reproducible from chain data alone. That is a genuinely strong availability story (§57.3) and has no analogue in any off-chain-content design.

It is also a dependency graph with no versioning, in which the deletion of a referenced inscription is impossible but the *misinterpretation* of one is a software release away.

67. Fungible Tokens on an Asset Protocol

67.1 The Construction

A convention grew for representing fungible tokens as inscriptions carrying text: a deploy inscription declaring a ticker and a supply, mint inscriptions claiming portions, transfer inscriptions moving balances. Balances live entirely in an indexer's interpretation of the sequence.

67.2 Why It Worked and Why It Was Bad

It worked because it required nothing: no new protocol, no consensus change, no permission. Anyone could deploy a token by publishing text.

It was bad in exactly the ways the taxonomy of Chapter 19 predicts. Balances are an indexer's opinion. The first-is-valid mint rule created fee races that congested blocks. Every transfer required an inscription, which meant asset transfers were priced like asset creations. And the entire supply of a token depended on indexers agreeing about which mints were valid.

The response was Runes (§29.2), built by the same author explicitly to do the same job in one compact output without the UTXO bloat. That sequence — convention, congestion, purpose-built replacement — is the most compressed available illustration of the cost of building assets on a chain that does not know about them.

67.3 The Lesson for This Document

The construction of Part VIII does not attempt fungible tokens, and this chapter is why.

Fungibility requires splitting and merging. The Action structure gives exactly one successor per note (§40.4), which is perfect for indivisible items and useless for divisible ones: a balance that must split into two halves needs two successors, and the seal chain provides one.

A fungible construction would require the per-asset value commitments of §26.1 — which is to say, it would require the consensus-level asset layer, which is orphaned (Chapter 25). **This document therefore claims non-fungible assets only, and the claim is structural rather than a scoping decision.** Any future document extending it to fungibility must first explain how one seal becomes two, and the honest answer today is that it does not.

68. The Market Structure That Emerged

68.1 Partially-Signed Transactions as the Trading Primitive

Inscription trading settled on a pattern: a seller signs an input-output pair committing to sell their inscription for a price, using a signature mode that commits to their own input and their own output while leaving the rest of the transaction open. A buyer completes the transaction with their payment and broadcasts. The exchange is atomic because it is one transaction.

This is the pattern Chapter 27 is trying to reach, and the comparison explains precisely why it is hard here. The mechanism depends on a signature mode that signs a subset of the transaction — a script-level feature. Shielded transactions have no scripts and no signature modes; an Action’s authorisation covers the Action and the binding signature covers the bundle.

The partially-created-transaction format (§27.2) is the analogue, and the open question of §27.3 is whether it composes for two distrusting parties rather than for one party across two devices. The inscription market’s success with the script-level version is the reason to think the shielded version is worth the experiment: **the market structure that works for assets on a UTXO chain is single-transaction atomic swap, and a system that cannot do it settles through custody.**

68.2 Marketplaces as Indexers

Because ownership is an indexer’s opinion (§28.1), inscription marketplaces are indexers first and venues second. Their competitive advantage is the quality of their interpretation.

Under Part VIII’s construction, the venue cannot be the source of truth about ownership even if it wants to be, because ownership is not public. The venue’s role collapses to matching and settlement. This is architecturally cleaner and commercially harder: an indexer has a moat, a matching engine does not.

That is a real strategic cost of the design and it should be stated in the same document that claims the design’s benefits.

Part XXII — Why the Account Model Does Not Port

The dominant asset standards on other chains are account-based, and the most common question about any UTXO asset design is why it cannot simply do what they do. This Part answers it properly, because the answer illuminates the constraint that shapes everything here.

69. What the Account Model Provides

69.1 The Mechanism

A contract holds a mapping from item identifier to owner address. Transfer is a function call that writes the mapping. Ownership queries read it.

Three properties follow immediately and none of them is available here:

Global public state. Anyone can query any item's owner at any time, for free, with no cooperating party. This is the public lookup of Chapter 23, and it exists because the mapping is stored in consensus state that every validator holds.

Arbitrary logic. Royalties, allowlists, time locks, conditional transfers — anything expressible as code executes at transfer time, enforced by consensus.

Atomic composition. A single call can transfer an item, move payment, and update a third contract, all atomically, with conditions.

69.2 What It Costs

Every one of those properties is the same property: **consensus stores and executes application state**. The cost is that every validator stores every item's owner forever, every transfer's semantics are public, and the chain's state grows with application usage.

And the privacy cost is total. There is no version of the account model with hidden ownership, because the mapping is the ownership and the mapping is state.

70. Why the Port Fails

70.1 No State

A UTXO chain has no persistent application state. There is no mapping to write. The nearest equivalent is the set of unspent outputs, which the chain maintains for its own purposes and which carries no application semantics.

Every asset design in this document is a strategy for simulating state that the chain does not have: an indexer simulates it off chain from published events (Chapter 19, architecture one); client-side validation simulates it in the recipient's head from a proof chain (architecture two); a consensus asset layer would make the chain hold it natively (architecture three).

70.2 No Execution

There is no transfer function to attach logic to. A shielded spend is authorised by a signature and constrained by a circuit; neither can be extended by an application.

This is why royalties are unenforceable in every UTXO asset design, including this one. A royalty requires that a transfer be *conditional* on a payment, and conditionality requires execution. The inscription ecosystem discovered this the hard way — marketplace-enforced royalties were bypassed the moment a marketplace chose not to enforce them, because nothing in the chain required it.

Under the construction of Part VIII, royalties are not merely unenforceable but undetectable, since transfers are private. A venue can enforce its own royalty on trades it brokers and can do nothing about trades it does not see. This should be said to issuers plainly before they build a business model on it.

70.3 No Composability

An account-model item can be used as collateral, fractionalised, or wrapped, because a contract can hold it and other contracts can interact with that contract.

Here, an item is a note, and a note is held by whoever has its spending key. "Holding" by a program is not a concept the chain has. Composability of the kind the account model provides is not reduced here, it is absent.

71. What the UTXO Model Provides Instead

The comparison is not one-sided, and it would be dishonest to present it that way.

No global state growth. The chain does not store item ownership, so a million items cost validators nothing beyond the transactions that created them. Account-model chains have spent years fighting state growth from exactly this.

No contract risk. There is no code to be wrong. The account model's largest realised losses have come from defective contracts, and an item whose ownership is enforced by the same circuit that enforces the currency has no separate attack surface.

Privacy is possible at all. This is the decisive one. The account model forecloses hidden ownership structurally; the UTXO model, on a chain with a shielded pool, permits it. The entire subject of this document exists in the gap.

Ownership does not depend on an issuer. In the account model, a contract's admin functions frequently permit the issuer to transfer, freeze, or modify items. Here the holder holds a key and nobody else can act on it — not the issuer, not the venue, not the chain.

72. The Honest Scorecard

	Account model	This construction
Public ownership query	yes, free, global	no (Chapter 23)
Ownership privacy	none	yes, except prior holders (§41)
Transfer logic	arbitrary	none
Royalties	enforceable	not enforceable, not detectable
Composability	high	none
Validator state cost	grows with items	zero
Issuer control after mint	usually present	none
Atomic settlement	native	unresolved (Chapter 27)
Fungible tokens	yes	no (§67.3)

Reading this table honestly: the account model is better at almost everything except the one thing this document is about. A project that does not need ownership privacy should not use this construction, and should probably not be on this chain.

That is a narrower claim than whitepapers usually make and it is the accurate one.

Part XXIII — What Is Actually Known About Linkability

A design whose central claim is privacy must engage with the research that attacks privacy on this chain. This Part does so, and it begins with a correction to this project's own earlier writing, because the most commonly cited figure in the field is routinely quoted in a way that is wrong.

73. The Figure That Is Misquoted

73.1 The Claim

A widely-repeated result holds that a large majority of shielded transactions can be linked by a small set of heuristics. The number is quoted constantly, in the present tense, as a description of the chain's privacy today.

73.2 What It Actually Measured

The bulk of the linkable volume came from two sources.

Founder-reward transactions. During the chain's first four years, a portion of every block reward went to a defined set of recipients who shielded and deshielded on a regular, structurally distinctive schedule. These transactions were linkable because they were *institutional and periodic*, not because the shielded pool leaks.

Regularity in early pool usage. In a pool with few users and simple usage patterns, timing and value heuristics work well. They work less well as the pool fills.

The founder reward ended in November 2020. The measurement period predates that, and every subsequent structural change — a new pool, a different fee model, materially different usage — postdates it.

73.3 The Correct Statement

Quoting the figure in the present tense is an error, and this project has made it. The result is a valid historical measurement of a chain that no longer exists in that configuration.

The general lesson is one this document has hit twice already, in §25.3 and here: *a claim's provenance determines its scope*. A measurement is a measurement of the system that was measured. Carrying it forward across a protocol change and a governance change requires an argument that nobody has made.

74. What the Defensible Results Say

Three findings survive scrutiny and they are the ones a design should be evaluated against.

74.1 The Measured False-Positive Rate

One line of work reports an explicit false-positive rate for its linking heuristics on this chain — a figure meaningfully above zero.

Its importance is out of proportion to its size, because **it is the only such figure in the literature**. Every other published heuristic is reported without one, which implicitly claims perfect precision. A heuristic that links transactions with a one-in-seven error rate is a very different object from one that links them reliably: it produces investigative leads, not evidence, and its output degrades as the pool grows.

Any evaluation of a privacy design should ask, of every attack cited against it, what the false-positive rate was. Usually the answer is that nobody measured.

74.2 Value Fingerprint Survival

The same line of work measures how often an exact value survives a simple deshielding operation in a form that permits matching. The figure is high — well above 0.9 for the simplest case of one note producing one output.

This is the empirical foundation of §36.2 and it is the single most operationally relevant number in the literature. It says that **the pool boundary, not the pool, is where privacy is lost**, and that it is lost through amounts rather than addresses.

It is also the number that kills the transparent-seal construction for a marketplace, because a marketplace publishes the value that makes the search targeted.

74.3 Pool Usage Composition

Studies of how value actually flows show that a substantial share of shielded-pool interaction is pass-through — value that enters and leaves rather than remaining — and that fully shielded transactions are a minority of transactions though not of value.

The implication for anonymity-set reasoning is direct. The effective anonymity set for a fully shielded transaction is not the set of all shielded transactions; it is the set of *similar* shielded transactions in a similar window. The residue of §39.3 is therefore smaller than the raw transaction count suggests, and a design should not claim otherwise.

75. Applying This to the Construction

75.1 What the Attacks Target

Every attack surveyed targets one of three things: a pool boundary crossing, a distinguishable usage pattern, or a reused identifier.

The construction of Part VIII crosses the boundary exactly once per collection, at genesis, in a transaction deliberately public and uncorrelated with any holder (§36.4). It produces transactions structurally identical to payments (§39.2). It uses a fresh account per item and never reuses one (§31.3).

On the three surfaces the literature attacks, the construction is on the strong side of each. That is a meaningful statement and it is weaker than "the construction is private," which would require ruling out attacks nobody has published.

75.2 What the Attacks Do Not Target, and Should

Two surfaces are under-studied and both apply here.

Anchor age. §30.4 and §33.2 note that a wallet's choice of how far behind the tip to anchor is a fingerprint. No published work measures anchor-age distributions across wallet implementations. It would be straightforward to do and would likely produce a usable classifier.

Action-count distributions. §39.3 notes that item transfers may use a different Action count than typical payments. Whether the resulting distinguisher is usable depends on the empirical distribution, which nobody has published either.

Both of these are attacks against this construction that the literature has not yet built. Naming them is better than waiting for somebody else to.

75.3 The Mitigation That Covers Both

Pad Action counts to the modal value and randomise anchor age within the acceptance window, matching the distribution of the dominant wallet.

Both cost nothing — padding is free under the fee floor (§51.1), and anchor choice is free by construction — and both should be defaults rather than options. An option that must be enabled is a partition of the user base into those who enabled it and those who did not, which is the same fingerprint problem in a different place (§24.4).

Part XXIV — The Layer Below

76. Network-Level Exposure

Everything in Part XXIII concerns what is visible in blocks. A separate and generally stronger adversary sees transactions before they are in blocks.

76.1 The Threat

A party with many peer connections observes which node first announced each transaction. For a node that originates its own transactions, the first announcer is frequently the originator, and the originator's network address is a strong identifier.

This defeats every property in Part XIII at once. It does not matter that a transaction is cryptographically unlinkable if the adversary saw which machine broadcast it.

76.2 Why It Is Out of Scope for the Protocol Argument

The protocol has no defence and cannot have one. Transaction propagation is a network-layer concern and the chain's cryptography is indifferent to it.

This is why Chapter 39 grants the adversary perfect network vision and then argues about what else they learn. The argument is: *even against an adversary who sees the network, the chain reveals nothing additional*. That is the correct scoping, and it must be paired with an honest statement that network exposure alone is often sufficient.

76.3 What a User Can Do

Route through an anonymising transport. Broadcast through a node the user does not otherwise use. Do not broadcast from the same network address used to browse the venue.

The last is the one users actually get wrong, and it is the one a venue can help with: a venue that offers to broadcast on the user's behalf removes the user's network exposure and adds the venue as a trusted party. Neither option dominates, and the choice should be the user's and should be explained.

76.4 The Venue's Own Exposure

A venue serving the marketplace web interface sees every visitor's address, every browse, every bid. This is ordinary web surveillance and it is not mitigated by anything in this document.

A privacy design whose front door logs everybody has a problem that no amount of circuit work fixes. The operational commitments — minimal logging, no third-party analytics, no content delivery network that sees request paths — are boring, unglamorous, and more important to most users' actual privacy than the contents of Part XI.

77. Light Clients

77.1 The Problem

A light client does not hold the chain. To check whether a nullifier is spent, it must ask somebody, and asking reveals which nullifier it cares about — which, for a verifier under §44.3, reveals which item they are checking (§60.1).

77.2 The Options

Ask a server you run. Solves it completely, available to almost nobody.

Ask with decoys. Batch the real query with plausible fakes. Reduces precision at linear cost, and the decoys must be drawn from a realistic distribution or they are distinguishable.

Private information retrieval. Solves it cryptographically at substantial bandwidth or computation cost. Deployed nowhere on this chain.

Download the nullifier set. The set grows without bound, which is the whole reason light clients exist.

77.3 The Honest Position

Verification under this construction requires a full node for full privacy, and the light-client story is a real gap.

It is worth noting that this gap is not specific to the construction. Every shielded wallet on this chain faces the analogous problem for its own notes, and the detection-key proposal of §32.3 exists to address it. A solution there would likely transfer.

Part XXV — How the Specifications Got Here

78. The Process, and What It Explains

Chapter 25 documented the orphaning of the asset layer through five surfaces. This chapter asks why a well-engineered specification with a working implementation ends up structurally stranded, because the answer bears on every future dependency a builder might take.

78.1 The Dependency Structure

Specifications on this chain reference each other by number and by requirement. A proposal defining a circuit depends on a proposal defining a transaction format, which depends on the fee specification, which depends on the consensus rules.

The dependency edges are **implicit**. There is no machine-checked graph, no build that fails when a dependency is withdrawn, and no automated notification. A proposal's own header can assert a deployment that its dependencies make impossible (§25.3), and nothing catches it.

78.2 What Follows for a Builder

Read the dependencies, not the header. The header is hand-maintained metadata. The dependency graph is structural and it is the ground truth.

Check the fee specification. A transaction type that no fee rule prices is a transaction type no node accepts. Removal of fee provisions is the earliest reliable signal that a proposal is dead.

Check identifier allocations. Reassignment of a reserved identifier is the strongest signal available, because it creates a collision that cannot be resolved if the original proposal ships.

Check the upgrade's scope statement, not its wish list. An upgrade that excludes transaction format changes cannot carry a proposal that is a transaction format change, regardless of what anyone hopes.

These four checks would have correctly predicted the asset layer's status two years before it became common knowledge, and they take an afternoon.

78.3 The Governance Signal

An advisory poll of coinholders ranked priorities, and the asset layer did not win.

The correct reading is careful. The poll was advisory; participation was concentrated; and the question was framed as a choice among priorities rather than as a judgment on the work. It is better evidence of an absence of organised demand than of a considered rejection.

It is nonetheless a signal, and combined with the four structural signals it is decisive. **A specification with no format, no fee rule, no identifier, no scheduled upgrade, and no constituency is not pending. It is shelved.**

79. What This Implies About Building

79.1 The Dependency Rule

Do not take a critical-path dependency on a draft.

This sounds obvious and is violated constantly, usually in the form of a roadmap whose second phase requires a specification that is not scheduled. This document's own earlier drafts did it.

The test is simple: **if the specification never ships, does the product still exist?** If the answer is no, the product's timeline is unbounded.

79.2 What This Document Does

Every construction in Parts VIII through XIV runs on mainnet as it is today. Every dependency on a draft is confined to Part XX, is labelled as such, and is framed as an improvement to something that already works rather than as a prerequisite.

The three drafts that would help are enumerated in Chapter 62 with what each buys and what is done instead. None of them is on the critical path.

79.3 The Countervailing Risk

Building only on what exists has its own failure mode: the shipped feature can be withdrawn. Chapter 35 documents exactly this — a capability removed from a live pool by a flag set to zero.

The defence is §35.4's rule: **depend on structure, not on permission**. The nullifier uniqueness that this construction rests on is not a feature that can be turned off, because turning it off would permit double-spending the currency. That is the strongest available foundation on this chain, and choosing it was the single most consequential decision in the design.

Part XXVI — The Primitives

Part XI described the structures. This Part describes what they are built from. The material is standard and is included because every claim in Parts XV and XIII reduces to a property of one of these primitives, and a reader auditing those claims needs to know which assumption each one rests on.

80. The Curve Cycle

80.1 Why Two Curves

A proof system whose verifier is expressed as an arithmetic circuit needs the verifier's field to match the circuit's field. One curve cannot do this: a curve's base field and scalar field differ, so verifying a proof over curve A inside a circuit over curve A requires simulating one field inside the other, which is expensive.

Two curves arranged in a cycle solve it. Curve A 's scalar field is curve B 's base field, and vice versa. Verification of a proof over one curve is natural inside a circuit over the other, and recursion alternates between them.

80.2 What It Buys Here

The cycle is what makes the proof system's recursion possible without a trusted setup, and recursion is what would make §64.3's flattened proof feasible. It is not exercised by the current design — a single Action bundle uses one proof over one curve — but it is the reason the gap-closing circuit of Chapter 64 is engineering rather than research.

80.3 What a Break Would Cost

A discrete-logarithm break on either curve breaks everything: commitments become openable, nullifiers become forgeable, proofs become falsifiable. This is a single point of failure shared with the currency itself, and the construction of Part VIII adds no new assumption here.

That is worth stating as a positive property. **The construction rests on the same assumptions as the money.** A break that compromises items compromises the currency first, which means item holders are exactly as exposed as coin holders and no more.

§1. The Hash Functions, and Why There Are Several

Three distinct hash constructions appear in the shielded pool, and confusing them is a common source of implementation error.

§1.1 The Circuit-Friendly Hash

Used for nullifier derivation, where it is invoked inside the circuit and cost is measured in constraints. Its design targets low multiplicative complexity over the circuit's field — it is deliberately *not* a general-purpose hash and should never be used as one outside the circuit.

Its security rests on assumptions less studied than those underpinning conventional hashes. This is a real and acknowledged trade: the protocol accepts a younger assumption in exchange for a circuit that is feasible to prove.

For the construction of Part VIII this matters at exactly one point. The forgery argument of §34.4, case 3, requires that an attacker cannot find a note whose nullifier equals a target value — a preimage resistance claim about this hash. If the hash falls to preimage attacks, that case fails and successors become forgeable. Cases 1 and 2 still hold and still reject the forgery as a duplicate nullifier, so the construction degrades rather than collapsing, but the clean argument is gone.

§1.2 The Commitment Hash

Used for note commitments and for the commitment tree. It is a different construction from §81.1, optimised for the lookup-based circuit techniques the proof system supports, and it takes a randomness input that makes commitments hiding.

The tree uses it 32 times per spend (§33.1), which is why it dominates proving cost.

§1.3 The General-Purpose Hash

Used for key derivation, for transaction identifiers, and everywhere outside the circuit. Conventional, well-studied, and fast in software.

§1.4 The Domain Separation Discipline

Every invocation of every hash is domain-separated by a personalisation string. Two hashes of the same input for different purposes produce different outputs.

This is not decoration. Without it, a value computed for one purpose could be reinterpreted as a value for another — a note commitment reused as a nullifier, a key derivation reused as a signature challenge. Domain separation is what makes the protocol's many uses of the same primitives independent.

An implementation of Part VIII's construction must respect it exactly. The verification of §44.3 recomputes commitments and nullifiers, and a mismatched personalisation string produces a

value that is wrong in a way that looks like a corrupt disclosure.

82. Commitments

82.1 What a Commitment Must Do

Hiding: the commitment reveals nothing about the value. **Binding:** the committer cannot later open it to a different value.

Perfect versions of both simultaneously are impossible, and which one is perfect determines what an unbounded adversary can do. The shielded pool makes different choices in different places and the choices are deliberate.

82.2 Note Commitments

A note commitment binds the recipient's address, the value, the randomness ρ , and a commitment randomness. It appears in the Action as `cmx` and in the tree as a leaf.

The construction of Part VIII treats a note commitment as an item's seal identifier (§46.1). The binding property is what makes this sound: an issuer cannot later claim a different note was item 5, because `cm5` in the manifest opens to exactly one note.

The hiding property is what makes the manifest safe to publish: `cm5` is in a public manifest and reveals nothing about the item's value, its address, or its holder.

Both properties are load-bearing and they point in opposite directions, which is why the choice between perfect hiding and perfect binding matters. The note commitment is computationally binding and statistically hiding — an unbounded adversary could open it to a different value but could never learn the original. For a manifest that is public forever, that is the right way round: the value stays hidden against any future adversary, and the binding only needs to survive until the collection stops mattering.

82.3 Value Commitments

Value commitments are homomorphic: the commitment to a sum is the sum of the commitments. This is what permits balance to be checked without any value being revealed — sum every Action's value commitment, and the result opens to the bundle's declared net value (§30.4).

The homomorphism is also exactly what the asset layer's per-asset generators would break deliberately (§26.1): commitments under different generators do not combine, so values of different assets cannot cancel.

82.4 The Binding Signature

The bundle's binding signature is signed under a key derived from the sum of all value commitment randomnesses. Producing it requires knowing every randomness, which requires having constructed every Action.

Its purpose is to prevent an attacker from splicing Actions from different transactions into a new one. Each Action's proof is independently valid; only the binding signature ties them into a set with a consistent balance.

This matters for §61.3's co-construction experiment. Two parties building one transaction must jointly produce the binding signature, which requires combining randomnnesses without either party learning the other's. Whether the partially-created-transaction format supports this is precisely the open question, and this is the specific mechanism it turns on.

83. Signatures and Re-Randomisation

83.1 The Scheme

Spend authorisation uses a signature scheme supporting key re-randomisation: given a key pair, anyone holding the public key can derive a re-randomised public key, and the private key holder can derive the matching private key.

83.2 Why It Is Necessary

Without it, every spend by a holder would present the same verification key and all their spends would be linkable (§30.2). Re-randomisation makes each spend present an unrelated key while remaining verifiable.

The randomiser travels inside the proof rather than in the clear: the circuit constrains that $\mathbf{rk}$ is a valid re-randomisation of the $\mathbf{ak}$ in the spent note's key material. An observer sees a fresh key and a signature under it, with no way to relate it to any other.

83.3 The Capability Boundary, Restated

§40.6 argued that a disclosure conveying $\mathbf{nk}$ does not permit theft. The mechanism is here: spending requires a signature under a re-randomisation of $\mathbf{ak}$, which requires $\mathbf{ask}$, which is not in the disclosure.

The disclosure $(\mathbf{ak}, \mathbf{nk}, \mathbf{rivk})$ contains $\mathbf{ak}$, the *validating* key, not $\mathbf{ask}$, the *authorising* key. The difference is one letter in the name and the entire security of the construction. An implementation that discloses $\mathbf{ask}$ by confusion hands every buyer the ability to steal from every prior seller, and the resulting bug would look like a working system until somebody noticed.

This belongs in the failure-mode list of §45 and it is arguably the most dangerous entry, which is why it is stated three times in this document.

84. The Proof System

84.1 What It Proves

For each Action: that the spent note's commitment is in the tree at the anchor; that the nullifier is correctly derived from that note and nk ; that rk re-randomises the note's ak ; that the created note's commitment is correctly derived; that the created note's p is the spent note's nullifier; and that the value commitment is correctly formed.

The fifth constraint is the one this document is about. It is in the circuit for the protocol's own reason — guaranteeing p never repeats (§34.3) — and it is what makes the seal chain consensus-enforced rather than conventional.

84.2 No Trusted Setup

The system requires no per-circuit trusted ceremony. Its reference string is derived from public randomness.

The security significance is that there is no toxic waste: no secret whose compromise would permit undetectable counterfeiting forever. Earlier pools on this chain required ceremonies, and the permanent contingent liability those created is one of the reasons this pool exists.

84.3 Aggregation

One proof covers every Action in a bundle. This is why a four-Action transaction is not four times the proof size, and it is why the marginal cost of padding with dummies (§30.1) is small.

84.4 What a Soundness Break Costs

The uniqueness argument of §47.2 assumes soundness. Without it, an adversary produces an Action spending a note they do not hold, or with an arbitrary p , and the seal chain is forgeable.

It also permits counterfeiting the currency, which would be detected by the turnstile (§37.2) but not prevented for pool-internal activity. **Item holders and coin holders share this exposure identically**, which is the same point as §80.3 and is worth the repetition: the construction adds no assumption that the money does not already make.

85. The Key Agreement

85.1 The Scheme

Note encryption uses a Diffie–Hellman key agreement on the same curve as the addresses, with the sender’s ephemeral key in the Action and the recipient’s address point as the static half.

85.2 Why Trial Decryption Is Unavoidable

The recipient’s only way to test whether a note is theirs is to perform the agreement and attempt decryption. There is no tag, no hint, and no index, because any of those would be a public function of the recipient (§32.2).

85.3 The Outgoing Ciphertext

A second, smaller ciphertext encrypts the note’s key material under a key derived from the sender’s outgoing viewing key, so a sender can recover their own sends from the chain.

This is the mechanism that makes self-addressed consignment storage work (§44.4) and therefore the mechanism behind the seed-only recovery advantage of §29.4. It is easy to overlook — it exists for wallet convenience — and it is doing significant architectural work here.

§6. Where the Assumptions Actually Live

Collecting them, because an auditor should be able to see the whole set at once:

Property claimed	Rests on
Uniqueness (§47)	Proof soundness; nullifier determinism; consensus duplicate check
No forged successor (§40.3)	Circuit-friendly hash preimage resistance; consensus duplicate check
Ownership privacy (§49)	Curve discrete log; commitment hiding; ciphertext indistinguishability
Manifest binding (§50)	Commitment binding; general-purpose hash collision resistance
No theft from disclosure (§40.6)	Signature unforgeability; <code>ak</code> / <code>ask</code> separation
Transfer integrity (§48)	All of the above

Every row except the second shares its assumption with the currency. The second — preimage resistance of the circuit-friendly hash — is the one place the construction leans on something the currency leans on less heavily, and §81.1 explains why a break there degrades rather than collapses the construction.

That is the complete assumption set. There is no additional trusted party, no additional cryptographic assumption, and no assumption about the behaviour of the issuer, the venue, or any other participant.

Part XXVII — Running It

Specifications describe what a system is. This Part describes what it takes to keep one alive, because the gap between the two is where most projects of this kind end.

§7. The Operational Commitments

§7.1 What Must Never Go Down

The chain. Not the project's responsibility, and the project's uptime is bounded by it.

Content availability. §57.2's operational commitment. If the images stop resolving, the items still exist and verify, and the product is broken. This is the project's responsibility and it is permanent.

The manifest. Same, and smaller, and therefore easier: a manifest is kilobytes and should be replicated everywhere, including inside the venue's own source repository, where it is versioned and impossible to lose quietly.

§7.2 What Can Go Down Without Loss

The venue. Listings and bids are ephemeral. A venue outage stops trading and destroys nothing, provided settlement is not mid-flight.

The indexer. Collection discovery and browsing. Nothing depends on it for correctness (§60.2).

Everything else. By design. The test of §42.1 — verification works with the venue unreachable — is what makes this true, and it should be run as a test rather than asserted as a property.

§7.3 The Asymmetry Worth Noticing

The permanent commitments are cheap and boring: keep some bytes available. The expensive, complex infrastructure is the part that can fail without consequence.

This is the correct shape for a system of this kind and it is unusual. Most marketplaces have it inverted — the expensive infrastructure is the source of truth, and its failure loses the assets.

88. Migration and Versioning

88.1 What Cannot Change

The manifest, once anchored. Its digest is in a confirmed transaction and an item's identity depends on it.

The verification procedure of §44.3, for existing collections. A change to how disclosures verify is a change to what a disclosure means, and disclosures already exist in holders' hands.

88.2 What Can

The consignment encoding (§44.4), provided old encodings remain readable. A version byte at the head of the encoding costs one byte and makes this tractable; omitting it makes every future change a flag day.

The wallet, the venue, the indexer, the content storage. None is load-bearing for correctness.

88.3 The Upgrade That Would Force a Migration

If the gap-closing circuit of Chapter 64 is built, holders would want to move from disclosure-based proofs to circuit proofs. The migration is clean: the circuit proves a statement about the same chain the disclosures describe, so both work simultaneously and holders migrate at their own pace.

If the consensus asset layer ever ships, migration is not clean. Items would have to be re-issued as native assets, which means the old items are burned and new ones minted, which means the provenance chain restarts. There is no way to carry a client-side chain into a consensus asset without the consensus layer knowing about the old chain, which it will not.

This should be said to issuers now: a collection created under this construction does not automatically become a native asset if native assets arrive. It becomes a legacy collection that a new one can reference but not continue.

89. What This Project Owes Its Users

A short chapter, because the list is short and each item is a commitment rather than a description.

The custody is custody. Until Chapter 27's question is answered, the venue holds funds between payment and delivery, and the interface says so in that word (§42.2, §61.1).

The open security finding is open. A documented issue affecting account security has not been closed at the time of writing. The correct disclosure is that it is open, with the amount at risk stated, rather than a claim that it is managed.

The privacy claim is bounded. It is bounded by prior holders (§41), by venue-published timing (§49.2), and by network-level exposure (§76). Each is stated where it applies and none is hidden in a limitations appendix.

The terminology debt is owed. Parts I through VI name the wrong pool (§35.2). The claims survive; the names do not. It is a correction this document owes and has not yet made.

Nothing here is audited. The construction is novel (§34.3), the argument is this document's own, and no third party has reviewed either. A novel cryptographic construction with an unaudited implementation is exactly the thing this document's own analysis of other systems would treat sceptically, and it should be treated the same way.

Appendix A — A Complete Lifecycle

This appendix walks one item from creation to third sale, naming every value and every check. It is the operational content of Chapters 43 through 45 in concrete form.

A.1 Setup

The issuer holds seed S . The collection will contain three items.

Derive. From S , derive accounts at indices 100, 101, 102 under the hardened path (§31.4). Call them A_0 , A_1 , A_2 . From each, derive the full viewing key and one diversified address.

The starting index 100 is recorded in the manifest. This is what lets a restoring wallet find the accounts without external state (§59.1).

Prepare metadata. For each item, produce a metadata byte string containing the item's name, its traits, and the digest of its content. Compute each string's digest. Publish the strings under content addressing (§57.1).

A.2 Genesis

The issuance transaction. One shielded transaction with three Actions, each creating a minimal-value note addressed to A_0 , A_1 , A_2 respectively. Plus change. Call it T_0 at height h_0 .

After confirmation, the issuer decrypts their own outputs (via the outgoing ciphertext, \$85.3) and records the three note commitments cm_0 , cm_1 , cm_2 .

The manifest.

```
collection: "Example"
genesis:    (filled in after anchoring)
start_index: 100
items:
  0: cm = cm0 , metadata_digest = d0
  1: cm = cm1 , metadata_digest = d1
  2: cm = cm2 , metadata_digest = d2
```

Note the ordering problem: the manifest names the genesis transaction, but the genesis transaction is the anchor, which commits to the manifest. This is circular and the resolution is that the `genesis` field is omitted from the hashed bytes and filled in by readers from the transaction they found the anchor in. An implementation that includes it in the digest cannot produce a valid manifest.

Publish the manifest under content addressing. Compute $\text{digest}(M)$.

The anchor. One transparent transaction with one data output containing a short protocol tag and $\text{digest}(M)$ — 32 bytes plus a few, inside the 80-byte budget (\$24.1). Call it T_a . **Its transaction identifier G is the collection's identity.**

A.3 First Sale

Alice buys item 1.

Alice supplies an address. She derives a fresh account B_1 from her own seed at an unused index and sends the issuer a diversified address for it.

The issuer builds the transfer. One shielded transaction whose first Action spends the note at A_1 and creates a note to B_1 's address **in that same Action** (§44.1). Additional Actions carry the payment and the change.

The issuer asserts, on the built transaction before signing, that the spend and the output occupy the same Action index. If not, abort.

The issuer discloses. One entry:

```
hop 1: (ak, nk, rinv) of  $A_1$     – 96 bytes
```

plus G , the item index 1, and the manifest.

Alice verifies.

1. Fetch T_a by G . Read the data output. Extract $\text{digest}(M)$.
2. Fetch the manifest. Hash it. Compare to $\text{digest}(M)$. Equal, or abort.
3. Read $M[1].\text{cm} = \text{cm}_1$ and $M[1].\text{metadata_digest} = d_1$.
4. Derive ivk from the disclosed $(\text{ak}, \text{nk}, \text{rinv})$ of A_1 .
5. Confirm A_1 's address is consistent with ivk — this binds nk to the note (§44.3 step 2).
6. Find T_0 and trial-decrypt the relevant Action's ciphertext with ivk . Succeeds.
7. Recompute the note commitment from the decrypted plaintext. Equals cm_1 , or abort.
8. Confirm cm_1 is in the tree.
9. Compute the note's nullifier from nk . Call it nf_1 .
10. Find nf_1 on chain. It is in the transfer transaction. Read that Action's output commitment. Call it cm_1' .
11. Decrypt that output with Alice's own ivk for B_1 . Succeeds — this is her note.
12. Recompute its commitment. Equals cm_1' , or abort.
13. Compute her note's nullifier. Confirm it is **absent** from the nullifier set.

Steps 1–3 anchor. Steps 4–8 establish hop 1 is genuinely the manifest's note. Steps 9–12 establish the chain. Step 13 establishes present ownership.

Alice stores. She writes the consignment — one hop, 96 bytes, plus G and the index — into a memo on a note addressed to herself (§44.4). She also caches the metadata and content (§57.2).

A.4 Second Sale

Alice sells to Bob.

Same shape. Bob supplies a fresh address C_1 . Alice builds a transfer spending her note and creating Bob's in the same Action. She asserts placement.

Alice discloses two hops.

```
hop 1: (ak, nk, rivk) of  $A_1$       – the issuer's account
hop 2: (ak, nk, rivk) of  $B_1$       – Alice's account
```

192 bytes.

Bob verifies steps 1–8 for hop 1, then steps 9–12 to move to hop 2, then 9–12 again to move to his own note, then step 13.

What Bob now knows. That the item descends from the manifest. That it passed through two accounts. The viewing keys of both — which, because each account held exactly one note (\$41.2), tells him nothing about the issuer's or Alice's other activity.

What Bob cannot do. Spend from either account: the disclosure contains ak , not ask (\$83.3).

A.5 Third Sale

Bob sells to Carol. Three hops, 288 bytes. Identical procedure.

Consignment storage at this point. 288 bytes of keys plus the collection reference fits in one 512-byte memo. At six hops the naive encoding overflows.

The compressed encoding. A holder whose accounts all descend from one seed can store derivation indices rather than keys — a few bytes per hop instead of 96 — for their *own* hops. They cannot do this for prior holders' hops, whose keys are not derivable from any seed the holder has. So the consignment is a mix: full keys for inherited hops, indices for own hops. In practice a holder has one own hop, so the saving is small and the ceiling is real.

The chunked encoding. Write the consignment across n self-addressed notes, each memo carrying a sequence number and a chunk. Recovery scans the holder's own account, finds the chunks, and reassembles. This is unbounded and is the correct general answer.

A.6 What an Observer Saw

Across the entire lifecycle:

- One transparent transaction with an 80-byte data output, at height h_0
- One shielded transaction with four Actions, at h_0
- Three shielded transactions with two to four Actions each, at later heights
- No value balance other than the issuer's funding
- No repeated identifier of any kind

An observer knows a collection exists, knows its manifest digest, and — if they fetch the manifest — knows there are three items and what they look like. They do not know who holds any of them, whether any have traded, or which of the thousands of shielded transactions in the period were the trades, except through the timing residue of \$39.3 if the venue published the sales.

Appendix B — Verification, Specified

This appendix gives the verification procedure precisely enough to implement from. It is normative in the sense that an implementation disagreeing with it produces different answers about ownership, which is the one thing that must not vary.

B.1 Types

```
Digest      = 32 bytes
Commitment  = 32 bytes      -- a note commitment, cmx
Nullifier   = 32 bytes
TxId        = 32 bytes
Height      = unsigned integer

FullViewingKey = { ak: 32, nk: 32, rivk: 32 }      -- 96 bytes

ManifestEntry = { cm: Commitment, metadata_digest: Digest }

Manifest = {
  collection:  utf8 string
  start_index: unsigned integer
  items:       map from index to ManifestEntry
}

Consignment = {
  genesis: TxId
  index:   unsigned integer
  hops:    list of FullViewingKey      -- ordered, genesis-first
}
```

The consignment carries no commitments, no heights, and no transaction identifiers beyond the genesis. Everything else is recomputed. This is deliberate: **a consignment field that the verifier trusts rather than recomputes is a field an attacker controls.**

B.2 Chain Interface

The verifier requires exactly five operations against a node it trusts:

```
getTransaction(txid)      -> Transaction
commitmentInTree(cm)     -> bool
findNullifier(nf)        -> Option<TxId, action_index>
nullifierIsSpent(nf)     -> bool
getAction(txid, index)   -> Action
```

`findNullifier` and `nullifierIsSpent` are the two that a light client cannot perform privately (§77.1).

No other operation is permitted. In particular, **no operation against the venue** (§42.1).

B.3 The Procedure

```

verify(consignment, node) -> Result<Outcome, Error>:

-- Phase 1: anchor
anchorTx = node.getTransaction(consignment.genesis)
if anchorTx is none:                fail GenesisNotFound
dataOutputs = anchorTx.outputs filtered to data-carrier scripts
if count(dataOutputs) != 1:         fail MalformedAnchor
payload = dataOutputs[0].payload
if payload does not start with PROTOCOL_TAG: fail NotOurAnchor
manifestDigest = payload[len(PROTOCOL_TAG) ..]
if len(manifestDigest) != 32:       fail MalformedAnchor

manifestBytes = fetchManifest(manifestDigest)    -- out of band
if hash(manifestBytes) != manifestDigest: fail ManifestMismatch
manifest = parse(manifestBytes)                 -- AFTER hashing
entry = manifest.items[consignment.index]
if entry is none:                               fail NoSuchItem

-- Phase 2: the chain
expectedCm = entry.cm
previousNullifier = none

for i, fvk in enumerate(consignment.hops):

    ivk = deriveIvk(fvk.ak, fvk.nk, fvk.rivk)

    -- locate the action that created this hop's note
    if i == 0:
        if not node.commitmentInTree(expectedCm): fail NotInTree
        (txid, idx) = locateCommitment(expectedCm)
    else:
        located = node.findNullifier(previousNullifier)
        if located is none:                fail ChainBroken(i)
        (txid, idx) = located

    action = node.getAction(txid, idx)

    -- decrypt
    plaintext = tryDecrypt(action.encCiphertext,
                           action.ephemeralKey, ivk)
    if plaintext is none:                  fail NotDecryptable(i)

```

```
if plaintext.recipient != addressFor(ivk, plaintext.diversifier):
    fail KeyNotBound(i)

    -- recompute and check
    cm = noteCommitment(plaintext)
    if cm != action.cmx:                fail CommitmentMismatch(i)
    if i == 0 and cm != expectedCm:    fail NotTheManifestNote

    nf = nullifier(plaintext, fvk.nk)
    previousNullifier = nf

    -- Phase 3: present ownership
    if node.nullifierIsSpent(previousNullifier):
        return Outcome::Spent
    return Outcome::CurrentlyHeld
```

B.4 The Checks, and What Each Stops

Line	Stops
<code>count(dataOutputs) != 1</code>	An anchor with two commitments, ambiguous by construction
<code>hash(manifestBytes) != manifestDigest</code>	A substituted manifest (§40.5)
<code>parse</code> after hashing	Canonicalisation bugs (§56.3)
<code>commitmentInTree</code>	A manifest naming a commitment that was never created
<code>plaintext.recipient != addressFor(ivk, ...)</code>	A disclosed key that does not actually own the note
<code>cm != action.cmx</code>	A tampered plaintext
<code>cm != expectedCm</code> at <code>i=0</code>	A chain rooted somewhere other than the manifest
<code>findNullifier</code> returns none	A chain that stops (§40.1)
<code>nullifierIsSpent</code> at the end	The stale-proof attack (§40.2)

The two bolded rows are the ones a naive implementation omits. The first is subtle: without it, a seller could disclose *any* key that happens to decrypt, and the binding between the key and the address would not be checked. The second is the difference between proving history and proving ownership.

B.5 What Callers Must Do With the Outcome

`Outcome::Spent` means the chain is valid and the last hop's note has been spent — the seller no longer holds it. **This is not an error and must not be reported as one.** It is the correct answer to "does this party hold the item," and it is *no*.

Conflating `Spent` with a verification failure is a bug with a specific bad consequence: it makes the stale-proof case indistinguishable from a corrupt disclosure, which trains users to retry rather than to walk away.

B.6 Re-verification Before Settlement

A verification result is a statement about a chain state. It expires.

A venue must re-run Phase 3 immediately before settling (§61.2):

```
recheck(lastNullifier, node) -> bool:  
    return not node.nullifierIsSpent(lastNullifier)
```

This is one lookup and it closes the observed window, though not the race (§48.2).

B.7 Constructing a Transfer

The mirror procedure, with the one assertion that matters:

```
buildTransfer(itemNote, buyerAddress, payment, wallet) -> Transaction:

tx = wallet.newTransaction()

actionIndex = tx.addAction(spend = itemNote,
                           output = { to: buyerAddress, value: DUST })

-- MANDATORY
assert tx.actions[actionIndex].spend is itemNote
assert tx.actions[actionIndex].output.recipient is buyerAddress

tx.addPaymentActions(payment)
tx.padToModalActionCount()           -- $75.3
tx.chooseAnchor(randomisedWithinWindow) -- $75.3

assert not tx.spends_any(wallet.itemNotes except itemNote) -- $45 item 3

return tx
```

The two `assert` statements after `addAction` are the difference between a working system and a silently broken one (\$45 item 1). They must not be compiled out.

Appendix C — Conformance

An implementation that passes these tests is not correct. An implementation that fails any of them is definitely wrong. The set is chosen to cover exactly the failure modes of \$45, each of which produces software that works until it does not.

C.1 Derivation

C.1.1 — Item accounts are seed-derived. Create a collection. Destroy all local state except the seed and the manifest. Re-derive. Every account must reproduce, and every manifest commitment must be decryptable by its re-derived key.

Fails if: accounts were generated from fresh entropy (§45 item 7).

C.1.2 — Indices are unique. Create a collection of n items. Assert the derivation indices are pairwise distinct.

Fails if: an index counter is reset or reused (§45 item 2).

C.1.3 — Indices do not collide across collections. Create two collections from one seed. Assert no index appears in both.

Fails if: the index counter is per-collection rather than per-seed.

C.2 Transfer

C.2.1 — Same-Action placement. Build a transfer. Inspect the built transaction before signing. Assert the item spend and the buyer output are at the same Action index.

Fails if: the builder shuffles (§45 item 1). **This is the single most important test in the suite,** because the failure is silent and is discovered by a third party months later.

C.2.2 — End-to-end round trip. Build a transfer, broadcast it on a test network, then verify the resulting consignment from a clean verifier. Must return `CurrentlyHeld`.

Fails if: anything in C.2.1, or any derivation mismatch, or a domain-separation error (§81.4).

C.2.3 — Item notes are not spendable. Fund a wallet holding an item. Request a payment that would require the item note to cover it. Assert the wallet reports insufficient funds rather than spending the item.

Fails if: coin selection pools item notes (§45 item 3).

C.2.4 — Item notes are not selected as change. Same setup, with a payment the wallet *can* cover. Assert the item note is not consumed.

Fails if: the above, in the more common form.

C.3 Verification

C.3.1 — Stale proof is rejected as Spent. Verify a consignment. Sell the item. Verify the *old* consignment again. Must return `Outcome::Spent`, not an error, not `CurrentlyHeld`.

Fails if: Phase 3 is omitted (§45 item 8), or if `Spent` is conflated with failure (§B.5).

C.3.2 — Unbound key is rejected. Construct a consignment in which one hop's disclosed key decrypts the ciphertext but the address check of §B.3 would fail. Assert `KeyNotBound`.

Fails if: the binding check is omitted. Constructing the test case requires care and is worth the effort, because this check is the one most often skipped.

C.3.3 — Broken chain is rejected. Construct a transfer with the spend and output in *different* Actions. Verify. Assert `ChainBroken`.

Fails if: the verifier accepts any successor rather than the p-determined one — which would mean the construction's core property is not being checked at all.

C.3.4 — Substituted manifest is rejected. Alter one byte of the manifest. Verify. Assert `ManifestMismatch`.

C.3.5 — Wrong genesis is rejected. Verify a valid consignment against a different collection's genesis. Assert `NoSuchItem` or `NotTheManifestNote`.

C.3.6 — Parse-after-hash. Produce two byte encodings of the same logical manifest differing only in whitespace. Assert they produce different digests, and that the verifier accepts only the one the anchor commits to.

Fails if: the implementation canonicalises before hashing (§56.3).

C.4 Independence

C.4.1 — Verification works with the venue down. Run the verifier with all venue endpoints unreachable, against a local node. Must succeed.

Fails if: the verifier calls the venue (§45 item 5). **This is an architectural test and its failure is not a bug but a design collapse** — an implementation that fails it has built Chapter 19's first architecture.

C.4.2 — Verification works with the indexer down. Same, for the indexer.

C.4.3 — Content resolution is not required for verification. Make content storage unreachable. Verification of ownership must still succeed; only content display degrades.

Fails if: content availability is on the ownership path, which would make §57.2's operational commitment load-bearing for correctness rather than for usability.

C.5 Recovery

C.5.1 — Seed-only restore. Destroy all state except the seed. Restore. Assert every held item is located, and that a consignment sufficient to sell each one is recovered.

Fails if: consignments were stored only locally (§45 item 4). This is the test that distinguishes this construction from RGB (§29.4), and an implementation that fails it has given up the one advantage.

C.5.2 — Restore finds items received from others. As above, for items acquired by purchase rather than issuance — the harder case, since the inherited hops' keys are not seed-derivable and must come from the memo.

C.5.3 — Chunked consignment reassembly. Create an item with enough hops to overflow one memo. Restore from seed. Assert reassembly.

C.6 Privacy

C.6.1 — Transfers are structurally ordinary. Capture a transfer transaction and a plain shielded payment of the same Action count. Assert the serialised transactions are indistinguishable field-by-field in structure — same field set, same sizes, no extra outputs.

Fails if: the implementation adds a marker, a data output, or a distinguishing memo.

C.6.2 — Action count is padded. Assert transfers use the modal Action count for the network, not the minimum required (\$75.3).

C.6.3 — Anchor age is randomised. Build many transfers. Assert the anchor-age distribution matches the target distribution rather than being constant.

C.6.4 — No account reuse across items. Hold two items. Assert their accounts share no key material, and that disclosing one reveals nothing decryptable about the other.

Fails if: \$45 item 2, in the form that causes the worst privacy outcome.

C.7 Interface

C.7.1 — Genesis identifier is displayed. Assert the collection's transaction identifier appears in the item view (§45 item 6, §50.3).

C.7.2 — Custody is labelled. Assert the payment flow uses the word custody, not escrow (§42.2).

C.7.3 — The disclosure leak is disclosed. Assert the sale flow tells the seller that the buyer will learn the item's full prior history (§41).

The last is not a software property and it belongs in the suite anyway, because it is the one thing a user cannot discover for themselves and cannot undo.

Appendix D — Open Questions

Each entry states the question, why it matters, what would answer it, and what this document assumes in the meantime. They are ordered by how much they would change the design.

D.1 Two-Party Transaction Co-Construction

The question. Can two mutually distrusting parties jointly construct a single shielded transaction that spends a note from each and creates a note for each, without either party being able to extract a spendable object mid-flight, and without either disclosing a spending key?

Why it matters. It eliminates custody (Chapter 61) and makes trades atomic. It is the single largest improvement available without new cryptography.

What would answer it. An experiment, not a proof. Take the partially-created-transaction format, attempt the construction between two processes, and observe where it breaks. Three sub-questions:

1. Can the proof be generated by one party when the witness for one Action belongs to the other? The proof covers all Actions in a bundle (§84.3), so somebody must prove over both — and proving over an Action requires its witness.
2. Can the binding signature be produced jointly? It requires the sum of value commitment randomnesses (§82.4), which means combining secrets without either party learning the other's.
3. Is there an intermediate state in which one party holds a broadcastable transaction favourable only to them?

What is assumed meanwhile. Venue custody, with the discipline of §61.1 and the word custody in the interface.

Prior probability. Moderate. The format was built for hardware-wallet signing, which is one principal across two devices rather than two principals. Sub-question 1 is the likely blocker, and the likely workaround is a two-proof structure that the bundle format may not permit.

D.2 Succinct Nullifier Non-Membership

The question. Can “this nullifier is not in the nullifier set at height h ” be proven succinctly, against a commitment that a verifier can compute from chain data alone?

Why it matters. It is the missing piece of §64.1’s circuit. Without it, the gap-closing proof cannot include the present-ownership check, which is the check that makes verification meaningful (§40.2).

What would answer it. The nullifier set is not currently accumulated in any form supporting succinct non-membership. Options:

- An off-chain accumulator maintained by the prover, which the verifier must trust — defeats the purpose.
- An off-chain accumulator maintained by many independent parties who publish roots, which converts the trust into a majority assumption.
- A consensus change adding such an accumulator, which returns to Chapter 25’s timeline problem.
- A weaker statement: prove membership of the *spend* rather than non-membership, inverting the check. This requires the prover to prove a negative about an unbounded set by other means and does not obviously work.

What is assumed meanwhile. The verifier holds the chain and performs the check directly (§B.2).

Prior probability. Low without a consensus change. This is the genuinely hard open problem in the design and it should not be presented as an engineering detail.

D.3 Anchor-Age Fingerprinting

The question. Do wallet implementations have distinguishable anchor-age distributions, and is the resulting classifier accurate enough to partition the anonymity set?

Why it matters. If yes, every shielded transaction carries a wallet fingerprint, and a construction whose transactions come from one implementation is identifiable regardless of everything in Part XIII.

What would answer it. A measurement study. Collect anchor ages across a period, cluster, and compare against known wallet release timings. This is a weekend of work against public data and nobody has published it.

What is assumed meanwhile. That the fingerprint exists. Mitigation: randomise within the acceptance window, matching the dominant wallet's distribution (§75.3).

Prior probability of a usable fingerprint. High. Wallets differ in how they fetch and cache tree state, and those differences are not tuned for uniformity because nobody has pointed out that they should be.

D.4 Action-Count Distributions

The question. What is the distribution of Action counts on the network, and does an item transfer's count sit outside the mode?

Why it matters. §39.3's second residue. If item transfers are a distinguishable count, the timing attack gets a second dimension.

What would answer it. The same measurement study as D.3.

What is assumed meanwhile. Pad to the modal count (§75.3). Since padding is free under the fee floor (§51.1), this costs nothing even if the concern turns out to be unfounded.

D.5 Fungibility

The question. Is there a construction giving divisible assets with the properties of §46.3?

Why it matters. It is the difference between a collectibles system and an asset layer.

What is known. The Action structure gives one successor per note (§40.4). A balance that splits needs two. The seal chain therefore does not extend to fungibility directly.

What might work, and why it probably does not. A balance could be represented as a *set* of notes, each carrying a denomination, with splits expressed as burning one seal and opening two new ones under a new manifest — but “under a new manifest” means an issuer signature per split, which is a permissioned system.

Alternatively, the manifest could commit to a supply and each seal to a denomination, with a split proven by a circuit showing conservation. This is the asset layer (§26.1) rebuilt client-side, and it needs §D.2 plus a conservation circuit.

What is assumed meanwhile. Non-fungible assets only, stated as structural rather than scoping (§67.3).

D.6 Light-Client Verification

The question. Can a verifier without a full node perform Phase 3 of §B.3 without revealing which nullifier they are checking?

Why it matters. Without it, verification requires a full node, which excludes most users (§77.3).

Options and their costs. Decoy batching reduces precision linearly and requires a realistic decoy distribution. Private information retrieval solves it at substantial cost and is deployed nowhere here. The detection-key proposal (§32.3) would help with scanning but not with this.

What is assumed meanwhile. Full node for full privacy; a documented, labelled degradation otherwise.

D.7 Multi-Item Disclosure

The question. Can a holder prove ownership of n items with less than n independent disclosures, without linking the items to each other?

Why it matters. A seller listing a collection currently discloses each item separately, and a buyer purchasing several receives separate chains. The separateness is a privacy feature (§C.6.4) and an efficiency cost.

What is unclear. Whether the tension is real. Since the items are in unrelated accounts, any aggregation is an aggregation *by the holder*, which is precisely the linkage the separation exists to prevent. The likely answer is that the question is malformed and the cost should be paid.

D.8 Issuer Equivocation

The question. Can an issuer be prevented from anchoring two manifests, rather than merely detected?

Why it matters. §40.5's residue and §50.3's weakness.

What is known. Prevention requires a consensus rule binding an issuer identity to at most one manifest, which is a consensus change. Detection is available now — two anchors are two transaction identifiers — and relies on buyers checking.

What might improve it without consensus. A transparency log of anchors, gossiped between venues, so that a second anchor by a known issuer is immediately visible. This is fork-consistency rather than non-equivocation (§23.5) and is the same limitation that makes anchored registries unsatisfactory — but it is strictly better than nothing and it is cheap.

Part XXVIII — Specific Adversaries

Chapter 39 defined an abstract adversary. This Part works through four concrete ones, because abstract adversaries have abstract capabilities and real ones have budgets, legal powers, and specific goals.

91. The Chain Analytics Firm

Goal. Sell attribution. Produce a report saying that a particular party holds particular assets, accurate enough to be bought.

Capabilities. Full chain history, substantial compute, commercial network monitoring, exchange relationships giving transparent-side identity for some addresses, and the venue's public data.

What they get. The timing residue (§39.3). For each sale the venue publishes, a candidate set of shielded transactions in the window.

What they cannot do. Chain the candidate sets. Each sale is independent; there is no on-chain link between one transfer and the next (§39.4). A candidate set for sale n tells them nothing about sale $n+1$.

What breaks this. Volume. If a collection trades rarely and shielded transaction volume is low, the candidate set can approach one. **The defence is not cryptographic — it is that the construction's anonymity depends on the pool's traffic, which the project does not control.**

This should be said to users: on a quiet chain, timing is a real leak, and a user who transacts at an unusual hour is more exposed than one who does not.

What the firm actually sells instead. Almost certainly the pool-boundary analysis of §36 against transparent-seal systems, which is far easier. The construction of Part VIII is a hard target relative to the available alternatives, and analytics firms work on the easy ones.

92. The Compelled Venue

Goal. A legal order requires the venue to disclose what it knows.

What the venue knows. Listings, bids, sale times, prices, the network addresses of visitors, and any account identifiers users supplied. If the venue brokered disclosures, potentially the disclosures themselves.

What the venue does not know. Which shielded transaction corresponds to which sale, unless it constructed the transaction. Which chain addresses belong to which user, unless the user told it.

The design consequence. The venue should be structured so that compulsion yields as little as possible:

- Never hold disclosures. They pass between counterparties, not through the venue.
- Never require a chain address for browsing or bidding. The bid is a commitment to an address, not the address (a pattern already used in this project).
- Minimal logging, no third-party analytics, no delivery network that sees paths (§76.4).
- Settle promptly, so the custodial window is short and the set of funds subject to an order is small (§61.1).

What remains. Sale times and prices, which are published anyway, and whatever the visitor's network exposure revealed. The venue cannot be made to know nothing; it can be made to know only what it publishes.

The honest statement. A venue is a legal entity in a jurisdiction and the cryptography does not change that. A user whose threat model includes the venue's jurisdiction should transact peer-to-peer, which the construction permits and the product does not encourage.

93. The Malicious Issuer

Goal. Sell more of an item than exists, or reclaim items after sale.

What they cannot do. Reclaim. Once an item note is transferred, the issuer has no key to it (§71). There is no admin function, no freeze, and no recovery — which is a property the account model usually lacks and which is worth stating as a feature.

What they cannot do. Create a second valid successor. Uniqueness is a consensus property and does not depend on the issuer (§47.3).

What they can do. Anchor a second manifest (§40.5, §D.8). This does not double-sell item 5; it creates a second collection containing an item numbered 5. Buyers who check the genesis identifier are unaffected. Buyers who do not are defrauded.

What they can also do. Withhold content. The manifest commits to digests; the issuer is usually the only party hosting the bytes initially. An issuer who takes the content down leaves holders with verifiable ownership of an unresolvable object.

The mitigation, and its honesty. Holder-side caching (§57.2) converts this from a single point of failure into a distributed one *after* holders have fetched. It does nothing for a holder who bought and never fetched. A prudent client fetches and pins on purchase, and this should be default rather than optional.

94. The Patient Global Adversary

Goal. Complete deanonymisation, unlimited budget, indefinite timeframe.

Capabilities. Everything in Chapter 39, plus network-level observation of the entire internet, plus the ability to run venues, plus the ability to participate as a buyer.

What they get from participating. Every disclosure for every item they ever buy — which is the leak of \$41, used deliberately. An adversary who buys one item from a collection learns that item's complete prior chain.

What that buys them. The viewing keys of accounts that, by the one-account-per-item rule, held exactly one note each (§41.2). They learn that six anonymous hops occurred. They do not learn who the holders were, and they do not learn anything about those holders' other activity.

Unless the implementation reuses accounts, in which case they learn everything those accounts ever received. This is why \$45 item 2 is ranked second and why C.6.4 exists.

What they get from network observation. Probably everything, for users who did not route through an anonymising transport (§76.1). This adversary defeats the construction at the network layer, not the cryptographic one, and the construction has no defence.

The honest statement. Against an adversary with global network visibility, no chain-level privacy design is sufficient by itself. The construction's claim is that it adds nothing to what such an adversary already has, and that against every weaker adversary it holds. That is the correct scope, and a whitepaper claiming more would be wrong.

95. What the Four Have in Common

None of them breaks the cryptography. Each of them either exploits something the *deployment* publishes — sale times, prices, network addresses — or exploits a specific implementation error.

This is the usual shape of real attacks and it is why Part XIV's failure-mode list and Appendix C's test suite are longer than Part XV's proofs. **The proofs are the easy part. The implementation is where the assets are lost.**

Part XXIX — The Same Collection, Five Ways

Comparison tables compress. This Part decompresses: one hypothetical collection — a thousand items, images averaging 200 KB, expected to trade — is specified five ways, with what each would actually cost and actually leak.

97. Way One: Inscriptions on a Transparent Chain

97.1 Construction

Each item is inscribed: a commit transaction creating an output whose spending condition includes the envelope, then a reveal spending it. Content goes in the envelope, subject to the witness discount.

For 200 KB images, a thousand items is 200 MB of block space. At the discount this prices as 50 MB of ordinary transaction data.

97.2 What It Costs

Two transactions per item, two thousand total. Block space dominates: 200 MB is a material fraction of a day's capacity on the chain in question, and the fee market responds accordingly.

The practical answer is that nobody inscribes 200 KB images a thousand times. Real collections either use much smaller images, or use recursion (§66.3) to share a generator, or inscribe pointers and store content elsewhere — at which point the availability advantage of §57.3 evaporates and the comparison changes completely.

97.3 What It Leaks

Everything. Every item's content, every holder's address, every transfer, every price paid on chain.

97.4 What It Gets Right

Content permanence, provided the content is actually inscribed. Recovery from seed. No consignment, no disclosure, no availability commitment.

98. Way Two: Account-Model Contract

98.1 Construction

A contract mapping item identifier to owner. Content on content-addressed storage, metadata URLs in the contract.

98.2 What It Costs

One deployment, a thousand mint calls or one batch mint, one call per transfer. Storage growth on the chain proportional to items held.

Cheap per operation on a chain built for it.

98.3 What It Leaks

Ownership, completely and permanently, as a queryable public mapping. Transfer history. Price, if traded through an on-chain venue.

98.4 What It Gets Right

Everything operational (§69.1). Public lookup, royalties, composability, atomic settlement, fungible variants, mature tooling.

98.5 Why It Is Excluded Here

It cannot be private. Not "is not private by default" — cannot be, because the mapping *is* the ownership and the mapping is consensus state (§69.2).

A project that does not require privacy should use this. Saying so is not a weakness of this document's argument; it is the argument's precondition.

99. Way Three: Transparent Seals on This Chain

99.1 Construction

Each item is a transparent output. Transfer spends it and creates a successor. Funding comes from the shielded pool so that the funder is not transparently linked.

99.2 What It Costs

One transaction per transfer, cheap. An 83-byte data output at genesis for the manifest, or per-item inscription in input scripts (§54.1) if content must be on chain.

99.3 What It Leaks

The item's complete transfer graph, publicly. Every holder's transparent address at the moment they held it.

And — this is the part that is usually missed — **the shielded funding does not help**, because the attack goes through the pool boundary rather than through addresses (§36.2). The listing price is public, the fee is deterministic, and the value fingerprint survives deshielding at a measured rate above 0.9 (§74.2). An analyst knows what value to search for.

99.4 The Correct Claim For It

"No transparent funding history." That is true and it is much weaker than privacy.

It must not be described as private ownership. This document's project considered this construction, and the reason it was rejected is exactly §99.3.

99.5 What It Gets Right

Public lookup. Anyone can see who holds what, which is what a marketplace's floor price, holder distribution, and rarity analytics all require. Giving that up is the real cost of Way Five, and this is the way that keeps it.

100. Way Four: Client-Side Validation on a Transparent Chain

100.1 Construction

Asset state off chain, commitments in taproot outputs or equivalent. Transfers proven by consignment.

100.2 What It Costs

One transaction per transfer. Consignment of 0.4–1.9 KB per hop, growing with history. No seed-only recovery.

100.3 What It Leaks

Not ownership directly — but the anchoring outputs are public, their spends are public, and the *shape* of asset activity is visible. An observer sees that some asset moved, when, and between which outputs, without knowing which asset or who owns it.

That is meaningfully better than Ways One through Three and meaningfully worse than Way Five, and the difference is precisely that the seal identifiers are public because the UTXO model makes them so (§29.3).

100.4 What It Gets Right

Maturity. This is deployed, audited, tooled, and has users. A project choosing it is choosing a known quantity, which is worth a great deal against a novel construction with no audit (§89).

101. Way Five: The Construction of Part VIII

101.1 Construction

One seed-derived account per item. Genesis notes in a batched shielded transaction. One anchor. Transfers as ordinary shielded payments with same-Action placement. Consignments of 96 bytes per hop in self-addressed memos.

101.2 What It Costs

Issuance: roughly a thousand Actions across a handful of transactions, plus one anchor — the whole collection for a fraction of a coin (§51.2).

Transfer: one ordinary shielded transaction, identical in cost to a payment (§51.3).

Storage: 96 bytes per hop, in pre-paid memo space (§51.5).

Verification: free, against the verifier's own node (§51.4).

101.3 What It Leaks

Per-event timing, if the venue publishes sales (§39.3). Complete prior history to each subsequent holder (§41). Nothing else.

101.4 What It Gives Up

Public lookup — no floor price from chain data, no holder distribution, no rarity analytics computed by a third party (Chapter 23). Royalties, undetectably (§70.2). Atomic settlement, pending \$D.1. Fungibility, structurally (§67.3). Content permanence, in exchange for an operational commitment (§57.2).

And it gives up maturity: novel, unaudited, and resting on an argument this document makes for the first time (§89).

102. Choosing

	Way 1	Way 2	Way 3	Way 4	Way 5
Ownership private	no	no	no	partly	yes
Public lookup	yes	yes	yes	partly	no
Content permanent	yes	no	no	no	no
Per-hop proof	none	none	none	0.4–1.9 KB	96 B
Seed recovery	yes	yes	yes	no	yes
Atomic settlement	yes	yes	yes	yes	no
Royalties	no	yes	no	no	no
Fungible variant	yes	yes	yes	yes	no
Audited	yes	yes	n/a	yes	no

Read this honestly and the recommendation is narrow: **Way Five is correct when ownership privacy is a requirement and public lookup is not.** Outside that intersection, another way is better, usually Way Two.

The intersection is not empty — it is exactly the set of people who chose a privacy chain on purpose, which §54.3 shows is a real population that paid extra for a weaker version of it. But it is narrow, and a document that pretended otherwise would be selling rather than specifying.

Part XXX — Objections

Each objection is stated as an informed critic would state it, then answered. Where the answer is a concession, it says so.

103. "This Is Just RGB With Extra Steps"

The objection. Client-side validation on a chain with single-use seals. RGB did this in 2019, is deployed, and is audited.

The answer. The architecture is the same and two things differ.

First, the seal's successor pointer. In RGB, a seal is a UTXO and the successor is determined by an application-level commitment — the protocol has to *say* which output continues the chain. Here, the successor is determined by p , which the circuit constrains for the protocol's own reasons (§34.3). The difference is between a convention checked by the validator's software and a constraint checked by consensus.

Second, the consequences of that. Because the verifier already runs a node (they must, to check payment), and because the pointer is a chain-native value, the consignment carries only enough to *locate* what the node already has — 96 bytes rather than a kilobyte. And because the memo field is chain state decryptable from the seed, recovery from a backup is possible, which RGB explicitly cannot do (§29.4).

The concession. RGB is more general, more mature, and audited. A project wanting arbitrary contract logic should use it. This construction does one thing.

104. "You Have Not Solved Anything, You Have Just Hidden It"

The objection. Of course you get privacy if nobody can check anything. The hard problem is public verifiability *with* privacy, and you have declined it.

The answer. Partly conceded, and the concession is Chapter 23, which proves the decline is forced rather than chosen: Zcash has exactly two double-spend-protected resources, and the public one destroys privacy by construction.

The part that is not conceded is the implication that nothing is verified. Transfer integrity holds against anyone given the disclosure, and it holds *because of consensus*, not because of the discloser's honesty (§23.1, case ii). A lying seller cannot manufacture a second valid history, because the second history requires a second consumption of a resource the chain permits to be consumed once.

What is genuinely given up is the ability of an *arbitrary* observer to check, which client-side validation never had — it only appears to have it on transparent chains because seal identifiers happen to be public there (§103).

105. "The History Leak Makes This Useless"

The objection. Every buyer learns the whole chain. For most use cases that is worse than a public ledger, because at least a public ledger is symmetric.

The answer. Conceded for use cases where prior ownership is sensitive. §41.3 says not to use this construction for those.

Two qualifications. The leak is per-item and bounded to accounts holding one note each (§41.2), so a buyer learns "six anonymous hops" rather than six people's financial histories — *provided* the one-account-per-item rule holds, which is why it is ranked second in §45 and tested in C.6.4.

And the leak is closable by the circuit of Chapter 64, which is engineering rather than research except for the open problem of §D.2.

106. "Nobody Will Run a Node"

The objection. Verification requires a full node. Almost nobody runs one. In practice everyone will trust an API, and you will have built Way Three with extra steps.

The answer. This is the strongest objection in the list and it is not fully answerable.

What is true: verification requires a node for full privacy (§77.3), the light-client story is a genuine gap (§D.6), and an ecosystem that defaults to a hosted verifier has reintroduced the trusted third party.

What mitigates it: the verifier can run against *any* node, including one the user starts for the purpose and stops afterwards. Verification is $O(1)$ in chain length (§38.4), so the cost is the node's sync, not the verification. And the architectural test of C.4.1 at least guarantees that a user *can* verify independently, which is the difference between a design that permits self-verification and one that forecloses it.

What does not mitigate it: most users will not, and a design whose security properties are available only to the diligent is, in aggregate, weaker than its specification.

107. "Timing Analysis Destroys This"

The objection. The venue publishes sale times. Shielded volume is low. The candidate set is small.

The answer. Conceded as the residual leak (§39.3, §91). Three qualifications.

It is a *deployment* leak, not a protocol leak — a venue that delayed or batched publication would shrink it, at product cost.

It does not chain. Each sale's candidate set is independent, so the adversary gets guesses, not a history (§39.4).

It scales with the pool's traffic, which the project does not control and which is the same dependency every user of the shielded pool already has.

What would strengthen the answer is a measurement: what *is* the candidate set size for a realistic sale? Nobody has published it and this document does not claim a number.

108. "The Construction Is Unaudited and Novel"

The objection. A new cryptographic construction, argued for the first time in this document, with no third-party review, in a domain where novel constructions routinely fail.

The answer. Fully conceded (§89). It is stated in the document's own commitments chapter rather than buried.

Two things reduce the exposure relative to a typical novel construction. The assumption set is the currency's assumption set plus one (§86) — there is no new primitive, no new proof system, and no new consensus rule. And the observation at the core is a reading of an existing specification's existing sentence (§34.3), not a new mechanism; if the reading is wrong, it is wrong in a way that is checkable by anyone who reads the same sentence.

That is not an audit. It is an argument for why the audit is more likely to succeed than for a construction that invents machinery.

109. "Why Not Just Wait for the Asset Layer"

The objection. The specification exists, the cryptography is reviewed, the implementation works. Building an inferior thing now is wasted effort.

The answer. Chapter 25 is the answer, in five pieces of evidence: no transaction format, no fee rule, a reassigned identifier, exclusion from the next upgrade's scope, and a negative governance signal.

The honest form of the objection is "wait" without a date, and §79.1 is the response: if the specification never ships, does the product exist? Under "wait," no.

The concession is §88.3: a collection built now does not become a native asset later. It becomes a legacy collection. That is a real cost of not waiting and issuers should be told.

110. "Custody Means You Are Just a Bank"

The objection. The whole privacy argument is decoration if the venue holds the money.

The answer. Largely conceded, and it is the reason Chapter 27 exists and §D.1 is ranked first.

What reduces it: the custody window is between payment and delivery, not indefinite; custody applies to the *money*, not to the *items*, which are never held by the venue (§93); and the construction permits peer-to-peer trades that skip the venue entirely.

What does not reduce it: for trades through the venue, which will be most of them, it is custody, and the interface says so (§C.7.2).

111. "This Does Not Scale"

The objection. Scanning cost, consignment growth, proving cost per transfer.

The answer. Each has a specific bound.

Scanning: $O(1)$ per item per block in steady state, after birthday-bounded initial location (§59.2). A hundred items is a hundred index lookups per block.

Consignment: 96 bytes per hop, chunked across memos when it overflows (§A.5). Unbounded but small.

Proving: identical to an ordinary shielded payment, because it is one (§38.3). The construction adds nothing.

The real scaling limit is the same as the chain's: shielded transaction throughput and mobile proving time (§38.5). The construction inherits it and does not worsen it.

Appendix E — Notation

Symbols used throughout, with the section that introduces each.

E.1 Protocol Values

Symbol	Meaning	Introduced
cm	Note commitment	§30.2
cmx	Note commitment as serialised in an Action	§30.2
nf	Nullifier	§30.2
p	A created note's randomness; equals the spent note's nullifier	§34.3
rk	Randomised spend validation key	§30.2
ak	Spend validating key	§31.1
ask	Spend authorising key	§31.1
nk	Nullifier deriving key	§31.1
rivk	Commitment randomness base for viewing key derivation	§31.1
ivk	Incoming viewing key	§31.1
dk	Diversifier key	§31.1
pk_d	Diversified transmission key	§31.3
g_d	Diversified base point	§31.3

E.2 Construction Values

Symbol	Meaning	Introduced
G	Genesis transaction identifier; the collection's identity	\$46.1
M	Manifest	\$46.1
$M[i].cm$	The commitment sealing item i	\$46.1
d_i	Metadata digest for item i	\$56.1
N_j	The note at hop j of a chain	\$46.1
A_i	The issuer's account for item i	\$A.1
h_0	Genesis height	\$43.4
k	Number of hops in a chain	\$46.1

E.3 Conventions

- **Hop numbering is genesis-first.** Hop 1 is the issuer's holding; hop k is the current one.
- **"The chain"** unqualified means the blockchain, not a seal chain. Seal chains are always qualified.
- **"The pool"** unqualified means the current transferable shielded pool, not the sealed one (§35.2).
- Byte counts are of serialised on-chain representations unless stated.
- Fee figures use the released fee schedule, not the draft (§24.4).

E.4 Properties

Name	Statement	Where
P1 — Anchoring	The item set is publicly committed and fixed	§46.3, §50
P2 — Privacy	Holders are not publicly determinable	§46.3, §49
P3 — No mutable registry	No party can alter ownership by changing a record	§46.3
P4 — Transfer integrity	An accepted disclosure implies present ownership	§46.3, §48

Appendix F — Glossary

Terms as used in this document. Where common usage differs, the difference is noted, because several of these are used loosely elsewhere in ways that cause real confusion.

Action. The unit of shielded state change: one spend and one output, fused. 820 bytes. Not to be confused with "logical action," the fee-accounting unit. §30.

Anchor (chain). The historical commitment-tree root a bundle's proofs are relative to. §30.4.

Anchor (construction). The transparent transaction committing to a collection's manifest digest. §43.4. The two senses are unrelated and are distinguished by context throughout; where ambiguity was possible, the sense is stated.

Anonymity set. The set of parties an observer cannot distinguish between. Not a fixed quantity — it depends on the observer's auxiliary information, which is why §39.1 specifies the observer before claiming anything.

Binding (commitment). The property that a committer cannot open a commitment to two different values. §82.1.

Binding signature. The bundle-level signature tying the aggregate value commitment to the declared value balance. §82.4.

Birthday. The height before which an account cannot hold anything, used to bound scanning. §59.2.

Bundle. The collection of Actions in a transaction, sharing one proof, one anchor, one value balance, one binding signature. §30.4.

Client-side validation. An architecture in which asset state is validated by the recipient from a proof chain rather than by consensus. §19, §29.3.

Collection. A pair of a genesis transaction identifier and a manifest. Two manifests are two collections even if their contents are identical. §46.1.

Commit-reveal. The two-transaction pattern required to publish data in a spending script. §28.3.

Commitment tree. The append-only Merkle tree of note commitments, depth 32. §33.

Consignment. The data a recipient needs to validate an asset transfer in a client-side-validation system. Here: the ordered list of full viewing keys, 96 bytes per hop, plus the collection reference. §44.2.

Cursed inscription. An inscription assigned a negative number by an indexer convention later retired. §28.5. Relevant here only as evidence that indexer conventions are mutable.

Custody. Holding another party's funds. Used deliberately in place of "escrow," which implies a conditional release that this system cannot provide. §42.2.

Data-carrier output. A transparent output whose script is provably unspendable and carries data. Limited to 83 bytes of script, one per transaction. §24.1.

Detection key. A proposed key conferring the ability to detect that a note is yours without decrypting it. Not available. §32.3.

Diversified address. One of many addresses derivable from one key, all detectable by the same incoming viewing key. §31.3.

Dummy Action. An Action spending a zero-value note the sender created, used to pad Action counts. Cryptographically indistinguishable from a real Action. §30.1.

Envelope. The `OP_FALSE OP_IF ... OP_ENDIF` construct carrying inscription data. §28.1.

Escrow. *Not used in this document for what this system does.* See Custody.

Finalisation. In the orphaned asset specification, the flag permanently closing an asset's supply. §26.2.

Fork consistency. The property that equivocation by a server is detectable by comparing views, as opposed to non-equivocation, which prevents it. §23.5.

Full viewing key. The triple `(ak, nk, rrvk)`. Confers decryption *and* spend detection, and not spending. 96 bytes. §31.2.

Homomorphic. Of a commitment scheme: the commitment to a sum equals the sum of the commitments. §82.3.

Incoming viewing key. Confers decryption of received notes only. Does *not* confer spend detection — this separation is load-bearing. §31.2.

Indexer. Off-chain software assigning application meaning to chain data. In convention-based architectures it is the source of truth; here it is not permitted to be. §60.

Inscription. Data embedded in a spending script under a protocol convention. §28.

Item. An index together with the collection containing it. §46.1.

Jubilee. The block height at which the cursed-inscription rule was retired. Cited only as the clearest example of an ownership convention changing by software release. §28.5.

Logical action. The fee-accounting unit. Not the same as an Action, though for shielded transactions the counts coincide. §24.3.

Manifest. The document mapping item indices to sealing commitments and metadata digests. §43.3.

Memo. The 512-byte field inside every shielded output's note ciphertext. Fixed size regardless of contents, hence pre-paid. §24.2.

Memo bundle. A proposed replacement raising capacity to 16 KiB and adding a publicly-decryptable mode. Draft. §24.5.

Nullifier. The value published when a note is spent, deterministic in the note and `nk`, unlinkable to the note without `nk`. §34.

Nullifier set. The consensus-maintained set of all published nullifiers. Duplicates are rejected; this is the entirety of shielded double-spend prevention. §34.1.

One-account-per-item. The rule that each item is held in its own account derived at its own index. The most consequential implementation rule in the document. §31.3.

Partially-created transaction. A serialisation of a shielded transaction under construction, passed between parties. §27.2.

Pool boundary. The interface between transparent and shielded value, where amounts are published exactly. Where essentially all practical deanonymisation occurs. §36.

Present-ownership check. Step 8 of verification: the last hop's nullifier is absent from the nullifier set. Distinguishes proving history from proving ownership. §44.3.

Recursion (inscriptions). An inscription referencing others' content at render time. §66.3.

Recursion (proofs). A proof verifying another proof, enabling constant-size aggregation. §80.2.

Re-randomisation. Deriving a fresh public key from an existing one such that the private key holder can still sign. Prevents spend linkability. §83.1.

Sat theory. The convention tracking individual base units to determine which output an inscription follows. §65.

Seal. A resource that can be consumed once. Here: a note, named by its commitment, closed by its nullifier. §34.3.

Seal chain. The sequence of notes linked by $p = nf$. §34.3.

Sealed pool. The previous shielded pool, closed to third-party recipients by the 2026 upgrade. §35.1.

Spend unlinkability. The specified property that the incoming viewing key does not confer the ability to detect spends. §31.2.

Successor. The unique note whose p equals a given note's nullifier. §34.4.

Trial decryption. Attempting decryption of every note to find one's own. Unavoidable; an index would leak. §32.2.

Turnstile. The public per-pool value accounting that makes inflation detectable without revealing holdings. §37.

Value balance. The public signed integer giving a bundle's net flow across the pool boundary. §30.4.

Value commitment. A homomorphic commitment to an Action's value, aggregated for the balance check. §82.3.

Witness (cryptographic). The private input to a proof. §38.2.

Witness (Bitcoin). The segregated transaction portion excluded from the transaction identifier and priced at a discount. Zcash has neither. §28.4.

Zero-knowledge set. A publicly committed set supporting membership and non-membership proofs revealing nothing else — but with proofs produced by the set holder, so enumeration remains online. §23.5.

Appendix G — Reading the Specifications

This document makes many claims about what specifications say. This appendix says how to check them, and — more usefully — how to avoid the specific errors this project made while reading them.

G.1 The Order to Read In

1. **The protocol specification**, for the Action structure, the key hierarchy, the commitment tree, and the nullifier. Everything in Part XI is there.
2. **The Action proposal**, for the sentence about p (§34.3). This is the single most important primary source for this document.
3. **The fee specification**, for the accounting in Chapter 24. Note which version you are reading (§24.4).
4. **The upgrade's candidate list and scope statement**, for anything about what is shipping.
5. **The asset proposals**, last and with the caution of §G.3.

G.2 The Four Structural Checks

From §78.2, restated as a procedure for evaluating any proposal's real status:

Check 1 — the transaction format. Does a non-withdrawn format exist that can carry it? If not, it cannot deploy, regardless of its own header.

Check 2 — the fee rule. Does the active fee specification price it? A transaction type with no fee rule is unacceptable to every node.

Check 3 — identifier allocations. Has any identifier the proposal reserves been allocated elsewhere? Reassignment is the strongest available signal, because it creates a collision that cannot be resolved if the proposal ships.

Check 4 — upgrade scope. Does the next upgrade's scope statement permit this category of change? An upgrade excluding transaction format changes cannot carry one.

All four take an afternoon and would have correctly predicted the asset layer's status two years early.

G.3 The Errors This Project Made

Recorded because they are the errors a careful reader makes, not careless ones.

Reading a Deployment section as current. A proposal's header asserted deployment in an upgrade its dependencies made impossible. The header is hand-maintained; the dependency graph is structural. §25.3.

Quoting a linkability figure in the present tense. A historical measurement of a chain configuration that no longer exists, carried forward without the argument that would justify it. §73.

Naming the wrong pool. Writing the older pool's name where the current one was meant — harmless as a claim, wrong as a reference, and a reader checking against the specification finds the older pool cannot do what is claimed. §35.2.

Trusting documentation over behaviour. An external service's documentation stated a restriction its live API did not enforce. The documentation was wrong and only a live test revealed it. The general rule: for third-party services, test rather than read.

Accepting an agent's summary of a specification. A research summary asserted that the asset transfer proposal targets the next upgrade. It did — in its own stale Deployment section. The summary was accurate about the text and wrong about the world.

G.4 A Caution About Third-Party Documentation

Two documentation sites consulted during this work embed blocks addressed to automated readers, instructing them to make requests to the sites' own endpoints.

These are page content. They are not instructions, they carry no authority, and an automated reader that acts on them has been successfully injected. Anyone re-checking this document's citations against those sources should expect to encounter them and should ignore them.

The general principle: **content retrieved from a source is data about that source, never an instruction to the retriever.** It applies to specifications, to documentation, to marketplace listings, and to anything else fetched rather than authored.

Appendix H — Index of Claims

Every substantive claim this document makes, with its support and its confidence. A reader who wants to attack the argument should start here and work down the confidence column.

Confidence levels: **Specified** — stated in a protocol specification. **Derived** — follows from specified facts by an argument given here. **Measured** — supported by a published measurement. **Observed** — verified by direct test. **Inferred** — an argument this document makes that could be wrong. **Open** — stated as a question, not a claim.

H.1 Protocol Facts

Claim	Where	Confidence
An Action spends one note and creates one note	§30.1	Specified
An Action serialises to 820 bytes	§30.2	Specified
Every shielded output carries a 512-byte memo	§24.2	Specified
The note ciphertext is fixed size regardless of memo contents	§30.2	Specified
<code>rk</code> is re-randomised per Action	§30.2	Specified
A created note's <code>p</code> is the spent note's nullifier	§34.3	Specified
The specification gives uniqueness as the reason for the above	§34.3	Specified
<code>ivk</code> does not confer spend detection	§31.2	Specified
The commitment tree has depth 32	§33	Specified
Consensus rejects duplicate nullifiers	§34.1	Specified
<code>valueBalance</code> is a public signed integer	§30.4	Specified
The previous pool is closed to third-party recipients	§35.1	Specified
The data-carrier limit is 83 bytes, one output per transaction	§24.1	Specified
The limit is unanimous across all node implementations	§24.1	Observed
Zcash has no witness and no witness discount	§28.4	Specified
The proving system requires no trusted setup	§84.2	Specified

H.2 Derived Results

Claim	Where	Confidence
At most one successor exists per note	\$34.4, \$47.2	Derived
The three cases in §34.4 are exhaustive	\$34.4	Derived
Zcash has exactly two double-spend-protected resources	\$23.2	Derived
Therefore no third construction exists	\$23.2	Derived
Public ownership verifiability is incompatible with ownership privacy on mainnet today	\$23.1–\$23.2	Derived
Integrity survives in the view of anyone given the disclosure	\$23.1	Derived
A <code>nk</code> -bearing disclosure defeats the stale-proof attack	\$40.2	Derived
A disclosure conveys detection, not spending	\$40.6, \$83.3	Derived
An item cannot be split	\$40.4	Derived
Fungibility does not follow from this construction	\$67.3	Derived
Verification is $O(1)$ in chain length	\$38.4	Derived
The construction adds no proving cost over a payment	\$38.3	Derived
The construction's assumption set is the currency's plus one	\$86	Derived

H.3 Economic Figures

Claim	Where	Confidence
The shielded memo costs ~19.5 zatoshi/byte	\$24.3	Derived
A transparent data output costs ~250 zatoshi/byte	\$24.3	Derived
The shielded channel is >12× cheaper per byte	\$24.3	Derived
The second Action is free under the fee floor	\$51.1	Derived
Padding with dummies costs nothing	\$51.1	Derived
A 100-item collection issues for a fraction of a coin	\$51.2	Derived
The marginal fee constant differs between released and draft schedules	\$24.4	Specified

H.4 Comparative Claims

Claim	Where	Confidence
Per-hop proof is 96 bytes	\$29.5	Derived
Taproot Assets proofs run 0.4–1.9 KB per hop	\$29.5	Inferred from published descriptions
RGB cannot recover from seed alone	\$29.4	Specified (by RGB)
This construction can, via self-addressed memos	\$29.4	Derived
Inscriptions require two transactions per creation	\$28.3	Specified
Ordinals' numbering was changed by a software release	\$28.5	Observed
Every deployed system's asset meaning is decided by mutable software	\$29.5	Inferred

H.5 Specification Status

Claim	Where	Confidence
The v6 transaction format proposal is Withdrawn	\$25.1	Specified
Its successor states it need not support custom assets	\$25.2	Specified
The fee specification's asset clauses were removed	\$25.2	Specified
The asset layer's reserved lead byte was reassigned	\$25.2	Specified
The next upgrade's scope excludes transaction format changes	\$25.2	Specified
The asset layer is absent from the upgrade's candidates	\$25.2	Specified
A coinholder poll did not select it	\$25.2	Observed
Therefore the asset layer's timeline is unbounded	\$25.4	Derived
The transfer proposal's Deployment section is stale	\$25.3	Derived
Memo bundles are a draft dependent on a draft format	\$24.5	Specified
Payment-conditional transfer is a draft with the same dependency	\$27.4	Specified

H.6 Privacy Claims

Claim	Where	Confidence
An item transfer is structurally identical to a payment	\$39.2	Derived
Per-event uncertainty does not compound into a history	\$39.4	Derived
The residual leak is venue-published timing	\$39.3, \$49.2	Derived
Every prior holder learns the full chain	\$41.1	Derived
That leak is closed only by consensus assets or the \$64 circuit	\$41.4	Derived
The widely-quoted linkability figure is misapplied in the present tense	\$73	Inferred
Value fingerprints survive simple deshielding at a high rate	\$74.2	Measured
The only published false-positive rate for these heuristics is meaningfully above zero	\$74.1	Measured
A fresh transparent address does not defend against value matching	\$36.2	Derived
Anchor age is likely a wallet fingerprint	\$75.2, \$D.3	Open
Action count may be a distinguisher	\$75.2, \$D.4	Open

H.7 The Record

Claim	Where	Confidence
Over 100,000 inscriptions were created on this chain	\$54.3	Observed
They used input scripts, not data outputs	\$54.1	Observed
Roughly one in seven were shielded-funded	\$54.3	Observed
Input scripts are excluded from the transaction identifier	\$54.2	Specified
Inscriptions were destroyed by wallets spending them as change	\$54.4	Observed
Content identifiers were embedded on this chain a decade earlier	\$54.5	Observed
The OP_RETURN count does not detect scriptSig inscriptions	\$54.1	Derived

The last row deserves a note. An earlier draft asserted a causal link between an activity spike and inscription volume on the basis of data-output counts. Since the deployed protocol uses input scripts, data-output counts cannot detect it, and the correlation was not evidence for the claim. The passage now presents both sources and labels the causal link as inferred. It is recorded here because a document that lists its own corrections is easier to trust than one that does not.

H.8 Open

Question	Where
Two-party co-construction	\$D.1
Succinct nullifier non-membership	\$D.2
Anchor-age fingerprinting	\$D.3
Action-count distributions	\$D.4
Fungibility	\$D.5
Light-client verification	\$D.6
Multi-item disclosure	\$D.7
Issuer equivocation prevention	\$D.8
Realistic timing candidate-set size	\$I07

Appendix I — Deployment Checklist

Ordered by when it must be done. Each item names the section that explains it and the test that proves it.

I.1 Before Any Code

- ☐ Decide whether ownership privacy is actually a requirement. If not, use the account model (§98.5).
- ☐ Decide whether public lookup is required. If yes, this construction is wrong (§102).
- ☐ Decide whether fungibility is required. If yes, this construction is wrong (§67.3).
- ☐ Accept that the construction is unaudited and novel (§89, §108).
- ☐ Accept the permanent content-availability commitment (§57.2, §87.1).

I.2 Key Management

- ☐ Item accounts derived from the seed at hardened indices — C.1.1
- ☐ Indices unique within a collection — C.1.2
- ☐ Indices unique across collections from one seed — C.1.3
- ☐ Starting index recorded in the manifest (§A.1)
- ☐ Never derive item accounts from fresh entropy (§45 item 7)

I.3 Issuance

- ☐ Metadata serialised deterministically; hash the published bytes, never a re-serialisation (§56.3, C.3.6)
- ☐ Manifest excludes the genesis field from its own digest (§A.2)
- ☐ Manifest replicated, including in the source repository (§87.1)
- ☐ Content published under content addressing (§57.1)
- ☐ Exactly one data output in the anchor transaction (§B.4)
- ☐ Anchor once — never per item, never per transfer (§43.4)
- ☐ Genesis identifier recorded and displayed everywhere (§43.4, C.7.1)

I.4 Wallet

- ☐ Same-Action placement asserted before signing — C.2.1 (**the most important single test**)
- ☐ Item notes excluded at the coin-selection layer, not by a dialog — C.2.3, C.2.4
- ☐ Birthday-bounded scanning per item account (§59.2)
- ☐ Steady-state holding costs a nullifier lookup, not a decryption (§59.2)
- ☐ Consignment written to a self-addressed memo on every transfer — C.5.1
- ☐ Chunked encoding implemented for chains exceeding one memo — C.5.3
- ☐ Action count padded to the network mode — C.6.2
- ☐ Anchor age randomised within the acceptance window — C.6.3
- ☐ `ask` disclosed, never `ask` (§83.3)

I.5 Verifier

- ☐ All eight steps of §4.3 implemented, including step 8
- ☐ The key-binding check implemented — C.3.2
- ☐ `Spent` returned as an outcome, not an error (§B.5)
- ☐ Parse after hashing, never before — C.3.6
- ☐ No network call to the venue — C.4.1 (**architectural**)
- ☐ No network call to the indexer — C.4.2
- ☐ Content unreachable does not block ownership verification — C.4.3
- ☐ Works against any node the user names (§B.2)

I.6 Venue

- ☐ Re-run Phase 3 immediately before settlement (§B.6, §61.2)
- ☐ Custody labelled custody — C.7.2
- ☐ Custodial funds not pooled with operating funds (§61.1)
- ☐ Settle on confirmation, not on a batch schedule (§61.1)
- ☐ Disclosures never pass through the venue (§92)
- ☐ Bids stored against a commitment to an address, not the address (§92)
- ☐ Minimal logging; no third-party analytics; no path-visible delivery network (§76.4)
- ☐ The history leak disclosed to sellers before sale — C.7.3

I.7 Recovery

- ☐ Restore from seed alone locates every held item — C.5.1
- ☐ Restore recovers consignments for purchased items — C.5.2
- ☐ Restore reassembles chunked consignments — C.5.3
- ☐ A restore drill is run before launch, not after the first loss

I.8 Disclosure

- ☐ Open security findings stated as open, with amounts (§89)
- ☐ The privacy claim's three bounds stated where they apply (§89)
- ☐ The terminology debt acknowledged or discharged (§35.2)
- ☐ Issuers told that a collection built now does not become a native asset later (§88.3)
- ☐ Issuers told that royalties are unenforceable and undetectable (§70.2)

Appendix J — Every Number in One Place

Collected for checking. Where a figure is derived, the derivation is named; where measured, the measurement; where a range, the range.

J.1 Sizes

Quantity	Value	Source
Action, serialised	820 B	\$30.2
Note ciphertext	580 B	\$30.2
Memo	512 B	\$24.2
Outgoing ciphertext	80 B	\$30.2
Nullifier, commitment, each key	32 B	\$30.2, \$31.1
Full viewing key	96 B	\$31.2
Spend authorisation signature	64 B	\$30.2
Data-carrier script, maximum	83 B	\$24.1
Data-carrier payload, maximum	80 B	\$24.1
Input script, standardness limit	~1,650 B	\$24.1
Coinbase script	2–100 B	\$24.1
Commitment tree depth	32	\$33
Commitment tree capacity	$\sim 4.3 \times 10^9$	\$33
Memo bundle capacity (draft)	16 KiB	\$24.5
Memo bundle chunk cap (draft)	64	\$24.5

J.2 Costs

Quantity	Value	Source
Marginal fee per logical action (released)	5,000 zat	\$24.4
Marginal fee per logical action (draft)	1,000 zat	\$24.4
Logical action floor	2	\$24.3
Baseline transparent payment	10,000 zat	\$24.3
Two-action shielded transfer	10,000 zat	\$24.3
Transparent payment with 80-byte data output	20,000 zat	\$24.3
Transparent payment with 32-byte commitment	15,000 zat	\$24.3
Cost per byte, transparent data output	~250 zat	\$24.3
Cost per byte, one memo in a 2-action transfer	~19.5 zat	\$24.3
Cost per byte, both memos	~9.8 zat	\$24.3
Ratio, shielded advantage	>12×	\$24.3

J.3 Proof and Verification

Quantity	Value	Source
Proving time per Action	~1–2 s, desktop CPU	\$38.2
Proving memory	hundreds of MB	\$38.2
Verification time	negligible	\$38.2
Consignment per hop	96 B	\$29.5
Hops per 512-byte memo, naive	5	\$44.4
Verification complexity	$O(\text{hops})$	\$38.4
Taproot Assets proof per hop	0.4–1.9 KB	\$29.5
RGB consignment	~1 KB and up	\$29.5

J.4 Steady-State Operation

Quantity	Value	Source
Scanning, initial, per item account	from its birthday	\$59.2
Scanning, steady state, per item	1 nullifier lookup per block	\$59.2
Actions per transfer	2–4, padded to mode	\$51.3, \$75.3
Transactions per transfer	1	\$51.3
Transactions per inscription (comparison)	2	\$28.3
Anchors per collection	1	\$43.4
Data outputs per anchor	1	\$B.4

J.5 The Record

Quantity	Value	Source
Inscriptions created on this chain	>111,000	\$54.3
Shielded-funded fraction	~1 in 7	\$54.3
Value fingerprint survival, 1-note deshield	>0.9	\$74.2
Only published false-positive rate	meaningfully above zero	\$74.1

Appendix K — Error Taxonomy

Every failure the verifier can report, what it means, whether it is the seller's fault, and what the user should do. An implementation that collapses these into one "verification failed" trains users to retry, which is the wrong response to most of them.

K.1 Anchor Errors

Error	Means	Fault	User action
GenesisNotFound	No transaction with that identifier	Seller, or a typo	Re-request the collection reference
MalformedAnchor	Wrong number of data outputs, or wrong payload length	Issuer	Do not buy. The collection is malformed.
NotOurAnchor	The data output is not this protocol's	Seller, or a wrong identifier	Re-request
ManifestMismatch	The fetched manifest does not hash to the anchored digest	Substitution attempt, or a corrupted fetch	Re-fetch from another source. If it persists, do not buy.
NoSuchItem	The manifest has no entry at that index	Seller	Do not buy

ManifestMismatch is the one to treat as hostile. It is the signature of \$40.5's attack, and a second fetch from an independent source distinguishes corruption from substitution.

K.2 Chain Errors

Error	Means	Fault	User action
NotInTree	The manifest's commitment was never created on chain	Issuer	Do not buy. The collection is fictitious.
NotTheManifestNote	Hop 1 does not open to the manifest's commitment	Seller, deliberately	Do not buy
NotDecryptable(i)	A disclosed key does not decrypt its hop	Seller, or a truncated transfer	Re-request the consignment
KeyNotBound(i)	The key decrypts but does not own the address	Seller, deliberately	Do not buy
CommitmentMismatch(i)	Recomputed commitment differs from the Action's	Implementation bug, or tampering	Report. Do not buy.
ChainBroken(i)	No Action published the previous hop's nullifier	Seller, or a broken transfer (§44.1)	Do not buy. Tell the seller their wallet is defective.

`ChainBroken` is worth a specific note. The most likely cause is not fraud — it is a seller whose wallet did not enforce same-Action placement (§45 item 1). The item exists and they hold it; it simply cannot be proven to descend from genesis. Telling them so is more useful than refusing silently, because they can report it to their wallet vendor and nobody else's items get broken the same way.

K.3 Outcomes, Not Errors

Outcome	Means	User action
CurrentlyHeld	The chain is valid and the last note is unspent	Proceed
Spent	The chain is valid and the last note has been spent	Do not buy. This is not an error.

SB.5 explains why conflating `Spent` with an error is harmful. To restate: it is the correct answer to a legitimate question, and presenting it as a malfunction teaches users to retry a transaction they should abandon.

K.4 Environmental

Condition	Means	User action
Node unreachable	Cannot verify	Verification is not possible; do not proceed on trust
Node not fully synced	Nullifier set incomplete	Do not proceed. A partial nullifier set makes <code>Spent</code> indistinguishable from <code>CurrentlyHeld</code> .
Content unreachable	Ownership verifies, content does not display	Proceed if willing; note \$42.5

The middle row is the dangerous one, because a partially-synced node returns a *plausible* wrong answer rather than an error. An implementation must check sync status before Phase 3 and refuse rather than guess.

Appendix L — Timeline

Context for Parts IX, X and XVII, on one axis. Dates are given where they are firm and omitted where they are approximate.

L.1 Foundations

When	What	Relevance
2003	Zero-knowledge sets introduced	The strong form of the problem Chapter 23 declines (§23.5)
2005 onward	DNS authenticated denial of existence, three successive mechanisms	The eighteen-year natural experiment (§23.5)
2014	Counterparty establishes convention-based assets on a UTXO chain	The oldest answer, and the oldest failure mode (§29.1)
2014	Data-carrier relay limit settles at 83 bytes	The constraint still binding today (§24.1)
2016	Zcash launches with a shielded pool requiring a trusted setup	The liability later removed (§84.2)
2017	Content identifiers embedded in Zcash transactions	The forgotten precedent (§54.5)
2019	RGB establishes maximal client-side validation	The most ambitious comparison (§29.4)

L.2 The Current Pool

When	What	Relevance
2022	The Orchard pool activates: fused Actions, no trusted setup, p = nullifier	Everything in Part XI
2022	Structured-memo proposal opened	Still a draft with zero allocations (§24.2)
2020 (Nov)	Founder reward ends	Invalidates the present-tense use of the linkability figure (§73.2)

L.3 Inscriptions

When	What	Relevance
2023	Ordinals launches; envelope, commit-reveal, witness discount	Chapter 28
2023	Cursed inscriptions accrete by indexer accident	\$28.5
2023	Fungible-token convention on inscriptions; fee races	\$67
2023	The Jubilee: numbering changed by a software release	The clearest evidence that conventions are mutable (\$28.5)
2024	Runes ships as the purpose-built replacement	\$29.2
2023–2025	Zerdinals: >111,000 inscriptions on Zcash via input scripts	Chapter 54

L.4 The Asset Layer

When	What	Relevance
—	ZIP 226/227 drafted: per-asset commitments, transparent issuance	Chapter 26
—	ZIP 230 (v6 format) drafted as the carrier	\$25.1
—	ZIP 230 Withdrawn	\$25.2
—	Successor format states it need not support custom assets	\$25.2
—	Fee specification's asset clauses removed	\$25.2
—	Reserved lead byte reassigned to an unrelated specification	\$25.2 — the strongest signal
2026 (May)	Counterfeiting vulnerability disclosed in the reference implementation	\$26.5
—	Coinholder poll does not select the asset layer	\$25.2
—	Next upgrade's candidate list excludes it; scope excludes format changes	\$25.2

L.5 The Transition

When	What	Relevance
2025 (Oct)	Bitcoin Core raises its data-carrier default to effectively uncapped	The opposite direction of travel (§24.1)
2026 (Jan)	A Zcash node implementation introduces the 83-byte cap it never enforced	§24.1
2026 (Jul 28)	NU6.3 "Ironwood": new pool activates, previous pool sealed	Chapter 35
2026 (Sep)	This document's construction identified	§34.3

L.6 What Is Not On This Timeline

No date exists for the asset layer's deployment, for memo bundles, or for payment-conditional transfer. That absence is the subject of Chapter 25 and the reason for Part VIII.

Appendix M — Specification Map

Which document says what, and what depends on what. Numbers are given for lookup; the four structural checks of §G.2 are what determines status, not these headers.

M.1 Load-Bearing for This Construction

Document	Status	What this document uses it for
Protocol specification	Active	Action structure, key hierarchy, tree, nullifier — all of Part XI
ZIP 224 (Orchard Actions)	Final	The ρ = nullifier sentence (§34.3) — the single most important citation
ZIP 32 (key derivation)	Active	Hierarchy and hardened paths (§31)
ZIP 316 (unified addresses)	Active	Diversified addresses, receiver types (§31.3)
ZIP 317 (fees)	Active	All economics (Chapter 24, Chapter 51)
Consensus rules, NU6.3	Active	The sealing (§35.1)

Every one of these is Final or Active. **The construction depends on no draft.**

M.2 Referenced, Not Depended On

Document	Status	Where discussed
ZIP 226 (asset transfer)	Draft	\$26.1, \$25.1 — note the stale Deployment section (\$25.3)
ZIP 227 (asset issuance)	Draft	\$26.2
ZIP 230 (v6 format)	Withdrawn	\$25.1 — the orphaning
ZIP 228 (payment-conditional)	Draft	\$27.4, \$62.3
Memo bundle proposal	Draft	\$24.5, \$62.2
Structured memo proposal	Draft, zero allocations	\$24.2
Detection key proposal	Draft	\$32.3, \$D.6

M.3 Dependency Structure

```
consensus rules
  ↑
transaction format ← fee specification
  ↑                ↑
Action definition   (asset clauses: REMOVED)
  ↑
key derivation
  ↑
address format

asset transfer —depends on→ v6 format [WITHDRAWN]
asset issuance —depends on→ v6 format [WITHDRAWN]
payment-conditional —→ v6 format [WITHDRAWN]
memo bundles —→ v6 format [WITHDRAWN]
```

Four drafts, one withdrawn dependency. This diagram is the whole of Chapter 25 in one figure, and it is the reason Part VIII exists.

M.4 How to Check This Yourself

Apply §G.2's four checks to any row in M.2. For each: does a live format carry it, does the fee specification price it, is its identifier still its own, and does the next upgrade's scope admit it?

For every row in M.2 the answer to at least three of the four is no.

Part XXXI — If the Open Questions Resolve

The design is not static. Each open question of Appendix D has two answers, and the system looks different under each. This Part works through the consequences, so that a reader can judge how much of the argument survives a bad outcome.

113. If Co-Construction Works

What changes. Custody disappears. Trades become single atomic transactions. §110's objection evaporates. The venue becomes a matching service holding nothing.

What it costs. A protocol between two parties, with all the usual hazards: grieving by abandoning half-built transactions, denial of service by flooding, and the need for a timeout that does not leak.

What stays the same. Everything in Parts XI through XIII. Privacy, uniqueness, verification, and the disclosure leak are untouched — co-construction is a settlement mechanism, not an ownership mechanism.

Confidence this happens. Moderate. The blocker is likely to be joint proving (§D.1, sub-question 1).

114. If Co-Construction Does Not Work

What changes. Nothing about the construction; everything about the product. The venue holds funds indefinitely as a structural fact rather than a temporary one, and \$110 stands as a permanent objection.

The fallback worth building. Reduce the window to its minimum: the buyer pays, the venue verifies and settles in the same operation, and custody is measured in seconds rather than in confirmations. This does not eliminate the trust; it bounds the exposure.

The second fallback. Peer-to-peer trades with a mutually trusted third party of the counterparties' choosing, which the construction permits and which is strictly better than a single venue for users who have someone to trust.

What this means for the claim. The document's four properties (§46.3) are about ownership, not settlement, and they hold either way. A reader should not treat §D.1's failure as undermining Parts VIII through XV.

115. If Succinct Non-Membership Is Solved

What changes. The circuit of Chapter 64 becomes buildable. Holders can prove present ownership to anyone with no disclosure. **The leak of §41 closes entirely**, which is the largest single improvement available to the design.

Voluntary public provenance becomes possible (§64.2): a holder who *wants* to publish can, and nobody can publish falsely. Most of what markets use a public index for — proving a notable holder held something — becomes available without giving up the privacy of holders who do not opt in.

What does not change. There is still no public *lookup*. Nobody can ask “who holds item 5.” Chapter 23’s boundary is unmoved, because the proof requires a prover.

Confidence. Low without a consensus change (§D.2). This is the hard problem.

116. If Anchor Age Turns Out to Fingerprint

What changes. Every shielded transaction carries a wallet identifier, and the anonymity set partitions by software. A construction whose transactions all come from one implementation is identifiable as such.

What mitigates it. §75.3's randomisation, applied before the measurement is published rather than after. This costs nothing and should already be default.

The broader point. This is a risk the construction shares with every shielded wallet on the chain, and a finding here would be a finding about the ecosystem rather than about this design. That is cold comfort to a user, and it is the reason §D.3 recommends someone run the study.

117. If the Asset Layer Ships

What changes. §26.4's leak closure, §26.2's public supply, and the elimination of consignments entirely. The construction of Part VIII becomes a legacy mechanism.

What does not change. Existing collections. §88.3 is explicit: items created under this construction do not migrate. They remain valid, verifiable, and tradeable by the procedure of Part XIV, in perpetuity, on a chain that also supports something better.

Whether this is bad. For an issuer, it means a collection created now is permanently on the old mechanism. For the argument of this document, it changes nothing — Chapter 25's point was never that the asset layer is wrong, only that its timeline is unbounded, and a design that works today does not become wrong when a better one arrives.

118. If Nothing Resolves

The base case, and the one to plan for. Custody persists, the history leak persists, verification requires a node, fungibility remains impossible, and no specification ships.

What still works. Issuance, transfer, and verification on mainnet, with the four properties of \$46.3, at the cost of an ordinary shielded payment, recoverable from a seed.

That is the whole claim of this document, and it is the version that requires nothing from anyone.

Appendix N — Worked Derivations

Every number in Appendix J that is derived rather than specified, with the arithmetic. Included so that a reader can check them rather than trust them.

N.1 Fee Per Logical Action

The fee is the marginal rate times the greater of the logical action count and the floor.

```
fee = marginal × max(logical_actions, 2)
marginal = 5,000 zatoshi      (released schedule, §24.4)
floor    = 2
```

Baseline transparent payment. One input, one output, one change output. Logical actions: by the transparent accounting, one. Below the floor, so:

```
fee = 5,000 × 2 = 10,000 zatoshi
```

Two-action shielded transfer. Two Actions, at or above the floor:

```
fee = 5,000 × 2 = 10,000 zatoshi
```

Identical. This is the source of §51.3's claim that a transfer costs what a payment costs, and it is worth noticing that the *shielded* transaction is not more expensive despite doing more work.

The second Action is free. A one-Action transaction pays the floor, which is the two-Action price. Padding a one-Action transaction to two costs nothing, which is §51.1 and the economic reason dummy padding (§30.1) is actually used.

N.2 The Data Output Premium

A transparent payment carrying an 80-byte data output:

```
inputs: 1
outputs: 2 (the data output, and change)
```

The transparent contribution is computed from serialized sizes rather than from counts, and the 83-byte script pushes the output's contribution up. The result, by the accounting of §24.3:

```
logical_actions = 4
fee = 5,000 × 4 = 20,000 zatoshi
```

Against the 10,000 baseline, the data output costs 10,000 zatoshi for 80 bytes of payload:

```
10,000 / 80 = 125 zatoshi per byte (marginal)
20,000 / 80 = 250 zatoshi per byte (total, attributing the whole fee)
```

§24.3 uses the total attribution, which is the conservative direction for the comparison this document is making — it makes the transparent channel look as good as possible and the shielded advantage is still an order of magnitude.

A 32-byte commitment instead of 80 bytes of payload drops the output below a size threshold and yields three logical actions:

```
fee = 5,000 × 3 = 15,000 zatoshi
```

This is the anchor transaction's cost (§51.2), and it is paid exactly once per collection.

N.3 The Memo Per-Byte Figure

A two-Action shielded transfer costs 10,000 satoshi and carries two memos of 512 bytes each.

One memo used:

$$10,000 / 512 = 19.53 \text{ satoshi per byte}$$

Both memos used:

$$10,000 / 1,024 = 9.77 \text{ satoshi per byte}$$

The ratio against the transparent channel:

$$\begin{array}{ll} 250 / 19.53 = 12.8\times & \text{(one memo)} \\ 250 / 9.77 = 25.6\times & \text{(both memos)} \end{array}$$

\$24.3 quotes 12.8×, the conservative figure.

Why this is the right comparison and not a trick. The 10,000 satoshi buys a transfer of value that the sender wanted to make anyway. The memo bytes are carried by a ciphertext of fixed size (\$30.2) that is paid for whether or not it is used. The marginal cost of the payload is therefore *zero*, and the 19.53 figure is an attribution of the whole fee to bytes that came free with it.

The transparent channel has no such property: the 80 bytes are additional output size that would not otherwise exist, and they raise the fee. The comparison understates the advantage.

N.4 Issuance Cost for a Collection

For n items, batched b items per transaction:

```
transactions = ceil(n / b) + 1      (+1 for the anchor)
shielded_fee = 5,000 × (b + 1) × ceil(n / b)  (+1 per tx for change)
anchor_fee   = 15,000
```

For $n = 100$, $b = 25$:

```
transactions = 4 + 1 = 5
shielded_fee = 5,000 × 26 × 4 = 520,000 zatoshi
anchor_fee   = 15,000 zatoshi
total        = 535,000 zatoshi = 0.00535 ZEC
```

Plus the minimal value locked in each item note, which is recoverable only by destroying the item and is therefore a permanent cost. At the smallest sendable value this is negligible; an implementation that uses a larger value locks that value forever and should not.

For $n = 1,000$, $b = 25$:

```
transactions = 40 + 1 = 41
shielded_fee = 5,000 × 26 × 40 = 5,200,000 zatoshi
total        = 0.0522 ZEC
```

§51.2's claim that issuance is not a cost centre.

N.5 Consignment Growth

Naive encoding, 96 bytes per hop, 512-byte memo:

```
hops_per_memo = floor((512 - overhead) / 96)
```

With ~30 bytes of overhead for the collection reference, index, and version byte:

```
hops_per_memo = floor(482 / 96) = 5
```

\$44.4's figure. Beyond five hops, chunking is required:

```
memos_needed = ceil(hops / 5)
```

A twenty-hop item needs four memos, which is four self-addressed notes, which is four Actions across however many transactions the holder makes anyway. Since a transfer already creates a self-addressed note for bookkeeping (\$59.5), the marginal cost is Actions, and under \$N.1 the first spare Action is free.

N.6 Scanning Cost

Initial location, per item account. Trial decryption from the account's birthday to the tip:

```
cost = (tip_height - birthday) × avg_actions_per_block × decrypt_cost
```

For an account created recently, the first factor is small. For the wallet's main account it is the full chain, which is the ordinary shielded wallet cost and not attributable to this construction.

Steady state, per item. Once located, the account holds exactly one note whose nullifier is known. Per block:

```
cost = 1 nullifier set lookup
```

Not a decryption. \$59.2's claim, and the difference between a usable implementation and one that multiplies scanning cost by the number of items held.

A hundred items: one hundred index lookups per block. Negligible.

The error to avoid: treating item accounts as ordinary accounts requiring full trial decryption each block. That gives:

```
cost = 100 × avg_actions_per_block × decrypt_cost    per block
```

which is a hundred times the cost of the wallet itself and is unusable. This is not hypothetical — it is what a naive implementation does.

N.7 Verification Cost

Per hop: one key derivation, one trial decryption against a *known* Action, one commitment recomputation, one tree membership lookup, one nullifier computation, one nullifier set lookup.

```
cost_per_hop ≈ 1 scalar mult + 1 symmetric decrypt + 2 hashes + 2 index reads
```

All constant-time in the chain's length. The trial decryption is against one known Action rather than against every Action, which is the difference between verification and scanning.

```
total = hops × cost_per_hop
```

§38.4's claim. A ten-hop item verifies in milliseconds, dominated by the two index reads per hop.

Part XXXII — Reading This Document

A document of this length is read in pieces. This chapter says which pieces, for whom.

120. If You Are Auditing the Construction

Read, in order: Chapter 34 (the nullifier and the three cases), Chapter 47 (the uniqueness argument), Chapter 86 (the assumption set), Appendix H (the claim index, sorted by confidence).

Attack, in order:

1. §34.4's exhaustiveness. Is there a fourth way a note comes into existence?
2. §23.2's exhaustiveness. Is there a third double-spend-protected resource?
3. §44.3 step 2. Does the key-binding check actually bind, or can a seller substitute a key that decrypts without owning?
4. §81.1's preimage assumption, which is the one place the construction leans harder than the currency does.
5. §48.2's fourth case, which this document concedes is open.

The single citation to verify first: §34.3's sentence from the Action specification. Everything else follows from it, and if it says something other than what this document quotes, the construction collapses.

121. If You Are Implementing

Read: Part XIV (chapters 43–45), Appendix B (the specified procedure), Appendix C (the tests), Appendix K (the errors), Appendix I (the checklist).

Do first: C.2.1, the same-Action placement test. It is the failure that produces working-looking software whose output is broken, and the breakage is discovered by a third party months later (§45 item 1).

Do second: C.5.1, the seed-only restore. It is the test that distinguishes this construction from the mature alternatives, and an implementation that fails it has given up the advantage while keeping the novelty (§29.4).

Do not skip: the key-binding check (C.3.2) and the present-ownership check (C.3.1). Both are easy to omit and both are load-bearing.

Read twice: §83.3 on `ak` versus `ask`. The distinction is one letter and the consequence is total.

122. If You Are Issuing a Collection

Read: Chapter 43 (the procedure), §57.2 (the availability commitment), §70.2 (royalties), §88.3 (migration), §102 (whether to use this at all).

Understand before committing:

- Royalties are not enforceable and not detectable (§70.2). A business model that assumes them will not work.
- Content availability is your permanent obligation (§57.2). The cryptography does not store your images.
- A collection built now does not become a native asset later (§88.3). It becomes a legacy collection.
- Your buyers learn the item's full prior history (§41). If that is a problem for your collection, do not use this.
- You cannot reclaim, freeze, or modify items after mint (§93). This is a feature and it is irreversible.

The one thing to get right: anchor once, and publish the genesis transaction identifier everywhere (§43.4). It is your collection's identity, and a buyer who does not check it can be defrauded by anyone (§40.5).

123. If You Are Buying

Read: §44.3 (what verification checks), Appendix K (what the errors mean), §41 (what you learn and what you give up).

Know:

- Verification against your own node is the only verification that means anything (§42.1). If the interface verifies for you, you are trusting the interface.
- **Spent** is an answer, not a malfunction (§B.5). If you see it, walk away.
- A node that is not fully synced gives plausible wrong answers (§K.4). Check sync before you check ownership.
- You will learn the item's prior history, and future buyers will learn yours (§41).
- Between paying and receiving, the venue holds your money (§42.2). That is custody, not escrow.

The one thing to check: the genesis transaction identifier matches the collection you think you are buying into.

124. If You Are Deciding Whether Any of This Applies

Read only: Chapter 19 (the three architectures), Chapter 23 (the boundary), Chapter 102 (the five ways), Chapter 89 (what this project owes its users).

The decision reduces to two questions. Is ownership privacy a requirement? Is public lookup a requirement?

If privacy is not required, use the account model — it is better at everything else (§98.4). If public lookup *is* required, this construction cannot provide it and no amount of engineering will change that (Chapter 23). Only the intersection — privacy required, public lookup not — is this document's subject, and §102 says so plainly.

Part XXXIII — Paying From Outside

A privacy chain is a small island. Most buyers hold something else, and telling them to acquire the native asset first is telling most of them to leave. This Part specifies how a buyer pays in another asset and the venue still receives shielded value, and it is included because the design has one non-obvious constraint that every naive attempt violates.

126. The Constraint

126.1 Memos Do Not Survive Conversion

The ordinary way a payment identifies itself is the memo: the buyer sends value with a tag, and the venue matches the tag to the order.

No memo survives a conversion. The buyer sends asset X to an intermediary; the intermediary sends the native asset to the venue. The second transaction is the intermediary's, constructed by the intermediary, with a memo field the intermediary controls and does not expose. There is no protocol by which a memo attached to the first transaction appears in the second, and no intermediary offers one.

This is not a limitation of any particular service. It is structural: the two transactions are on different chains with different formats, and the intermediary is the author of the second.

126.2 What Replaces It

The intermediary supplies a **unique deposit address per conversion**. The venue therefore:

1. Creates the order.
2. Requests a conversion quote, naming its own shielded address as the destination.
3. Receives a deposit address unique to that quote.
4. Records the mapping from deposit address to order.
5. Shows the buyer the deposit address.

When value arrives at the venue's shielded address, the venue does not identify the order by inspecting the payment. It identifies it by polling the intermediary for the status of each outstanding quote, and the quote *is* the order key.

The order identity lives in the venue's record of the deposit address, not in anything on either chain. This is the design's one piece of off-chain bookkeeping and it is unavoidable.

126.3 The Amount Must Be Fixed on the Output Side

A conversion quoted on the input side — "send 0.01 of asset X, receive whatever it converts to" — delivers an amount that varies with the rate between quote and execution. The venue then receives an amount that does not equal the listing price, and must either refund the difference, accept underpayment, or reject.

A conversion quoted on the **output** side — "receive exactly 1.5 of the native asset, send whatever that costs" — delivers the exact listing price. The buyer absorbs the rate variance, which is the correct allocation: the buyer chose to pay in another asset.

An implementation that does not use output-side quoting will spend its life reconciling near-misses.

127. What Was Verified

Two facts about the intermediary used in this project's implementation were established by live testing rather than by reading documentation, and both contradict the documentation.

Shielded destinations are accepted. The documentation states that destinations must be transparent. A live quote naming a shielded unified address was accepted and returned a valid deposit address. The documentation is wrong.

Output-side quoting is available. Also undocumented in the form needed, also confirmed by a live quote.

The general lesson is the one in §G.3: **for third-party services, test rather than read.** Documentation describes an intention; the API describes the behaviour, and only one of them is what your code meets.

128. Where the Venue's Fee Goes

128.1 The Naive Error

Intermediaries typically offer an application-fee parameter: the integrator names a percentage and the intermediary deducts it.

Using it here **double-charges**. The venue already takes its percentage from the seller's proceeds — that mechanism exists and is independent of how the buyer paid. Adding an intermediary-level fee takes a second cut from the same trade.

128.2 The Correct Arrangement

Quote the conversion for the **full listing price**, with no application fee. The venue receives the whole price and deducts its percentage from the seller's proceeds exactly as it does for a native-asset payment.

The buyer pays the listing price plus the intermediary's own spread, which is the intermediary's business and is disclosed by the quote. The venue's economics are identical regardless of payment asset, which is the property worth having: **one fee path, exercised by every trade, tested once.**

129. Settlement Through the Same Path

129.1 Why This Matters

A cross-asset payment that settles through a separate code path is a second implementation of settlement, with its own bugs, its own payout logic, and its own opportunity to get the accounting wrong.

The design avoids this. When the intermediary reports success, the venue **synthesises a payment record** — the transaction identifier, the amount, and the memo the buyer would have sent had they paid natively — and feeds it to the same settlement function that handles native payments.

Everything downstream is unchanged: order matching, fee deduction, payout accrual, and the seller's balance. The cross-asset path is a translation layer, not a parallel system.

129.2 What This Buys

The settlement logic has one implementation and one test suite. A bug in fee calculation is a bug in one place. A change to payout policy applies to both paths automatically.

It also means the cross-asset path inherits the settlement path's correctness properties without re-proving them — which matters, because settlement is where money is.

130. Refunds Are in the Wrong Asset

A bid funded by conversion and later withdrawn refunds in the **native** asset, not the one the buyer sent.

This follows from the conversion being one-way: the venue holds native value and has no mechanism to convert back on the buyer's behalf, nor any reason to take that rate risk.

It must be disclosed before the buyer commits. A refund in an unexpected asset is a surprise that generates support load and, entirely reasonably, anger. The disclosure belongs on the funding screen, in the flow, not in terms of service.

131. Minting

Minting is issuance with a buyer, and it has two failure modes that ordinary sales do not.

131.1 The Treasury Question

A sale routes value to the seller. A mint routes value to the issuer. These are different destinations and the code must distinguish them.

The resolution used here is that an order carries a **treasury designation** — market or mint — resolved when the order is created, and the payment destination is a function of that designation rather than a constant. An implementation with a single hard-coded venue address works correctly for sales and silently misroutes mint proceeds.

This was a real defect in this project's implementation and it was found by a test that asserted the destination rather than by inspection.

131.2 The Allocation Race

A mint allocates an item from a finite supply. Two buyers paying simultaneously must not receive the same item.

Under this construction the allocation is not a database row; it is the issuer's choice of which item note to spend. The race is therefore resolved where it should be — by the chain — because two transfers spending the same item note produce a duplicate nullifier and the second is rejected (§34.4, case 2).

This is worth noticing as a structural benefit. In a conventional mint, the allocation race is an application-level concurrency problem solved by locking. Here the losing transaction is rejected by consensus, and the correct application behaviour is to retry with a different item rather than to prevent the race.

131.3 Commit-Reveal for Fair Allocation

If item metadata is published before allocation, buyers can target desirable items and the mint degrades into a race for the best ones.

The mitigation is to publish commitments to item metadata at genesis and reveal the metadata after allocation. The manifest already commits to metadata digests (§56.1); withholding the metadata bytes until after the mint is sufficient, with no protocol change.

The cost is that buyers cannot verify what they are buying until after they have bought it — which is the same trade every blind mint makes, and which should be labelled as such.

131.4 What a Mint Buyer Must Still Check

The genesis identifier (§43.4). The manifest's digest against the anchor. That the item they received verifies (§44.3) — which, for a freshly minted item, is a one-hop chain and is the

cheapest verification the system has.

A mint interface that skips verification because “we issued it ourselves” has trained its users to skip the check that protects them from an issuer who is not the one they think (§40.5).

132. What This Part Does Not Solve

The buyer's privacy on the other chain. A conversion begins with a transaction on a transparent chain, visible in the ordinary way. The construction's privacy begins at the venue's shielded address, not at the buyer's wallet.

A buyer whose threat model includes their own chain's observers gains nothing from the shielded destination — they have simply moved value from a public chain to a private one, and the departure is public. This is the pool-boundary observation of Chapter 36 in an inter-chain register, and it should be said to users rather than implied away.

The intermediary's knowledge. The intermediary knows the source asset, the amount, the deposit address, and the destination. It is a trusted third party for the duration of the conversion and a data source afterwards.

The mitigation is that the destination is the *venue's* shielded address, not the buyer's, so the intermediary does not learn which item or which holder. It learns that somebody converted into the venue.

Appendix 0 — Order Lifecycle

The states an order passes through, what moves it, and what is true in each. Included because the failure modes of a marketplace are mostly state-machine failures, and an unstated state machine is one nobody can review.

0.1 States

State	Means	Money held by	Item held by
draft	Seller is composing	seller	seller
listed	Publicly offered	seller	seller
awaiting_funds	Buyer committed, payment not confirmed	buyer	seller
funded	Payment confirmed	venue (custody)	seller
verifying	Re-running the present-ownership check	venue	seller
transferring	Seller's transfer broadcast, unconfirmed	venue	in flight
settled	Transfer confirmed, proceeds sent	seller	buyer
refunding	Something failed; returning funds	venue	seller
refunded	Terminal	buyer	seller
withdrawn	Seller delisted before commitment	seller	seller

The `funded` and `verifying` states are where custody exists. \$61.1's discipline is about making them short.

0.2 Transitions

From	To	Trigger	Must be true
draft	listed	Seller publishes	Verification returns <code>CurrentlyHeld</code>
listed	withdrawn	Seller delists	No buyer committed
listed	awaiting_funds	Buyer commits	Price and item unchanged
awaiting_funds	funded	Payment confirms	Amount matches; memo or deposit address matches (\$126.2)
awaiting_funds	refunding	Timeout	—
funded	verifying	Immediate	—
verifying	transferring	Re-check passes	Last nullifier still absent (\$B.6)
verifying	refunding	Re-check fails	Seller sold elsewhere (\$48.2 case 4)
transferring	settled	Transfer confirms	Buyer's verification passes
transferring	refunding	Transfer fails or times out	—
refunding	refunded	Refund confirms	In the native asset (\$130)

0.3 The Two Transitions That Matter

verifying → **transferring** is the one that prevents paying for an item the seller no longer holds. It is one nullifier lookup and it must happen after the money confirms, not before. An implementation that verifies at listing time and settles on a schedule has a window measured in hours.

transferring → **settled** must wait for the buyer's own verification, not merely for confirmation. A transfer can confirm and still produce a broken chain if same-Action placement failed (§44.1). Settling on confirmation alone pays a seller whose transfer did not do what it was supposed to.

0.4 What Cannot Be Reversed

Once `transferring` is broadcast, the item has moved. If settlement then fails, the venue owes the seller money and the buyer has the item.

The mitigation is ordering: confirm the money first, then move the item. This makes the venue's exposure the *seller's proceeds* rather than the *item*, which is the correct direction — money is fungible and recoverable, an item is not.

0.5 Where the Race Remains

§48.2's fourth case. Between `verifying` passing and `transferring` broadcasting, the seller can broadcast a competing transfer. Whichever confirms first wins; the other is rejected as a duplicate nullifier.

If the seller's competing transfer wins, the venue holds the buyer's money and the item went elsewhere. The venue refunds. The buyer has lost time and the seller has double-sold successfully against one of two buyers.

Only atomic settlement closes this (§D.1). Until then, the mitigation is that the window is milliseconds and the seller gains nothing they could not gain more easily by simply not listing.

Appendix P — Data Model

What the venue must store, and — more importantly — what it must not.

P.1 Collections

```
collections:
  genesis_txid      TEXT PRIMARY KEY      -- the collection's identity (§43.4)
  manifest_digest   BYTEA NOT NULL
  manifest_uri      TEXT NOT NULL
  anchor_height     INTEGER NOT NULL
  item_count        INTEGER NOT NULL
  treasury          TEXT NOT NULL         -- 'market' | 'mint' (§131.1)
```

No holder column. There cannot be one — the venue does not know and cannot learn (Chapter 23).

P.2 Listings

```
listings:
  id          BIGSERIAL PRIMARY KEY
  genesis_txid TEXT NOT NULL
  item_index  INTEGER NOT NULL
  price_zat   BIGINT NOT NULL
  memo_tag    TEXT NOT NULL UNIQUE -- the native-payment matcher
  last_nullifier BYTEA NOT NULL    -- for the 0.3 re-check
  state       TEXT NOT NULL
  created_at   TIMESTAMPTZ NOT NULL
```

`last_nullifier` is the only chain-derived secret-adjacent value stored, and it is not secret — it is public once spent, and before that it is a value the seller computed and chose to share so the venue can perform the re-check.

It is nonetheless a correlation key. A venue holding `last_nullifier` for every listing can, once those nullifiers appear on chain, identify exactly which transactions were sales. This is a real leak from the venue's database and it is the price of the \$0.3 re-check.

The mitigation is retention: delete `last_nullifier` on reaching a terminal state. The re-check needs it for minutes, not forever.

P.3 Bids

```
offers:
  id          BIGSERIAL PRIMARY KEY
  genesis_txid TEXT NOT NULL
  item_index  INTEGER          -- NULL for a collection-wide bid
  price_zat   BIGINT NOT NULL
  memo_tag    TEXT NOT NULL UNIQUE
  address_commit BYTEA NOT NULL -- commitment, NOT the address (§92)
  state       TEXT NOT NULL
```

`address_commit` rather than the address is the pattern §92 recommends against compulsion. The bidder reveals the address when the bid is taken; until then the venue holds a commitment it cannot open.

P.4 Cross-Asset Orders

```
bridge_orders:
  deposit_address  TEXT PRIMARY KEY    -- the order key (§126.2)
  memo_tag         TEXT NOT NULL       -- links to the listing or offer
  deliver_address  TEXT NOT NULL
  origin_asset     TEXT NOT NULL
  amount_out_zat   BIGINT NOT NULL     -- output-side quote (§126.3)
  status          TEXT NOT NULL
  created_at       TIMESTAMPTZ NOT NULL
  delivered_at     TIMESTAMPTZ
```

The primary key is the deposit address because that is what the intermediary keys on and therefore what the venue must key on to reconcile.

P.5 What Must Not Be Stored

Never store	Why
Consignments	They pass between counterparties; a venue holding them holds every item's full history (§92)
Viewing keys of any kind	Same, and worse — a full viewing key confers spend detection
Buyer chain addresses before a bid is taken	§92; store a commitment
Seller chain addresses	Not needed; proceeds go to an address supplied per settlement
<code>last_nullifier</code> after settlement	§P.2; it becomes a sale identifier
Request paths in access logs	§76.4

The first two are absolute. A venue that holds consignments has made itself the single most valuable target in the system, and the compulsion analysis of Chapter 92 becomes: it knows everything.

Appendix Q — Document Map

Each Part, in one line, with what it establishes and what depends on it.

Part	Establishes	Depended on by
I–VI	The project, its staging, and its original premises	— (note the terminology debt, §35.2)
VII (19–21)	The three architectures; the inscription record; why Ordinals cannot port	VIII, X, XVII
VIII (22–24)	The construction ; the impossibility boundary; data-carrying economics	Everything after
IX (25–27)	The asset layer is orphaned; what it would give; settlement is unresolved	XX, XXX
X (28–29)	Ordinals dissected; the other four designs compared	XXI, XXII, XXIX
XI (30–34)	The Action, keys, encryption, tree, the nullifier	XV, XXVI, all of XIV
XII (35–38)	The sealing; pool boundaries; the turnstile; proving costs	XIII, XVI
XIII (39–42)	The adversary; attacks that fail; the leak that remains	XXVIII, XXX
XIV (43–45)	How to build it ; failure modes ranked	Appendices B, C, I
XV (46–50)	Formal definitions and arguments for the four properties	XXX, Appendix H

XVI (51–53)	What each operation costs	XXIX, Appendix N
XVII (54–55)	What was actually deployed on this chain, and what it implies	VIII’s design choices
XVIII (56–58)	Content commitment, availability, why mutability is excluded	XIV, XIX
XIX (59–61)	Wallet, indexer, settlement infrastructure	Appendix I
XX (62–64)	The three upgrades that would help; the circuit that closes the gap	XXXI
XXI (65–68)	What the inscription ecosystem learned	XXII, XXIX
XXII (69–72)	Why the account model does not port	XXIX
XXIII (73–75)	What is actually known about linkability; a correction	XIII, XXVIII
XXIV (76–77)	Network exposure; light clients	XXVIII, Appendix D
XXV (78–79)	How the specifications got here; the four structural checks	Appendices G, M
XXVI (80–86)	The primitives, and where every assumption lives	XV, Appendix H
XXVII (87–89)	Operations; migration; what this project owes its users	Appendix I
XXVIII (91–95)	Four concrete adversaries	XXX
XXIX (97–102)	The same collection five ways; when to use this at all	XXXII
XXX (103–111)	Objections, answered or conceded	—
XXXI (113–118)	What changes if each open question resolves	Appendix D
XXXII (120–124)	How to read this document, by role	—
XXXIII (126–132)	Paying from another chain; minting	Appendices O, P

Q.1 The Shortest Honest Path Through

A reader with an hour: §19 (architectures), §23 (the boundary), §34 (the nullifier), §44 (the procedure), §102 (when to use it), §89 (what is owed).

That is the argument. Everything else is support, derivation, comparison, or operational detail.

Q.2 The Load-Bearing Citation

One sentence in one specification (§34.3, reference [2]). If it says something other than what is quoted, the construction has no foundation and the rest of the document is an elaborate error.

It is Final, it is public, and it takes a minute to check. **Check it.**

Appendix R — Data Channels Across Chains

Chapter 24 gave Zcash’s numbers. This appendix places them alongside the chains this document compares against, because “Zcash cannot carry data” is a claim that turns out to be half-right in a specific and interesting way.

R.1 Public Channels

Chain	Channel	Capacity	Per-tx limit	Discount	Notes
Bitcoin	Data output	uncapped (since Oct 2025)	policy	none	\$24.1
Bitcoin	Witness script	~4 MB block budget	per-push 520 B	4×	The inscription channel (\$28.4)
Bitcoin	Input script (legacy)	~1,650 B	standardness	none	Malleable (\$54.2)
Zcash	Data output	83 B script / 80 B payload	exactly 1	none	Unanimous across implementations
Zcash	Input script	~1,650 B	standardness	none	The Zerdinals channel (\$54.1); malleable
Zcash	Coinbase script	2–100 B	miner only	none	Not general purpose

R.2 Private Channels

Chain	Channel	Capacity	Readable by	Cost per byte
Bitcoin	none	—	—	—
Zcash	Shielded memo	512 B per output	recipient only	~19.5 zat
Zcash	Memo bundle (draft)	16 KiB per tx	recipient, or public if flagged	not priced

This is the table that reframes the comparison. Zcash’s public channel is the smallest in the set and its private channel has no counterpart at all. A protocol that needs public bytes should not be here; a protocol that needs private bytes has nowhere else to go.

R.3 The Inversion

The usual framing is that Bitcoin is permissive about data and Zcash is restrictive. Per byte of *public* data that is true. Per byte of *deliverable* data it inverts:

	Public bytes available	Private bytes available
Bitcoin	effectively unlimited	0
Zcash	80 per transaction	1,024 per two-action transaction

And the private bytes are cheaper than Bitcoin’s *discounted* public bytes in relative terms, because they are pre-paid by a ciphertext whose size is fixed for privacy reasons (§N.3).

R.4 Direction of Travel

When	Chain	Change
Oct 2025	Bitcoin	Data-carrier default raised to effectively uncapped
Jan 2026	Zcash	A node implementation <i>introduces</i> the 83-byte cap it never enforced
—	Zcash	No proposal to restrict, expand, or remove the cap has ever been filed

The two chains moved in opposite directions within three months. Neither move was a response to the other, and the coincidence is worth noting only because it makes the divergence in inscription activity (§28.4) look cultural when it is mostly a pricing fact that got more extreme.

Appendix S — Auditing a Claim Like This One

The construction of Part VIII is the kind of claim that appears regularly and is usually wrong. This appendix is the checklist this document would want applied to itself, stated generally enough to apply to the next one.

\$.1 The Six Questions

1. What resource enforces uniqueness, and is it public?

Every uniqueness guarantee runs through a resource that consensus can name, and every such resource is public (§23.1). A design claiming uniqueness without naming one is claiming magic. A design that names a public one and also claims privacy is claiming a contradiction unless the binding to the resource is hidden.

This document's answer: the nullifier. Public, unlinkable, binding hidden (§23.2).

2. Is the successor pointer a convention or a constraint?

The question "which thing continues this thing" is answered either by software agreement or by something the chain computes. The first is mutable by software release (§28.5); the second is not.

This document's answer: p, constrained in circuit (§34.3). **This is the only novel part and it should receive the most scrutiny.**

3. Who must cooperate for a verifier to get a true answer?

If the answer includes any party other than the chain, that party is trusted, and the design should be described as trusted.

This document's answer: the discloser must supply the disclosure, and after that nobody — a lying discloser cannot manufacture a second valid history (§23.1). Verification touches only the verifier's node (§42.1, C.4.1).

4. What does each new participant learn permanently?

Disclosure-based systems leak to recipients. The question is what, and whether it is bounded.

This document's answer: the item's full prior chain, bounded to accounts holding one note each (§41). Stated as a disqualifying limitation for some use cases (§41.3).

5. What is the assumption set, relative to the underlying chain's?

A construction adding assumptions is riskier than one that does not. Count them.

This document's answer: the currency's assumptions plus preimage resistance of one hash, with a stated degradation rather than collapse if it fails (§86, §81.1).

6. Does it recover from a seed?

Client-side-validation systems generally do not, and this is the property users discover they needed after a device dies.

This document's answer: yes, via self-addressed memos, and it is tested (C.5.1). It is also the single feature distinguishing this from the mature alternatives (§29.4), so a reader should be sceptical and run the test.

S.2 Warning Signs in Any Such Document

Sign	What it usually means
"Private and publicly verifiable" without a proof	One of the two is false; see §23.1
A proof size quoted without saying what the verifier already holds	The comparison is not like-for-like (§29.3)
"Escrow" for a mechanism with no conditional spending	It is custody (§42.2)
A roadmap phase depending on a draft specification	The timeline is unbounded (§79.1)
A linkability figure quoted in the present tense	Check what it measured and when (§73)
A specification's Deployment section cited as current	Check the dependency graph (§G.2)
No stated false-positive rate for a cited attack	Nobody measured (§74.1)
A limitations section shorter than the claims section	—

\$.3 What This Document Would Fail On

Applied honestly to itself:

- **Unaudited.** No third party has reviewed the construction or any implementation (§89, §108).
- **One unverified critical claim.** Two-party co-construction (§D.1), on which the settlement story depends.
- **One unsolved problem presented as future work.** Succinct nullifier non-membership (§D.2), which gates the gap-closing circuit.
- **Two named attacks against itself that nobody has run.** Anchor-age and Action-count fingerprinting (§D.3, §D.4).
- **A terminology debt in its own earlier chapters.** §35.2.
- **A corrected overclaim in its own record.** §H.7.

A reader who wanted to be unkind could write the critique from this list, and that is the point of including it.

Part XXXIV — What This Changed

A whitepaper that does not say what it revised is a marketing document. This chapter records what the analysis changed about the project's own plan, because the changes are the evidence that the analysis was done rather than assembled.

134. Designs Abandoned

Memo-passing in the previous pool. The first design sent items as annotated notes between holders. It worked when written and became impossible at a fixed block height when the pool was sealed (§35.2). Nothing survived it except the lesson: depend on structure, not on permission (§35.4).

Transparent seals with shielded funding. The second design used transparent outputs as seals and funded them from the pool so the funder was unlinked. It was abandoned when the boundary analysis showed the attack goes through amounts rather than addresses (§36.2), that a marketplace publishes exactly the amount an analyst needs, and that the item itself is a non-expiring join key. The correct claim for it — “no transparent funding history” — is much weaker than privacy, and describing it as private ownership would have been false (§99.4).

Waiting for the asset layer. The original roadmap’s second phase required a specification that turns out to have no transaction format, no fee rule, no identifier, no scheduled upgrade, and no constituency (Chapter 25). The phase was not delayed; it was removed.

Mutable metadata. Requested as a product feature and excluded because it reintroduces the mutable registry the architecture exists to avoid (§58.2).

135. Claims Withdrawn

"Closure is unverifiable by construction." Too strong. Closure is unverifiable by an *arbitrary* observer and verifiable by anyone given 96 bytes. Client-side validation never required arbitrary-observer verifiability; on transparent chains it only appears to have it because seal identifiers happen to be public there (§103).

The linkability figure in the present tense. A historical measurement of a configuration that no longer exists, quoted as current (§73). Replaced with the defensible results of Chapter 74.

"The correlation is not inferential." An activity spike was attributed to inscription volume using data-output counts. The deployed protocol uses input scripts, so those counts cannot detect it. Rewritten to present both sources and label the causal link as inferred (§H.7).

"The next upgrade is the one assets ship in." Wrong, and it came from a specification's own stale Deployment section (§25.3). It was caught before publication.

136. What Was Added Because of the Analysis

The present-ownership check. Step 8 of verification (§44.3) exists because the stale-proof attack (§40.2) was identified during the formal treatment. An earlier version verified history only and would have accepted a disclosure from a seller who had already sold.

The one-account-per-item rule. §31.3, promoted from hygiene to a load-bearing rule when the disclosure leak was analysed (§41.2) and found to be bounded only if accounts are not reused.

The same-Action assertion. §44.1, added when it became clear the failure is silent and is discovered by a third party months later.

The key-binding check. §44.3 step 2, added when it was noticed that a key which merely decrypts is not a key which owns.

Treasury designation. §131.1, added after a test asserted the payment destination and found mint proceeds routed to the wrong place.

137. What Remains Undone

The terminology sweep. Parts I–VI name the previous pool where they mean the current one (§35.2). The claims survive; the names do not. It is a mechanical correction and it is owed.

The co-construction experiment. §D.1. Until it is run, settlement is custody and the document says so.

The open security finding. §89. Documented, unclosed, with the amount at risk stated rather than managed.

An audit. Of the construction and of any implementation. Nothing here has had one (§108).

Appendix T — Wire Formats

The document references a manifest, an anchor payload, and a consignment throughout without specifying them. This appendix does. The formats are deliberately boring: fixed-width fields, explicit lengths, a version byte, and no self-describing structure, because §56.3 argues that the way to eliminate canonicalisation bugs is to refuse to canonicalise.

T.1 The Anchor Payload

Placed in the single data output of the genesis transaction (§43.4).

offset	size	field	
0	4	magic	"ZBT1" (0x5A 0x42 0x54 0x31)
4	1	version	0x01
5	32	manifest_digest	BLAKE2b-256 of the manifest bytes
37	1	flags	bit 0: metadata withheld until mint (§131.3) bits 1-7: MUST be zero
38		total	

38 bytes, inside the 80-byte payload budget (§24.1) with 42 bytes spare. The spare is deliberate and unused: a future version may need it, and a format that exactly fills its budget has no room to evolve.

Parsing rule. A payload whose first four bytes are not the magic is not this protocol's and must be ignored, not rejected — other protocols share the channel.

Validation rule. A payload with the magic, a known version, and a wrong length is `MalformedAnchor` (§K.1). An unknown version is `MalformedAnchor` as well, not an ignorable condition: an implementation that cannot parse a collection must not claim it verified one.

T.2 The Manifest

A byte string. Hashed as published; parsed only after the hash matches (§B.3).

offset	size	field	
-----	----	-----	
0	4	magic	"ZBM1"
4	1	version	0x01
5	2	name_len	big-endian u16
7	n	name	UTF-8, not normalised
7+n	4	start_index	big-endian u32 (§43.1)
11+n	4	item_count	big-endian u32
15+n	?	items	item_count × 64 bytes
?	1	issuance_policy	0x00 closed, 0x01 open (advisory)
?	2	note_len	big-endian u16
?	m	note	UTF-8, free text

Each item entry:

offset	size	field	
-----	----	-----	
0	32	cm	note commitment (§46.1)
32	32	metadata_digest	BLAKE2b-256 of the metadata bytes

The genesis identifier is not in the manifest. It cannot be — the manifest is committed by the anchor, and the anchor's identifier is not known until the anchor is confirmed (§A.2). Readers obtain it from the transaction they found the payload in. An implementation that includes a genesis field in the hashed bytes cannot produce a valid manifest, and this is the single most likely first bug.

issuance_policy is advisory. It is a statement of intent by the issuer, enforced by nothing (§50.3). It exists so that a buyer can see that the issuer claimed a closed supply, and so that a second anchor by the same issuer is visibly a broken promise rather than an ambiguity.

Items are ordered by index, starting at zero, contiguous. Index i is at offset $15 + n + 64i$. There is no index field per item because the position is the index; a format carrying both permits them to disagree.

T.3 Metadata

Not specified here. The manifest commits to a digest of a byte string, and what that byte string contains is the collection's business.

Two rules only:

It must contain the content's digest if the item has content, or the commitment chain of §56.1 does not reach the content.

It must be hashed as published. Whatever format is chosen, the digest is of the bytes served, not of a re-serialisation (§56.3, C.3.6).

A collection may use a structured format, a plain text format, or an image directly. The protocol does not care and should not.

T.4 The Consignment

Stored in a self-addressed memo (§44.4), or transmitted between counterparties.

offset	size	field	
0	4	magic	"ZBC1"
4	1	version	0x01
5	1	chunk_index	0-based (§A.5)
6	1	chunk_count	total chunks
7	32	genesis_txid	
39	4	item_index	big-endian u32
43	2	hop_count	big-endian u16, total across chunks
45	?	hops	this chunk's hops, 96 bytes each

Header is 45 bytes, leaving 467 bytes for hops in a 512-byte memo — four full hops of 96 bytes with 83 bytes unused, or five hops if the header is carried once and subsequent chunks use a short header.

The short header for continuation chunks:

offset	size	field	
0	4	magic	"ZBC1"
4	1	version	0x01
5	1	chunk_index	
6	1	chunk_count	
7	32	genesis_txid	retained for reassembly safety
39		total	

39 bytes, leaving 473, which is four hops. §N.5's figure of five assumed ~30 bytes of overhead; the specified header is larger, and the honest figure is **four hops per memo** for the first chunk and four for continuations.

This is worth correcting explicitly rather than leaving the earlier estimate standing: a twenty-hop item needs five memos, not four.

Each hop entry:

```
offset  size  field
-----  ----  -
      0    32  ak          spend VALIDATING key (§83.3)
     32    32  nk
     64    32  rivk
```

ak, not **ask**. The format has no field for a spending key and must never acquire one (§83.3).

T.5 Reassembly

```
reassemble(chunks) -> Consignment or Error:
    if any chunk's magic or version differs:      fail
    if any chunk's genesis_txid differs:          fail
    if chunk_count differs across chunks:         fail
    if the set of chunk_index is not 0..count-1:   fail Incomplete
    sort by chunk_index
    concatenate hop bytes
    if total hops != hop_count from chunk 0:      fail
    return assembled
```

Incomplete is a recoverable state, not an error: a wallet restoring from seed may find chunks across several blocks and should wait for the scan to finish before concluding a consignment is lost.

T.6 Versioning

Every format carries magic and a version byte. The rules:

- An unknown magic is not this protocol's: **ignore**.
- A known magic with an unknown version: **refuse**, do not guess.
- A known version with a wrong length: **refuse**.

The asymmetry between ignore and refuse matters. Ignoring foreign data lets the channel be shared. Refusing unknown versions prevents an old implementation from producing a confident wrong answer about a collection it does not understand — which, in a system whose entire value is provability, is worse than producing no answer.

This is the same principle as the odd/even tag numbering of §28.2, reached independently and applied more conservatively: Ordinals permits ignoring unknown odd tags because a wrong interpretation of an inscription is cosmetic. Here a wrong interpretation is a claim about ownership, so nothing unknown is ignorable.

T.7 What Is Deliberately Absent

Absent	Why
Timestamps	Derivable from the chain; a stored one can disagree
Heights	Same
Commitments in the consignment	The verifier recomputes them; a carried value is attacker-controlled (§B.1)
Nullifiers in the consignment	Same
Issuer signature on the manifest	The anchor is the authentication; a signature adds a key to manage and proves nothing extra
Any field the verifier would trust rather than check	§B.1

The last row is the principle. **Every field in these formats is either recomputed and checked, or is advisory and labelled as such.** A format with a field that is neither is a format with an attack in it.

Appendix U — Trace of a Seal Chain

The relationship $\rho = \text{nf}$ is stated repeatedly and never shown. This appendix traces it concretely. Values are illustrative placeholders in the shape of real ones; the structure is what matters.

U.1 Genesis

The issuer creates the item note for item 1 in Action 1 of transaction T_0 .

```
T0 , Action 1
  spend:  some funding note F
          nullifier(F) = nf_F = 0x7a3c...e91b
  output: note N1
          recipient = A1's address
          value      = 1 zatoshi
          p          = nf_F = 0x7a3c...e91b    ← the circuit sets this
          rcm        = fresh randomness
          commitment cm(N1) = 0x4f80...22ad
```

Note what happened. The issuer did not choose p . The circuit set it to the nullifier of whatever note funded the Action. **The item's first note is already the successor of the funding note** — the chain does not begin at the item, it begins wherever the money came from, and the manifest simply declares one link in an already-existing chain to be the start of an item.

The manifest records:

```
item 1 → cm = 0x4f80...22ad , metadata_digest = 0x9c1e...0447
```

U.2 The First Transfer

Alice buys. The issuer spends N_1 and creates N_2 in the same Action.

```
T1 , Action 0
  spend:  note N1
          nullifier(N1) = nf1 = 0xb206...7f35    ← computed from N1 and nk(A1)
  output: note N2
          recipient = B1's address (Alice)
          ρ         = nf1 = 0xb206...7f35        ← the circuit sets this
          commitment cm(N2) = 0xe17d...5b0c
```

The chain link is now visible: $N_2.\rho == \text{nullifier}(N_1)$.

What the chain publishes: nf_1 (in the Action's nullifier field) and $\text{cm}(N_2)$ (in the Action's cmx field). Both public, in the same Action, and *not visibly related* — the relation is inside N_2 , which is inside the commitment, which is hiding.

What an observer sees: one nullifier, one commitment, no connection.

What a verifier holding $\text{nk}(A_1)$ sees: they compute nf_1 themselves, find it on chain, read the output of that Action, and now know $\text{cm}(N_2)$ is the successor. They can only do this because they were handed a key.

U.3 The Second Transfer

Alice sells to Bob.

T_2 , Action 2

```
spend:   note  $N_2$ 
         nullifier( $N_2$ ) =  $nf_2$  = 0x3388...ca70

output:  note  $N_3$ 
         recipient =  $C_1$ 's address (Bob)
          $\rho$          =  $nf_2$  = 0x3388...ca70
         commitment  $cm(N_3)$  = 0xa591...1de8
```

U.4 The Chain, Assembled

```

manifest → cm(N1)=0x4f80...22ad
      |   nf1 = 0xb206...7f35   published in T1 Action 0
      ↓
      cm(N2)=0xe17d...5b0c       with p = 0xb206...7f35 ✓
      |   nf2 = 0x3388...ca70   published in T2 Action 2
      ↓
      cm(N3)=0xa591...1de8       with p = 0x3388...ca70 ✓
      |   nf3 = 0x55e4...9002   NOT on chain
      ↓
      Bob currently holds

```

Each ✓ is a check the verifier performs by decrypting the note and comparing its p to the nullifier they computed for the previous hop. Each is enforced in-circuit at the time the transfer was made — the verifier is confirming a constraint the chain already checked, not establishing a new fact.

The final line is the present-ownership check (§44.3 step 8): nf_3 is absent, so N_3 is unspent, so Bob still holds it.

U.5 Why a Second N_2 Cannot Exist

Suppose an attacker wants a different successor to N_2 . They need a note with $p = 0x3388...ca70$.

Same bundle. Another Action in T_2 producing that p must spend a note whose nullifier is $0x3388...ca70$. That is N_2 , already spent in Action 2. Duplicate nullifier within one transaction. Rejected.

Later transaction. Same requirement, and $0x3388...ca70$ is now in the nullifier set. Rejected.

Dummy. A dummy Action's output p is the nullifier of the fabricated zero-value note the attacker created. To make that equal $0x3388...ca70$ they must find a note whose nullifier hashes to a chosen value — a preimage. And if they found one, spending it would publish $0x3388...ca70$, which is a duplicate. Rejected twice.

Three cases, exhaustive over how a note comes into existence. §34.4.

U.6 What Alice Discloses to Bob

```
ZBC1 | v1 | chunk 0/1 | genesis=0x... | item=1 | hops=2  
hop 1: ak(A1) 0x... nk(A1) 0x... rivk(A1) 0x... ← the issuer's account  
hop 2: ak(B1) 0x... nk(B1) 0x... rivk(B1) 0x... ← Alice's account
```

192 bytes of keys, 45 bytes of header. Fits in one memo with room for two more hops (§T.4).

What Bob can do with it: everything in §U.4.

What Bob cannot do with it: spend from `A1` or `B1`. The disclosure contains `ak`, the validating key. Spending requires `ask`, the authorising key, which is not in the format and must never be (§T.4, §83.3).

U.7 The Same Trace, From the Observer's Side

An observer holding no keys sees, across the three transactions:

```
T0  4 actions  nullifiers: nf_F, ...  commitments: 0x4f80..., ...  balance: +ε
T1  2 actions  nullifiers: 0xb206..., ...  commitments: 0xe17d..., ...  balance: 0
T2  4 actions  nullifiers: 0x3388..., ...  commitments: 0xa591..., ...  balance: 0
```

Three shielded transactions. No field says "item." No field relates any nullifier to any commitment. The value balance is zero for both transfers, so no value crossed the pool boundary and there is nothing to match against a published price.

The only thing tying T_1 and T_2 to the item is the timing of whatever the venue published, which is \$39.3's residue and the whole of the leak.

Appendix V — The Construction on One Page

For readers who want the whole thing without the argument.

V.1 The Primitive

Seal	A shielded note, named by its commitment <code>cm</code>
Closing	Spending the note, which publishes its nullifier <code>nf</code>
Successor	The unique note whose p equals <code>nf</code>
Enforced by	The Action circuit, which constrains $p =$ the spent note's nullifier
Uniqueness from	The consensus nullifier set, which rejects duplicates

V.2 The Lifecycle

Step	Action	On chain	Off chain
Derive	One account per item, from the seed at hardened indices	—	index record
Issue	One minimal note per item, batched	n Actions	note commitments
Anchor	One data output with the manifest digest	38 bytes, once	the manifest
Transfer	Spend item note, create buyer's note, same Action	one shielded tx	append a hop
Disclose	Full viewing keys, genesis-first	—	96 B/hop
Verify	Eight steps against own node	—	milliseconds
Store	Consignment in a self-addressed memo	pre-paid bytes	—

V.3 The Properties

	Holds	Because
Anchoring	✓	Manifest digest in a confirmed transaction
Privacy	✓	Transfers are ordinary shielded payments
No mutable registry	✓	Ownership is a note, not a record
Transfer integrity	✓	$p = nf$, enforced in circuit, checked by the buyer
Public lookup	X	Chapter 23 — structurally impossible
History privacy from buyers	X	§41 — closed only by §64's circuit
Atomic settlement	X	§D.1 — open
Fungibility	X	§67.3 — structural

V.4 The Numbers

Per-hop proof	96 bytes
Transfer cost	one shielded transaction — same as a payment
Verification	$O(\text{hops})$, milliseconds, local
Anchor	38 bytes, once per collection
Consignment per memo	4 hops
Issuance, 1,000 items	~0.05 ZEC
Consensus changes required	0
New cryptographic assumptions	1 (hash preimage, degrading not collapsing)

V.5 The Three Things to Check Before Trusting This

1. **§34.3's sentence**, in the Final specification. One minute.
2. **§34.4's three cases**. Is there a fourth way a note comes into existence?
3. **§23.2's exhaustiveness**. Is there a third double-spend-protected resource on this chain?

If all three hold, the construction holds. If any fails, it does not.

138. Closing: What This System Is Betting On

Every design in this document follows from a single claim, and it is worth stating plainly at the end so that it can be argued with.

The claim. The public ownership ledger that every collectibles platform is built on is not a feature of digital collectibles. It is an artifact of the chains they happened to launch on, and it produces most of the harm: the doxxed collector, the scraped portfolio, the targeted phish, the front-run bid, the wallet that becomes a trophy case pointed at its owner. Remove that layer and the useful parts survive — provenance, verifiable scarcity, transferability, a market — because none of them ever required knowing who holds what.

138.1 What Changed While This Was Written

An earlier version of this chapter said that the shielded asset layer was the first protocol where that removal is not a compromise, and that the work was waiting on a network upgrade.

That is no longer true and Chapter 25 is the evidence. The transaction format was withdrawn. The successor format states it need not carry assets. The fee specification's asset clauses were removed. The reserved identifier was reassigned to something else. The next upgrade's scope excludes transaction format changes, and its candidate list does not include the asset layer. A coinholder poll did not select it.

None of that is a rejection on technical grounds. The cryptography is sound and the implementation works. But a specification with no format, no fee rule, no identifier, no scheduled upgrade, and no constituency is not pending — and a roadmap whose second phase depends on it has a second phase of unknown length.

So the document was rewritten around a different question: **not what becomes possible when the asset layer ships, but what is possible on mainnet as it is today.**

138.2 The Answer

The Action structure already contains a single-use seal chain. A created note's randomness is set to the nullifier of the note spent in the same Action, and the specification says why: to ensure it is unique. That makes every note in the shielded pool carry, in a field the circuit constrains and no user chooses, the identity of the note whose destruction paid for it.

The pool is therefore not a set of independent notes. It is a forest of chains, each link forged inside a proof that consensus verifies — the same proof that prevents the currency from being counterfeited.

An item is a note. Its successor is the unique note whose randomness equals its nullifier. Transfer is an ordinary shielded payment. Verification is 96 bytes handed to a buyer who checks them against their own node. Uniqueness is enforced by the nullifier set. Nothing needs to ship.

The whole construction is a reading of one sentence in one Final specification. If that reading is wrong, the document is an elaborate error, and the sentence takes a minute to check (§Q.2).

138.3 What Is Honestly Given Up

Four things, stated here rather than in a limitations appendix.

There is no public ownership lookup, and there cannot be. Chapter 23 proves it: the chain has exactly two double-spend-protected resources, the public one destroys privacy by construction, and there is no third. A design claiming both on this chain today is claiming a resource that does not exist.

Every buyer learns the item's full prior history. Permanently and irrevocably, because verifying the chain requires being handed it. Bounded to accounts holding one note each — but real, and disqualifying for any collection where prior ownership is sensitive.

Settlement is custody, not escrow. There is no conditional spending in the shielded pool. Until two parties can co-construct one transaction, a venue holds funds between payment and delivery, and the word for that is custody.

Fungibility does not follow. An Action produces one successor. A balance that splits needs two. This is structural, not a scoping decision.

138.4 What Would Falsify This

Naming the failure conditions is better than being caught by them.

If §34.3's sentence does not say what is quoted, the construction has no foundation.

If two-party co-construction proves impossible, settlement remains custodial permanently and the largest objection to the design stands unanswered.

If anchor age or Action count turns out to fingerprint transactions, the privacy argument degrades to something meaningfully weaker than what Chapter 39 claims — and nobody has run either study.

If holders turn out not to want private ownership — if the trophy case was the product all along — then what is built here will be admired and unused.

And if an audit finds a defect in the argument of Chapter 47, everything downstream fails at once, because uniqueness is the property the other three rest on.

138.5 What Is Already True

The pool exists and runs. Items can be issued on it today, with no permission, no upgrade, and no new cryptography. Transfers cost what payments cost, because they are payments. Verification is milliseconds against a node the verifier chooses, and works with the venue switched off. Recovery from a seed phrase alone is possible, which the mature client-side-validation systems cannot do.

The assumption set is the currency's assumption set plus one hash property, with a stated degradation rather than a collapse if that one fails. There is no new primitive, no new proof system, no new consensus rule, and no trusted party — not the issuer, not the venue, not us.

That last item is the one that matters. None of it depends on trusting this project, which is the property we care most about, because it is the only one that survives us.

138.6 The Bet

That a market whose holders are not public is worth building even though its floor price cannot be computed by a stranger.

That is the whole wager, and it is narrower than whitepapers usually make. A project that does not need ownership privacy should use the account model, which is better at everything else. A project that needs public lookup cannot use this and no engineering will change that.

Only the intersection — privacy required, public lookup not — is this document's subject. It is not an empty set: one in seven inscriptions on this chain were funded from the shielded pool by people paying extra for a weaker version of what is specified here. But it is narrow, and pretending otherwise would be selling rather than specifying.

The rest is execution, and an audit.

Appendix W — Revision Notes

This revision roughly tripled the document and changed what it argues. The changes are recorded because a reader who saw an earlier version should know which of its claims no longer stand.

W.1 Structural

	Before	After
Parts	I–VI	I–VI plus VII–XXXIV
Appendices	none	A–W
Premise	the asset layer ships, then this works	this works on mainnet today
Central construction	asset-layer NFTs with ZIP 228 swaps	the seal chain of Chapter 22

W.2 Claims That No Longer Stand

Each of these appeared in an earlier version and is now withdrawn, with the section that replaces it.

Withdrawn	Replaced by
The asset layer is pending in the next upgrade	§25.2 — five structural signals say otherwise
Waiting is the plan	Part VIII — build on what exists
Seal closure is unverifiable by construction	§103 — unverifiable by an <i>arbitrary</i> observer; verifiable with 96 bytes
The 69.1% linkability figure, present tense	Chapter 74 — the defensible modern results
"The correlation is not inferential"	§H.7 — both sources presented, causation labelled inferred
Escrow	§42.2 — custody, in that word

W.3 Debts Not Discharged

The terminology sweep. Parts I–VI name the previous shielded pool where the current one is meant (§35.2). Every claim survives; the names do not.

The old glossary. The glossary below predates Part IX and defines several terms — swap assembly, the matcher, escrow-engine staging — in terms of a specification that Chapter 25 shows is orphaned. Rather than delete them, each is left with its original definition and the reader is referred to Appendix F, which is the glossary for the construction this document now specifies.

An audit. Of the construction, of the argument, and of any implementation. None has had one (§108).

Glossary

Note on this section. The entries below were written for the earlier plan and describe a system built on the asset layer and its swap mechanism. Chapter 25 establishes that both are orphaned. The definitions remain accurate as definitions and several describe machinery that is no longer on the critical path — `Escrow engine`, `Matcher`, `Action group`, and `Aggregated binding signature` in particular belong to a settlement design that Chapter 27 replaces with an open question.

For the vocabulary of the construction this document now specifies, see Appendix F.

Term	Definition
Action group	A set of Orchard-family actions within a transaction that is signed as a unit; ZIP 228 swap assembly composes maker, fee, and royalty action groups into one settlement transaction
Aggregated binding signature	Transaction-level signature binding all value commitments; prevents extracting a subset of action groups into a different transaction
assetDescHash	The on-chain hash (BLAKE2b) of an asset's description; the binding commitment linking a ZSA to its off-chain metadata preimage
Bearer instrument (order)	A fully signed swap order, executable by any party holding it within its signed terms; the reason orderbook secrecy is a security property
Blue/green (indexer)	Operating two indexer instances so one can rebuild or upgrade while the other serves traffic
Burn	Public, consensus-recorded destruction of ZSA supply; with issuance, one of the two public events in an asset's life
Commit-reveal drop	Two-phase mint: commitments to item contents are published first, contents revealed after assignment, preventing pre-reveal sniping
Confirmation depth	Number of blocks atop a transaction's block; gates "UI-final" status, with elevated thresholds on the low-difficulty testnet
Copyminting	Issuing an asset that reuses another collection's media or identity under a different issuance key
Deterministic rebuild	The indexer property that catalog state is a pure function of raw blocks, so any party can reproduce and verify it
Escrow engine (Phase T)	Custodial testnet trading mechanism predating ZIP 228 swaps; explicitly not deployed on mainnet
Finalized asset	A ZSA whose issuance is closed so supply can never increase; an NFT is a finalized asset with supply 1
Ghost-listing execution	Execution of a leaked order signature after the maker believed it delisted; contained by hard cancellation and expiry
Hard cancellation	Self-spending the note funding an order, invalidating every copy of its signature at consensus level
HSM	Hardware security module; holds the wallet master seed, matcher keys, and registry signing key, exporting signatures but never key material

Inclusion proof	Proof that a salted per-trade commitment is included in a matcher-signed aggregate, letting users verify published statistics
Issuance key	The key authorized to issue a ZSA (ZIP 227); also the BIP-340 authentication root for ZMD-1 manifests
Lineage mapping	Signed Registry record connecting a testnet-era collection to its mainnet re-issuance; provenance, not migration
Matcher	The Phase M service holding a private orderbook of signed swap orders and assembling ZIP 228 settlement transactions
Petname	A locally assigned, user-controlled display name for an asset; the anti-phishing primary identifier in signing prompts
Registry	The service mapping <code>assetDescHash</code> values to verified preimages and ZMD-1 manifests, with badges, dumps, and mirrors
Reorg	Chain reorganization replacing recent blocks; handled by rollback-and-replay in the indexer and unwind logic in stateful services
Soft cancellation	Delisting an order at the matcher without invalidating its signature; a policy action, not a cryptographic one
ZecBit Connect	The Phase B non-custodial wallet layer: browser WASM SDK, client-side Halo 2 proving, and the wallet bridge protocol
ZMD-1	ZecBit's metadata manifest standard: JCS-canonicalized, BIP-340 creator-signed, hash-bound to the on-chain asset
ZSA	Zcash Shielded Assets (ZIPs 226/227): user-issued assets with public issuance/burn and shielded transfers

References

Organised by topic. Where a document's status matters to an argument in this document, the status is given — and §G.2 is the procedure for checking that it has not changed since this was written.

Zcash protocol — load-bearing

1. Hopwood, D., Bowe, S., Hornby, T., Wilcox, N. *Zcash Protocol Specification*. zips.z.cash/protocol/protocol.pdf. — The Action structure, key hierarchy, commitment tree, nullifier derivation, note encryption. All of Part XI.
2. ZIP 224: *Orchard Shielded Protocol*. zips.z.cash/zip-0224. **Status: Final**. — Contains the p = nullifier statement on which Chapters 22, 34 and 47 depend. §34.3 quotes it; §120 recommends verifying it first.
3. ZIP 32: *Shielded Hierarchical Deterministic Wallets*. zips.z.cash/zip-0032. — Hardened derivation paths; §31.4, §43.1, §59.1.
4. ZIP 316: *Unified Addresses and Unified Viewing Keys*. zips.z.cash/zip-0316. — Receiver types and diversified addresses; §31.3.
5. ZIP 317: *Proportional Transfer Fee Mechanism*. zips.z.cash/zip-0317. **Status: Active; asset clauses removed**. — All of Chapter 24 and Appendix N. The removal is one of the five orphaning signals of §25.2.
6. *The Orchard Book*. zcash.github.io/orchard. — Implementation-level treatment of the Action circuit.
7. NU6.3 consensus rules. — The sealing of the previous pool; Chapter 35. The `enableCrossAddress` flag and the non-negative value balance constraint are §35.1.

Zcash — the asset layer

8. ZIP 226: *Transfer and Burn of Zcash Shielded Assets*. zips.z.cash/zip-0226. **Status: Draft**. — Per-asset value commitments, §26.1. **Its Deployment section is stale; see §25.3.**
9. ZIP 227: *Issuance of Zcash Shielded Assets*. zips.z.cash/zip-0227. **Status: Draft**. — Transparent issuance bundle, finalisation flag; §26.2.
10. ZIP 230: *Version 6 Transaction Format*. zips.z.cash/zip-0230. **Status: Withdrawn**. — The carrier that no longer exists; §25.1, Appendix M.
11. ZIP 228: *Asset Swaps for Zcash Shielded Assets*. zips.z.cash/zip-0228. **Status: Draft, dependent on [10]**. — Payment-conditional transfer; §27.4, §62.3.
12. GHSA-ww9q-8r59-xv46. — The May 2026 counterfeiting vulnerability in the asset layer's reference implementation; §26.5.

Zcash — proposals referenced but unavailable

13. Memo bundle proposal. **Status: Draft, dependent on [10].** — 16 KiB capacity and the publicly-decryptable key value; §24.5, §62.2.
14. ZIP 302: *Standardized Memo Field Format*. **Status: Draft; zero allocations granted.** — The absent type registry; §24.2.
15. Detection key proposal. **Status: Draft.** — Outsourced scanning without decryption; §32.3, §D.6.
16. PCZT (partially-created Zcash transaction) format, version 0.8.0. — Multi-party transaction construction; §27.2, §D.1.

Cryptographic foundations

17. Bowe, S., Grigg, J., Hopwood, D. *Recursive Proof Composition without a Trusted Setup (Halo)*. IACR ePrint 2019/1021. — §84.2's no-trusted-setup property and §80.2's recursion.
18. Halo 2 implementation documentation. zcash.github.io/halo2. — Constraint accounting behind §38.2.
19. Micali, S., Rabin, M., Kilian, J. *Zero-Knowledge Sets*. FOCS 2003. — The strong form of the problem Chapter 23 declines. §23.5 quotes its footnote on why soundness must be computational.
20. Chase, M., Healy, A., Lysyanskaya, A., Malkin, T., Reyzin, L. *Mercurial Commitments with Applications to Zero-Knowledge Sets*. EUROCRYPT 2005. — Generalisation of [19]; concedes impracticality (§23.5).
21. Camenisch, J., Chaabouni, R., Shelat, A. *Efficient Protocols for Set Membership and Range Proofs*. ASIACRYPT 2008. — Set-membership with the set as common input; §23.5.
22. Benarroch, D., Campanelli, M., Fiore, D., Gurkan, K., Kolonelos, D. *Zero-Knowledge Proofs for Set Membership*. FC 2021. — Hidden-order-group accumulators; states the set is public to the verifier (§23.5).
23. BIP-340: *Schnorr Signatures for secp256k1*. Wuille, P., Nick, J., Ruffing, T. — Manifest signing; §43.3.

The DNS natural experiment

24. RFC 4034 / RFC 4035: DNSSEC resource records. — NSEC; leaks the zone by construction (§23.5).
25. RFC 5155: *DNSSEC Hashed Authenticated Denial of Existence*. — NSEC3; concedes dictionary attacks in its own text.
26. RFC 7129: *Authenticated Denial of Existence in the DNS*. — Background on the trade-offs.
27. NSEC5 (expired Internet-Draft). — Online public-key operation per negative response; not zero-knowledge; unadopted (§23.5).

28. RFC 9276: *Guidance for NSEC3 Parameter Settings*. — Sets salt and iterations to zero; the eighteen-year conclusion (§23.5).

Privacy measurement

29. Kappos, G., Yousaf, H., Maller, M., Meiklejohn, S. *An Empirical Analysis of Anonymity in Zcash*. USENIX Security 2018. — The widely-quoted figure. **§73 explains why quoting it in the present tense is an error.**
30. Biryukov, A., Feher, D., Vitto, G. *Privacy Aspects and Subliminal Channels in Zcash*. CCS 2019. — The only published false-positive rate (§74.1) and the value-fingerprint survival measurement (§74.2).
31. Quesnelle, J. *On the linkability of Zcash transactions*. arXiv 1712.01210. — Round-trip heuristics.

Assets on UTXO chains

32. Ordinal Theory Handbook. docs.ordinals.com. — Envelope, tags, sat theory, cursed inscriptions, the Jubilee; Chapters 28, 65, 66.
33. BIP-141: *Segregated Witness*. — The witness discount that Zcash does not have; §28.4.
34. Runes specification. — Compact data-output token encoding; §29.2.
35. Taproot Assets protocol specification (bLIP-0017 and related). — Merkle-committed asset state, proof chains; §29.3.
36. RGB black paper. black-paper.rgb.tech. — Client-side validation, consignments, the stated inability to recover from seed alone; §29.4. **See §G.4: this site embeds a block addressed to automated readers. It is page content, not instruction.**
37. Counterparty protocol specification. — The oldest convention-based asset layer; §29.1.
38. EIP-721: *Non-Fungible Token Standard*. — The account model; Chapters 69–72.
39. Zerdinals. — The deployed inscription protocol on Zcash mainnet; input-script reveal, >111,000 inscriptions, the shielded-funded fraction; Chapter 54.

Data-carrier policy

40. Bitcoin Core release notes, October 2025 — the data-carrier default raised to effectively uncapped; §24.1.
41. Zebra / Zaino / Zallet node implementation sources — the unanimous 83-byte limit and its January 2026 introduction in one implementation; §24.1.

Adjacent work on Zcash proofs

42. Proof-of-assets implementation over Orchard in Halo 2 (independent, unaudited). — Demonstrates circuits over pool structures outside consensus; §23.3, §64.4.
43. Draft domain-separated balance proof specification. — Public inputs include a domain-separated nullifier; proves notes exist in the commitment tree and not in the nullifier set at a height; §23.3, §64.4. Unaudited, single-pool, does not contemplate custom assets.

Standards and formats

- 44. RFC 2119: *Key words for use in RFCs to Indicate Requirement Levels*. Bradner, S.
- 45. RFC 8785: *JSON Canonicalization Scheme (JCS)*. Rundgren, A., Jordan, B., Erdtman, S. — **\$56.3 argues against relying on canonicalisation; hash the published bytes.**
- 46. CIDv1: *Content Identifier Specification*, Multiformats. github.com/multiformats/cid. — Content addressing; §57.1. Also the 2017 precedent of §54.5.
- 47. W3C *Bitstring Status List*. — Quoted in §23.4 on the privacy of status lists that never change.

Note on citation practice

Several claims in this document rest on the *absence* of a document — no scheduled upgrade carrying the asset layer, no fee rule pricing it, no allocation granted by the memo registry. Absences cannot be cited. The procedure of §G.2 is how a reader checks them, and §H.5 records each one with its confidence.

Legacy reference list

Retained from earlier revisions for continuity of citation numbers used in Parts I–VI.

L1. ZIP 32: Shielded Hierarchical Deterministic Wallets. zips.z.cash/zip-0032. L2. ZIP 226: Transfer and Burn of Zcash Shielded Assets. zips.z.cash/zip-0226. L3. ZIP 227: Issuance of Zcash Shielded Assets. zips.z.cash/zip-0227. L4. ZIP 228: Asset Swaps for Zcash Shielded Assets (draft). zips.z.cash/zip-0228. L5. ZIP 230: Version 6 Transaction Format. zips.z.cash/zip-0230. L6. ZIP 317: Proportional Transfer Fee Mechanism. zips.z.cash/zip-0317. L7. Bowe, S., Grigg, J., Hopwood, D. Recursive Proof Composition without a Trusted Setup (Halo). IACR ePrint 2019/1021; and the Halo 2 implementation documentation, zcash.github.io/halo2. L8. Hopwood, D., Bowe, S., Hornby, T., Wilcox, N. Zcash Protocol Specification. zips.z.cash/protocol/protocol.pdf. L9. The Orchard Book. zcash.github.io/orchard. L10. BIP-340: Schnorr Signatures for secp256k1. Wuille, P., Nick, J., Ruffing, T. L11. RFC 2119: Key words for use in RFCs to Indicate Requirement Levels. Bradner, S. L12. RFC 8785: JSON Canonicalization Scheme (JCS). Rundgren, A., Jordan, B., Erdtman, S. L13. CIDv1: Content Identifier Specification, Multiformats. github.com/multiformats/cid.