Bitcoin Stamps Protocol: A Technical Whitepaper

Abstract

Bitcoin Stamps is a metaprotocol for creating permanent, immutable digital assets on Bitcoin through direct UTXO storage. Unlike witness-data approaches, Bitcoin Stamps embed asset data in transaction outputs using bare multisig and P2WSH encoding, ensuring universal node storage and consensus-critical permanence.

The protocol evolved from Counterparty foundations (block 779,652) through native Bitcoin encoding (block 793,068) to P2WSH optimization via OLGA (block 865,000). Built on account-based asset tracking, Bitcoin Stamps support fungible tokens (SRC-20), non-fungible assets (base stamps), decentralized naming (SRC-101), and composable recursion (SRC-721).

Core Innovation: Leveraging Bitcoin's UTXO set for permanent data storage, making asset data consensus-critical and unprunable. All full nodes must store stamp data to validate transactions, guaranteeing permanence as long as Bitcoin exists.

Key Properties:
- **UTXO-based permanence**: Data stored in spendable outputs, not witness segments
- **Consensus-critical storage**: Required for transaction validation across all nodes
- **Account-based assets**: Counterparty-style balance tracking, not UTXO-bound tokens
- **Multi-protocol support**: Extensible architecture for tokens, names, and recursion
- **Cost-optimized encoding**: OLGA P2WSH reduces fees 30-95% vs bare multisig

Architecture: The protocol separates data encoding (UTXO layer) from asset tracking (account layer). Stamps create permanent records in Bitcoin's UTXO set while maintaining balances through Counterparty-proven account ledger. This hybrid approach combines Bitcoin's permanence with practical asset management.

Table of Contents

10. **Appendices** — Reference implementations, test vectors, block timeline

---

Document Structure

This whitepaper consists of multiple sections:

- **[introduction.md](./introduction.md)** — Protocol history from Counterparty origins (block 779,652) through native encoding (793,068) to OLGA optimization (865,000)
- **[architecture.md](./architecture.md)** — Technical architecture: UTXO storage model, bare multisig vs P2WSH encoding, account-based asset tracking
- **[token-standards.md](./token-standards.md)** — SRC-20, SRC-721, SRC-721r, SRC-101 specifications
- **[economics.md](./economics.md)** — UTXO permanence economics, storage costs, fee analysis
- **[improvement-proposals.md](./improvement-proposals.md)** — SIP governance framework and active proposals (SIP-0001 through SIP-0008)
- **[implementation.md](./implementation.md)** — Indexer architecture, consensus mechanisms, validation logic
- **[security.md](./security.md)** — Threat model, attack vectors, immutability guarantees
- **[future.md](./future.md)** — Roadmap for conditional transfers, privacy enhancements, cross-chain bridges

---

Quick Reference

Genesis Block: 779,652 (March 29, 2023) — First Bitcoin Stamp by Mikeinspace
Native Encoding: 793,068 (April 20, 2023) — Direct Bitcoin encoding begins
Counterparty Cutoff: 796,000 (August 15, 2023) — SRC-20 consensus rule
OLGA Activation: 865,000 (October 15, 2023) — P2WSH optimization available

Foundation: Built on Counterparty protocol (est. 2014) for proven account-based asset tracking
Storage Model: UTXO-based (consensus-critical, unprunable)
Asset Model: Account-based (balances tracked per address, not per UTXO)

This whitepaper serves as the canonical technical specification for Bitcoin Stamps protocol. All implementations should reference this document for protocol compliance.

1. Introduction

1.1 Motivation

Bitcoin's primary innovation is permanent, censorship-resistant value storage backed by proof-of-work consensus. While Bitcoin enables programmable transactions through Script, the network primarily serves as monetary infrastructure. Bitcoin Stamps extends this permanence to arbitrary data—images, tokens, names—by embedding information directly in Bitcoin's UTXO set.

Core Problem: Digital assets require permanent storage to retain value. Traditional NFT platforms rely on IPFS, Arweave, or centralized servers—all subject to failure modes outside asset holders' control. Even Bitcoin-based solutions using witness data lack permanence guarantees since nodes can prune witness segments after validation.

Solution: Store asset data in transaction outputs (UTXOs) rather than witness data or external systems. Bitcoin's consensus rules require all full nodes to maintain the UTXO set for transaction validation, making UTXO-embedded data:
- **Consensus-critical**: Required for network operation
- **Unprunable**: Cannot be removed without breaking validation
- **Universal**: Stored by every full node globally
- **Permanent**: Survives as long as Bitcoin exists

Design Philosophy: Embrace Bitcoin's constraints rather than fight them. Higher fees for UTXO storage reflect true economic cost of permanent Bitcoin storage. Protocol design prioritizes permanence over convenience, aligning with Bitcoin's long-term value proposition.

1.2 Historical Context

1.2.1 Counterparty Foundation (2014-2023)

Bitcoin Stamps builds on Counterparty protocol, established January 2014 as Bitcoin's first metaprotocol for asset creation. Counterparty introduced:
- **Account-based assets**: Balance ledger tracked per address, not per UTXO
- **OP_RETURN encoding**: Embed metadata in 80-byte provably unspendable outputs
- **Decentralized exchange**: On-chain order books and atomic swaps
- **10+ years production**: Battle-tested architecture handling millions of transactions

Counterparty proved account-based asset tracking works at scale on Bitcoin. Rather than track which UTXOs contain tokens (complex, privacy-leaking), maintain address-level balances (simple, efficient, private).

Critical insight: SRC-20 tokens inherit Counterparty's account model. Token ownership is tracked per address in indexer state, NOT embedded in specific UTXOs. This is foundational to Bitcoin Stamps architecture.

1.2.2 Genesis: Block 779,652 (March 29, 2023)

Mikeinspace created the first Bitcoin Stamp—a laser-eyes pixel art embedded via Counterparty transaction. This stamp used traditional OP_RETURN encoding but sparked recognition: Bitcoin could permanently store visual art, not just monetary metadata.

Innovation: Frame digital art as permanent Bitcoin artifacts rather than ephemeral files. If art data lives in Bitcoin's UTXO set, it inherits Bitcoin's permanence and censorship resistance.

Community formation: The Original Trinity (Mikeinspace, Arwyn, Reinamora) recognized potential for permanent digital culture on Bitcoin. Within days, Stampchain.io launched as reference indexer and minting interface, establishing infrastructure for ecosystem growth.

1.2.3 Cultural Milestone: KEVIN (Blocks 783,718 & 788,041)

Block 783,718 (March 15, 2023): Arwyn created KEVIN (Stamp #4258) as homage to Rare Pepe culture. The artwork unexpectedly exhibited "ghost-like" behavior—appearing in unexpected system locations, inspiring organic derivative works. KEVIN evolved from artistic experiment to community symbol.

Block 788,041 (April 20, 2023): Arwyn deployed KEVIN as first SRC-20 token (Stamp #18,516), formalizing fungible token standard atop Bitcoin Stamps. This dual nature (unique stamp #4258 + fungible token) established pattern: stamps provide non-fungible foundation, SRC-20 adds fungible layer.

Cultural impact: KEVIN demonstrated fair launch principles—no pre-mine, equal minting access, community-driven distribution. These values became protocol philosophy: "we are all Kevin" (echoing Mayan "In Lak'ech Ala K'in"—"I am you, you are me"). Over 2,300 holders grew organically without marketing or speculation.

1.2.4 Technical Evolution: Block 793,068 (April 20, 2023)

First stamp using native Bitcoin bare multisig encoding rather than Counterparty OP_RETURN. This transition marked protocol independence—stamps no longer required Counterparty infrastructure, only Bitcoin itself.

Bare multisig encoding:
```
OP_1 <pubkey1> <pubkey2> <pubkey3> OP_3 OP_CHECKMULTISIG
```
Each "pubkey" is 32 bytes of image/data. A 2-of-3 multisig provides 64 bytes usable data per output. Multiple outputs chain together for larger assets.

Advantages:
- Direct Bitcoin encoding without metaprotocol dependencies
- UTXO-based storage (consensus-critical, unprunable)
- No witness data—data is part of transaction validation itself
- Simplified indexer logic (scan multisig outputs, decode data)

Tradeoffs: Higher fees (4x witness discount lost) but guaranteed permanence. Design choice: pay for true permanence rather than optimize for cost.

1.2.5 Asset Standards: Blocks 788,041 - 796,000

SRC-20 fungible tokens (block 788,041): JSON metadata in stamp encoding defines DEPLOY, MINT, TRANSFER operations. Indexers maintain account balances per Counterparty model—ownership tracked by address, not UTXO.

SRC-721 recursion (block 792,370): Stamps can reference other stamps by ID, enabling composable artwork. A stamp might combine background #1234 + character #5678 + effects #9012, creating infinite combinations from finite on-chain components.

Counterparty cutoff (block 796,000): Community consensus rule—SRC-20 tokens on Counterparty only valid until block 796,000. After this, only Bitcoin-native encoded tokens recognized. Ensures protocol independence while honoring early adopters.

1.2.6 Optimization: OLGA at Block 865,000 (October 15, 2023)

Reinamora introduced OLGA (Optimal Large Graphics Arrangement)—P2WSH encoding replacing bare multisig for 30-95% cost reduction.

P2WSH structure:
```
OP_0 <32-byte-hash-of-witness-script>
```
Witness script contains data, hashed and stored in output. More efficient than bare multisig pubkeys in output scripts.

Key insight: P2WSH witness scripts are still consensus-critical (unlike witness data for signatures). Scripts must be provided to spend P2WSH outputs, so nodes must store them for UTXO validation. Data remains unprunable and permanent.

Cost reduction mechanism:
- Bare multisig: 3 fake pubkeys (96 bytes) per output in transaction data
- P2WSH: 32-byte hash per output, actual data in witness script
- Witness discount: 4:1 reduction (witness data counted at 1/4 weight)
- Result: 60-80% fee reduction for typical stamps

OLGA benefits:
- Maintains UTXO permanence (witness scripts are consensus-critical)
- Dramatically reduces creation costs (broader accessibility)
- Better miner priority (more efficient byte usage)
- Universal compatibility (works across all stamp protocols)

1.3 Protocol Overview

Bitcoin Stamps protocol comprises:

1. **Data encoding layer**: Bare multisig (pre-865,000) or P2WSH/OLGA (post-865,000) for embedding data in UTXOs
2. **Asset tracking layer**: Account-based ledger (Counterparty-style) for ownership and balances
3. **Standards layer**: SRC-20 (tokens), SRC-721 (recursion), SRC-101 (names) defining asset semantics
4. **Indexer layer**: Software parsing stamp transactions, maintaining asset state, serving APIs

Critical distinction: Encoding determines WHERE data is stored (UTXOs). Asset tracking determines WHO owns WHAT (accounts). These layers are independent—SRC-20 tokens use UTXO storage for transaction permanence but account balances for ownership tracking.

1.4 Design Principles

Permanence over cost: Pay Bitcoin's true storage cost rather than rely on prunable witness data or external systems. Expensive stamps reflect accurate economics of permanent Bitcoin storage.

Simplicity over features: Account-based assets simpler than UTXO-bound tokens. Proven Counterparty model beats novel approaches requiring complex state tracking.

Bitcoin-native alignment: Work with Bitcoin's economic incentives (UTXO storage fees support miners) rather than fight them (clever witness hacks ultimately prunable).

Community governance: Fair launches (no pre-mines), organic growth (no VC funding), cultural values (authenticity over speculation). KEVIN's success demonstrates aligned incentives create sustainable ecosystems.

Extensibility: Base stamp protocol provides permanence primitive. Standards like SRC-20/721/101 add semantics without modifying underlying encoding. Future protocols can leverage same UTXO permanence.

1.5 Document Scope

This whitepaper specifies:
- UTXO storage architecture (Section 2)
- Bare multisig encoding (Section 3.1)
- P2WSH/OLGA encoding (Section 3.2)
- SRC-20 token standard (Section 4.1)
- SRC-721 recursion (Section 4.2)
- SRC-101 naming (Section 4.3)
- Implementation guidelines (Section 5)
- Security analysis (Section 6)
- Economic model (Section 7)
- Comparative protocol analysis (Section 8)

Out of scope: Wallet integration details, specific indexer implementations, user interface design, market dynamics. Focus is protocol specification for implementers.

1.6 Terminology

- **Stamp**: Non-fungible digital asset permanently stored in Bitcoin UTXO set via bare multisig or P2WSH encoding
- **SRC-20**: Fungible token standard atop stamps, using account-based balance tracking
- **UTXO set**: Set of all unspent transaction outputs in Bitcoin; consensus-critical data structure required for transaction validation
- **Account-based**: Asset ownership tracked per address (account balance) rather than per UTXO (UTXO-bound tokens)
- **Bare multisig**: Native Bitcoin multisig scripts used to encode data in fake pubkeys
- **P2WSH/OLGA**: Pay-to-Witness-Script-Hash outputs storing data in witness scripts (consensus-critical but weight-discounted)
- **Counterparty**: First Bitcoin metaprotocol (est. 2014); provides account-based asset model inherited by Bitcoin Stamps
- **Indexer**: Software parsing stamp transactions from Bitcoin blockchain, maintaining asset state database, serving API queries

---

Next: [Protocol Architecture →](./architecture.md)

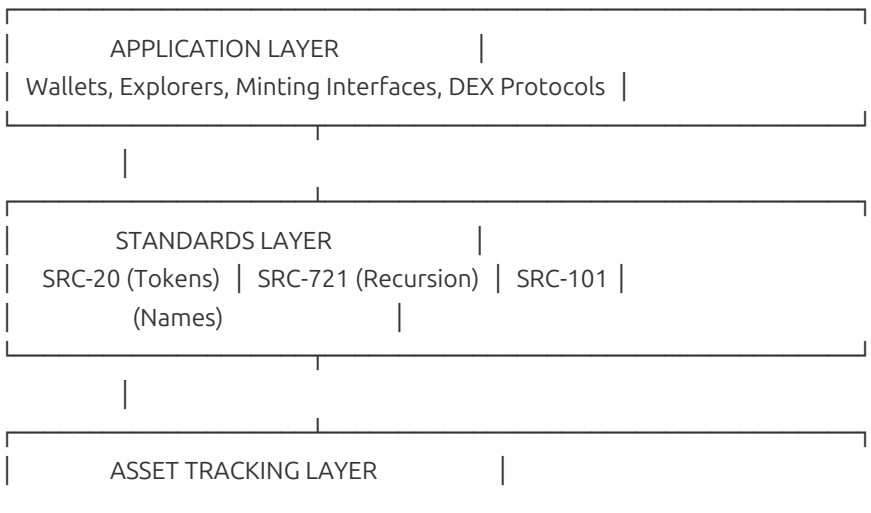
---

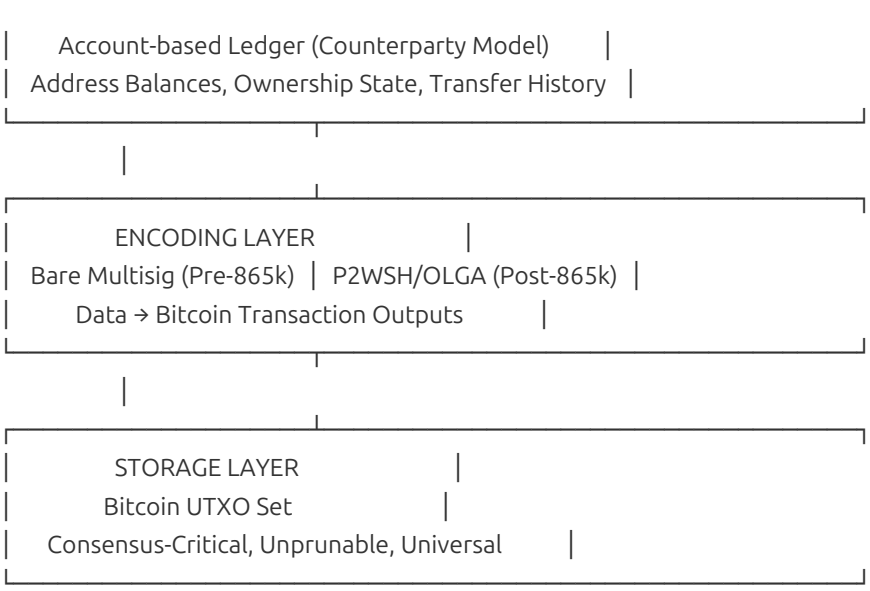
2. Protocol Architecture

2.1 Architectural Overview

Bitcoin Stamps employs a layered architecture separating concerns:

`





`

Key architectural principle: Data encoding (UTXO storage) is independent from asset tracking (account balances). Stamps permanently embed transaction data in Bitcoin outputs while maintaining ownership through account ledger.

2.2 UTXO Storage Model

2.2.1 Bitcoin UTXO Set

Bitcoin maintains an **Unspent Transaction Output (UTXO) set**—the complete list of all unspent outputs on the blockchain. This set is:

- **Consensus-critical**: Required for validating new transactions (ensure inputs reference valid UTXOs)
- **Universal**: Every full node maintains identical UTXO set
- **Unprunable**: Cannot be deleted without breaking transaction validation
- **Permanent**: Persists as long as Bitcoin network operates

UTXO structure:
```rust
struct UTXO {
    txid: [u8; 32],         // Transaction ID
    vout: u32,              // Output index
    amount: u64,            // Satoshis
    scriptPubKey: Vec<u8>,  // Locking script
    height: u32,            // Block height
}
```

Validation requirement: To validate a transaction spending UTXO X, nodes must:
1. Verify X exists in UTXO set
2. Check spending transaction provides valid unlock script
3. Verify amount conservation (inputs ≥ outputs + fees)
4. Execute output scripts to verify spending conditions

Critical insight: Any data embedded in `scriptPubKey` must be stored by all nodes to validate future spends. This makes UTXO-embedded data consensus-critical and unprunable.

2.2.2 Why UTXO Storage Guarantees Permanence

Contrast with witness data (SegWit):

Witness data (signatures, witness scripts):
- Required only during transaction validation
- After validation, nodes can prune witness data
- Not part of transaction hash (malleability fix)
- Not consensus-critical for future transactions
- **Result**: Witness data is prunable, not guaranteed permanent

UTXO data (scriptPubKey, amounts):
- Required for all future transaction validation
- Cannot be pruned without breaking validation
- Part of transaction hash (UTXO uniquely identified by txid:vout)
- Consensus-critical for network operation
- **Result**: UTXO data is unprunable, guaranteed permanent

Bitcoin Stamps strategy: Embed asset data in scriptPubKey (output scripts) rather than witness data. This makes stamp data UTXO-embedded and thus consensus-critical.

2.2.3 UTXO Set Size Implications

UTXO storage has real cost—every full node stores entire UTXO set in fast-access databases. As of 2026:
- ~150M UTXOs globally (~6GB UTXO database)
- Each stamp adds 1-20 UTXOs depending on data size
- Trade-off: Higher fees for permanent storage vs lower fees for prunable witness data

Design philosophy: Accept higher costs for true permanence. Bitcoin Stamps reflects accurate economics of permanent Bitcoin storage. Protocols using witness data or external storage have hidden costs (pruning risk, service maintenance, infrastructure failure).

2.3 Encoding Layer Architecture

2.3.1 Bare Multisig Encoding (Blocks 779,652 - 865,000)

Structure: Use Bitcoin's native multisig scripts to encode data.

Multisig script format:
```
OP_1 <pubkey1> <pubkey2> <pubkey3> OP_3 OP_CHECKMULTISIG
```

Data encoding:
- `<pubkey1>`, `<pubkey2>`, `<pubkey3>` are 33-byte compressed pubkey format
- Actually contain stamp data, not real public keys
- 2-of-3 multisig: keys 1 & 2 are data (66 bytes), key 3 is real signing key
- Multiple outputs chained for larger data

Example (simplified):
```python
# Encode 64 bytes of image data in bare multisig

output_script = OP_1 + data[0:33] + data[33:66] + real_pubkey + OP_3 + OP_CHECKMULTISIG
```

Characteristics:
- **Permanent**: Data in scriptPubKey, part of UTXO set
- **Consensus-critical**: Required to spend multisig UTXO
- **Expensive**: Full transaction weight (4 WU per byte)
- **Simple**: Native Bitcoin scripts, no special rules
- **Universal**: Any Bitcoin node can validate

Limitations:
- High fees due to no witness discount
- Large stamps require many outputs (cost scales linearly)
- Multisig scripts flagged by some mempool policies (relay issues)

2.3.2 P2WSH/OLGA Encoding (Block 865,000+)

OLGA (Optimal Large Graphics Arrangement) uses Pay-to-Witness-Script-Hash for 30-95% cost reduction.

Structure:
```
Output script: OP_0 <32-byte-script-hash>
Witness: <actual-witness-script>
```

Data encoding:
```python
# Encode data in P2WSH witness script

witness_script = data_chunks + OP_DROP_sequence + <conditions>
script_hash = SHA256(witness_script)
output_script = OP_0 + script_hash

# To spend: provide witness_script in witness field
```

witness = [signatures, witness_script]
`

Witness script construction:
`
<data_chunk_1> OP_DROP <data_chunk_2> OP_DROP ... <signature_check>
`
Data chunks are pushed to stack and dropped, leaving only signature verification logic.

Characteristics:
- **Still consensus-critical**: Witness script must be provided to spend P2WSH output
- **Weight discount**: Witness data counted at 1/4 weight (WU)
- **Cost reduction**: 30-95% vs bare multisig
- **Same permanence**: Witness scripts stored in UTXO set (not prunable)
- **Better relay**: P2WSH is standard, no mempool policy issues

Key distinction: P2WSH witness *scripts* (containing data) are consensus-critical, unlike witness *signatures* (prunable). To spend P2WSH output, validator must:
1. Hash provided witness script
2. Compare to hash in output script
3. Execute witness script
4. Verify conditions satisfied

Result: Witness scripts cannot be pruned—required for validation. Stamp data embedded in witness scripts remains permanent and unprunable.

2.3.3 Encoding Layer Comparison

Dimension	Bare Multisig	P2WSH/OLGA
Permanence	UTXO-embedded	UTXO-embedded (witness script)
Consensus-critical	Yes	Yes (script hash validation)
Prunable	× No	× No (scripts required for spending)
Cost	High (4 WU/byte)	Low (1 WU/byte witness)
Relay	Potential issues	Standard P2WSH
Complexity	Simple	Moderate (witness construction)
Block range	779,652 - present	865,000 - present

Protocol evolution: OLGA doesn't replace bare multisig—both remain valid. Stamps can use either encoding; indexers must support both. OLGA is optimization, not consensus change.

2.4 Account-Based Asset Tracking

2.4.1 The Account Model

Critical architectural decision: Bitcoin Stamps uses **account-based** asset tracking, NOT UTXO-based.

Account-based (Bitcoin Stamps, Counterparty):
```python
# State: simple address → balance mapping
balances = {
    "bc1q...xyz": {"KEVIN": 1000, "STAMP": 50},
    "bc1q...abc": {"KEVIN": 500}
}

# Transfer: update sender and receiver balances
def transfer(from_addr, to_addr, asset, amount):
    balances[from_addr][asset] -= amount
    balances[to_addr][asset] += amount
```

UTXO-based (Colored Coins, theoretical models):
```python
# State: track which UTXOs contain which tokens
token_utxos = {
    "txid1:vout0": {"asset": "TOKEN", "amount": 100},
    "txid2:vout1": {"asset": "TOKEN", "amount": 200}
}

# Transfer: complicated UTXO tracking across inputs/outputs
def transfer(tx):
    input_tokens = sum(token_utxos[input] for input in tx.inputs)
    # Allocate to outputs (complex rules for multi-input, change, fees)
    distribute_to_outputs(tx.outputs, input_tokens)
```

Why account-based wins:

1. **Simplicity**: Address balances simpler than UTXO tracking across coin mixing
2. **Privacy**: Don't reveal which specific coins hold tokens
3. **Efficiency**: Single DB query for balance vs scanning UTXO set
4. **Proven**: Counterparty ran 10+ years on account model
5. **UX**: Users understand "address balance" better than "token-bearing UTXO"

Common misconception: "SRC-20 tokens are locked in UTXOs." **False**. SRC-20 balances are tracked per address in indexer database. Tokens aren't "in" any specific UTXO—ownership is account-based.

2.4.2 Asset State Management

Indexer responsibilities:
1. Scan Bitcoin blocks for stamp transactions
2. Decode stamp data (bare multisig or P2WSH)
3. Parse asset operations (DEPLOY, MINT, TRANSFER)
4. Update account balances per consensus rules
5. Serve API queries for balances, history, metadata

State schema (simplified):
```sql
-- Account balances
CREATE TABLE balances (
  address TEXT,
  asset TEXT,
  amount NUMERIC,
  PRIMARY KEY (address, asset)
);

-- Transfer history
CREATE TABLE transfers (
  txid TEXT,
  block_height INTEGER,
  from_address TEXT,
  to_address TEXT,
  asset TEXT,
  amount NUMERIC,
  timestamp INTEGER
);

-- Asset metadata
CREATE TABLE assets (
  asset_name TEXT PRIMARY KEY,
  deploy_txid TEXT,
  deploy_block INTEGER,
  total_supply NUMERIC,
  divisible BOOLEAN,
  locked BOOLEAN
);
```

Consensus rules (SRC-20 example):
```python
def process_src20_transfer(tx, from_addr, to_addr, asset, amount):
  # Validation
  if balances[from_addr][asset] < amount:
    return False  # Insufficient balance

  # State update
  balances[from_addr][asset] -= amount
  balances[to_addr][asset] += amount

  # History
  transfers.append({
    'txid': tx.txid,
    'from': from_addr,
    'to': to_addr,
```

```
        'asset': asset,
        'amount': amount,
        'block': tx.block_height
    })

    return True
```

Reorganization handling:
```python
def handle_reorg(old_chain_tip, new_chain_tip):
    # Rollback state to fork point
    fork_height = find_fork_point(old_chain_tip, new_chain_tip)
    rollback_to_height(fork_height)

    # Replay blocks from fork point to new tip
    for block in range(fork_height + 1, new_chain_tip.height + 1):
        process_block(block)
```

2.4.3 Transfer Mechanism

Transaction flow:

1. **User action**: Send 100 KEVIN tokens to recipient
2. **Transaction construction**:
   - Create Bitcoin transaction from sender's address
   - Embed SRC-20 TRANSFER operation in stamp encoding:
     ```json
     {
       "p": "src-20",
       "op": "transfer",
       "tick": "KEVIN",
       "amt": "100",
       "to": "bc1q...recipient"
     }
     ```
   - Broadcast to Bitcoin network
3. **Block confirmation**: Transaction included in Bitcoin block
4. **Indexer processing**:
   - Detects stamp transaction
   - Decodes TRANSFER operation
   - Validates sender has 100+ KEVIN balance
   - Updates balances: sender -100, recipient +100
5. **User query**: Recipient checks balance via indexer API → sees 100 KEVIN

Key point: The Bitcoin transaction itself only stores the TRANSFER instruction. Actual balance updates happen in indexer state. Indexers independently compute same state by replaying transactions.

Consensus: Multiple indexers process same blockchain, arrive at identical balances. If indexers disagree, indicates implementation bug—consensus rules must be deterministic.

2.5 Layer Separation

2.5.1 Encoding ≠ Ownership

Encoding layer (UTXO storage):
- Determines WHERE data is stored (which UTXOs)
- Ensures permanence (consensus-critical storage)
- Handles Bitcoin transaction construction
- **Example**: Bare multisig or P2WSH encoding

Asset tracking layer (account balances):
- Determines WHO owns WHAT (balances per address)
- Manages transfer logic and validation
- Maintains asset metadata and history
- **Example**: SRC-20 balance ledger

Independence: You can change encoding (bare multisig → P2WSH) without changing asset tracking. You can add new asset standards (SRC-721, SRC-101) without changing encoding.

2.5.2 Standards Layer Flexibility

Base stamps protocol:
- Defines encoding methods (bare multisig, P2WSH)
- No inherent asset semantics
- Just permanent data storage on Bitcoin

Standards define semantics:
- **SRC-20**: Fungible tokens with DEPLOY/MINT/TRANSFER operations
- **SRC-721**: Recursion standard for composable stamps
- **SRC-101**: Decentralized naming system
- **Future standards**: Can add new semantics without protocol changes

Example: A single stamp transaction can embed:
```json
{
  "stamp": {
    "image": "base64_data",
    "src20": {"op": "mint", "tick": "TOKEN", "amt": "1000"},
    "src721": {"parent": 1234, "trait": "golden"}
  }
}
```
Multiple standards operate on same underlying UTXO permanence.

2.6 Architecture Summary

Layered design:
1. **Storage**: Bitcoin UTXO set (consensus-critical permanence)
2. **Encoding**: Bare multisig or P2WSH (data → UTXOs)
3. **Asset tracking**: Account-based ledger (Counterparty model)
4. **Standards**: SRC-20/721/101 defining asset semantics
5. **Applications**: Wallets, DEXs, explorers building on indexer APIs

Key innovations:
- **UTXO permanence**: Leverage Bitcoin's consensus requirements for guaranteed storage
- **Account simplicity**: Counterparty-proven model avoids UTXO tracking complexity
- **Layer separation**: Encoding independent from asset logic; standards independent from protocol
- **P2WSH optimization**: OLGA reduces costs 30-95% while maintaining permanence

Tradeoffs:
- **Higher fees**: True cost of permanent Bitcoin storage (vs prunable witness tricks)
- **Indexer dependency**: Need off-chain state computation (vs pure Bitcoin validation)
- **Larger UTXO set**: Global node storage impact (vs transient witness data)

Design philosophy: Embrace Bitcoin's constraints, pay true costs, achieve genuine permanence. No clever hacks—just aligned incentives and honest economics.

Next: [Data Encoding Methods →](#) (Section 3)
Previous: [← Introduction](./introduction.md)

---

4. Token Standards

The Bitcoin Stamps protocol supports three distinct token standards, each optimized for specific use cases while maintaining the core principle of UTXO-based immutability. All standards leverage Bitcoin's Proof-of-Work consensus mechanism, ensuring data integrity once confirmed.

4.1 SRC-20: Fungible Token Standard

Overview

SRC-20 is an account-based fungible token protocol that enables fair, accessible token creation with only standard Bitcoin miner fees. Inspired by BRC-20 but designed with Stamps' immutability guarantees, SRC-20 operates directly on the Bitcoin blockchain without dependency on Counterparty since block 796,000.

Critical Design Note: SRC-20 is **account-based**. Balances are tracked per address in indexer state, NOT per UTXO. This distinguishes it from UTXO-based protocols where tokens are locked in specific transaction outputs.

First Deployment

The KEVIN token, deployed by Reinamora at block 788041, represents the genesis SRC-20 deployment.

Transaction Structure

SRC-20 transactions follow standardized JSON encoding embedded in Bitcoin transaction outputs. Required fields include:

- `p` : Protocol identifier ("src-20")
- `op` : Operation type (deploy, mint, transfer)
- `tick` : Token ticker symbol
- Additional operation-specific parameters

Operations

DEPLOY: Initializes a new token collection with supply limits, per-mint caps, and optional pricing.

MINT: Creates new token units within deployment constraints. Minting continues until max supply is reached.

TRANSFER: Moves tokens between addresses. The indexer validates sender balance before updating account states.

Validation and Indexing

The indexer validates transactions through multi-step verification:

1. **Length Verification**: First two bytes represent expected decoded data length in hex
2. **JSON Validation**: Transaction must parse as valid JSON with required fields
3. **Balance Check**: For transfers, sender must hold sufficient balance
4. **State Update**: Successful transactions update the account-based balance ledger

Invalid transactions receive no stamp number and do not affect user balances. The Rust-based parser provides 20-50x performance improvement over pure Python implementations.

Economic Model

SRC-20 deployments incur only Bitcoin miner fees, eliminating token burn requirements or auxiliary cryptocurrency costs. This "fair launch" model ensures accessibility while maintaining immutability through UTXO set storage.

4.2 SRC-721: Layered NFT Standard

Overview

SRC-721 addresses the economic challenge of high-resolution NFT collections by introducing a layered composition architecture. Instead of embedding complete images per mint, collections store reusable layer components once, then reference them through lightweight JSON manifests.

Architecture

Layer Storage: Collections deploy up to 10 layered stamp images using standard Stamps protocol. Each layer is independently stamped with full immutability guarantees.

Composition Manifests: Users mint small JSON files (~100-500 bytes) that reference pre-stamped layers, specifying:
- Layer stamp IDs
- Stacking order (z-index)
- Optional layer transformations
- Metadata fields

Rendering: Client applications reconstruct final artwork by retrieving and compositing referenced layers in specified order.

Benefits

1. **Cost Efficiency**: 60-70% reduction in per-NFT minting costs through layer reuse
2. **High Fidelity**: Supports indexed color palettes and high-resolution assets per layer
3. **Composability**: Enables 10K PFP projects and generative art collections
4. **Immutability**: Both layers and manifests are permanently stored in UTXO set

Transaction Fields

Required fields for valid SRC-721 transactions:

- `p` : "src-721"
- `op` : Operation type (deploy, mint)

- `layers`: Array of stamp IDs comprising the composition
- `attributes`: Metadata describing trait composition

First Implementation

The AVIME collection by Derp Herpenstein, deployed at block 788041, pioneered the SRC-721 standard.

4.3 SRC-721r: Recursive Rendering Standard

Evolution from SRC-721

SRC-721r extends the layered model by incorporating **on-chain JavaScript libraries** for complex recursive rendering. This enables animated, interactive, and algorithmically generated artwork while maintaining complete on-chain data storage.

Technical Capabilities

JavaScript Runtime: Manifests can include or reference stamped JavaScript libraries that execute client-side to produce final artwork.

Recursive Composition: Supports:
- Nested layer hierarchies
- Algorithmic pattern generation
- Animation sequences
- Interactive elements responding to block data or timestamps

Library Reuse: Common rendering functions (e.g., noise generators, easing functions) are stamped once and referenced across collections.

Use Cases

- Generative art projects with algorithmic variation
- Animated collections with on-chain animation logic
- Interactive NFTs responding to blockchain state
- Complex visual effects requiring computational rendering

Security Considerations

All JavaScript executes client-side in sandboxed environments. The protocol does not introduce execution risk

to the Bitcoin network itself, as rendering is strictly a presentation-layer concern.

4.4 SRC-101: Domain Registration Standard

Overview

SRC-101 provides a Bitcoin-native domain name service leveraging Stamps' immutability to solve UTXO-linked asset challenges. Jointly developed by Bitname and Stamp teams, it enables permanent, address-tied naming while supporting the entire Bitcoin ecosystem including Layer 2 solutions.

Core Design

Domain names are stamped directly onto the Bitcoin blockchain as permanent records tied to user addresses. This separates name ownership from UTXO management, preventing accidental spending of domain-bearing transaction outputs.

Operations

DEPLOY
Creates a name service collection with deployment parameters:

- `name` : Collection identifier
- `tick` : Token symbol (e.g., "BNS")
- `owner` : Must match transaction signer
- `pri` : Price in satoshis per mint
- `max` : Supply limit (0 = unlimited)
- `lim` : Maximum 10 mint operations per transaction
- Optional whitelist with discount rates

MINT
Registers individual domain names:

- References deploy transaction hash
- `tokenid` : Name in hexadecimal format
- `dua` : Duration in years before expiration
- `toaddress` : Recipient (may differ from transaction signer)

TRANSFER
Moves domain ownership between addresses:

- Transaction signer must be current owner
- `toaddress` : New recipient address
- Supports all Bitcoin address types (Legacy, SegWit, Taproot)

SETRECORD

Associates resolver data with domains:

- Supported record types: "address" (resolution target) and "txt" (arbitrary metadata)
- Signer must be service owner
- Multiple records permitted; duplicate keys overwrite previous values

RENEW
Extends domain lease period:

- Requires owner authorization
- Payment in satoshis per deployment pricing
- Extends expiration by specified duration

TRANSFEROWNERSHIP
Transfers administrative control of the name service:

- Service owner only
- New owner assumes deployment-level permissions

Address Interoperability

SRC-101 supports resolution and interconversion of all Bitcoin address types, enabling seamless integration with:
- Mainnet (Legacy, P2SH, P2WPKH, P2WSH, Taproot)
- Layer 2 protocols (Lightning Network, sidechains)
- Bitcoin ecosystem extensions

Economic Model

Deployers set per-mint pricing in satoshis, creating sustainable name services without reliance on external fee structures. Renewal fees provide ongoing revenue while ensuring active namespace use.

4.5 Cross-Protocol Guarantees

All token standards share fundamental properties:

1. **Immutability**: Data stored in UTXO set cannot be pruned or modified
2. **Consensus Security**: Protected by Bitcoin's Proof-of-Work
3. **Indexer Validation**: Multiple independent indexer implementations can verify state
4. **No Burn Requirements**: Only Bitcoin miner fees required
5. **Open Source**: Reference indexer and validation logic publicly available

These guarantees distinguish Stamps-based protocols from witness-data alternatives that compromise on permanence or introduce auxiliary dependencies.

References:

- [Bitcoin Stamps Indexer Repository](https://github.com/stampchain-io/btc_stamps)
- [SRC-101 Specification](https://bitname.gitbook.io/bitname/src-101)
- [Stampchain FAQ](https://stampchain.io/faq)
- [SRC-20 Token Standard Overview](https://trustmachines.co/learn/what-is-the-src-20-token-standard/)
- [Bitcoin Stamps vs Ordinals Analysis](https://coinpedia.org/guest-post/bitcoin-stamps-vs-ordinals-deep-dive-into-future-of-on-chain-permanence/)

---

5. Economic Model

The Bitcoin Stamps protocol's economic model is fundamentally shaped by its design choice to store data in the UTXO set rather than witness data. This section analyzes the permanence guarantees, cost structures, and economic tradeoffs inherent to this architecture.

5.1 UTXO Set Permanence Guarantees

Architectural Foundation

Bitcoin Stamps are stored directly in Bitcoin's Unspent Transaction Output (UTXO) set, which full nodes maintain in memory or indexed storage for efficient transaction validation. This contrasts with witness-data protocols (e.g., Ordinals) that leverage SegWit's discounted witness field for data inscription.

Permanence Mechanism

Unprunable by Design: UTXO set entries cannot be pruned from full nodes without breaking consensus rules. While nodes can prune historical block data and witness information after verification, they must retain all unspent outputs to validate new transactions.

Stamp UTXOs: Once created, Stamp-bearing UTXOs are expected to remain unspent indefinitely, ensuring:
1. Data persists in the globally replicated UTXO set
2. No dependency on archival node policies
3. Immunity to pruning configurations

Counterparty Integration: Classic Stamps leverage Counterparty's bare multisig (P2MS) outputs, which chunk

image data across multiple outputs. By avoiding OP_RETURN (limited to 80 bytes and prunable), Stamps achieve true immutability.

Economic Implications of Permanence

UTXO Set Bloat: Every Stamp contributes to permanent UTXO set growth, imposing ongoing storage costs on all full nodes. As of 2026, the UTXO set exceeds 10GB, with protocols like Stamps representing a measurable fraction.

Node Operation Costs: Validators bear the cost of storing Stamp UTXOs perpetually, creating a commons dilemma where minters externalize storage costs to the network.

Economic Finality: Permanence ensures that Stamp data survives even catastrophic scenarios (e.g., protocol deprecation, indexer abandonment). The data exists independently of any external service.

5.2 Storage Format Evolution

Bare Multisig (OP_MULTISIG)

Original Format: Early Stamps used Counterparty's bare multisig encoding:
- Base64-encode image binary
- Split encoded data into 33-byte chunks
- Embed chunks as fake public keys in multisig outputs (e.g., 1-of-3, 2-of-3)

Size Limits: Maximum 7KB per Stamp due to standard transaction size constraints.

Weight Calculation: Multisig data is stored in the base transaction block, counting as 4 weight units per byte under SegWit's accounting.

P2WSH Migration

Efficiency Gains: Pay-to-Witness-Script-Hash (P2WSH) outputs store data in the witness field, which receives a 75% discount under SegWit rules:
- Base block data: 4 weight units per byte
- Witness data: 1 weight unit per byte

Cost Reduction: P2WSH-based Stamps pay ~25% of bare multisig fees for equivalent data size.

Pruning Concern: Witness data is technically prunable by nodes that don't serve historical blocks. However, archival nodes and blockchain explorers retain witness data, ensuring practical permanence.

Stamps P2WSH Variant: Stamps protocol adopted P2WSH for certain formats while maintaining UTXO set references, balancing cost efficiency with permanence goals.

OLGA Encoding

Breakthrough Optimization: Introduced at block 833000, OLGA (Optimized Low Gas Architecture) eliminates Base64 encoding:

Technical Innovation:
- Stores raw binary data directly in transaction outputs
- Removes 33% overhead from Base64 conversion
- Achieves 50% transaction size reduction vs. OP_MULTISIG
- Reduces minting costs by 60-70%

Size Expansion: Maximum file size increased to 64KB, enabling higher-fidelity artwork and larger datasets.

Adoption: OLGA became the standard for new Stamps due to dramatic cost savings without compromising immutability.

5.3 Miner Fee Economics

Fee Market Competition

Base Layer Fees: Stamp minters compete in Bitcoin's fee market alongside financial transactions. During congestion (e.g., Ordinals inscription waves, halving periods), Stamp costs scale proportionally.

Fee Rate Dynamics:
- Low congestion: 1-5 sat/vByte (Stamps cost $0.50-$5 per KB)
- Medium congestion: 20-50 sat/vByte (Stamps cost $10-$30 per KB)
- High congestion: 100-500 sat/vByte (Stamps cost $60-$300 per KB)

Batching Economies: Minting multiple Stamps in a single transaction amortizes overhead:
- Single Stamp: ~300 bytes overhead + data
- 10 Stamps: ~300 bytes overhead + (10 × data), reducing per-Stamp cost

Cost Structure Analysis

Per-Stamp Breakdown (OLGA format, 5KB image, 20 sat/vByte):

Component	Size	Cost
Transaction overhead	150 bytes	3,000 sats
OLGA data (5KB)	5,000 bytes	100,000 sats
Output creation	50 bytes	1,000 sats
Total	**5,200 bytes**	**~104,000 sats (~$62 @ $60K BTC)**

Comparative Costs:
- Ordinals inscription (5KB): ~26,000 sats (~$16) — 75% cheaper due to witness discount

- Classic Stamp (Base64): ~180,000 sats (~$108) — 73% more expensive due to encoding overhead
- OLGA Stamp: ~104,000 sats (~$62) — balanced cost-permanence tradeoff

Miner Revenue Impact

Protocol Contribution: During 2023-2024 inscription waves, data-heavy protocols contributed 5-15% of miner fee revenue, with Stamps representing a smaller but consistent fraction.

Incentive Alignment: Stamp minters directly compensate miners for permanent block space allocation, aligning economic incentives without protocol subsidies.

5.4 Storage Cost Comparison

Bitcoin Stamps vs. Ordinals

Attribute	Bitcoin Stamps	Ordinals (Inscriptions)
Storage Location	UTXO set (base block data or P2WSH witness)	Witness data (SegWit)
Prunability	Unprunable (UTXO set)	Technically prunable (witness)
Cost Multiplier	4x (OP_MULTISIG) to 1x (P2WSH OLGA)	1x (witness discount)
Size Limit	64KB (OLGA), 7KB (legacy)	~400KB (block size constraints)
Node Impact	Perpetual UTXO set growth	Witness data pruning reduces impact
Economic Model	Minter pays permanent externality	Minter pays discounted temporary cost

UTXO Set Growth Implications

Long-Term Costs: As of 2026, storing 1GB of UTXO data costs validators:
- SSD storage: ~$0.10/GB/year
- RAM caching (performance nodes): ~$5/GB/year

Scaling Concerns: If Stamps adoption scales to 100GB UTXO footprint, validators face:
- $10/year storage costs (SSD)
- $500/year RAM costs (high-performance nodes)

These costs are externalized to the network, raising debate over sustainable protocol economics.

Alternative Protocols

IPFS + Bitcoin Anchoring: Store data off-chain (IPFS), anchor hashes on Bitcoin:
- Cost: ~200 bytes per anchor (~$2 at 20 sat/vByte)
- Tradeoff: Requires IPFS network availability; not truly immutable

Arweave + Bitcoin Verification: Permanent storage layer with Bitcoin proof references:
- Cost: ~$5-$10 per MB on Arweave
- Tradeoff: Dependency on Arweave network; cross-chain trust assumptions

Stamps Advantage: True Bitcoin-native permanence without external dependencies, at the cost of higher fees and UTXO set impact.

5.5 Economic Sustainability

Protocol Fee Structure

No Native Fees: Stamps protocol itself collects no fees. All costs are miner fees paid to Bitcoin validators.

Token Economics (SRC-20/721/101):
- **Deploy Fees**: Set by deployer; collected in satoshis by minting smart contracts or indexer-enforced logic
- **Royalties**: Not enforced at protocol level; marketplace-dependent
- **Renewal Fees** (SRC-101): Deployer-set pricing for domain lease extensions

Miner Incentive Alignment

Short-Term: Stamps generate direct fee revenue for miners, incentivizing block inclusion during low-congestion periods.

Long-Term: UTXO set growth imposes costs on future miners/validators. If externalized costs exceed fee revenue, validators may advocate for protocol-level restrictions.

Market-Driven Equilibrium

Fee Market Regulation: High congestion naturally limits Stamp creation as costs rise, creating self-regulating supply dynamics.

Quality vs. Quantity: Expensive minting favors high-value assets (rare art, critical data) over spam, improving signal-to-noise ratio.

Indexer Sustainability: Open-source indexer model ensures community-driven validation without centralized service dependencies. Multiple independent indexers can verify state, preventing single points of failure.

5.6 Economic Tradeoffs Summary

Advantages

1. **True Immutability**: UTXO-based storage guarantees permanence without reliance on archival nodes
2. **Censorship Resistance**: Data survives even if protocol indexers cease operation
3. **Bitcoin-Native Security**: Inherits full Proof-of-Work consensus guarantees
4. **No Auxiliary Dependencies**: Only Bitcoin miner fees required; no token burns or external fees

Disadvantages

1. **High Costs**: 1-4x more expensive than witness-based alternatives
2. **UTXO Set Externality**: Imposes permanent storage costs on all validators
3. **Scaling Constraints**: Limited to ~64KB per asset (OLGA), vs. 400KB for Ordinals
4. **Fee Market Competition**: Vulnerable to congestion-driven cost spikes

Strategic Positioning

Bitcoin Stamps occupies the "maximum permanence" niche within Bitcoin's data inscription ecosystem. Users willing to pay premium costs for uncompromising immutability choose Stamps over cheaper, less permanent alternatives. This positions the protocol as a premium store-of-value layer for digital artifacts requiring absolute permanence guarantees.

5.7 Future Economic Considerations

UTXO Set Management Proposals

Spent Output Archiving: Future Bitcoin soft forks may introduce mechanisms to archive spent outputs while maintaining cryptographic proofs, potentially affecting Stamp permanence.

Fee Policy Changes: BIP proposals targeting data-heavy transactions could introduce additional costs or restrictions on multisig/P2WSH data embedding.

Stamps Adaptation: Protocol must monitor Bitcoin Core development to ensure continued viability under potential consensus rule changes.

Layer 2 Integration

Lightning Network: Stamps could leverage LN for microtransactions involving SRC-20 tokens, though atomic swaps face account-based model challenges.

Sidechains: Federated sidechains (e.g., Liquid) may support Stamps-compatible standards with different cost structures.

Rollups: Bitcoin rollup proposals (e.g., BitVM) could enable Stamps-like permanence at reduced on-chain footprint.

Competitive Landscape Evolution

As Bitcoin's data inscription ecosystem matures, protocols will differentiate along cost-permanence-functionality axes. Stamps' commitment to UTXO-based immutability positions it as the "gold standard" for applications where permanence justifies premium costs—archival NFTs, legal records, decentralized identity systems, and foundational digital artifacts.

References:

- [Bitcoin UTXO Set Research](https://research.mempool.space/utxo-set-report/)
- [SegWit Witness Discount Analysis](https://bitcoinmagazine.com/technical/the-witness-discount-why-some-bytes-are-cheaper-than-others)
- [Bitcoin Stamps vs Ordinals Permanence Analysis](https://coinpedia.org/guest-post/bitcoin-stamps-vs-ordinals-deep-dive-into-future-of-on-chain-permanence/)
- [Economically Unspendable Bitcoin UTXOs](https://blog.lopp.net/economically-unspendable-bitcoin-utxos/)
- [Bitcoin Core SegWit Costs and Risks](https://bitcoincore.org/en/2016/10/28/segwit-costs/)
- [Bitcoin Stamps FAQ](https://stampchain.io/faq)

---

6. Stamps Improvement Proposals (SIPs)

6.1 SIP Governance Framework

Bitcoin Stamps protocol evolves through community-driven Stamps Improvement Proposals (SIPs). This governance model balances protocol stability with extensibility, enabling vetted enhancements while preserving core immutability guarantees.

6.1.1 SIP Lifecycle

Draft: Proposal submitted as GitHub Issue with specification outline. Author presents motivation, technical design, and backward compatibility analysis.

Review: Community discussion period (minimum 14 days). Technical reviewers evaluate:
- Specification clarity and completeness
- Implementation feasibility
- Security implications
- Impact on existing stamps and indexers
- Alignment with protocol philosophy

Accepted: Proposal achieves rough consensus among core developers and major indexer implementations. Specification finalized with version number (SIP-XXXX).

Activated: Implementation deployed with activation block height set 4+ weeks in future. Advance notice ensures all indexers, wallets, and services update before consensus rule changes take effect.

Final: Activation block height reached. New rules enforced by all compliant indexers. Proposal becomes immutable specification.

Superseded: Later SIP replaces or invalidates earlier proposal. Original SIP remains in historical record but no longer active.

6.1.2 Activation Lead Time

Critical Safety Mechanism: All consensus-changing SIPs must specify activation block height at least **4 weeks (approximately 4,032 blocks)** after acceptance.

Rationale:
- Indexer operators need time to upgrade software
- Wallet developers must integrate new transaction formats
- Service providers require testing and deployment cycles
- Community members must understand changes before activation

Historical Precedent: Block 796,000 (SRC-20 Counterparty cutoff) and block 865,000 (OLGA activation) both provided multi-week advance notice, ensuring smooth transitions without network disruption.

6.1.3 Consensus Requirements

Indexer Consensus: Bitcoin Stamps has no on-chain consensus mechanism. Protocol rules are enforced by indexer implementations. SIP activation requires:
- **Reference Indexer**: stampchain.io (official implementation) must deploy support
- **Secondary Indexers**: At least 2 independent implementations demonstrate compatibility
- **Community Signaling**: No significant objections from major stakeholders

Backward Compatibility: SIPs should maintain compatibility with existing stamps whenever possible. Breaking changes require strong justification and comprehensive migration path.

6.1.4 GitHub Issue Tracking

All SIPs are tracked as GitHub Issues in the Bitcoin Stamps repository:
- **Repository**: https://github.com/stampchain-io/btc_stamps
- **Issue Labels**: `SIP`, `enhancement`, `consensus-change`
- **Discussion Forum**: GitHub Discussions for preliminary ideas before formal SIP submission

6.2 Active SIPs

6.2.1 SIP-0001: SRC-20 HTLC (Hash Time-Locked Contracts)

GitHub Issue: [#685](https://github.com/stampchain-io/btc_stamps/issues/685)

Status: Draft (as of 2026-02)

Motivation: Enable trustless atomic swaps and escrow services for SRC-20 tokens through hash time-locked contracts. Supports cross-asset exchanges and conditional transfers without requiring external oracles or modifying Bitcoin consensus.

Technical Design:

SIP-0001 introduces three new SRC-20 operations:

1. `conditional_transfer` — Create HTLC with hashlock and/or timelock:
```json
{
  "p": "src-20",
  "op": "conditional_transfer",
  "tick": "KEVIN",
  "amt": "1000",
  "to": "bc1q...recipient",
  "hashlock": "a4b9c8d7e6f5...sha256hash",
  "timelock": 900000
}
```
- **hashlock** (optional): SHA-256 hash — recipient must reveal preimage to claim
- **timelock** (optional): Block height — sender can refund after this block if unclaimed
- At least one of hashlock/timelock required
- Tokens deducted from sender immediately, held in indexer escrow state

2. `claim` — Recipient claims tokens with preimage:
```json
{
  "p": "src-20",
  "op": "claim",
  "tick": "KEVIN",
  "transfer_tx": "abc123...original_txid",
  "preimage": "secret_value"
```

}
```
- Indexer verifies `SHA-256(preimage)` matches hashlock
- Must be before timelock block height (if timelock set)
- Tokens credited to recipient

**3. `refund` — Sender reclaims tokens after timelock expires**:
```json
{
  "p": "src-20",
  "op": "refund",
  "tick": "KEVIN",
  "transfer_tx": "abc123...original_txid"
}
```
- Only valid after timelock block height reached
- Tokens returned to original sender

Use Cases:
- **Atomic swaps**: Cross-asset exchange (e.g., KEVIN ↔ STAMP) with cryptographic settlement
- **Escrow services**: Time-locked deposits with refund guarantees
- **Trustless bridge deposits**: Lock tokens with hashlock, mint on L2 with preimage reveal (see SIP-0003)
- **Time-locked vesting**: Gradual token unlock over time

Challenges:
- **Liveness requirement**: Both parties must be online during swap window
- **Timelock griefing**: Malicious actors can lock counterparty funds then abandon swap
- **Multi-step process**: Atomic swap requires 4 transactions (2 conditional_transfer + 2 claim)
- **Indexer validation complexity**: Requires SHA-256 verification and timelock enforcement

Activation Timeline: TBD pending community review and implementation testing.

6.2.2 SIP-0003: Cross-Chain Bridges

GitHub Issue: [#485](https://github.com/stampchain-io/btc_stamps/issues/485)

Status: Draft (as of 2026-02)

Motivation: Enable SRC-20 token movement between Bitcoin mainnet and Layer 2 protocols (Lightning Network, sidechains, rollups) while maintaining UTXO-based permanence guarantees for bridged asset records.

Architecture:
```
Bitcoin L1 (Stamps)  ↔  Bridge Contract  ↔  L2 Protocol
    |                       |                    |
  Lock asset          Mint wrapped token      Fast transfers
  (UTXO proof)        (bridge attestation)    (off-chain)
```

Bridge Operations:

1. **Lock** (L1 → L2):
   - User sends SRC-20 transfer to bridge address
   - Bridge operators verify transaction and UTXO proof
```

- L2 mints equivalent wrapped token to user address

2. **Unlock** (L2 → L1):
- User burns wrapped token on L2
- Bridge operators create SRC-20 transfer from bridge address to user
- Bitcoin transaction permanently records bridge event

Security Model:
- **Federated multisig**: M-of-N bridge operators hold Bitcoin keys
- **Fraud proofs**: Users can challenge invalid bridge operations
- **Timelock withdrawals**: Delay allows dispute resolution

Implementation Requirements:
- Bridge indexer module for cross-chain state verification
- Oracle network for L2 state attestation
- Emergency pause mechanism for security incidents

Activation Timeline: Pending security audit and testnet deployment (target Q3 2026).

6.2.3 SIP-0004: Privacy Enhancements

GitHub Issue: [#687](https://github.com/stampchain-io/btc_stamps/issues/687)

Status: Draft (as of 2026-02)

Motivation: Improve SRC-20 transfer privacy through cryptographic commitments while maintaining indexer verifiability. Address concern that account-based model exposes address balances publicly.

Privacy Techniques:

1. Confidential Amounts:
```python
# Pedersen commitments hide transfer amounts
commitment = amount * G + blinding_factor * H

# Indexer verifies: commitment_in == commitment_out (balance preserved)
# Amount remains hidden from public queries
```

2. Stealth Addresses:
```python
```

One-time address per transfer

stealth_addr = hash(sender_secret + recipient_pubkey)

Only recipient can detect and claim transfer

Breaks on-chain address linkage

```

**3. Range Proofs**:
```python

# Prove amount is positive without revealing value

prove(0 < amount < max_supply)

# Prevents negative balance attacks

# Maintains confidentiality

```

Tradeoffs:
- **Proof size**: Range proofs add 1-2KB per transfer (higher fees)
- **Validation cost**: Indexers must verify cryptographic proofs (slower sync)
- **Regulatory risk**: Privacy features may face jurisdictional challenges
- **Complexity**: Wallet implementations require cryptographic libraries

Phased Rollout:
- **Phase 1**: Optional confidential amounts for willing users
- **Phase 2**: Stealth address support in major wallets
- **Phase 3**: Full privacy by default with opt-out mechanism

Activation Timeline: Specification under development (target 2027).

6.2.4 SIP-0005: Binary Transfer Format for SRC-20

GitHub Issue: [#688](https://github.com/stampchain-io/btc_stamps/issues/688)

Status: Draft (as of 2026-02)
```

Motivation: Replace JSON-encoded SRC-20 transactions with compact binary format. Reduce transaction size by approximately 63%, lowering minting costs and increasing throughput.

Format Specification:
```
Binary SRC-20 Transfer Format (44 bytes total):
<prefix:6><version:1><op:1><tick:20><amount:8><decimals:8> = 44 bytes raw
```

Field Breakdown:
- **prefix** (6 bytes): `stamp:` — indexer detection marker (ASCII: `73 74 61 6D 70 3A`)
- **version** (1 byte): `0x01` for format version 1
- **op** (1 byte): Operation code
  - `0x01`: DEPLOY
  - `0x02`: MINT
  - `0x03`: TRANSFER
- **tick** (20 bytes): UTF-8 ticker padded with null bytes
  - Example: "KEVIN" → `4B 45 56 49 4E` + 15 null bytes (`0x00`)
- **amount** (8 bytes): uint64 big-endian raw amount (not decimal-adjusted)
- **decimals** (8 bytes): uint64 big-endian decimal precision

Detection Logic:
```python
if data[:6] == b'stamp:' and data[6] == 0x01:
    # Binary format
    parse_binary(data)
else:
    # JSON format (backward compatible)
    parse_json(data)
```

Benefits:
- **~63% size reduction**: 44 bytes binary vs ~120 bytes JSON
- **Faster indexer parsing**: Binary deserialization vs JSON parsing
- **Lower transaction fees**: Smaller data size reduces on-chain costs
- **Increased data density**: More stamps per block

Migration Strategy:
- Binary format optional after activation
- JSON format remains valid indefinitely (backward compatibility)
- Indexers must support both formats simultaneously
- Wallets can choose format based on user preference

Activation Timeline: TBD pending final specification review.

6.2.5 SIP-0006: Native SRC-20 AMM (Automated Market Maker)

GitHub Issue: [#689](https://github.com/stampchain-io/btc_stamps/issues/689)

Status: Draft (as of 2026-02)

Motivation: Enable trustless on-chain token swaps without order books or centralized exchanges. The account-based SRC-20 model is ideal for AMM implementation since balance updates are atomic indexer operations, eliminating UTXO coordination complexity.

Technical Design:

SIP-0006 introduces four new SRC-20 operations for constant product market maker (Uniswap V2-style):

1. `create_pool` — Deploy new liquidity pool:
```json
{
  "p": "src-20",
  "op": "create_pool",
  "tick_a": "KEVIN",
  "tick_b": "STAMP",
  "fee_tier": 30
}
```
- **fee_tier**: Fee in basis points (10 = 0.1%, 30 = 0.3%, 100 = 1.0%)
- Creates LP token with tick: `LP:KEVIN/STAMP`

2. `add_liquidity` — Deposit token pair to pool:
```json
{
  "p": "src-20",
  "op": "add_liquidity",
  "pool": "LP:KEVIN/STAMP",
  "amt_a": "1000",
  "amt_b": "5000"
}
```
- Deposits proportional to current pool ratio
- Mints LP tokens to liquidity provider
- LP tokens are standard SRC-20 (transferable, tradeable)

3. `remove_liquidity` — Withdraw from pool:
```json
{
  "p": "src-20",
  "op": "remove_liquidity",
  "pool": "LP:KEVIN/STAMP",
  "lp_amt": "500"
}
```
- Burns LP tokens
- Returns proportional share of pool reserves

4. `swap` — Exchange tokens:
```json
{
  "p": "src-20",
  "op": "swap",
  "pool": "LP:KEVIN/STAMP",
  "from_tick": "KEVIN",
  "amt_in": "100"
}
```

Swap Pricing Formula (Constant Product):
```
```

amt_out = (reserve_out × amt_in_with_fee) / (reserve_in + amt_in_with_fee)

where:
  amt_in_with_fee = amt_in × (10000 - fee_bps)

Example (0.3% fee tier):
  amt_in_with_fee = 100 × (10000 - 30) / 10000 = 99.7
`

LP Token Mechanics:
- LP tokens are standard SRC-20 tokens with tick format `LP:{tick_a}/{tick_b}`
- Fully transferable between addresses
- Can be traded on secondary markets
- Mintable/burnable ONLY through AMM operations (add/remove liquidity)
- Represent proportional claim on pool reserves

Phased Rollout:
- **Phase 1**: SRC-20/SRC-20 pools (fully trustless, no external dependencies)
- **Phase 2**: wBTC pools (requires SIP-0007 wrapped asset standard)
- **Phase 3**: Stablecoin pools (requires SIP-0003 bridge for USDT/USDC)

Benefits:
- **Trustless**: No intermediaries, no custody risk
- **Permissionless**: Anyone can create pools or provide liquidity
- **Atomic operations**: Swaps execute in single indexer transaction
- **Capital efficient**: Liquidity providers earn fees on all trades

Challenges:
- **Impermanent loss**: Liquidity providers exposed to price divergence
- **MEV risk**: Indexer ordering can enable front-running (mitigated by transaction fee priority)
- **Pool fragmentation**: Multiple fee tiers for same pair splits liquidity

Activation Timeline: TBD pending community review and Phase 1 implementation.

6.2.6 SIP-0008: Dual Transaction Parsing — Combined SRC-20 Transfer + Stamp Issuance

GitHub Issue: [#692](https://github.com/stampchain-io/btc_stamps/issues/692) (originated from [#554](https://github.com/stampchain-io/btc_stamps/issues/554))

Author: DerpHerpenstein

Status: Draft

Phase: 1 (Foundation) | **Estimated Effort**: 2-3 weeks

Motivation: Currently, a single Bitcoin transaction can only perform one stamp operation — either issue a new stamp OR execute an SRC-20 transfer. Users who want to do both must create two separate transactions, paying double the fees. SIP-0008 enables a single transaction to contain both a stamp issuance and an SRC-20 transfer, reducing costs and enabling new composable workflows.

Technical Design:

The indexer currently processes each transaction for a single stamp operation. SIP-0008 extends the transaction parser to detect and process multiple stamp payloads within a single transaction:

```
Transaction outputs:
  Output 0: SRC-20 transfer payload (bare multisig or P2WSH)
  Output 1: Stamp image data (bare multisig or P2WSH)
  Output 2: Change output
```

Parsing Rules:
1. **Output scanning**: Indexer scans all outputs for stamp-compatible payloads
2. **Payload classification**: Each payload classified as SRC-20 operation or stamp issuance based on content type detection
3. **Ordered execution**: SRC-20 transfers processed before stamp issuance (deterministic ordering)
4. **Atomic processing**: Both operations succeed or both fail — no partial execution
5. **Backward compatibility**: Single-operation transactions continue to work unchanged

Soft Dependency: SIP-0005 (Binary Transfer Format) — binary encoding makes dual payloads more size-efficient, but SIP-0008 works with JSON encoding as well.

Use Cases:
- **Mint-and-transfer**: Create a stamp and immediately send SRC-20 tokens in one transaction
- **Composable workflows**: Agent-driven pipelines that batch stamp operations for efficiency
- **Fee optimization**: Single transaction fee instead of two for combined operations

Activation Timeline: TBD pending community review and Phase 1 implementation.

6.3 Superseded SIPs

6.3.1 SIP-0002: SRC-20 UTXO Binding & Transfer Format v2.0

GitHub Issue: [#484](https://github.com/stampchain-io/btc_stamps/issues/484)

Status: Superseded (by SIP-0001)

Original Motivation: Bind SRC-20 token balances to specific Bitcoin UTXOs to enable single-transaction PSBT-based atomic swaps without multi-step HTLC protocols.

Proposed Design:
```json
{
  "p": "src-20",
  "op": "bind_utxo",
  "tick": "KEVIN",
  "amt": "1000",
  "utxo": "txid:vout"
}
```
- Tokens would be locked to specific UTXO

- Spending the UTXO would automatically transfer bound tokens
- Enabled single-step atomic swaps via PSBT co-signing

Rejection Rationale:
- **Fundamental loss risk**: If user spends bound UTXO in normal Bitcoin transaction, SRC-20 tokens could be lost
  - Bitcoin consensus has no knowledge of SRC-20 state
  - Wallets cannot prevent accidental UTXO spending
  - Loss prevention is impossible without modifying Bitcoin protocol
- **Non-deterministic rescue operations**: Indexer "token recovery" would break consensus determinism
- **SIP-0001 provides superior solution**: HTLC covers all atomic swap use cases without loss risk
- **Complexity vs benefit**: UTXO coordination adds significant implementation burden for marginal UX improvement

Superseded By: SIP-0001 (HTLC) provides trustless atomic swaps without binding tokens to UTXOs, eliminating loss risk while maintaining full functionality.

Lessons Learned:
- Account-based models should not be forcibly bound to UTXO mechanics
- Protocol safety (loss prevention) outweighs UX convenience (single-step swaps)
- Multi-step protocols (HTLC) acceptable when they eliminate fundamental risks

6.4 SIP Process Best Practices

6.4.1 Proposal Template

Title: [SIP-XXXX] Brief descriptive title

Author: GitHub username / contact info

Status: Draft

Type: Standards Track / Informational / Process

Created: YYYY-MM-DD

Sections:
1. **Abstract**: One-paragraph summary
2. **Motivation**: Problem being solved, use cases
3. **Specification**: Technical design, data formats, validation rules
4. **Rationale**: Design decisions, alternatives considered
5. **Backward Compatibility**: Impact on existing stamps/indexers
6. **Test Cases**: Reference implementation tests
7. **Security Considerations**: Attack vectors, mitigations
8. **Activation**: Proposed block height, coordination plan

6.4.2 Review Criteria

Technical Soundness:
- Specification is complete and unambiguous
- Implementation is feasible with existing Bitcoin constraints
- No cryptographic or protocol vulnerabilities

Protocol Alignment:
- Preserves UTXO-based permanence guarantees
- Maintains account-based asset model
- Follows Bitcoin-native encoding principles
- Respects community governance values

Ecosystem Impact:
- Breaking changes justified and necessary
- Migration path documented for affected users
- Indexer implementation complexity is reasonable
- Wallet/service integration burden is acceptable

Community Support:
- Rough consensus among developers
- No strong objections from major stakeholders
- Clear demand from users and builders

6.4.3 Implementation Requirements

Reference Implementation: All accepted SIPs must include:
- Working code in stampchain.io indexer repository
- Comprehensive test suite with edge cases
- Documentation for indexer operators
- Example transactions on testnet

Multi-Indexer Compatibility: At least 2 independent indexer implementations must successfully validate SIP test cases before activation.

Regression Testing: New SIP implementations must pass full historical sync test (genesis block → current tip) without breaking existing stamp validation.

6.5 Open Research Areas

6.5.1 Zero-Knowledge Proofs

Research Question: Can zk-SNARKs enable private SRC-20 transfers with succinct on-chain proofs?

Potential Benefits:
- Strong privacy (ZCash-level confidentiality)
- Compact proofs (200-500 bytes regardless of transfer complexity)
- Trustless verification by indexers

Challenges:
- Trusted setup requirements (or STARK alternatives)
- Proof generation complexity for wallet implementations
- Validation performance impact on indexer sync speed

Status: Exploratory research; no formal SIP yet.

6.5.2 Recursive Stamps v2

Research Question: Can stamps reference external Bitcoin data (taproot scripts, DLCs) to enable advanced smart contracts?

Potential Applications:
- Stamps triggered by DLC oracle outcomes
- Integration with BitVM computation verification
- Lightning Network settlement to stamp ownership

Challenges:
- Cross-protocol coordination complexity
- Security assumptions for external data sources
- Indexer validation of external state

Status: Concept phase; community feedback sought.

6.5.3 Rollup Integration

Research Question: Can Bitcoin rollups (BitVM, Sovereign SDK) support Stamps-compatible assets with L1 permanence guarantees?

Potential Architecture:
- L2 transactions executed off-chain
- Periodic L1 commitment (Merkle root stamped on Bitcoin)
- L2 state reconstructible from L1 commitments

Benefits:
- High throughput (1000s of transfers per second)
- Low per-transfer cost (amortized L1 fees)
- Maintained UTXO permanence for rollup commitments

Challenges:
- Data availability (ensure L2 state accessible)
- Fraud proof mechanisms (dispute resolution)
- Indexer complexity (track both L1 and L2 state)

Status: Monitoring BitVM development; formal SIP pending rollup maturity.

6.6 SIP Summary Table

SIP	Title	Status	GitHub	Target Activation
0001	SRC-20 Conditional Transfers (HTLC)	Draft	[#685](https://github.com/stampchain-io/btc_stamps/issues/685)	TBD
0002	SRC-20 UTXO Binding & Transfer Format v2.0	Superseded (by SIP-0001)	[#484](https://github.com/stampchain-io/btc_stamps/issues/484)	N/A
0003	SRC-20 Cross-Chain Bridge Specification	Draft	[#485](https://github.com/stampchain-io/btc_stamps/issues/485)	TBD
0004	Shielded SRC-20 — Privacy Extension	Draft	[#687](https://github.com/stampchain-io/btc_stamps/issues/687)	2027+ (phased)
0005	Binary Transfer Format for SRC-20	Draft	[#688](https://github.com/stampchain-io/btc_stamps/issues/688)	TBD
0006	Native SRC-20 AMM (Automated Market Maker)	Draft	[#689](https://github.com/stampchain-io/btc_stamps/issues/689)	TBD
0008	Dual Transaction Parsing	Draft	[#692](https://github.com/stampchain-io/btc_stamps/issues/692)	TBD

References:
- [Bitcoin Stamps GitHub Repository](https://github.com/stampchain-io/btc_stamps)
- [SIP-0000: SIP Purpose and Guidelines](https://github.com/stampchain-io/btc_stamps/issues/686)
- [Counterparty Improvement Proposals (CIPs)](https://github.com/CounterpartyXCP/cips) — Inspiration for SIP governance model

---

Next: [Implementation Details →](./implementation.md)
Previous: [← Economic Model](./economics.md)

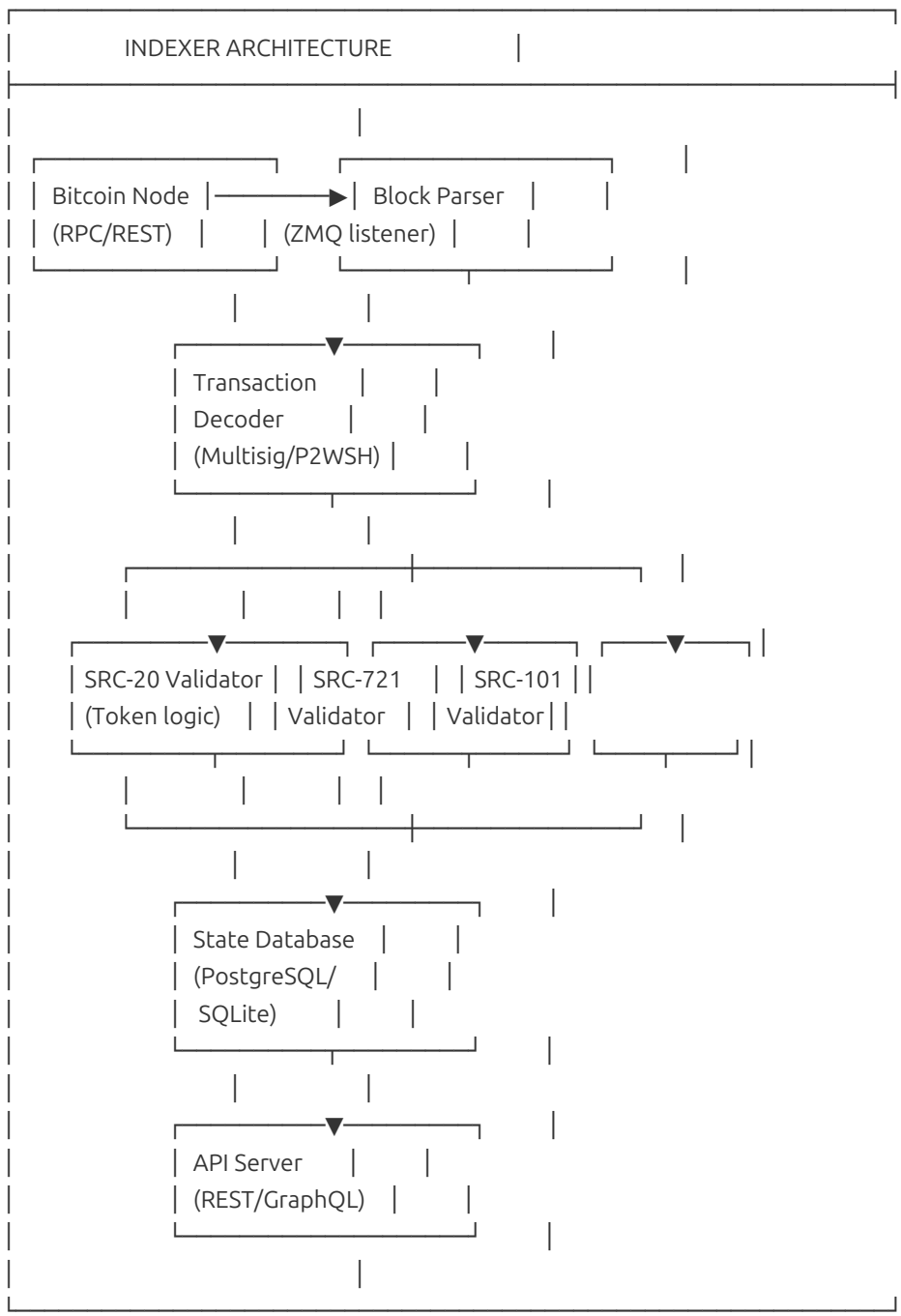
---

7. Implementation

7.1 Indexer Architecture

Bitcoin Stamps protocol relies on **off-chain indexers** to parse stamp transactions, validate operations, and maintain asset state. Unlike Bitcoin's native UTXO consensus, stamp validity is determined by indexer implementations following deterministic validation rules.

7.1.1 Core Components



7.1.2 Block Processing Pipeline

1. Block Discovery:
```python
```

ZMQ subscription for real-time blocks

zmq_socket.subscribe("hashblock")

while True:
  block_hash = zmq_socket.recv()
  block = bitcoin_rpc.getblock(block_hash, 2) # Verbosity 2: full tx data
  process_block(block)
`

2. Transaction Filtering:
` python
def is_stamp_transaction(tx):
  # Check for bare multisig outputs
  for vout in tx['vout']:
    script = vout['scriptPubKey']
    if script['type'] == 'multisig':
      return True
    # Check for P2WSH outputs (OLGA)
    if script['type'] == 'witness_v0_scripthash':
      return True
  return False
`

3. Data Extraction:
` python
def extract_stamp_data(tx):
  data_chunks = []

  # Bare multisig extraction
  for vout in tx['vout']:
    if vout['scriptPubKey']['type'] == 'multisig':
      # Extract fake pubkeys (33 bytes each)
      pubkeys = vout['scriptPubKey']['asm'].split()
      for pk in pubkeys[1:-2]: # Skip OP_1, OP_N, OP_CHECKMULTISIG
        data_chunks.append(bytes.fromhex(pk))

  # P2WSH witness extraction
  for vin in tx['vin']:
    if 'txinwitness' in vin:
      witness_script = vin['txinwitness'][-1] # Last item is script
      # Parse witness script for data chunks
      chunks = parse_witness_script(witness_script)
      data_chunks.extend(chunks)

  # Concatenate and decode
  raw_data = b''.join(data_chunks)
  return decode_stamp_format(raw_data)
`

4. Validation:
` python
def validate_stamp(tx, stamp_data, block_height):
  # Check format validity
  if not is_valid_json(stamp_data):

```
    return False

  parsed = json.loads(stamp_data)
  protocol = parsed.get('p')

  # Route to protocol-specific validator
  if protocol == 'src-20':
    return validate_src20(tx, parsed, block_height)
  elif protocol == 'src-721':
    return validate_src721(tx, parsed, block_height)
  elif protocol == 'src-101':
    return validate_src101(tx, parsed, block_height)

  return False  # Unknown protocol
```

5. State Update:
```python
def update_state(tx, stamp_data, block_height):
  parsed = json.loads(stamp_data)

  if parsed['op'] == 'deploy':
    create_asset(parsed, tx.txid, block_height)

  elif parsed['op'] == 'mint':
    increase_balance(
      address=tx.sender_address,
      asset=parsed['tick'],
      amount=parsed['amt']
    )

  elif parsed['op'] == 'transfer':
    transfer_balance(
      from_addr=tx.sender_address,
      to_addr=parsed['to'],
      asset=parsed['tick'],
      amount=parsed['amt']
    )
```

7.1.3 State Database Schema

Core Tables:
```sql
-- Asset registry
CREATE TABLE assets (
  asset_name TEXT PRIMARY KEY,
  deploy_txid TEXT NOT NULL,
  deploy_block INTEGER NOT NULL,
  deployer_address TEXT NOT NULL,
  max_supply NUMERIC,
  divisible BOOLEAN DEFAULT TRUE,
  locked BOOLEAN DEFAULT FALSE,
  metadata JSONB
```

```
);

-- Account balances (account-based model)
CREATE TABLE balances (
    address TEXT NOT NULL,
    asset TEXT NOT NULL REFERENCES assets(asset_name),
    amount NUMERIC NOT NULL DEFAULT 0,
    last_updated_block INTEGER NOT NULL,
    PRIMARY KEY (address, asset)
);

-- Transfer history
CREATE TABLE transfers (
    txid TEXT PRIMARY KEY,
    block_height INTEGER NOT NULL,
    timestamp INTEGER NOT NULL,
    from_address TEXT NOT NULL,
    to_address TEXT NOT NULL,
    asset TEXT NOT NULL REFERENCES assets(asset_name),
    amount NUMERIC NOT NULL,
    status TEXT NOT NULL  -- 'valid', 'invalid'
);

-- Stamp metadata
CREATE TABLE stamps (
    stamp_id SERIAL PRIMARY KEY,
    txid TEXT NOT NULL,
    block_height INTEGER NOT NULL,
    cpid TEXT,  -- Counterparty asset ID (if legacy)
    stamp_url TEXT,
    stamp_hash TEXT,
    stamp_mimetype TEXT,
    supply INTEGER DEFAULT 1,
    divisible BOOLEAN DEFAULT FALSE,
    locked BOOLEAN DEFAULT FALSE,
    creator_address TEXT NOT NULL,
    encoding TEXT NOT NULL  -- 'multisig', 'p2wsh', 'olga'
);

-- SRC-721 compositions
CREATE TABLE src721_layers (
    composition_id TEXT PRIMARY KEY,
    parent_stamp_ids INTEGER[] NOT NULL,
    layer_order INTEGER[] NOT NULL,
    attributes JSONB,
    rendered_hash TEXT
);
```

7.1.4 Reorganization Handling

Challenge: Bitcoin can experience chain reorganizations (reorgs) where blocks are replaced. Indexers must roll back state and replay new chain.

```python
def handle_reorganization(old_tip_height, new_tip_height, fork_height):
    """
    old_tip_height: Previous chain tip
    new_tip_height: New chain tip after reorg
    fork_height: Block where chains diverged
    """

    # Step 1: Roll back state to fork point
    with db.transaction():
        # Reverse all transfers after fork height
        reversed_transfers = db.query("""
            SELECT * FROM transfers
            WHERE block_height > $1
            ORDER BY block_height DESC
        """, fork_height)

        for transfer in reversed_transfers:
            # Undo transfer: reverse balance changes
            balances[transfer.from_address][transfer.asset] += transfer.amount
            balances[transfer.to_address][transfer.asset] -= transfer.amount

        # Delete rolled-back data
        db.execute("DELETE FROM transfers WHERE block_height > $1", fork_height)
        db.execute("DELETE FROM stamps WHERE block_height > $1", fork_height)

    # Step 2: Replay blocks from new chain
    for height in range(fork_height + 1, new_tip_height + 1):
        block_hash = bitcoin_rpc.getblockhash(height)
        block = bitcoin_rpc.getblock(block_hash, 2)
        process_block(block)

    logger.info(f"Reorg handled: fork at {fork_height}, replayed to {new_tip_height}")
```

Detection:
```python
def check_for_reorg(new_block):
    # Get current chain tip from DB
    current_tip = db.query("SELECT MAX(block_height) FROM transfers").scalar()

    # Get parent of new block
    new_block_parent = new_block['previousblockhash']

    # Check if parent matches our current tip
    expected_parent = db.query("""
        SELECT block_hash FROM blocks WHERE block_height = $1
    """, current_tip).scalar()

    if new_block_parent != expected_parent:
        # Reorg detected - find fork point
        fork_height = find_fork_point(new_block_parent)
        handle_reorganization(current_tip, new_block['height'], fork_height)
```

7.2 Consensus Model

7.2.1 Deterministic Validation

Critical Property: All indexers processing the same blockchain must arrive at identical state.

```python
# Example: SRC-20 transfer validation must be deterministic

def validate_src20_transfer(tx, parsed, block_height):
    # Rule 1: Sender must have sufficient balance
    sender = tx.sender_address
    asset = parsed['tick']
    amount = Decimal(parsed['amt'])

    if balances[sender][asset] < amount:
        return False  # Invalid: insufficient balance

    # Rule 2: Asset must exist
    if not asset_exists(asset):
        return False  # Invalid: unknown asset

    # Rule 3: Asset must not be locked
    if assets[asset].locked:
        return False  # Invalid: asset locked

    # Rule 4: Amount must respect divisibility
    if not assets[asset].divisible and amount != int(amount):
        return False  # Invalid: fractional amount for indivisible asset

    # All rules pass
    return True
```

Consensus Rules:
- Validation logic must be **order-dependent**: Process transactions in block order
- Floating-point arithmetic **forbidden**: Use fixed-point decimals (Python `Decimal`)
- No external data sources: Only blockchain data determines validity
- Edge cases must have **defined behavior**: No ambiguous outcomes

7.2.2 First-Seen Rule

Problem: Multiple transactions in same block may conflict (e.g., double-spend attempt).

Solution: Process transactions in block order (first-seen wins).

```python
def process_block(block):
    # Process transactions in order (tx index 0, 1, 2, ...)
    for tx_index, tx in enumerate(block['tx']):
        if is_stamp_transaction(tx):
            stamp_data = extract_stamp_data(tx)

            # Validate with current state
            if validate_stamp(tx, stamp_data, block['height']):
                update_state(tx, stamp_data, block['height'])
                assign_stamp_number(tx.txid)  # Only valid stamps get numbers
            else:
                log_invalid_stamp(tx.txid, "Validation failed")

    # Result: First valid transaction wins; later conflicts are invalid
```

Example:
```
Block 900,000 contains:
- Tx A (index 5): Transfer 1000 KEVIN from Alice to Bob
- Tx B (index 12): Transfer 1000 KEVIN from Alice to Carol

Alice balance: 1000 KEVIN

Processing:
1. Tx A validated (Alice has 1000 KEVIN) → Alice: 0, Bob: 1000
2. Tx B validated (Alice has 0 KEVIN) → INVALID (insufficient balance)

Result: Bob receives 1000 KEVIN, Carol receives nothing
```

7.2.3 Consensus Checkpoints

Purpose: Ensure indexer implementations agree on historical state.

Methodology: Community-generated state hashes at key block heights.

```python
# Checkpoint format

CHECKPOINTS = {
    796000: {  # Counterparty cutoff block
        'state_hash': 'a3f5c9e8d7b6...',  # Hash of all balances at block 796000
        'total_stamps': 18516,
        'total_assets': 142
    },
    865000: {  # OLGA activation block
        'state_hash': 'e8d7b6a3f5c9...',
        'total_stamps': 45203,
        'total_assets': 387
```

```
    }
}

def verify_checkpoint(block_height):
    if block_height not in CHECKPOINTS:
        return True  # No checkpoint at this height

    # Compute state hash
    current_state_hash = compute_state_hash()
    expected_hash = CHECKPOINTS[block_height]['state_hash']

    if current_state_hash != expected_hash:
        raise ConsensusError(
            f"State mismatch at block {block_height}: "
            f"expected {expected_hash}, got {current_state_hash}"
        )

    logger.info(f"Checkpoint verified at block {block_height}")
    return True

def compute_state_hash():
    # Deterministic hash of all balances
    all_balances = db.query("""
        SELECT address, asset, amount
        FROM balances
        ORDER BY address, asset
    """).fetchall()

    # Serialize to JSON with sorted keys
    state_json = json.dumps(all_balances, sort_keys=True)
    return hashlib.sha256(state_json.encode()).hexdigest()
```

7.2.4 Multi-Indexer Consensus

Reference Implementations:
1. **stampchain.io** (official): Python/Rust hybrid, PostgreSQL backend
2. **OpenStamps** (community): Independent implementation for validation
3. **Alternative indexers**: Third-party implementations for redundancy

Consensus Verification:
```bash
# Compare indexer outputs at block height 900,000

curl https://stampchain.io/api/balances/bc1q...xyz?block=900000

# Response: {"KEVIN": "1000.0", "STAMP": "50.0"}
```

curl https://openstamps.io/api/balances/bc1q...xyz?block=900000

Response: {"KEVIN": "1000.0", "STAMP": "50.0"}

If outputs differ → consensus bug, investigation required

```

**Divergence Protocol**:
1. Community reports divergence via GitHub Issue
2. Indexer operators freeze state at divergence block
3. Debug sessions compare validation logs step-by-step
4. Root cause identified (usually edge case in validation logic)
5. Reference implementation patched
6. All indexers update and re-sync from divergence point

## 7.3 Validation Logic

### 7.3.1 SRC-20 Validation

```python
def validate_src20(tx, parsed, block_height):
    op = parsed.get('op')

    if op == 'deploy':
        return validate_src20_deploy(parsed, tx, block_height)
    elif op == 'mint':
        return validate_src20_mint(parsed, tx, block_height)
    elif op == 'transfer':
        return validate_src20_transfer(parsed, tx, block_height)
    else:
        return False  # Unknown operation

def validate_src20_deploy(parsed, tx, block_height):
    # Required fields
    required = ['p', 'op', 'tick', 'max', 'lim']
    if not all(field in parsed for field in required):
        return False

    # Ticker constraints
    tick = parsed['tick']
    if not (1 <= len(tick) <= 5):  # 1-5 characters
        return False
    if not tick.isupper():  # Uppercase only
        return False
```
```

```
    # Check uniqueness
    if asset_exists(tick):
        return False  # Duplicate ticker

    # Supply constraints
    max_supply = Decimal(parsed['max'])
    mint_limit = Decimal(parsed['lim'])

    if max_supply <= 0 or mint_limit <= 0:
        return False
    if mint_limit > max_supply:
        return False

    # Counterparty cutoff rule
    if block_height > 796000:
        # After block 796,000, must use native Bitcoin encoding
        if uses_counterparty_encoding(tx):
            return False

    return True

def validate_src20_mint(parsed, tx, block_height):
    # Asset must exist
    asset = parsed['tick']
    if not asset_exists(asset):
        return False

    # Check supply constraints
    asset_info = get_asset(asset)
    current_supply = get_total_supply(asset)
    mint_amount = Decimal(parsed['amt'])

    # Respect per-mint limit
    if mint_amount > asset_info.mint_limit:
        return False

    # Respect max supply
    if current_supply + mint_amount > asset_info.max_supply:
        return False

    # Asset must not be locked
    if asset_info.locked:
        return False

    return True

def validate_src20_transfer(parsed, tx, block_height):
    sender = tx.sender_address
    asset = parsed['tick']
    amount = Decimal(parsed['amt'])

    # Asset must exist
    if not asset_exists(asset):
        return False

    # Sender must have balance
```

```
    if get_balance(sender, asset) < amount:
        return False

    # Amount must be positive
    if amount <= 0:
        return False

    # Respect divisibility
    asset_info = get_asset(asset)
    if not asset_info.divisible:
        if amount != int(amount):
            return False  # No fractional amounts

    return True
```

7.3.2 SRC-721 Validation

```python
def validate_src721(tx, parsed, block_height):
    # Required fields
    if 'layers' not in parsed or not isinstance(parsed['layers'], list):
        return False

    # Verify all referenced stamps exist
    for layer_stamp_id in parsed['layers']:
        if not stamp_exists(layer_stamp_id):
            return False  # Invalid: references non-existent stamp

    # Layer count limits (prevent DOS via huge compositions)
    if len(parsed['layers']) > 100:
        return False

    # Optional attributes validation
    if 'attributes' in parsed:
        if not isinstance(parsed['attributes'], dict):
            return False

    return True
```

7.3.3 Encoding Detection

```python
def detect_encoding(tx):
    """Determine stamp encoding method"""

    # Check for bare multisig
    for vout in tx['vout']:
        if vout['scriptPubKey']['type'] == 'multisig':
            return 'bare_multisig'
```

```
    # Check for P2WSH (OLGA)
    for vout in tx['vout']:
        if vout['scriptPubKey']['type'] == 'witness_v0_scripthash':
            # Verify witness script contains stamp data
            if is_olga_format(tx):
                return 'p2wsh_olga'

    # Check for legacy Counterparty OP_RETURN
    for vout in tx['vout']:
        if vout['scriptPubKey']['type'] == 'nulldata':
            if is_counterparty_format(tx):
                return 'counterparty_op_return'

    return None  # Not a stamp
```

7.4 Performance Optimization

7.4.1 Rust Parser Integration

Bottleneck: Python JSON parsing is slow for high-volume indexing.

Solution: Rust-based parser for critical path (stampchain.io implementation).

```rust
// Rust: Fast binary parsing and validation
use serde_json;

pub fn parse_stamp_data(raw_bytes: &[u8]) -> Result<StampData, ParseError> {
    // Validate length prefix
    let expected_len = u16::from_be_bytes([raw_bytes[0], raw_bytes[1]]);
    let actual_len = raw_bytes.len() - 2;

    if expected_len as usize != actual_len {
        return Err(ParseError::LengthMismatch);
    }

    // Parse JSON (serde_json is 20-50x faster than Python)
    let json_data = &raw_bytes[2..];
    let parsed: StampData = serde_json::from_slice(json_data)?;

    Ok(parsed)
}
```

Performance Impact:
- Full chain sync (genesis → block 900,000): 3 hours (Rust) vs 48 hours (pure Python)
- Real-time block processing: <100ms per block (Rust) vs 1-3 seconds (Python)

7.4.2 Database Indexing

```sql
-- Critical indexes for query performance
CREATE INDEX idx_balances_address ON balances(address);
CREATE INDEX idx_balances_asset ON balances(asset);
CREATE INDEX idx_transfers_block_height ON transfers(block_height);
CREATE INDEX idx_stamps_txid ON stamps(txid);
CREATE INDEX idx_stamps_creator ON stamps(creator_address);

-- Composite indexes for common queries
CREATE INDEX idx_transfers_asset_block ON transfers(asset, block_height);
CREATE INDEX idx_balances_address_asset ON balances(address, asset);
```

7.4.3 Caching Strategy

```python
# In-memory cache for hot data
from functools import lru_cache

@lru_cache(maxsize=10000)
def get_asset_info(asset_name):
    """Cache asset metadata (rarely changes)"""
    return db.query("SELECT * FROM assets WHERE asset_name = $1", asset_name)

@lru_cache(maxsize=100000)
def stamp_exists(stamp_id):
    """Cache stamp existence checks"""
    return db.query("SELECT 1 FROM stamps WHERE stamp_id = $1", stamp_id).scalar()

# Invalidate cache on state updates
def update_state(tx, stamp_data, block_height):
    # ... update database ...

    # Clear affected cache entries
    if stamp_data['op'] == 'deploy':
        get_asset_info.cache_clear()  # New asset added
```

7.5 API Layer

7.5.1 REST Endpoints

` python

Example API endpoints (stampchain.io)

```
@app.get("/api/v1/balance/{address}")
def get_balance(address: str, asset: str = None, block: int = None):
    """Get address balance(s) at specific block height"""
    if block:
        # Historical balance query
        return query_balance_at_block(address, asset, block)
    else:
        # Current balance
        return query_current_balance(address, asset)

@app.get("/api/v1/asset/{tick}")
def get_asset_info(tick: str):
    """Get asset metadata"""
    return {
        "tick": tick,
        "deploy_block": assets[tick].deploy_block,
        "max_supply": assets[tick].max_supply,
        "current_supply": get_total_supply(tick),
        "holders": count_holders(tick),
        "transfers": count_transfers(tick)
    }

@app.get("/api/v1/stamp/{stamp_id}")
def get_stamp(stamp_id: int):
    """Get stamp metadata and image data"""
    stamp = db.query("SELECT * FROM stamps WHERE stamp_id = $1", stamp_id)
    return {
        "stamp_id": stamp.stamp_id,
        "txid": stamp.txid,
        "block_height": stamp.block_height,
        "creator": stamp.creator_address,
        "image_url": stamp.stamp_url,
        "encoding": stamp.encoding
    }
```

7.5.2 WebSocket Real-Time Updates

` python

```
import asyncio
from websockets import serve
```

```
async def stream_new_stamps(websocket):
    """Stream newly confirmed stamps to connected clients"""
    while True:
        new_stamp = await stamp_queue.get()
        await websocket.send(json.dumps({
            "type": "new_stamp",
            "stamp_id": new_stamp.stamp_id,
            "txid": new_stamp.txid,
            "block_height": new_stamp.block_height
        }))
```

Client usage:

ws = new WebSocket("wss://stampchain.io/ws/stamps")

ws.onmessage = (event) => { console.log("New stamp:", event.data) }

```

---

### 7.6 Implementation Summary

**Architecture**: Off-chain indexers parse Bitcoin blockchain and maintain asset state in deterministic, verifiable manner.

**Consensus**: No on-chain enforcement—indexers independently validate and must agree on state through deterministic rules.

**Activation Lead Time**: Protocol upgrades (SIPs) require 4+ weeks notice, specified as activation block height.

**Performance**: Hybrid Python/Rust implementation achieves full chain sync in ~3 hours; real-time processing <100ms per block.

**Redundancy**: Multiple independent indexer implementations prevent single point of failure; community checkpoints ensure consensus.

---
```

References:
- [stampchain.io Indexer Source Code](https://github.com/stampchain-io/btc_stamps)
- [OpenStamps Independent Implementation](https://github.com/openstamps/indexer)
- [Bitcoin Core RPC Documentation](https://developer.bitcoin.org/reference/rpc/)
- [ZMQ Block Notifications](https://github.com/bitcoin/bitcoin/blob/master/doc/zmq.md)

Next: [Security Analysis →](./security.md)
Previous: [← SIPs](./improvement-proposals.md)

---

8. Security Analysis

8.1 Immutability Guarantees

Bitcoin Stamps' security model is fundamentally derived from its UTXO-based storage architecture. This section analyzes the threat model, attack vectors, and security properties inherited from Bitcoin.

8.1.1 UTXO Set Permanence

Security Property: Once a stamp transaction is confirmed in a Bitcoin block with sufficient depth, the embedded data is **effectively immutable** and **unprunable**.

Mechanism:
```
Stamp Transaction (Block N)
  ↓
Bitcoin Miners Confirm (6+ confirmations)
  ↓
UTXO Created (scriptPubKey contains stamp data)
  ↓
Full Nodes Store UTXO (required for validation)
  ↓
Data Persists (unprunable consensus-critical data)
```

Proof-of-Work Protection:
- To reverse a stamp (via blockchain reorganization), attacker must:
  1. Mine competing chain longer than confirmed depth
  2. Accumulate more proof-of-work than honest network
  3. Sustain 51% hashrate majority for extended period

Cost Analysis (6 confirmations = ~1 hour):
```python
# Attack cost to reverse 6-block-deep stamp (as of 2026)

network_hashrate = 600_000_000 TH/s  # ~600 exahash/s
block_reward = 3.125 BTC  # Post-2024 halving
btc_price = 60000  # USD

# Minimum cost to mine 7 blocks (replace 6 + extend by 1)

attack_cost = 7 * block_reward * btc_price
attack_cost = 7 * 3.125 * 60000 = $1,312,500 USD

# Reality: Requires acquiring 51% hashrate hardware (billions USD)

# Practical cost >> $1M due to hardware, electricity, opportunity cost
```

Result: Reversing confirmed stamps is economically infeasible for rational attackers.

8.1.2 UTXO vs Witness Data

Critical Distinction: Bitcoin Stamps store data in UTXO set, NOT witness data.

Property	UTXO Set Data (Stamps)	Witness Data (Ordinals)
Required for validation	✓ Yes (spending UTXO)	✗ No (signature verification only)
Prunable by nodes	✗ No (breaks validation)	✓ Yes (after tx validation)
Consensus-critical	✓ Yes	⚠ Partial (not for future txs)
Archival dependency	✗ None (all full nodes)	⚠ Requires archival nodes
Long-term permanence	✓ Guaranteed	⚠ Dependent on node policies

Security Implication: Stamps data survives even if:
- Archival nodes stop operating
- Witness data is pruned by majority of nodes
- Protocol indexers cease operation

Threat Scenario Analysis:

Scenario 1: Ordinals data loss
```
Year 2035: Bitcoin Core default config prunes witness data after 1 year
→ Majority of nodes delete old Ordinals inscription data
→ Only archival nodes retain full witness history
→ If archival nodes shut down, inscription data is lost
→ Ordinals become unrecoverable
```

Scenario 2: Stamps resilience
```
Year 2035: Same pruning scenario
→ Stamps data is in UTXO set (unprunable)
→ All full nodes retain stamp data (required for validation)
→ No dependency on archival nodes
→ Stamps remain permanently accessible
```

8.1.3 UTXO Spending Risk

Vulnerability: If stamp-bearing UTXO is spent, data remains in blockchain history but exits active UTXO set.

Mitigation: Stamp protocol uses **economically unspendable UTXOs**:
```python
# Example stamp output

scriptPubKey: OP_1 <fake_pubkey_1> <fake_pubkey_2> <real_pubkey> OP_3 OP_CHECKMULTISIG
value: 546 satoshis  # Dust limit

# Spending cost analysis

input_size = 150 bytes  # Approx size to spend this UTXO
fee_rate = 20 sat/vByte  # Typical fee rate
spend_cost = 150 * 20 = 3000 satoshis

# Economic rationality

output_value = 546 sats
```

spend_cost = 3000 sats
net_loss = 2454 sats

Conclusion: Economically irrational to spend (lose money)

```

**Result**: Stamp UTXOs are **economically unspendable** under normal fee markets, ensuring perpetual UTXO set residence.

**Exception**: During extremely low fee periods (<1 sat/vByte), spending may become economically viable.
However:
- Most stamp creators use addresses without private keys (burn addresses)
- Community norm: Do not spend stamp UTXOs
- Even if spent, data remains in blockchain history (recoverable via archival indexing)

### 8.1.4 Consensus-Layer Protection

**Property**: Stamps inherit Bitcoin's Proof-of-Work security.

**Attack Resistance**:
1. **51% Attack**: Requires sustained majority hashrate control (>$10B hardware investment)
2. **Sybil Attack**: PoW makes creating fake blocks prohibitively expensive
3. **Eclipse Attack**: Does not affect confirmed stamp data (only network propagation)
4. **Censorship**: Miners can censor new stamps, but cannot erase confirmed ones

**Finality**: After ~6 confirmations (~1 hour), stamp data has same security as Bitcoin monetary transactions. No known attack can reverse deeply confirmed stamps without breaking Bitcoin itself.

## 8.2 Indexer Security Model

### 8.2.1 Trust Assumptions

**Centralization Risk**: Unlike Bitcoin's native consensus, stamp *validity* is determined by off-chain indexers.

**Trust Model**:
```
Bitcoin Layer: Trustless (PoW consensus)
  ↓ (data storage)
Stamp Data: Permanently stored (guaranteed)
  ↓ (interpretation)
Indexer Layer: Trust-minimized (open-source, multi-implementation)
  ↓ (presentation)
Application Layer: Varies (wallet/explorer trust)
```
```

Key Insight: Users must trust indexer *validation logic*, not data availability. Data is guaranteed by Bitcoin; only interpretation requires indexer trust.

8.2.2 Indexer Attack Vectors

Attack 1: Malicious Indexer

Scenario: Rogue indexer reports false balances.

```python
# Honest indexer
get_balance("bc1q...xyz", "KEVIN") → 1000 KEVIN

# Malicious indexer
get_balance("bc1q...xyz", "KEVIN") → 999999 KEVIN  # False balance
```

Mitigation:
1. **Multi-indexer verification**: Users query multiple independent indexers
2. **Open-source validation**: Anyone can verify balances by running own indexer
3. **Consensus checkpoints**: Community-verified state hashes at key blocks
4. **Reputation systems**: Wallets prioritize trusted indexers (stampchain.io, OpenStamps)

Result: Attack detected when balances diverge across indexers. Malicious indexer loses reputation; honest indexers remain authoritative.

Attack 2: State Divergence Bug

Scenario: Bug in indexer code causes state divergence across implementations.

```python
# Indexer A (buggy edge case handling)
process_transfer(amount="1000.00000001")  # Accepts fractional indivisible token
→ balance_A["KEVIN"] = 1000.00000001

# Indexer B (correct validation)
process_transfer(amount="1000.00000001")  # Rejects invalid transfer
→ balance_B["KEVIN"] = 1000
```

```

*Detection*:
```bash

# Community monitoring

curl https://stampchain.io/api/balance/bc1q...xyz → {"KEVIN": "1000.00000001"}
curl https://openstamps.io/api/balance/bc1q...xyz → {"KEVIN": "1000"}

# Divergence alert triggered

```

Mitigation:
1. **Consensus checkpoints**: Pre-computed state hashes at milestone blocks
2. **Test suites**: Comprehensive edge case testing
3. **Multi-language implementations**: Python, Rust, Go reduce likelihood of identical bugs
4. **Bug bounty programs**: Incentivize discovery and reporting

Resolution:
1. Freeze indexer state at divergence block
2. Debug session: Compare validation logs transaction-by-transaction
3. Identify root cause (usually edge case in validation logic)
4. Patch reference implementation
5. All indexers re-sync from divergence point
6. Community consensus on canonical state

Attack 3: Eclipse Attack on Indexer

Scenario: Attacker isolates indexer's Bitcoin node, feeds fake blocks.

```
Attacker → Fake Bitcoin blocks → Isolated indexer node → False stamp state
```

Mitigation:
1. **Multiple Bitcoin node connections**: Indexer connects to diverse nodes
2. **Checkpoint validation**: Verify block hashes match known checkpoints
3. **Network diversity**: Connect to nodes across different ISPs, geolocations
4. **Block header verification**: Validate cumulative PoW matches expected difficulty

Result: Isolated indexer detects anomaly (PoW mismatch, checkpoint failure) and alerts operator.

8.2.3 Data Availability

Property: Stamp data is available as long as Bitcoin network operates.

Availability Guarantees:
```

1. **Full Nodes**: ~50,000 Bitcoin full nodes globally store UTXO set
2. **Geographic Distribution**: Nodes across 100+ countries
3. **Independent Operators**: Diverse node operators (mining pools, exchanges, enthusiasts)
4. **Redundancy**: Single node failure has no impact (1000s of backups)

Failure Scenario Analysis:

Scenario: All indexers shut down
```
→ Stamp data remains in Bitcoin UTXO set (unchanged)
→ Any party can launch new indexer, sync from genesis
→ Asset balances reconstructible from blockchain
→ Protocol continues functioning (trustless recovery)
```

Scenario: Catastrophic Bitcoin network failure
```
→ If Bitcoin dies, stamps die with it (accepted risk)
→ No protocol can survive underlying blockchain failure
→ Stamps permanence = Bitcoin permanence (aligned incentives)
```

8.3 Protocol-Specific Vulnerabilities

8.3.1 Front-Running Attacks

Vulnerability: Attacker observes pending stamp transaction (mempool), submits higher-fee competing transaction.

Example:
```
Alice broadcasts: MINT 1000 KEVIN (fee: 10 sat/vByte)
  ↓ (mempool)
Bob observes transaction, broadcasts: MINT 1000 KEVIN (fee: 50 sat/vByte)
  ↓ (next block)
Bob's transaction confirms first → Bob receives KEVIN
Alice's transaction confirms second → Alice receives nothing (max supply reached)
```

Mitigation:
1. **Privacy**: Use private transaction relay (direct miner submission)
2. **High fees**: Pay competitive fee rate to discourage front-running
3. **MEV-resistance**: SRC-20 minting is first-come-first-served (no extractable value in ordering)
4. **Batch minting**: Deploy + mint in same transaction (atomic operation)

Limitation: Front-running is inherent to public mempool. Complete mitigation requires private mempools (availability/centralization tradeoff).

8.3.2 Replay Attacks

Vulnerability: Reuse of stamp transaction on chain forks (e.g., contentious hard fork).

Scenario:
```
Bitcoin forks into Chain A and Chain B
Alice's stamp transaction valid on both chains
→ Stamp created on Chain A
→ Same stamp replayed on Chain B (unintended duplication)
```

Mitigation:
1. **Chain-specific indexers**: Community designates canonical chain (longest PoW)
2. **Replay protection**: Future SIPs may include chain ID in transactions
3. **Economic disincentive**: Forked chains typically have low value (no incentive to replay)

Historical Example: Bitcoin Cash (2017) and Bitcoin SV (2018) forks had separate Counterparty ecosystems. No significant stamp replay issues due to community consensus on Bitcoin mainnet.

8.3.3 Ticker Squatting

Vulnerability: Malicious actor deploys popular ticker before legitimate project.

Example:
```
Attacker deploys "STAMP" token (malicious)
  ↓ (1 month later)
Legitimate STAMP project launches
  ↓ (ticker already taken)
Legitimate project must use alternative ticker ("STAMP2", "STMP")
```

Mitigation:
1. **First-come-first-served**: Protocol design accepts ticker squatting as valid
2. **Community curation**: Indexers/wallets flag known malicious tickers
3. **Metadata verification**: Users verify deploy block, deployer address
4. **Naming services**: SRC-101 enables human-readable names (alternative to tickers)
5. **Social consensus**: Community recognizes legitimate projects regardless of ticker

Accepted Risk: Bitcoin Stamps follows permissionless ethos—anyone can deploy any ticker. Scam prevention is social/application layer responsibility, not protocol enforcement.

8.3.4 Dust Attack

Vulnerability: Attacker sends tiny stamp token amounts to many addresses, tracking UTXO linkage.

Example:
```
Attacker sends 0.00000001 KEVIN to 10,000 addresses
  ↓
```

Tracks which addresses consolidate UTXOs (reveals address clustering)
  ↓
Deanonymizes user identity via address linkage
`

Mitigation:
1. **Ignore dust**: Wallets can hide balances below threshold
2. **Coin control**: Users avoid consolidating dust with main balance
3. **Privacy protocols**: SIP-0004 (confidential transfers) breaks linkage
4. **CoinJoin integration**: Mix UTXOs before consolidation

Limitation: Account-based model means dust tokens don't create on-chain linkage (no UTXOs to track). Dust attack less effective against stamps than UTXO-based tokens.

8.4 Attack Cost Analysis

8.4.1 Stamp Reversal Attack

Goal: Delete or modify confirmed stamp data.

Required Attack: 51% attack on Bitcoin network.

Cost (as of 2026):
` python

Current Bitcoin hashrate

total_hashrate = 600_000_000 TH/s # 600 exahash/s

To achieve 51% majority

required_hashrate = 600_000_000 * 0.51 / 0.49 = 624_489_796 TH/s

Hardware cost (Antminer S19 XP: 140 TH/s, $5000 each)

miners_needed = 624_489_796 / 140 = 4,460,641 miners
hardware_cost = 4,460,641 * 5000 = $22,303,205,000 (~$22 billion USD)

Operational cost (electricity: $0.05/kWh, 3.25 kW per miner)

daily_power_cost = 4,460,641 *3.25* 24 * 0.05 = $17,344,000/day

Attack duration to reverse 6-deep stamp

attack_duration = 1 hour (mine 7 blocks)
attack_cost = $22.3B (hardware) + $720k (electricity) ≈ $22.3 billion

Opportunity cost (forgoing legitimate mining revenue)

blocks_mined = 7
revenue_lost = 7 *3.125 BTC* $60,000 = $1,312,500
`

Total Attack Cost: ~$22 billion USD (hardware) + ongoing electricity + lost revenue.

Conclusion: Economically irrational for all but nation-state attackers. Stamp data is secured by Bitcoin's cumulative PoW.

8.4.2 Indexer Manipulation Attack

Goal: Trick users into accepting false stamp balances.

Attack Vector: Operate malicious indexer reporting inflated balances.

Cost: ~$10,000 (server costs) + development time.

Mitigation Cost: $0 (users query multiple indexers for free).

Success Probability: Near zero (users detect divergence across indexers).

Conclusion: Low-cost attack with negligible success probability. Not economically viable.

8.4.3 Ticker Squatting Attack

Goal: Profit from squatting popular tickers before legitimate projects.

Cost: ~$50-$500 per ticker (deploy transaction fee).

Potential Profit: Speculative (reselling ticker to project, or scam exit).

Mitigation: Community curation, wallet warnings, metadata verification.

Conclusion: Low-cost nuisance attack. Profitable only if users fail to verify legitimacy. Social layer mitigation effective.

8.5 Threat Model Summary

8.5.1 Security Hierarchy

Layer 1: Bitcoin Consensus (Trustless)
- Stamp data permanence guaranteed by PoW
- Reversal requires >$20B attack (infeasible)
- Data availability as long as Bitcoin operates

Layer 2: Indexer Validation (Trust-Minimized)
- ⚠ Requires trust in indexer validation logic
- Mitigated by multi-indexer consensus
- Open-source, verifiable by anyone
- ⚠ State divergence bugs possible (rare, detectable, fixable)

Layer 3: Application Layer (Trust-Dependent)
- ⚠ Wallets/explorers may report false data
- ⚠ Users must verify application integrity
- Mitigated by using reputable services

8.5.2 Risk Matrix

Threat	Likelihood	Impact	Mitigation	Residual Risk
51% attack	Very Low	Critical	Bitcoin PoW	Negligible
UTXO pruning	None	N/A	Consensus-critical storage	None
Indexer bug	Low	Medium	Multi-indexer consensus	Low
Malicious indexer	Medium	Low	User verification	Very Low
Front-running	Medium	Low	Privacy tools	Medium
Ticker squatting	High	Low	Social consensus	Low
Replay attack	Very Low	Low	Chain consensus	Very Low

8.5.3 Security Recommendations

For Users:
1. **Verify balances across multiple indexers** (stampchain.io, OpenStamps)
2. **Use reputable wallets** with established track record
3. **Check deploy metadata** (block height, deployer address) before transacting
4. **Run own indexer** for maximum trustlessness (advanced users)

For Developers:
1. **Implement multi-indexer queries** in applications
2. **Display divergence warnings** if indexers disagree
3. **Validate consensus checkpoints** during indexer sync
4. **Contribute to test suites** for edge case coverage

For Indexer Operators:
1. **Connect to diverse Bitcoin nodes** (prevent eclipse attacks)
2. **Verify consensus checkpoints** at milestone blocks
3. **Publish state hashes** for community verification
4. **Run comprehensive test suites** before deploying updates

8.6 Comparison with Other Protocols

8.6.1 Bitcoin Stamps vs Ordinals

Security Property	Bitcoin Stamps	Ordinals (Inscriptions)
Data permanence	Guaranteed (UTXO set)	⚠ Dependent (witness pruning)
Consensus enforcement	× Indexer-based	× Indexer-based
Pruning risk	None	⚠ Possible (witness data)
51% attack protection	Full Bitcoin PoW	Full Bitcoin PoW
Archival dependency	None (full nodes sufficient)	⚠ Requires archival nodes
Long-term guarantee	As long as Bitcoin exists	⚠ Depends on node policies

8.6.2 Bitcoin Stamps vs Counterparty

Security Property	Bitcoin Stamps	Counterparty
Data storage	UTXO set (multisig/P2WSH)	⚠ OP_RETURN (80 bytes, prunable)
Asset model	Account-based (inherited)	Account-based
Protocol maturity	⚠ Young (est. 2023)	Mature (est. 2014)
Indexer diversity	⚠ Limited implementations	Multiple implementations
Permanence guarantee	UTXO-based	⚠ OP_RETURN (smaller, prunable)

Key Difference: Counterparty uses 80-byte OP_RETURN outputs (provably unspendable, smaller data). Bitcoin Stamps use multisig/P2WSH for larger data and stronger permanence guarantees.

8.7 Future Security Considerations

8.7.1 Quantum Computing Threat

Threat: Quantum computers (Shor's algorithm) can break ECDSA signatures, potentially allowing theft of funds from known public keys.

Impact on Stamps:
- Stamp data permanence unaffected (data is public, not secret)
- UTXO spending risk if quantum attacker derives private keys
- Indexer validation logic unaffected (no cryptographic secrets)

Mitigation:
- Use burn addresses (no private key exists → quantum-proof)
- Future stamps may use quantum-resistant signatures (post-quantum cryptography)
- Bitcoin-level mitigation (soft fork to quantum-resistant signatures) protects all stamps

8.7.2 Bitcoin Protocol Changes

Threat: Future Bitcoin soft/hard forks may affect stamp permanence guarantees.

Potential Risks:
- UTXO set pruning mechanisms (BIP proposal: stateless validation)
- Changes to multisig or P2WSH validation rules
- Block size reductions affecting stamp relay

Mitigation:
- Community monitoring of Bitcoin Core development
- Participation in BIP discussions affecting data storage
- Fork contingency plans (maintain support for longest PoW chain)

8.7.3 Regulatory Challenges

Threat: Jurisdictions may ban stamp creation or indexing.

Impact:
- Stamp data remains on-chain (cannot be removed by regulation)
- Indexers may shut down in restricted jurisdictions
- Wallets may delist stamp functionality

Mitigation:
- Geographic indexer distribution (censorship-resistant)
- Open-source code enables permissionless operation
- Tor/VPN access to indexers in permissive jurisdictions
- Decentralized indexer networks (future research)

References:
- [Bitcoin Security Model](https://en.bitcoin.it/wiki/Weaknesses)
- [51% Attack Cost Analysis](https://www.crypto51.app/)
- [UTXO Set Research](https://research.mempool.space/utxo-set-report/)

- [Counterparty Security Model](https://counterparty.io/docs/protocol_specification/)
- [Ordinals vs Stamps Permanence Debate](https://bitcoinmagazine.com/technical/bitcoin-stamps-vs-ordinals-permanence)

Next: [Future Directions →](./future.md)
Previous: [← Implementation](./implementation.md)

---

9. Future Directions

9.1 Roadmap Overview

Bitcoin Stamps protocol evolution focuses on three strategic pillars:

1. **DeFi Primitives**: Conditional transfers, escrows, atomic swaps
2. **Privacy Enhancements**: Confidential amounts, stealth addresses, zero-knowledge proofs
3. **Cross-Chain Bridges**: Layer 2 integration, sidechain interoperability

This roadmap prioritizes **backward compatibility**, **security-first design**, and **community-driven governance** through the SIP process.

9.1.1 Timeline (2026-2028)

```
2026 Q2: SIP-0005 Binary Data Format
  ↓ (40-60% cost reduction for stamp creation)

2026 Q3: SIP-0003 Cross-Chain Bridges (testnet)
  ↓ (Lightning Network, Liquid Network integration)

2027 Q1: SIP-0001 Conditional Transfers
  ↓ (Escrows, time-locked transfers, atomic swaps)

2027 Q2-Q4: SIP-0004 Privacy Enhancements (phased rollout)
  ↓ (Confidential amounts → Stealth addresses → Full privacy)
```

2028: Advanced research (zk-SNARKs, DLC integration, rollups)
```

### 9.1.2 Design Principles for Future Development

**1. Preserve UTXO Permanence**: All enhancements must maintain consensus-critical data storage in Bitcoin UTXO set.

**2. Account-Based Compatibility**: New features should work with existing account-based balance model (no forced migration to UTXO-based tokens).

**3. Indexer Feasibility**: Protocol extensions must be implementable by community indexers without excessive computational burden.

**4. Activation Lead Time**: Consensus changes require **4+ weeks advance notice** (specified block height) for ecosystem coordination.

**5. Graceful Degradation**: Legacy indexers/wallets that don't implement new features should continue functioning for existing stamps.

## 9.2 Conditional Transfers (SIP-0001)

### 9.2.1 Motivation

Current SRC-20 transfers are **immediate and unconditional**—once transaction confirms, recipient owns tokens. Many DeFi use cases require **programmable conditions**:

- **Escrows**: Transfer completes only if third-party oracle approves
- **Time-locks**: Tokens released at specific block height
- **Atomic swaps**: Cross-asset exchange settles simultaneously or reverts
- **Vesting schedules**: Gradual token unlock over time

### 9.2.2 Technical Design

**Transaction Format**:
```json
{
  "p": "src-20",
  "op": "conditional_transfer",
  "tick": "KEVIN",
  "amt": "1000",
  "to": "bc1q...recipient",
  "conditions": {
    "type": "timelock",
    "unlock_height": 950000

```
  }
}
```

Indexer Validation Logic:
```python
def process_conditional_transfer(tx, parsed, block_height):
    # Create pending transfer (not yet credited to recipient)
    pending_transfers[tx.txid] = {
        'from': tx.sender_address,
        'to': parsed['to'],
        'asset': parsed['tick'],
        'amount': parsed['amt'],
        'conditions': parsed['conditions'],
        'status': 'pending'
    }

    # Lock tokens in sender account (prevent double-spend)
    locked_balances[tx.sender_address][parsed['tick']] += parsed['amt']

def evaluate_pending_transfers(block_height):
    """Called every block to check if conditions are met"""
    for txid, transfer in pending_transfers.items():
        if check_conditions(transfer['conditions'], block_height):
            # Conditions satisfied - execute transfer
            balances[transfer['from']][transfer['asset']] -= transfer['amount']
            balances[transfer['to']][transfer['asset']] += transfer['amount']
            locked_balances[transfer['from']][transfer['asset']] -= transfer['amount']
            transfer['status'] = 'completed'
```

9.2.3 Condition Types

1. Time-Lock:
```json
{
  "type": "timelock",
  "unlock_height": 950000
}
```
Tokens released when blockchain reaches block 950,000.

2. Oracle Signature:
```json
{
  "type": "oracle",
  "oracle_pubkey": "02a3b5c7...",
  "required_message": "DELIVERY_CONFIRMED",
  "signature": null  // Provided later by oracle
}
```
Tokens released when oracle signs attestation. Use case: Escrow for physical goods delivery.

3. Multi-Signature Threshold:

```json
{
  "type": "multisig",
  "required_signatures": 2,
  "authorized_pubkeys": [
    "02a3b5...",  // Buyer
    "03d7e9...",  // Seller
    "04f1a2..."   // Arbitrator
  ]
}
```
Tokens released when 2-of-3 parties sign approval. Use case: Dispute resolution escrow.

4. Atomic Swap:
```json
{
  "type": "atomic_swap",
  "counterparty_tx": "txid_of_opposite_transfer",
  "timeout_height": 950100  // Revert if swap incomplete
}
```
Tokens transfer only if counterparty transaction also completes (cross-asset swap).

9.2.4 Security Considerations

Oracle Trust: Introduces third-party dependency. Mitigation:
- Multi-oracle schemes (M-of-N oracle consensus)
- Time-limited oracle authority (fallback to refund after timeout)
- Bonded oracles (stake tokens as collateral against misbehavior)

Griefing Attacks: Malicious sender creates conditional transfer but never fulfills condition, locking recipient's expectation.

Mitigation:
- Timeout clauses (auto-revert after X blocks)
- Sender reputation systems
- Require sender to lock tokens (cost to griefing)

Indexer Complexity: Tracking pending transfers increases state size and validation complexity.

Mitigation:
- Cap pending transfer lifetime (auto-expire after 4,032 blocks / ~1 month)
- Pruning of expired pending transfers
- Efficient database indexing for pending state

9.2.5 Use Cases Enabled

Decentralized Exchange (DEX):
```python
```

Alice wants to swap 1000 KEVIN for 500 STAMP

```
alice_tx = {
   "op": "conditional_transfer",
   "tick": "KEVIN",
   "amt": "1000",
   "to": "bc1q...bob",
   "conditions": {
      "type": "atomic_swap",
      "counterparty_tx": bob_tx.txid  # Bob's STAMP transfer
   }
}
```

Bob submits counterparty transaction

```
bob_tx = {
   "op": "conditional_transfer",
   "tick": "STAMP",
   "amt": "500",
   "to": "bc1q...alice",
   "conditions": {
      "type": "atomic_swap",
      "counterparty_tx": alice_tx.txid
   }
}
```

Both transactions confirm → indexer verifies mutual reference → swap executes

If only one confirms → timeout triggers revert → no party loses funds

`

Crowdfunding:
` python

Project raises 1M KEVIN tokens by block 960,000

Contributors send conditional transfers to project address

```
contribution = {
  "op": "conditional_transfer",
  "tick": "KEVIN",
  "amt": "10000",
  "to": "bc1q...project",
  "conditions": {
    "type": "threshold",
    "total_required": "1000000",
    "deadline_height": 960000
  }
}
```

At block 960,000:

If total contributions >= 1M KEVIN → all transfers execute (funding success)

If total < 1M KEVIN → all transfers revert (refund contributors)

```

**Vesting Schedule**:
```python

## Employee receives 12,000 KEVIN tokens vesting over 1 year (52,560 blocks)

## Monthly unlocks: 1,000 KEVIN per 4,380 blocks

```
for month in range(12):
  vesting_transfer = {
    "op": "conditional_transfer",
    "tick": "KEVIN",
    "amt": "1000",
    "to": "bc1q...employee",
    "conditions": {
      "type": "timelock",
      "unlock_height": current_height + (month + 1) * 4380
    }
  }
```

```

## 9.3 Privacy Enhancements (SIP-0004)

### 9.3.1 Privacy Challenges

**Current Model**: SRC-20 balances are **publicly queryable** via indexers. Anyone can:
- View all address balances for any asset
- Track transfer history between addresses
- Analyze holding patterns and whale movements

**Use Cases Requiring Privacy**:
- Corporate treasury management (competitor analysis risk)
- High-net-worth individuals (security/safety concerns)
- Confidential business transactions (trade secret protection)

### 9.3.2 Phased Privacy Rollout

**Phase 1: Confidential Amounts (2027 Q2)**

**Mechanism**: Pedersen commitments hide transfer amounts while allowing indexer verification.

```python

# Sender creates commitment

commitment = amount * G + blinding_factor * H  # Elliptic curve points

# Transfer transaction

{
  "op": "transfer",
  "tick": "KEVIN",
  "amt_commitment": "0x3a7f...",  # Commitment (public)
  "to": "bc1q...recipient",
  "range_proof": "0x9e2c..."  # Proves 0 < amount < max_supply
}

# Indexer validation
```
```

```
verify(commitment_in - commitment_out == 0)  # Balance preserved
verify(range_proof)  # No negative amounts
```

Amount remains hidden from public queries

```

```

Benefits:
- Transfer amounts private (only sender/recipient know)
- Balances remain confidential
- Indexer can verify validity without knowing amounts

Tradeoffs:
- Proof size: +1-2 KB per transfer (higher fees)
- Validation cost: 10-50x slower indexer sync
- Wallet complexity: Requires cryptographic libraries

Phase 2: Stealth Addresses (2027 Q3)

Mechanism: One-time addresses prevent address linkage.

```python

# Recipient publishes stealth address metadata

stealth_meta = {
    "view_key": "02a3b5...",  # Public view key
    "spend_key": "03d7e9..."  # Public spend key
}

# Sender generates one-time address

ephemeral_key = random_scalar()
one_time_address = derive_stealth_address(
    stealth_meta['view_key'],
    stealth_meta['spend_key'],
    ephemeral_key
)

# Transfer to one-time address

{
  "op": "transfer",
  "tick": "KEVIN",
  "amt": "1000",
  "to": one_time_address,  # Unlinked to recipient's known addresses
  "ephemeral_pubkey": ephemeral_key * G  # Allows recipient to detect
}
```

Recipient scans blockchain

for tx in new_transactions:
  if can_derive_private_key(tx.ephemeral_pubkey, my_view_key, my_spend_key):
    # This transfer is for me
    claim_funds(tx, derived_private_key)
`

Benefits:
- Breaks address linkage (sender doesn't know recipient's other addresses)
- Recipient can use single public identity for all transfers
- Third parties cannot track recipient activity

Tradeoffs:
- Recipient must scan all transactions (wallet sync overhead)
- Increased transaction size (~64 bytes for ephemeral key)
- Incompatible with light clients (full blockchain scan required)

Phase 3: Full Privacy by Default (2027 Q4)

Mechanism: Combine confidential amounts + stealth addresses + optional anonymity set mixing.

` python

Privacy-preserving transfer (all features enabled)

{
 "op": "transfer",
 "tick": "KEVIN",
 "amt_commitment": "0x3a7f...", # Amount hidden
 "to": one_time_address, # Address unlinked
 "range_proof": "0x9e2c...", # Validity proof
 "ephemeral_pubkey": "0x7b4d...", # Recipient detection key
 "decoy_inputs": ["0xa3c5...", "0xf8e2..."] # Mix with other txs (optional)
}
`

Result: Privacy level comparable to Monero, but on Bitcoin via stamps.

Opt-Out Mechanism: Users can choose transparent transfers for compliance/auditability:
` json
{
 "op": "transfer",
 "tick": "KEVIN",
 "amt": "1000", // Plain amount (no commitment)
 "to": "bc1q...", // Regular address (no stealth)
 "privacy": false // Explicitly opt out
}
`

9.3.3 Zero-Knowledge Proofs (Research)

Future Direction: zk-SNARKs for succinct privacy.

Potential Design:
```python
# Zero-knowledge transfer proof
zk_proof = prove(
    "I own X KEVIN tokens AND X > 1000 AND I am sending 1000 to recipient"
)

# Transfer transaction (200-500 bytes regardless of complexity)
{
  "op": "transfer",
  "tick": "KEVIN",
  "zk_proof": "0x3f7a...",  # Succinct proof
  "to_commitment": "0xe9c2...",  # Recipient identity hidden
  "nullifier": "0x5d8b..."  // Prevent double-spend (unique per tx)
}

# Indexer verification
verify_zk_proof(zk_proof)  # Fast verification (~1ms)
check_nullifier_not_spent(nullifier)

# No amount or sender knowledge required
```

Benefits:
- Strongest privacy (sender, recipient, amount all hidden)
- Compact proofs (200-500 bytes)
- Fast verification (1-10ms per proof)

Challenges:
- Trusted setup (or STARK alternative with larger proofs)
- Proof generation cost (30 seconds to 2 minutes per transfer)
- Wallet implementation complexity (circuit design, prover software)

Status: Exploratory research. No formal SIP yet. Monitoring advances in Bitcoin-compatible zk-proof systems (e.g., BitVM, zkCoins).

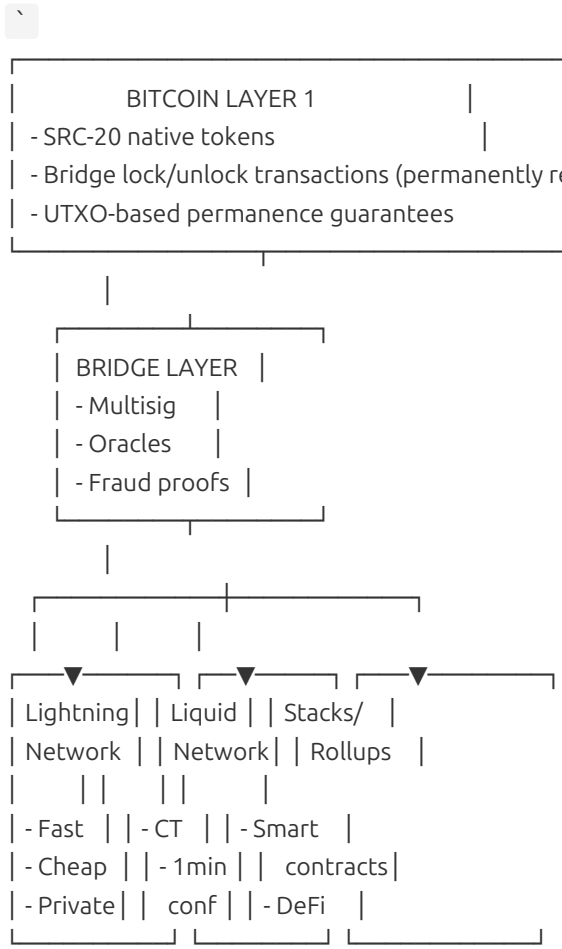
9.4 Cross-Chain Bridges (SIP-0003)

9.4.1 Motivation

Problem: Bitcoin Layer 1 has limited throughput (~7 transactions per second). Users need:
- Fast transfers (Lightning Network: 1000s TPS)
- Low fees (Layer 2: fractions of a cent)
- Scalability (sidechains: application-specific throughput)

Solution: Bridge SRC-20 tokens to Layer 2 protocols while maintaining Layer 1 permanence for bridge records.

9.4.2 Architecture



9.4.3 Bridge Operations

Lock (L1 → L2):
```python
```

User sends SRC-20 tokens to bridge address on Bitcoin L1

```
lock_tx = {
   "op": "transfer",
   "tick": "KEVIN",
   "amt": "1000",
   "to": "bc1q...bridge_address",  # Multisig controlled by bridge operators
   "bridge_metadata": {
      "target_chain": "lightning",
      "recipient_pubkey": "0x3a7f..."
   }
}
```

Bridge operators verify lock transaction

```
verify_lock(lock_tx)
```

L2 mints wrapped token

```
lightning_mint = {
   "asset": "KEVIN.BTC",  # Wrapped KEVIN
   "amount": 1000,
   "recipient": "0x3a7f...",
   "backing_tx": lock_tx.txid  # Reference to L1 lock
}
```
`

Unlock (L2 → L1):
` python

User burns wrapped token on L2

```
lightning_burn = {
   "asset": "KEVIN.BTC",
   "amount": 1000,
   "burn_proof": "0xe9c2..."  # Cryptographic proof of burn
}
```

Bridge operators verify burn proof

```
verify_burn_proof(lightning_burn)
```

Bridge creates L1 unlock transaction

```
unlock_tx = {
    "op": "transfer",
    "tick": "KEVIN",
    "amt": "1000",
    "from": "bc1q...bridge_address",  # Bridge multisig
    "to": "bc1q...user_address",
    "bridge_metadata": {
        "source_chain": "lightning",
        "burn_proof": lightning_burn.burn_proof
    }
}
```

9.4.4 Security Model

Federated Multisig:
```python
# Bridge controlled by M-of-N multisig (e.g., 7-of-10)

bridge_operators = [
    "stampchain.io",
    "openstamps.io",
    "bitcoin_magazine",
    "btc_dev_1",
    "btc_dev_2",
    # ... 10 total operators
]

# Unlock requires 7 signatures
required_signatures = 7
```

Fraud Proofs:
```python
# Anyone can challenge invalid unlock

def submit_fraud_proof(unlock_tx, proof):
    """
    Proves that unlock_tx does not have valid burn proof on L2
    If verified, slashes bridge operators' bond
    """
    if verify_fraud_proof(unlock_tx, proof):
        # Slash operators' bonds
        slash_bonds(unlock_tx.signers, amount=unlock_tx.amount * 1.1)
```

```
    # Revert unlock (bridge must refund)
    revert_unlock(unlock_tx)
    # Reward fraud proof submitter
    reward(proof.submitter, unlock_tx.amount * 0.1)
```

Challenge period: 1 week (1,008 blocks)

Users can withdraw after challenge period expires with no fraud proofs

```

```

Bond Requirements:
```python

# Bridge operators must post bond (collateral)

operator_bond = total_bridged_value * 1.2  # 120% collateralization

# Example:

total_locked_kevin = 10_000_000  # 10M KEVIN locked in bridge
kevin_price = 0.01  # $0.01 per KEVIN
total_value = 10_000_000 * 0.01 = $100,000
required_bond = $100,000 * 1.2 = $120,000 (in BTC or stablecoin)
```

9.4.5 Target Layer 2 Protocols

Lightning Network:
- **Benefits**: Instant transfers, near-zero fees, privacy
- **Challenges**: Channel liquidity management, routing complexity
- **Use Case**: Micropayments, fast retail transactions

Liquid Network:
- **Benefits**: 1-minute confirmations, confidential transactions, federated security
- **Challenges**: Federated trust assumptions (15-member federation)
- **Use Case**: Exchange settlements, trader liquidity

Stacks:
- **Benefits**: Smart contracts (Clarity language), direct Bitcoin finality
- **Challenges**: Microblock timing, contract complexity
- **Use Case**: DeFi applications (lending, DEXs, derivatives)

Future Rollups (BitVM, Sovereign SDK):
- **Benefits**: High throughput, fraud proofs, L1 data availability
- **Challenges**: Early-stage technology, unproven security
- **Use Case**: Scalable DeFi, gaming, high-frequency trading

9.4.6 Phased Deployment

Phase 1: Testnet (Q3 2026)
- Deploy bridge on Bitcoin testnet + Lightning testnet
- Community testing, bug bounties
- Security audits by independent firms

Phase 2: Limited Mainnet (Q4 2026)
- Launch with $100K bridge capacity limit
- Single asset (KEVIN) for initial testing
- 7-of-10 multisig, $120K bond requirement

Phase 3: Full Launch (Q1 2027)
- Remove capacity limits
- Support all SRC-20 tokens
- Add Liquid Network bridge
- Increase to 11-of-15 multisig for redundancy

Phase 4: Trustless Bridge (2028+)
- Research BitVM-based fraud proofs (no multisig)
- Explore zero-knowledge bridge validation
- Potential for fully trustless cross-chain transfers

9.5 Additional Research Areas

9.5.1 DLC (Discreet Log Contract) Integration

Concept: Stamp ownership can be conditional on DLC oracle outcomes.

Use Case Example:
```python
# Alice and Bob create DLC betting on BTC price
# Winner receives 1000 KEVIN tokens
```

```
dlc_conditional_transfer = {
  "op": "conditional_transfer",
  "tick": "KEVIN",
  "amt": "1000",
  "from": "escrow_address",
  "conditions": {
    "type": "dlc_oracle",
    "oracle_pubkey": "0x7a3c...",
    "event": "btc_price_2027_01_01",
    "payout_curve": {
      "< 50000": {"to": "bc1q...alice", "amt": "1000"},
      ">= 50000": {"to": "bc1q...bob", "amt": "1000"}
    }
  }
}
```

Benefits:
- Trustless betting markets
- Prediction markets using SRC-20 tokens
- Derivatives (options, futures) settled in stamps

Status: Conceptual. Requires DLC oracle standardization for stamp indexers.

9.5.2 Recursive Stamps v2

Enhancement: Stamps can reference external Bitcoin data (Taproot scripts, DLC outcomes).

Example:
```json
{
  "stamp_id": 123456,
  "type": "recursive",
  "base_image": "stamp:98765",  // Reference to another stamp
  "dynamic_layers": {
    "taproot_script": "bc1p...taproot_address",  // Execute Taproot script
    "render_function": "stamp:55555"  // JavaScript library stamp
  }
}
```

Use Cases:
- Stamps that change appearance based on Bitcoin block data
- NFTs with on-chain game state (DLC-driven evolution)
- Generative art responsive to network activity

Status: Early research. Community feedback sought on feasibility and demand.

9.5.3 Decentralized Indexer Network

Problem: Current indexers are centralized services (stampchain.io, OpenStamps). If all shut down, new users

cannot query balances until launching own indexer.

Solution: Peer-to-peer indexer network with incentivized data serving.

Architecture:
```
┌─────────────────────────────────────────────────────────┐
│      DECENTRALIZED INDEXER NETWORK                      │
├─────────────────────────────────────────────────────────┤
│                                                         │
│  Indexer Node A ↔ Indexer Node B ↔ Node C               │
│     ↕              ↕              ↕                     │
│  Validates blocks   Serves queries    Earns fees        │
│                                                         │
│  - Consensus via state hash attestations                │
│  - Incentives: Users pay sats for API queries           │
│  - Redundancy: 100s of nodes worldwide                  │
│  - Discovery: DHT-based peer finding                    │
└─────────────────────────────────────────────────────────┘
```

Incentive Mechanism:
```python
# User pays Lightning Network micropayment for query
user_query = {
    "address": "bc1q...xyz",
    "asset": "KEVIN",
    "payment": "10 sats"  // Pay 10 sats for balance query
}

# Indexer node serves query, receives payment
response = {
    "balance": "1000 KEVIN",
    "proof": merkle_proof,  // Cryptographic proof of correctness
    "payment_receipt": lightning_invoice
}

# User verifies proof (trustless query)
verify_merkle_proof(response.proof, consensus_checkpoint_hash)
```

Status: Conceptual research. Requires Lightning Network infrastructure and standardized proof formats.

9.5.4 SRC-20 Token Derivatives

Goal: Enable financial derivatives (options, futures, perpetuals) using SRC-20 tokens.

Example - Call Option:
```python

# Alice buys call option: Right to buy 1000 KEVIN at $0.02 by block 1,000,000

option = {
    "type": "call_option",
    "underlying": "KEVIN",
    "strike_price": "0.02 USD",
    "quantity": "1000",
    "expiry_height": 1000000,
    "premium": "50 KEVIN",  # Alice pays Bob 50 KEVIN upfront
    "seller": "bc1q...bob",
    "buyer": "bc1q...alice"
}

# At block 1,000,000:

# If KEVIN > $0.02 → Alice exercises, receives 1000 KEVIN at $0.02 (profit)

# If KEVIN < $0.02 → Alice doesn't exercise, loses 50 KEVIN premium
```

Implementation: Requires price oracles + conditional transfers (SIP-0001).

Status: Dependent on SIP-0001 activation. Community interest high for DeFi primitives.

9.6 Ecosystem Development

9.6.1 Wallet Integration

Current Wallets:
- Stampchain.io web wallet (official)
- Emblem Vault (hardware wallet integration)
- Hiro Wallet (Stacks ecosystem)

Future Goals:
- Native Bitcoin wallet support (Sparrow, Electrum, Blue Wallet)
- Hardware wallet firmware (Ledger, Trezor, Coldcard)
- Mobile-first wallets (iOS, Android)

Technical Requirements:
- BIP-44 derivation paths for stamp accounts
- PSBT (Partially Signed Bitcoin Transactions) support for multisig
- Indexer API standardization (consistent query format)

9.6.2 Marketplace Development

Current Marketplaces:
- Stampchain.io marketplace (stamp trading)
- Scarce.city (auction-based sales)
- Emblem Vault (cross-chain trading)

Future Enhancements:
- Decentralized order books (on-chain limit orders)
- Royalty enforcement (SIP proposal: optional creator fees)
- Batch trading (single transaction for multiple stamps)
- Cross-chain marketplaces (trade stamps for ETH/SOL NFTs via bridges)

9.6.3 Developer Tooling

Current Tools:
- Python SDK (stampchain-io/btc_stamps)
- Rust parser libraries
- JavaScript API clients

Needed Tooling:
- **TypeScript SDK**: Full-featured wallet integration library
- **GraphQL API**: Flexible querying for complex applications
- **Test frameworks**: Local indexer for development/testing
- **Documentation portal**: API references, tutorials, example projects

9.6.4 Education and Adoption

Community Initiatives:
- Weekly developer calls (protocol updates, Q&A)
- Hackathons (bounties for stamp-based applications)
- Educational content (video tutorials, written guides)

- University partnerships (blockchain courses featuring stamps)

Marketing Focus:
- Permanence guarantee (vs Ordinals, IPFS NFTs)
- Fair launch ethos (no VC funding, community-driven)
- Bitcoin-native security (inherits PoW, no alt-chain risk)

9.7 Long-Term Vision (2030+)

9.7.1 Bitcoin as Permanent Data Layer

Vision: Bitcoin Stamps establishes Bitcoin as the **canonical permanent data storage layer** for high-value digital artifacts.

Target Use Cases:
- **Legal records**: Contracts, deeds, certifications (immutable proof)
- **Identity systems**: Decentralized IDs (SRC-101 extensions)
- **Digital art archives**: Museum-quality NFTs (cultural preservation)
- **Scientific data**: Research publications, datasets (censorship-resistant)

Competitive Advantage: Only Bitcoin offers:
- 15+ years proven security (longest PoW history)
- Global node distribution (most decentralized network)
- Economic finality (highest attack cost)
- UTXO permanence (stamps guarantee)

9.7.2 DeFi on Bitcoin

Vision: Bitcoin Stamps enables **Bitcoin-native DeFi** without wrapping or bridging to other chains.

DeFi Primitives via Stamps:
- **Lending protocols**: Collateralized loans using SRC-20 tokens
- **Decentralized exchanges**: On-chain order books + atomic swaps
- **Stablecoins**: Algorithmic or collateralized stablecoins (SRC-20 format)
- **Yield farming**: Liquidity provision rewards via conditional transfers
- **Derivatives**: Options, futures, perpetuals (DLC integration)

Advantages over Ethereum DeFi:
- **Bitcoin security**: Strongest PoW consensus
- **No smart contract risk**: Indexer validation is deterministic (no reentrancy, overflow bugs)
- **Lower attack surface**: Simpler validation logic than Turing-complete contracts
- **Bitcoin-native UX**: No need to bridge assets to other chains

Challenges:
- Indexer trust (vs Ethereum on-chain execution)
- Limited programmability (vs Solidity flexibility)
- User education (Bitcoin culture skeptical of DeFi)

9.7.3 Integration with Future Bitcoin Upgrades

OP_CAT / Covenant Opcodes:
- If Bitcoin enables covenants, stamps could use on-chain validation
- Reduces indexer trust assumptions (rules enforced by Bitcoin Script)
- Example: Time-locked transfers enforced at consensus layer

Drivechains (BIP 300/301):
- Stamps could bridge to Bitcoin sidechains with two-way peg
- Enables high-throughput stamp transfers without federated trust
- Full Bitcoin security for sidechain assets

BitVM:
- Arbitrary computation verification via fraud proofs
- Could enable trustless bridges, zk-proof validation on Bitcoin
- Stamps would inherit BitVM security model (fraud-proof based)

Quantum Resistance:
- Bitcoin will likely adopt quantum-resistant signatures (post-quantum cryptography)
- Stamps automatically inherit protection (piggyback on Bitcoin upgrade)
- Future stamps may use quantum-proof cryptography for privacy features

9.8 Conclusion

Bitcoin Stamps protocol is positioned for sustainable long-term growth through:

1. **Technological Innovation**: Conditional transfers, privacy, bridges (2026-2028)
2. **Community Governance**: SIP process ensures decentralized decision-making
3. **Backward Compatibility**: New features don't break existing stamps
4. **Bitcoin Alignment**: Leverage Bitcoin's security, permanence, and decentralization

Guiding Philosophy: *"Build for permanence, optimize for accessibility, preserve decentralization."*

The future of Bitcoin Stamps is not predetermined—it will be shaped by community contributions, SIP proposals, and ecosystem builders. All are invited to participate in shaping the protocol's evolution.

Get Involved:
- **GitHub**: https://github.com/stampchain-io/btc_stamps (contribute code, submit SIPs)
- **Discord**: https://discord.gg/stampchain (community discussions)
- **Twitter**: @stampchain (protocol updates)
- **Developer Docs**: https://docs.stampchain.io (API references, tutorials)

References:
- [Bitcoin Stamps Roadmap](https://github.com/stampchain-io/btc_stamps/blob/main/ROADMAP.md)
- [Active SIPs](https://github.com/stampchain-io/btc_stamps/issues?q=label%3ASIP)
- [Lightning Network Specification](https://github.com/lightning/bolts)
- [BitVM Whitepaper](https://bitvm.org/bitvm.pdf)
- [Discreet Log Contracts](https://adiabat.github.io/dlc.pdf)

Previous: [← Security Analysis](./security.md)
Table of Contents: [↑ Whitepaper Index](./index.md)
