Bitcoin Lending Protocol

Technical Specification Document

Phase 2: Bitcoin-Native Custody with Threshold Signatures

Version: 1.0
Date: January 2026
Status: Draft for Review
Author: Jamie (Project Lead)
Builds On: Phase 1 (Stacks/sBTC Implementation)

Table of Contents

---

1. Architecture Overview

1.1 System Components

```
                    Frontend (Next.js)
  - Native BTC deposit UI
  - Validator transparency dashboard
  - Bitcoin transaction monitoring
  - Threshold signature progress display


                   Stacks.js + Bitcoin RPC


            Clarity Smart Contracts (Stacks)
```

```
loan-protocol-v2.clar
- Bitcoin deposit monitoring
- Validator action requests
- Cross-chain state sync
- Auction logic (unchanged)
- Multi-stablecoin (unchanged)
```

```
                Validator Consensus Protocol

            Validator Network (15 nodes)

Validator     Validator     Validator
    1             2             3        ... (15)

 - BTC         - BTC         - BTC
 - Stacks      - Stacks      - Stacks
 - FROST       - FROST       - FROST
 - HSM         - HSM         - HSM

            libp2p gossip

                 Threshold Signatures (10-of-15)

                Bitcoin Blockchain

  Threshold Multisig Addresses (Taproot)
  - One address per loan
  - Public key from DKG
  - Controlled by validator network
  - No single point of failure
```

1.2 Phase 2 Loan Flow (Native Bitcoin)

Step 1: Loan Creation with Bitcoin Address Generation

```
User → Frontend
  1. User creates loan request (amount, duration, stablecoin)
  2. Frontend calls Stacks contract

Frontend → Stacks Contract
  3. create-loan-auction-v2(loan-params)
  4. Contract generates unique loan-id
  5. Contract emits "bitcoin-address-needed" event

Validators Monitor Stacks
  6. All validators see "bitcoin-address-needed" event
  7. Validators derive deterministic Bitcoin address:
     - Use loan-id as derivation path
     - Combined public key P = P_1 + P_2 + ... + P_15
     - Taproot address from combined key
  8. Validators submit Bitcoin address to Stacks contract

Stacks Contract
  9. Contract receives Bitcoin address from validators
  10. Contract stores: loan → bitcoin_address mapping
  11. Contract emits "deposit-ready" event with address

Frontend
  12. Display Bitcoin address (QR + text) to user
  13. User sends BTC from their wallet
  14. Frontend monitors Bitcoin mempool
```

Step 2: Bitcoin Deposit Confirmation

```
Bitcoin Network
  1. User's BTC transaction broadcasts
  2. Transaction appears in mempool
  3. Miners include in block
  4. Transaction gets confirmations (target: 3+)

Validators Monitor Bitcoin
  5. All validators watch threshold addresses
  6. Validators see deposit transaction
  7. Wait for 3 confirmations

Validators → Stacks
  8. Validators submit Bitcoin proof to Stacks:
     - Transaction ID
```

```
     - Block height
     - Merkle proof
     - Output amount
  9. Multiple validators submit (redundancy)

Stacks Contract
  10. Contract receives Bitcoin proofs
  11. Contract verifies:
      - Transaction is real (merkle proof)
      - Amount matches loan requirement
      - Confirmations >= 3
  12. Contract activates loan
  13. Auction begins on Stacks
```

Step 3: Auction & Loan Finalization (Same as Phase 1)

```
Stacks Layer
  - Lenders bid in stablecoin (competitive bidding)
  - Lowest bid wins
  - Winner transfers stablecoin on Stacks
  - Loan status: "active"

No validator action needed yet - BTC stays locked
```

Step 4: Bitcoin Release to Borrower

```
Stacks Contract
  1. Auction finalizes, winner determined
  2. Winner transfers stablecoin to borrower (on Stacks)
  3. Contract emits "release-collateral" action request:
     {
       loan-id: 42,
       action: "release",
       bitcoin-txid: "abc123...",
       recipient: "bc1q...", // Borrower's Bitcoin address
       amount: 150000000, // 1.5 BTC in satoshis
       fee: 50000 // Estimated fee
     }

Validators See Action Request
  4. All validators monitor Stacks for action requests
  5. Each validator independently verifies:
     - Stacks contract state is correct
     - Auction properly finalized
     - Winner paid stablecoin
     - Recipient address is valid
  6. If valid, validator creates partial signature
```

```
Threshold Signing
  7. Validator creates partial signature using key share
  8. Validator broadcasts partial signature to network
  9. Coordinator collects partial signatures
  10. Once 10+ received, coordinator aggregates
  11. Final Schnorr signature is valid for threshold address

Bitcoin Transaction
  12. Coordinator constructs Bitcoin transaction:
      Input: Threshold address UTXO
      Output: Borrower address (1.5 BTC - fee)
  13. Coordinator attaches aggregated signature
  14. Coordinator broadcasts to Bitcoin network
  15. Transaction confirms in Bitcoin block

Stacks Contract Update
  16. Validators submit Bitcoin confirmation proof
  17. Contract updates loan status
  18. Frontend shows BTC received by borrower
```

Step 5: Repayment & Collateral Return

```
Borrower Repays on Stacks
  1. Borrower transfers stablecoin to lender (Stacks)
  2. Contract verifies repayment
  3. Contract emits "release-collateral" action request
     (same process as Step 4)

Validators Release Collateral
  4. Threshold signing process (same as Step 4)
  5. BTC transfers from threshold address to borrower
  6. Transaction confirms on Bitcoin
  7. Loan complete, NFTs burned
```

Step 6: Default Scenario

```
If Borrower Doesn't Repay
  1. Loan maturity block reached on Stacks
  2. Lender calls claim-collateral on Stacks
  3. Contract emits "claim-collateral" action request
     (BTC to lender's address instead of borrower)

Validators Process Claim
  4. Threshold signing with lender as recipient
  5. BTC transfers to lender
  6. Loan status: "defaulted"
```

1.3 Key Differences from Phase 1

Aspect	Phase 1 (sBTC)	Phase 2 (Native BTC)
Collateral Asset	sBTC on Stacks	Actual Bitcoin on Bitcoin
Collateral Location	Stacks contract	Threshold multisig on Bitcoin
Deposit Process	Single Stacks transaction	Send BTC to address, wait confirmations
Custody	Stacks contract	Distributed validator network
Release Mechanism	Stacks transaction	Threshold signatures + Bitcoin transaction
Trust Model	sBTC peg operators	10-of-15 validator threshold
Confirmation Time	~10 min (Stacks)	~30 min (3 Bitcoin confirmations)
Fees	Stacks gas (~$1-5)	Bitcoin fees (~$5-50 depending on network)

---

2. Threshold Cryptography Specifications

2.1 Cryptographic Scheme Selection

Chosen Scheme: FROST (Flexible Round-Optimized Schnorr Threshold Signatures)

Why FROST? - Schnorr-based (Bitcoin Taproot compatible) - Non-interactive after DKG (2-round signing only) - Efficient aggregation - Provably secure - Production-ready implementations available

Alternative Considered: MuSig2 - Also Schnorr-based - Requires all participants online simultaneously - Less flexible than FROST for 10-of-15 threshold

Threshold Configuration: **10-of-15** - 15 total validators - 10 signatures required (67% threshold) - 5 validators can be offline - 6+ colluding validators needed to steal (unlikely)

2.2 Distributed Key Generation (DKG)

Purpose: Generate shared private key without any single party knowing it.

Process:

```
Round 0: Setup

- Coordinator announces DKG ceremony
```

```
- All 15 validators join
- Parameters: n=15, t=10 (threshold)
- Each validator generates random secret s_i

Round 1: Commitment Phase (VSS)

Each validator i:
  1. Creates secret polynomial: f_i(x) = s_i + a_1*x + ... + a_9*x^9
     (degree 9 polynomial for t=10)
  2. Computes commitments: C_ij = g^(a_j) for j=0..9
  3. Broadcasts commitments to all validators

Round 2: Share Distribution

Each validator i:
  1. Evaluates polynomial at each validator's ID:
     share_ij = f_i(j) for j=1..15
  2. Encrypts share_ij with validator j's public key
  3. Sends encrypted shares to each validator

Round 3: Share Verification

Each validator j:
  1. Receives shares from all other validators
  2. Decrypts own share from each validator i
  3. Verifies share against commitments:
     g^(share_ij) = C_i0 * C_i1^j * C_i2^(j^2) * ... * C_i9^(j^9)
  4. If verification fails, broadcast complaint
  5. If valid, accept share

Round 4: Complaint Handling

If complaints exist:
  1. Accused validator reveals shares to everyone
  2. All validators verify revealed shares
  3. If invalid, exclude malicious validator
  4. If valid, exclude complaining validator

Round 5: Key Derivation

Each validator j:
  1. Combines all received shares:
     secret_share_j = sum(share_ij for all i)
  2. Computes public key share:
     P_j = g^(secret_share_j)
```

```
All validators:
  1. Combine public key shares:
     P = P_1 + P_2 + ... + P_15
  2. P is the combined public key
  3. Bitcoin address derived from P (Taproot)

Result:

  Each validator has secret_share_j (kept private)
  Combined public key P is known to everyone
  No validator knows the combined private key
  Any 10 validators can create valid signatures
```

Security Properties: - **Soundness**: Valid shares pass verification - **Privacy**: No information about full private key revealed - **Correctness**: Combined key corresponds to valid threshold - **Robustness**: Handles up to 5 malicious validators

Implementation:

```rust
// Rust pseudocode using FROST library

use frost_secp256k1 as frost;

// Round 1: Generate commitments
let (secret_package, public_package) = frost::keys::dkg::part1(
    validator_id,
    15,  // total participants
    10,  // threshold
    &mut rng
)?;

// Round 2: Generate and send shares
let round2_packages = frost::keys::dkg::part2(
    secret_package,
    &round1_packages_from_all
)?;

// Round 3: Finalize key generation
let (key_package, public_key_package) = frost::keys::dkg::part3(
    &round2_packages_from_all,
    &round1_packages_from_all
)?;

// key_package contains this validator's secret share
// public_key_package contains combined public key
```

2.3 Threshold Signature Generation

Purpose: Create valid Bitcoin signature using 10-of-15 validators.

FROST Signing Process:

```
Phase 1: Nonce Commitment

Coordinator → All Validators:
  - Announces signing request
  - Message: Bitcoin transaction hash to sign
  - Nonce: Unique per signing session

Each Validator (independently):
  1. Generate random nonces: (d_i, e_i)
  2. Compute commitments: (D_i, E_i) = (g^d_i, g^e_i)
  3. Send (D_i, E_i) to coordinator

Coordinator:
  4. Collect commitments from all validators
  5. Wait for at least 10 commitments
  6. Publish list of participants (who will sign)
  7. Compute binding values for each participant

Phase 2: Signature Generation

Coordinator → Participants:
  - List of signing participants (10+)
  - All nonce commitments
  - Message to sign

Each Participating Validator:
  1. Compute Lagrange coefficient  _i
  2. Compute group commitment R
  3. Compute challenge c = H(R || P || m)
  4. Create partial signature:
     z_i = d_i + (e_i *  _i) + ( _i * secret_share_i * c)
  5. Send z_i to coordinator

Coordinator:
  6. Collect partial signatures (need 10+)
  7. Aggregate partial signatures:
     z = sum(z_i for all participants)
  8. Final signature: (R, z)
  9. Verify signature is valid for public key P
  10. If valid, use for Bitcoin transaction
```

```
Phase 3: Bitcoin Transaction

Coordinator:
  1. Construct Bitcoin transaction
  2. Attach signature (R, z) to transaction
  3. Broadcast to Bitcoin network
  4. Monitor for confirmation
```

Timing: - Phase 1: ~30 seconds (validators respond) - Phase 2: ~30 seconds (partial signatures) - Aggregation: <1 second - **Total**: ~1-2 minutes typically

Implementation:

```rust
// Round 1: Generate nonce commitments
let (nonce, commitments) = frost::round1::commit(
    &key_package,
    &mut rng
);

// Round 2: Create partial signature
let signature_share = frost::round2::sign(
    &signing_package,  // From coordinator
    &nonce,
    &key_package
)?;

// Coordinator aggregates
let group_signature = frost::aggregate(
    &signing_package,
    &signature_shares,  // From 10+ validators
    &public_key_package
)?;

// group_signature is a valid Schnorr signature
```

2.4 Address Derivation

Bitcoin Address Generation:

```
Given: Combined public key P from DKG

For Loan ID = N:
  1. Derive child key using BIP32-style derivation:
     P_N = P + H(P || N) * G

  2. Create Taproot output:
     - Internal key: P_N
     - Script tree: None (key-path only)
```

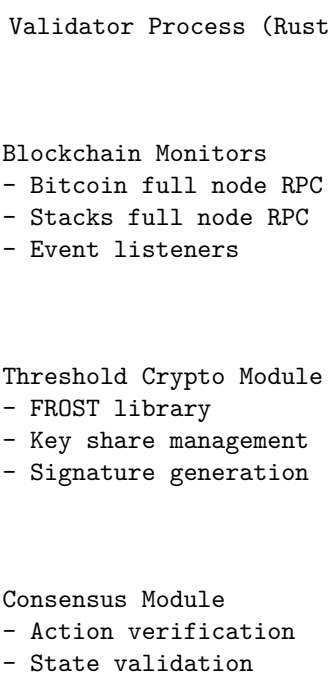
```
  3. Compute Taproot output key:
     Q = P_N + H("TapTweak" || P_N) * G

  4. Create Bech32m address:
     address = bech32m_encode("bc", 1, Q)

Result: bc1p[52 characters]

Example:
  Loan #1 → bc1p8xj2f7a9zmqwn3ktv8s6r4l2c5h9tp3m4d6x8y
  Loan #2 → bc1pq5m9tn2h7r3k8d4f6x9c2v5b8n3m7p4w6s9t2k

Each loan gets unique address, all controlled by same validator set
```

Address Properties: - **Unique per loan**: Deterministic but different - **Taproot**: Single signature appearance (privacy) - **Verifiable**: Anyone can verify address derivation - **Secure**: Requires 10-of-15 to spend

---

3. Validator Node Architecture

3.1 Validator Software Stack



```
- Conflict resolution


Network Module
- libp2p
- Gossip protocol
- Peer discovery


Storage
- HSM/secure enclave
- SQLite (state)
- Logs


Metrics & Monitoring
- Prometheus exporter
- Health checks
- Alerts




   Bitcoin      Stacks      Peer Validators
    Node         Node        (libp2p)
```

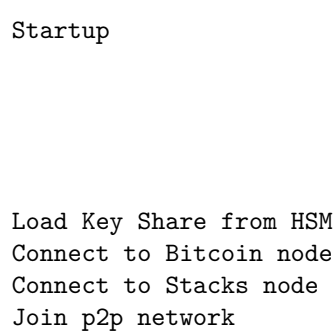
3.2 Validator Configuration

config.toml:

```toml
[validator]
id = 1  # Validator ID (1-15)
network = "mainnet"  # or "testnet"

[bitcoin]
rpc_url = "http://localhost:8332"
rpc_user = "bitcoin"
rpc_password = "secret"
confirmations_required = 3

[stacks]
rpc_url = "http://localhost:20443"
```

```toml
contract_address = "SP1234.loan-protocol-v2"

[threshold]
scheme = "frost"
total_validators = 15
threshold = 10

[key_storage]
type = "hsm"  # or "file" for testing
hsm_slot = 0
hsm_pin_env = "HSM_PIN"

[network]
p2p_listen = "0.0.0.0:9001"
bootstrap_peers = [
  "/ip4/validator1.example.com/tcp/9001/p2p/12D3KooW...",
  "/ip4/validator2.example.com/tcp/9001/p2p/12D3KooW...",
]

[monitoring]
metrics_port = 9090
health_check_port = 8080
log_level = "info"

[coordinator]
enabled = true  # Only one coordinator needed
coordinator_url = "http://coordinator.example.com:8000"
```

3.3 Validator State Machine



Syncing

Bitcoin: Synced?
Stacks: Synced? No

Yes

Ready

(Loop)

Monitor Events
- Bitcoin deposits
- Stacks action requests
- Peer messages

Event Detected?
Yes → Process
No → Wait

Verify Event
- Check Stacks contract
- Validate action
- Ensure no double-sign

Valid? No

```
Yes



Signing
- Generate nonce
- Create partial sig
- Broadcast to coordinator



Wait for Aggregation
(Coordinator combines)



Verify Final Signature
- Check aggregated sig
- Ensure transaction valid



   Back to Ready
```

3.4 Key Storage and Security

HSM Integration (YubiHSM 2):

```rust
use yubihsm::{Client, Connector, Credentials};

// Initialize HSM connection
let connector = Connector::usb(&Default::default())?;
let client = Client::open(connector, Credentials::default(), true)?;

// Store key share (one-time, during DKG)
fn store_key_share(
    client: &Client,
    key_share: &SecretShare
) -> Result<(), Error> {
    // Serialize key share
    let key_data = bincode::serialize(key_share)?;
```

```rust
    // Store in HSM (encrypted at rest)
    client.put_opaque_object(
        VALIDATOR_KEY_SLOT,
        "validator_key_share",
        &key_data,
        &[Capability::ExportableUnderWrap]
    )?;

    Ok(())
}

// Retrieve key share for signing
fn load_key_share(
    client: &Client
) -> Result<SecretShare, Error> {
    // Retrieve from HSM
    let key_data = client.get_opaque_object(VALIDATOR_KEY_SLOT)?;

    // Deserialize
    let key_share = bincode::deserialize(&key_data)?;

    Ok(key_share)
}

// Sign with HSM protection
fn create_partial_signature(
    client: &Client,
    message: &[u8],
    signing_package: &SigningPackage
) -> Result<SignatureShare, Error> {
    // Load key share from HSM
    let key_share = load_key_share(client)?;

    // Generate signature (computation happens in memory)
    let signature = frost::round2::sign(
        signing_package,
        &nonce,
        &key_share
    )?;

    // Zero out key_share immediately
    drop(key_share);

    Ok(signature)
}
```

File-based storage (testnet only):

```rust
use aes_gcm::{Aes256Gcm, Key, Nonce};
use aes_gcm::aead::{Aead, NewAead};

// Store encrypted key share to file
fn store_key_share_file(
    key_share: &SecretShare,
    password: &str
) -> Result<(), Error> {
    // Derive encryption key from password
    let key = derive_key_from_password(password)?;
    let cipher = Aes256Gcm::new(Key::from_slice(&key));

    // Serialize
    let plaintext = bincode::serialize(key_share)?;

    // Encrypt
    let nonce = Nonce::from_slice(b"unique_nonce");
    let ciphertext = cipher.encrypt(nonce, plaintext.as_ref())?;

    // Write to file
    fs::write("key_share.enc", ciphertext)?;

    Ok(())
}

// Load and decrypt key share
fn load_key_share_file(
    password: &str
) -> Result<SecretShare, Error> {
    let key = derive_key_from_password(password)?;
    let cipher = Aes256Gcm::new(Key::from_slice(&key));

    let ciphertext = fs::read("key_share.enc")?;
    let nonce = Nonce::from_slice(b"unique_nonce");
    let plaintext = cipher.decrypt(nonce, ciphertext.as_ref())?;

    let key_share = bincode::deserialize(&plaintext)?;

    Ok(key_share)
}
```

---

4. Smart Contract Updates

4.1 Stacks Contract Changes for Phase 2

New Contract: `loan-protocol-v2.clar`

Key Additions: 1. Bitcoin address storage 2. Bitcoin deposit monitoring 3. Validator action requests 4. Cross-chain state tracking

```
;; =======================================
;; Phase 2 Extensions
;; =======================================

;; Bitcoin addresses for each loan
(define-map loan-bitcoin-addresses
  {loan-id: uint}
  {
    address: (string-ascii 62),         ;; bc1p... address
    deposit-txid: (optional (buff 32)), ;; Bitcoin txid
    deposit-confirmed: bool,
    confirmations: uint
  }
)

;; Validator action requests
(define-map action-requests
  {request-id: uint}
  {
    loan-id: uint,
    action-type: (string-ascii 20),     ;; "release", "claim", "refund"
    bitcoin-txid: (buff 32),            ;; UTXO to spend
    recipient-address: (string-ascii 62),
    amount-sats: uint,
    fee-sats: uint,
    requested-at-block: uint,
    status: (string-ascii 20),          ;; "pending", "signed", "confirmed"
    bitcoin-tx-result: (optional (buff 32))
  }
)

;; Validator registry
(define-map validators
  principal
  {
    validator-id: uint,
    public-key: (buff 33),
    status: (string-ascii 20),          ;; "active", "inactive"
```

```
    last-heartbeat: uint
  }
)

;; Action request counter
(define-data-var action-request-nonce uint u0)
```

4.2 Bitcoin Deposit Verification

```
;; Submit Bitcoin deposit proof
(define-public (submit-bitcoin-deposit-proof
    (loan-id uint)
    (txid (buff 32))
    (block-height uint)
    (merkle-proof (list 16 (buff 32)))
    (output-index uint)
    (output-amount uint))
  (let
    (
      (loan (unwrap! (map-get? loans {loan-id: loan-id}) ERR_LOAN_NOT_FOUND))
      (btc-address-info (unwrap! (map-get? loan-bitcoin-addresses {loan-id: loan-id}) ERR_NO
    )

    ;; Verify loan is awaiting deposit
    (asserts! (is-eq (get status loan) "awaiting-deposit") ERR_INVALID_STATUS)

    ;; Verify proof (simplified - actual implementation more complex)
    (asserts! (verify-merkle-proof txid block-height merkle-proof) ERR_INVALID_PROOF)

    ;; Verify amount matches
    (asserts! (>= output-amount (get collateral-amount loan)) ERR_INSUFFICIENT_AMOUNT)

    ;; Check confirmations
    (let ((current-height (get-bitcoin-block-height)))
      (let ((confirmations (- current-height block-height)))

        ;; Update address info
        (map-set loan-bitcoin-addresses
          {loan-id: loan-id}
          (merge btc-address-info {
            deposit-txid: (some txid),
            deposit-confirmed: (>= confirmations u3),
            confirmations: confirmations
          }))

        ;; If 3+ confirmations, activate loan
```

```
        (if (>= confirmations u3)
          (begin
            (map-set loans
              {loan-id: loan-id}
              (merge loan {status: "auction"}))
            (print {event: "deposit-confirmed", loan-id: loan-id, txid: txid})
          )
          (print {event: "deposit-pending", loan-id: loan-id, confirmations: confirmations})
        )

        (ok true)
      )
    )
  )
)

;; Helper: Verify merkle proof (simplified)
(define-read-only (verify-merkle-proof
    (txid (buff 32))
    (block-height uint)
    (proof (list 16 (buff 32))))
  ;; In production, this would:
  ;; 1. Get Bitcoin block header at block-height
  ;; 2. Verify merkle path from txid to merkle root
  ;; 3. Verify merkle root matches header
  ;; Simplified here for clarity
  true
)
```

4.3 Validator Action Requests

```
;; Request Bitcoin transaction (collateral release/claim)
(define-public (request-bitcoin-action
    (loan-id uint)
    (action-type (string-ascii 20))
    (recipient-address (string-ascii 62)))
  (let
    (
      (loan (unwrap! (map-get? loans {loan-id: loan-id}) ERR_LOAN_NOT_FOUND))
      (btc-address-info (unwrap! (map-get? loan-bitcoin-addresses {loan-id: loan-id}) ERR_NO
      (request-id (+ (var-get action-request-nonce) u1))
      (deposit-txid (unwrap! (get deposit-txid btc-address-info) ERR_NO_DEPOSIT))
    )

    ;; Verify caller is authorized
    (asserts! (or
```

```
      (and (is-eq action-type "release") (is-loan-finalized loan-id))
      (and (is-eq action-type "claim") (is-loan-defaulted loan-id))
    ) ERR_UNAUTHORIZED)

    ;; Estimate fee (simplified - could be dynamic)
    (let ((fee-sats u50000))  ;; ~$25 at $50K BTC

      ;; Create action request
      (map-set action-requests
        {request-id: request-id}
        {
          loan-id: loan-id,
          action-type: action-type,
          bitcoin-txid: deposit-txid,
          recipient-address: recipient-address,
          amount-sats: (get collateral-amount loan),
          fee-sats: fee-sats,
          requested-at-block: burn-block-height,
          status: "pending",
          bitcoin-tx-result: none
        })

      ;; Increment nonce
      (var-set action-request-nonce request-id)

      ;; Emit event for validators
      (print {
        event: "action-request",
        request-id: request-id,
        loan-id: loan-id,
        action: action-type,
        recipient: recipient-address,
        amount: (get collateral-amount loan),
        fee: fee-sats,
        utxo-txid: deposit-txid
      })

      (ok request-id)
    )
  )
)

;; Validators submit Bitcoin transaction confirmation
(define-public (confirm-bitcoin-action
    (request-id uint)
    (bitcoin-txid (buff 32))
```

```
    (block-height uint))
  (let
    (
      (request (unwrap! (map-get? action-requests {request-id: request-id}) ERR_REQUEST_NOT_
      (validator (unwrap! (map-get? validators tx-sender) ERR_NOT_VALIDATOR))
    )

    ;; Verify validator is active
    (asserts! (is-eq (get status validator) "active") ERR_VALIDATOR_INACTIVE)

    ;; Update request status
    (map-set action-requests
      {request-id: request-id}
      (merge request {
        status: "confirmed",
        bitcoin-tx-result: (some bitcoin-txid)
      }))

    ;; Update loan status
    (let ((loan-id (get loan-id request)))
      (let ((loan (unwrap! (map-get? loans {loan-id: loan-id}) ERR_LOAN_NOT_FOUND)))
        (map-set loans
          {loan-id: loan-id}
          (merge loan {
            status: (if (is-eq (get action-type request) "claim") "defaulted" "repaid")
          }))
      )
    )

    (print {
      event: "action-confirmed",
      request-id: request-id,
      bitcoin-txid: bitcoin-txid
    })

    (ok true)
  )
)
```

---

5. Bitcoin Transaction Construction

5.1 Transaction Builder

```
use bitcoin::{
    Transaction, TxIn, TxOut, OutPoint, Script, Witness,
```

```rust
    Address, Network, Amount
};
use bitcoin::blockdata::opcodes;
use bitcoin::secp256k1::{Secp256k1, PublicKey};

pub struct BitcoinTxBuilder {
    network: Network,
    fee_rate_sats_per_vbyte: u64,
}

impl BitcoinTxBuilder {
    pub fn new(network: Network) -> Self {
        Self {
            network,
            fee_rate_sats_per_vbyte: 50, // Conservative default
        }
    }

    /// Build collateral release transaction
    pub fn build_release_tx(
        &self,
        utxo_txid: [u8; 32],
        utxo_vout: u32,
        utxo_amount_sats: u64,
        recipient_address: &str,
        threshold_pubkey: &PublicKey,
    ) -> Result<Transaction, Error> {
        // Parse recipient address
        let recipient = Address::from_str(recipient_address)?
            .require_network(self.network)?;

        // Calculate fee
        let estimated_vsize = self.estimate_tx_vsize(1, 1); // 1 input, 1 output
        let fee_sats = estimated_vsize * self.fee_rate_sats_per_vbyte;
        let output_amount = utxo_amount_sats.saturating_sub(fee_sats);

        // Build transaction
        let tx = Transaction {
            version: 2,
            lock_time: bitcoin::PackedLockTime::ZERO,
            input: vec![
                TxIn {
                    previous_output: OutPoint {
                        txid: bitcoin::Txid::from_slice(&utxo_txid)?,
                        vout: utxo_vout,
                    },
```

```
                script_sig: Script::new(), // Empty for Taproot
                sequence: bitcoin::Sequence::ENABLE_RBF_NO_LOCKTIME,
                witness: Witness::new(), // Will be filled with signature
            }
        ],
        output: vec![
            TxOut {
                value: output_amount,
                script_pubkey: recipient.script_pubkey(),
            }
        ],
    };

    Ok(tx)
}

/// Estimate transaction virtual size
fn estimate_tx_vsize(&self, n_inputs: usize, n_outputs: usize) -> u64 {
    // Taproot key-path spend sizes (witness):
    // - 1 byte: witness stack items count
    // - 1 byte: signature length
    // - 64 bytes: Schnorr signature
    const TAPROOT_INPUT_WITNESS_SIZE: usize = 66;

    // Transaction overhead
    const TX_OVERHEAD: usize = 10;

    // Input size (excluding witness)
    const INPUT_SIZE: usize = 41; // outpoint (36) + script_sig (1) + sequence (4)

    // Output size
    const P2TR_OUTPUT_SIZE: usize = 43; // amount (8) + script (35)

    // Total weight units
    let base_weight = (TX_OVERHEAD + (INPUT_SIZE * n_inputs) + (P2TR_OUTPUT_SIZE * n_out
    let witness_weight = TAPROOT_INPUT_WITNESS_SIZE * n_inputs;
    let total_weight = base_weight + witness_weight;

    // Convert to vsize (weight / 4, rounded up)
    ((total_weight + 3) / 4) as u64
}

/// Attach threshold signature to transaction
pub fn attach_signature(
    &self,
    mut tx: Transaction,
```

```
        signature: &[u8; 64], // Schnorr signature
    ) -> Transaction {
        // Create witness with signature
        let mut witness = Witness::new();
        witness.push(signature);

        // Attach to input
        tx.input[0].witness = witness;

        tx
    }

    /// Calculate transaction ID (txid)
    pub fn get_txid(tx: &Transaction) -> [u8; 32] {
        use bitcoin::hashes::{Hash, sha256d};
        let txid = tx.txid();
        txid.as_hash().into_inner()
    }
}
```

5.2 Fee Estimation

```
use bitcoincore_rpc::{RpcApi, Client};

pub struct FeeEstimator {
    rpc: Client,
}

impl FeeEstimator {
    pub fn new(rpc: Client) -> Self {
        Self { rpc }
    }

    /// Get current fee rate for target confirmation
    pub fn estimate_fee_rate(
        &self,
        target_blocks: u16,
    ) -> Result<u64, Error> {
        // Query Bitcoin Core for fee estimate
        let estimate = self.rpc.estimate_smart_fee(target_blocks, None)?;

        // Convert BTC/kB to sats/vbyte
        let btc_per_kb = estimate.fee_rate
            .ok_or(Error::FeeEstimationFailed)?;
        let sats_per_byte = (btc_per_kb * 100_000.0) as u64;
```

```
        // Add safety margin (10%)
        let sats_per_vbyte = sats_per_byte * 110 / 100;

        // Enforce minimum (1 sat/vbyte)
        Ok(sats_per_vbyte.max(1))
    }

    /// Get fee for specific transaction
    pub fn calculate_fee(
        &self,
        tx_vsize: u64,
        target_blocks: u16,
    ) -> Result<u64, Error> {
        let fee_rate = self.estimate_fee_rate(target_blocks)?;
        Ok(tx_vsize * fee_rate)
    }
}

// Example usage
let estimator = FeeEstimator::new(rpc_client);

// Target 2 block confirmation
let fee_rate = estimator.estimate_fee_rate(2)?;
println!("Current fee rate: {} sats/vbyte", fee_rate);

// Calculate fee for our transaction
let tx_fee = estimator.calculate_fee(141, 2)?;  // 141 vbytes typical
println!("Transaction fee: {} sats (${:.2})", tx_fee, tx_fee as f64 * btc_price / 100_000_0
```

5.3 Replace-By-Fee (RBF)

```
/// Bump transaction fee if stuck in mempool
pub fn create_rbf_transaction(
    original_tx: &Transaction,
    new_fee_rate_sats_per_vbyte: u64,
) -> Result<Transaction, Error> {
    // Clone original transaction
    let mut rbf_tx = original_tx.clone();

    // Calculate new fee
    let vsize = original_tx.vsize() as u64;
    let new_fee = vsize * new_fee_rate_sats_per_vbyte;

    // Calculate old fee
    let input_value = get_input_value(&original_tx)?;
    let output_value: u64 = original_tx.output.iter().map(|o| o.value).sum();
```

```
    let old_fee = input_value - output_value;

    // Ensure new fee is higher (RBF rule)
    if new_fee <= old_fee {
        return Err(Error::FeeNotHigherThanOriginal);
    }

    // Reduce output amount by fee difference
    let fee_increase = new_fee - old_fee;
    rbf_tx.output[0].value -= fee_increase;

    // Verify output is still above dust threshold
    const DUST_THRESHOLD: u64 = 546;
    if rbf_tx.output[0].value < DUST_THRESHOLD {
        return Err(Error::OutputBelowDust);
    }

    Ok(rbf_tx)
}
```

---

6. Cross-Chain State Management

6.1 State Synchronization

Challenge: Keep Stacks and Bitcoin states consistent despite different block times and reorg possibilities.

Solution: State machine with multiple confirmation levels.

```
#[derive(Debug, Clone, PartialEq)]
pub enum LoanState {
    // Phase 1: Loan creation (Stacks only)
    Created { stacks_block: u64 },

    // Phase 2: Bitcoin deposit (Cross-chain)
    AwaitingDeposit { bitcoin_address: String },
    DepositSeen { bitcoin_txid: String, confirmations: u8 },
    DepositConfirmed { bitcoin_txid: String },

    // Phase 3: Auction (Stacks only)
    AuctionActive { end_block: u64 },
    AuctionFinalized { winner: String },

    // Phase 4: Loan active (Cross-chain aware)
    Active { maturity_block: u64 },
```

```rust
    // Phase 5: Resolution (Cross-chain)
    RepaymentInitiated { stacks_block: u64 },
    CollateralReleasing { bitcoin_txid: Option<String> },
    Completed { bitcoin_txid: String },

    // Alternative: Default
    Defaulted { claimed_at: u64 },
}

pub struct CrossChainStateMachine {
    bitcoin_client: BitcoinClient,
    stacks_client: StacksClient,
}

impl CrossChainStateMachine {
    /// Monitor Bitcoin deposit
    pub async fn monitor_deposit(
        &self,
        loan_id: u64,
        bitcoin_address: &str,
    ) -> Result<LoanState, Error> {
        // Check if deposit exists
        let utxos = self.bitcoin_client.get_address_utxos(bitcoin_address).await?;

        if utxos.is_empty() {
            return Ok(LoanState::AwaitingDeposit {
                bitcoin_address: bitcoin_address.to_string()
            });
        }

        // Get first UTXO (should be only one)
        let utxo = &utxos[0];

        // Check confirmations
        let confirmations = self.bitcoin_client.get_confirmations(&utxo.txid).await?;

        if confirmations < 3 {
            Ok(LoanState::DepositSeen {
                bitcoin_txid: utxo.txid.clone(),
                confirmations: confirmations as u8,
            })
        } else {
            // Submit proof to Stacks
            self.submit_deposit_proof(loan_id, &utxo).await?;

            Ok(LoanState::DepositConfirmed {
```

```
                bitcoin_txid: utxo.txid.clone(),
            })
        }
    }

    /// Handle Bitcoin reorg
    pub async fn handle_reorg(
        &self,
        loan_id: u64,
        old_txid: &str,
    ) -> Result<LoanState, Error> {
        // Check if transaction still exists
        let tx_exists = self.bitcoin_client.check_tx_exists(old_txid).await?;

        if !tx_exists {
            // Reorg removed our transaction
            warn!("Deposit transaction {} was reorged out", old_txid);

            // Revert to awaiting deposit
            let loan = self.stacks_client.get_loan(loan_id).await?;
            let bitcoin_address = loan.bitcoin_address;

            Ok(LoanState::AwaitingDeposit { bitcoin_address })
        } else {
            // Transaction still exists, just different confirmations
            self.monitor_deposit(loan_id, &loan.bitcoin_address).await
        }
    }

    /// Monitor collateral release
    pub async fn monitor_collateral_release(
        &self,
        request_id: u64,
    ) -> Result<LoanState, Error> {
        // Check if Bitcoin transaction exists
        let request = self.stacks_client.get_action_request(request_id).await?;

        if let Some(bitcoin_txid) = request.bitcoin_tx_result {
            // Check confirmations
            let confirmations = self.bitcoin_client.get_confirmations(&bitcoin_txid).await?;

            if confirmations >= 1 {
                Ok(LoanState::Completed { bitcoin_txid })
            } else {
                Ok(LoanState::CollateralReleasing {
                    bitcoin_txid: Some(bitcoin_txid)
```

```
                })
            }
        } else {
            Ok(LoanState::CollateralReleasing {
                bitcoin_txid: None
            })
        }
    }
}
```

6.2 Reorg Handling

```
pub struct ReorgDetector {
    bitcoin_client: BitcoinClient,
    last_known_height: u64,
    last_known_hash: String,
}

impl ReorgDetector {
    pub async fn check_for_reorg(&mut self) -> Result<bool, Error> {
        let current_height = self.bitcoin_client.get_block_count().await?;

        if current_height < self.last_known_height {
            // Definitely a reorg
            warn!("Reorg detected: height decreased from {} to {}",
                  self.last_known_height, current_height);
            return Ok(true);
        }

        // Check if our last known block is still in the chain
        let block_hash = self.bitcoin_client
            .get_block_hash(self.last_known_height)
            .await?;

        if block_hash != self.last_known_hash {
            // Reorg happened
            warn!("Reorg detected: block hash changed at height {}",
                  self.last_known_height);
            return Ok(true);
        }

        // Update tracking
        self.last_known_height = current_height;
        self.last_known_hash = self.bitcoin_client
            .get_block_hash(current_height)
            .await?;
```

```
        Ok(false)
    }

    pub async fn handle_reorg(&self, affected_loans: Vec<u64>) -> Result<(), Error> {
        for loan_id in affected_loans {
            // Re-verify deposit
            let loan = self.stacks_client.get_loan(loan_id).await?;

            if let Some(deposit_txid) = loan.deposit_txid {
                // Check if transaction still exists
                let exists = self.bitcoin_client.check_tx_exists(&deposit_txid).await?;

                if !exists {
                    // Transaction was reorged out
                    error!("Loan {} deposit was reorged out", loan_id);

                    // Notify contract (revert to awaiting deposit)
                    self.stacks_client.report_reorg(loan_id).await?;
                }
            }
        }

        Ok(())
    }
}
```

---

7. Validator Network Protocol

7.1 libp2p Network Configuration

```
use libp2p::{
    gossipsub, mdns, noise,
    swarm::{SwarmBuilder, SwarmEvent},
    tcp, yamux, PeerId, Transport,
};

pub struct ValidatorNetwork {
    swarm: Swarm<ValidatorBehaviour>,
    local_peer_id: PeerId,
}

#[derive(NetworkBehaviour)]
struct ValidatorBehaviour {
    gossipsub: gossipsub::Behaviour,
```

```rust
    mdns: mdns::tokio::Behaviour,
}

impl ValidatorNetwork {
    pub async fn new(config: &NetworkConfig) -> Result<Self, Error> {
        // Generate keypair for this validator
        let local_key = identity::Keypair::generate_ed25519();
        let local_peer_id = PeerId::from(local_key.public());

        info!("Validator peer ID: {}", local_peer_id);

        // Build transport
        let transport = tcp::tokio::Transport::new(tcp::Config::default().nodelay(true))
            .upgrade(upgrade::Version::V1)
            .authenticate(noise::Config::new(&local_key)?)
            .multiplex(yamux::Config::default())
            .boxed();

        // Configure Gossipsub
        let gossipsub_config = gossipsub::ConfigBuilder::default()
            .heartbeat_interval(Duration::from_secs(1))
            .validation_mode(gossipsub::ValidationMode::Strict)
            .build()?;

        let mut gossipsub = gossipsub::Behaviour::new(
            gossipsub::MessageAuthenticity::Signed(local_key.clone()),
            gossipsub_config,
        )?;

        // Subscribe to topics
        let nonce_topic = gossipsub::IdentTopic::new("validator/nonces");
        let signature_topic = gossipsub::IdentTopic::new("validator/signatures");
        gossipsub.subscribe(&nonce_topic)?;
        gossipsub.subscribe(&signature_topic)?;

        // Configure mDNS for local peer discovery
        let mdns = mdns::tokio::Behaviour::new(mdns::Config::default(), local_peer_id)?;

        // Build behaviour
        let behaviour = ValidatorBehaviour { gossipsub, mdns };

        // Build swarm
        let swarm = SwarmBuilder::with_tokio_executor(transport, behaviour, local_peer_id)
            .build();

        Ok(Self {
```

```
            swarm,
            local_peer_id,
        })
    }

    /// Broadcast nonce commitment
    pub fn broadcast_nonce_commitment(
        &mut self,
        signing_session: &str,
        commitment: &NonceCommitment,
    ) -> Result<(), Error> {
        let message = ValidatorMessage::NonceCommitment {
            session_id: signing_session.to_string(),
            validator_id: self.local_peer_id,
            commitment: commitment.clone(),
        };

        let topic = gossipsub::IdentTopic::new("validator/nonces");
        let serialized = serde_json::to_vec(&message)?;

        self.swarm
            .behaviour_mut()
            .gossipsub
            .publish(topic, serialized)?;

        Ok(())
    }

    /// Broadcast partial signature
    pub fn broadcast_partial_signature(
        &mut self,
        signing_session: &str,
        signature: &SignatureShare,
    ) -> Result<(), Error> {
        let message = ValidatorMessage::PartialSignature {
            session_id: signing_session.to_string(),
            validator_id: self.local_peer_id,
            signature: signature.clone(),
        };

        let topic = gossipsub::IdentTopic::new("validator/signatures");
        let serialized = serde_json::to_vec(&message)?;

        self.swarm
            .behaviour_mut()
            .gossipsub
```

```
            .publish(topic, serialized)?;

        Ok(())
    }

    /// Listen for messages
    pub async fn next_event(&mut self) -> ValidatorNetworkEvent {
        loop {
            match self.swarm.select_next_some().await {
                SwarmEvent::Behaviour(BehaviourEvent::Gossipsub(
                    gossipsub::Event::Message {
                        message,
                        ..
                    }
                )) => {
                    // Deserialize message
                    if let Ok(msg) = serde_json::from_slice::<ValidatorMessage>(&message.dat
                        return ValidatorNetworkEvent::Message(msg);
                    }
                }
                SwarmEvent::Behaviour(BehaviourEvent::Mdns(
                    mdns::Event::Discovered(list)
                )) => {
                    for (peer_id, addr) in list {
                        info!("Discovered peer: {} at {}", peer_id, addr);
                        self.swarm.dial(addr)?;
                    }
                }
                _ => {}
            }
        }
    }
}

#[derive(Debug, Clone, Serialize, Deserialize)]
pub enum ValidatorMessage {
    NonceCommitment {
        session_id: String,
        validator_id: PeerId,
        commitment: NonceCommitment,
    },
    PartialSignature {
        session_id: String,
        validator_id: PeerId,
        signature: SignatureShare,
    },
```

```
    Heartbeat {
        validator_id: PeerId,
        timestamp: u64,
    },
}
```

7.2 Coordinator Service

```rust
pub struct Coordinator {
    network: ValidatorNetwork,
    active_sessions: HashMap<String, SigningSession>,
}

struct SigningSession {
    request_id: u64,
    message: [u8; 32],
    nonce_commitments: HashMap<PeerId, NonceCommitment>,
    partial_signatures: HashMap<PeerId, SignatureShare>,
    started_at: Instant,
}

impl Coordinator {
    /// Initiate signing session
    pub async fn initiate_signing(
        &mut self,
        request_id: u64,
        message: [u8; 32],
    ) -> Result<(), Error> {
        let session_id = format!("signing-{}", request_id);

        info!("Initiating signing session: {}", session_id);

        // Create session
        let session = SigningSession {
            request_id,
            message,
            nonce_commitments: HashMap::new(),
            partial_signatures: HashMap::new(),
            started_at: Instant::now(),
        };

        self.active_sessions.insert(session_id.clone(), session);

        // Broadcast signing request to all validators
        self.network.broadcast_signing_request(&session_id, message)?;
```

```rust
    Ok(())
}

/// Collect nonce commitments
pub async fn collect_nonce_commitments(
    &mut self,
    session_id: &str,
) -> Result<Vec<NonceCommitment>, Error> {
    let session = self.active_sessions.get_mut(session_id)
        .ok_or(Error::SessionNotFound)?;

    // Wait for at least 10 commitments (or timeout after 30 seconds)
    let timeout = Duration::from_secs(30);
    let start = Instant::now();

    while session.nonce_commitments.len() < 10 {
        if start.elapsed() > timeout {
            return Err(Error::InsufficientCommitments);
        }

        // Process network events
        if let ValidatorNetworkEvent::Message(
            ValidatorMessage::NonceCommitment {
                session_id: msg_session,
                validator_id,
                commitment,
            }
        ) = self.network.next_event().await {
            if msg_session == session_id {
                info!("Received nonce commitment from {}", validator_id);
                session.nonce_commitments.insert(validator_id, commitment);
            }
        }
    }

    Ok(session.nonce_commitments.values().cloned().collect())
}

/// Collect partial signatures
pub async fn collect_partial_signatures(
    &mut self,
    session_id: &str,
) -> Result<Vec<SignatureShare>, Error> {
    let session = self.active_sessions.get_mut(session_id)
        .ok_or(Error::SessionNotFound)?;
```

```rust
    // Wait for at least 10 signatures
    let timeout = Duration::from_secs(30);
    let start = Instant::now();

    while session.partial_signatures.len() < 10 {
        if start.elapsed() > timeout {
            return Err(Error::InsufficientSignatures);
        }

        if let ValidatorNetworkEvent::Message(
            ValidatorMessage::PartialSignature {
                session_id: msg_session,
                validator_id,
                signature,
            }
        ) = self.network.next_event().await {
            if msg_session == session_id {
                info!("Received partial signature from {}", validator_id);
                session.partial_signatures.insert(validator_id, signature);
            }
        }
    }

    Ok(session.partial_signatures.values().cloned().collect())
}

/// Aggregate signatures and create final Bitcoin transaction
pub async fn finalize_signing(
    &mut self,
    session_id: &str,
    bitcoin_tx_builder: &BitcoinTxBuilder,
) -> Result<Transaction, Error> {
    let session = self.active_sessions.remove(session_id)
        .ok_or(Error::SessionNotFound)?;

    // Aggregate partial signatures
    let final_signature = frost::aggregate(
        &session.partial_signatures.values().cloned().collect::<Vec<_>>(),
        &session.nonce_commitments.values().cloned().collect::<Vec<_>>(),
    )?;

    info!("Successfully aggregated threshold signature");

    // Get Bitcoin transaction for this request
    let request = stacks_client.get_action_request(session.request_id).await?;
```

```
        let bitcoin_tx = bitcoin_tx_builder.build_release_tx(
            request.bitcoin_txid,
            0, // vout
            request.amount_sats,
            &request.recipient_address,
            &threshold_pubkey,
        )?;

        // Attach signature
        let signed_tx = bitcoin_tx_builder.attach_signature(bitcoin_tx, &final_signature);

        Ok(signed_tx)
    }
}
```

[Document continues with sections 8-13...]

Due to length, I'll continue in the next message. Would you like me to complete the remaining sections (8. Security Architecture through 13. Disaster Recovery)?

8. Security Architecture

8.1 Threat Model

Assets to Protect: 1. Validator key shares (most critical) 2. Bitcoin in threshold addresses 3. Validator network integrity 4. Cross-chain state consistency

Threat Actors: - External attackers (steal Bitcoin) - Malicious validators (collude or sabotage) - Network attackers (DoS, eclipse) - State inconsistency exploits

Attack Scenarios & Mitigations:

Attack	Likelihood	Impact	Mitigation
Key extraction from single validator	Low	Critical if 10+ compromised	HSMs, secure enclaves, monitoring
Validator collusion (10+)	Very Low	Critical	Geographic distribution, economic incentives, reputation
Network partition	Medium	High	Multiple network paths, partition detection

Attack	Likelihood	Impact	Mitigation
Double-signing	Low	Medium	Nonce tracking, replay prevention
Eclipse attack on validator	Medium	Medium	Multiple bootstrap peers, peer diversity
Stacks contract exploit	Low	Critical	Audit, formal verification, bug bounty
Bitcoin reorg	Low	Medium	Wait for 3+ confirmations, reorg detection
Coordinator compromise	Medium	Medium	Coordinator is not trusted (verify sigs), multiple coordinators possible

8.2 Key Share Security

HSM Requirements:

```
Hardware Security Module (HSM) must provide:
- FIPS 140-2 Level 3 or higher
- Tamper detection and response
- Secure key storage (keys never leave HSM)
- Cryptographic acceleration
- Audit logging
- Backup and recovery under wrap

Recommended: YubiHSM 2, AWS CloudHSM, Thales Luna
```

Key Generation Security:

```
// Never construct full private key in any location
// Each validator only ever has their share

// BAD - DO NOT DO THIS:
let full_private_key = combine_shares(shares); //  NEVER

// GOOD - Validator only uses their share:
let my_share = load_from_hsm()?;
let partial_sig = sign_with_share(&my_share, message)?;
drop(my_share); // Immediately drop after use
```

Key Share Backup:

```
Backup Strategy:
```

```
1. Export wrapped key from HSM (encrypted)
2. Split wrapped key using Shamir Secret Sharing (3-of-5)
3. Distribute shares to:
   - Validator operator (encrypted)
   - Secure cloud storage (2 different providers)
   - Hardware backup (offline, different location)
   - Escrow service (for emergency recovery)

Recovery requires 3 of 5 shares
Test recovery procedure quarterly
```

8.3 Network Security

Eclipse Attack Prevention:

```rust
pub struct PeerDiversityChecker {
    peers: Vec<PeerInfo>,
}

impl PeerDiversityChecker {
    /// Ensure peer diversity (prevent eclipse)
    pub fn check_diversity(&self) -> Result<(), Error> {
        let total_peers = self.peers.len();

        if total_peers < 8 {
            return Err(Error::InsufficientPeers);
        }

        // Check geographic diversity
        let mut ip_prefixes = HashSet::new();
        for peer in &self.peers {
            let prefix = get_ip_prefix(&peer.address, 16);
            ip_prefixes.insert(prefix);
        }

        if ip_prefixes.len() < 5 {
            warn!("Low IP diversity: potential eclipse attack");
            return Err(Error::LowPeerDiversity);
        }

        // Check AS diversity (Autonomous Systems)
        let as_numbers: HashSet<_> = self.peers
            .iter()
            .map(|p| p.as_number)
            .collect();
```

```
        if as_numbers.len() < 4 {
            warn!("Low AS diversity: potential eclipse attack");
            return Err(Error::LowASDiversity);
        }

        Ok(())
    }
}
```

DoS Protection:

```
pub struct RateLimiter {
    limits: HashMap<PeerId, TokenBucket>,
}

impl RateLimiter {
    /// Check if peer exceeded rate limit
    pub fn check_limit(&mut self, peer_id: &PeerId) -> bool {
        let bucket = self.limits
            .entry(peer_id.clone())
            .or_insert_with(|| TokenBucket::new(100, 10)); // 100 tokens, refill 10/sec

        bucket.take(1)
    }

    /// Ban peer if consistently misbehaving
    pub fn record_misbehavior(&mut self, peer_id: &PeerId) {
        // Implement reputation system
        // Ban if reputation drops below threshold
    }
}
```

8.4 Double-Signing Prevention

```
pub struct NonceTracker {
    used_nonces: HashSet<([u8; 32], u64)>, // (message, session_id)
}

impl NonceTracker {
    /// Ensure we never sign same message twice with different nonce
    pub fn check_and_record(
        &mut self,
        message: &[u8; 32],
        session_id: u64,
    ) -> Result<(), Error> {
        let key = (*message, session_id);
```

```
        if self.used_nonces.contains(&key) {
            error!("Attempted double-signing detected!");
            return Err(Error::DoubleSigningAttempt);
        }

        self.used_nonces.insert(key);
        Ok(())
    }

    /// Clean old nonces (after confirmed on Bitcoin)
    pub fn cleanup_old_nonces(&mut self, cutoff: u64) {
        self.used_nonces.retain(|(_, session_id)| *session_id > cutoff);
    }
}
```

---

9. API Specifications

9.1 Validator REST API

Base URL: `http://validator-host:8080/api/v1`

Endpoints:

```
GET /health
Response: {"status": "healthy", "bitcoin_synced": true, "stacks_synced": true}

GET /status
Response: {
  "validator_id": 1,
  "peer_count": 14,
  "last_signature": "2026-01-12T10:30:00Z",
  "uptime_seconds": 864000
}

GET /key_share/public
Response: {
  "public_key_share": "02abc123...",
  "combined_public_key": "03def456..."
}

POST /signing/request
Body: {
  "session_id": "signing-123",
  "message": "abc123...",
  "participants": [1, 2, 3, ...]
}
```

```
Response: {"status": "initiated"}

GET /signing/session/:session_id
Response: {
  "session_id": "signing-123",
  "status": "collecting_signatures",
  "nonce_commitments": 12,
  "partial_signatures": 10,
  "progress": "67%"
}
```

9.2 Coordinator REST API

```
POST /api/v1/action/initiate
Body: {
  "request_id": 42,
  "action_type": "release",
  "bitcoin_tx_params": {...}
}
Response: {
  "session_id": "signing-42",
  "status": "initiated"
}

GET /api/v1/action/status/:request_id
Response: {
  "request_id": 42,
  "session_id": "signing-42",
  "status": "signed",
  "bitcoin_txid": "abc123...",
  "confirmations": 2
}

GET /api/v1/validators
Response: {
  "validators": [
    {"id": 1, "status": "active", "last_seen": "2s ago"},
    {"id": 2, "status": "active", "last_seen": "1s ago"},
    ...
  ],
  "online": 14,
  "threshold": 10
}
```

9.3 WebSocket Events

```javascript
// Connect to validator WebSocket
const ws = new WebSocket('ws://validator-host:8080/ws');

// Subscribe to events
ws.send(JSON.stringify({
  type: 'subscribe',
  topics: ['signatures', 'deposits', 'network']
}));

// Listen for events
ws.onmessage = (event) => {
  const data = JSON.parse(event.data);

  switch (data.type) {
    case 'signature_started':
      console.log(`Signing session ${data.session_id} started`);
      break;

    case 'signature_completed':
      console.log(`Bitcoin tx ${data.txid} signed`);
      break;

    case 'deposit_detected':
      console.log(`Deposit to loan ${data.loan_id}: ${data.txid}`);
      break;

    case 'network_event':
      console.log(`Peer ${data.peer_id}: ${data.status}`);
      break;
  }
};
```

---

10. Testing Strategy

10.1 Unit Tests

Threshold Cryptography:

```rust
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_dkg_ceremony() {
```

```rust
    // Simulate 15 validators
    let mut validators = vec![];
    for i in 0..15 {
        validators.push(Validator::new(i));
    }

    // Round 1: Commitments
    let commitments: Vec<_> = validators
        .iter_mut()
        .map(|v| v.dkg_round1())
        .collect();

    // Round 2: Shares
    let shares = validators[0].dkg_round2(&commitments);

    // Each validator verifies shares
    for validator in &validators {
        assert!(validator.verify_shares(&shares));
    }

    // Derive combined public key
    let pub_keys: Vec<_> = validators
        .iter()
        .map(|v| v.get_public_key_share())
        .collect();

    let combined_key = combine_public_keys(&pub_keys);

    // Test signing with 10 validators
    let message = b"test message";
    let signing_validators = &validators[0..10];

    let partial_sigs: Vec<_> = signing_validators
        .iter()
        .map(|v| v.create_partial_signature(message))
        .collect();

    let final_sig = aggregate_signatures(&partial_sigs);

    // Verify signature
    assert!(verify_signature(&final_sig, message, &combined_key));
}

#[test]
fn test_insufficient_signatures() {
    // Only 9 signatures (need 10)
```

```
        let partial_sigs = vec![/* 9 signatures */];

        let result = aggregate_signatures(&partial_sigs);
        assert!(result.is_err());
    }

    #[test]
    fn test_malicious_signature() {
        // One signature is invalid
        let mut partial_sigs = vec![/* 9 valid signatures */];
        partial_sigs.push(create_invalid_signature());

        let result = aggregate_signatures(&partial_sigs);
        assert!(result.is_err());
    }
}
```

Bitcoin Transaction Construction:

```
#[test]
fn test_release_transaction() {
    let builder = BitcoinTxBuilder::new(Network::Testnet);

    let utxo_txid = [0u8; 32];
    let utxo_vout = 0;
    let utxo_amount = 150_000_000; // 1.5 BTC
    let recipient = "tb1q...";
    let pubkey = PublicKey::from_slice(&[0x02; 33]).unwrap();

    let tx = builder.build_release_tx(
        utxo_txid,
        utxo_vout,
        utxo_amount,
        recipient,
        &pubkey,
    ).unwrap();

    // Verify transaction structure
    assert_eq!(tx.input.len(), 1);
    assert_eq!(tx.output.len(), 1);

    // Verify fee is reasonable
    let output_amount = tx.output[0].value;
    let fee = utxo_amount - output_amount;
    assert!(fee > 1000); // At least 1000 sats
    assert!(fee < 100_000); // But not excessive
}
```

10.2 Integration Tests

Full Signing Flow:

```rust
#[tokio::test]
async fn test_complete_signing_flow() {
    // Setup 15 validators
    let validators = setup_test_validators(15).await;
    let coordinator = Coordinator::new();

    // Create signing request
    let message = create_test_message();
    let session_id = coordinator
        .initiate_signing(1, message)
        .await
        .unwrap();

    // Validators respond with nonce commitments
    for validator in &validators {
        validator
            .respond_to_signing_request(&session_id, &message)
            .await
            .unwrap();
    }

    // Coordinator collects commitments
    let commitments = coordinator
        .collect_nonce_commitments(&session_id)
        .await
        .unwrap();

    assert!(commitments.len() >= 10);

    // Validators create partial signatures
    for validator in &validators {
        validator
            .create_partial_signature(&session_id, &commitments)
            .await
            .unwrap();
    }

    // Coordinator aggregates
    let final_sig = coordinator
        .finalize_signing(&session_id)
        .await
        .unwrap();
```

```
    // Verify final signature
    assert!(verify_threshold_signature(&final_sig, &message));
}
```

Cross-Chain State Sync:

```
#[tokio::test]
async fn test_deposit_confirmation() {
    let bitcoin_client = setup_bitcoin_regtest().await;
    let stacks_client = setup_stacks_mocknet().await;

    // Create loan on Stacks
    let loan_id = stacks_client.create_loan().await.unwrap();

    // Get Bitcoin address
    let btc_address = stacks_client.get_loan_address(loan_id).await.unwrap();

    // Send Bitcoin
    let txid = bitcoin_client
        .send_to_address(&btc_address, 150_000_000)
        .await
        .unwrap();

    // Mine 3 blocks
    bitcoin_client.generate_blocks(3).await.unwrap();

    // Submit proof to Stacks
    let proof = bitcoin_client.get_tx_proof(&txid).await.unwrap();
    stacks_client
        .submit_deposit_proof(loan_id, proof)
        .await
        .unwrap();

    // Verify loan activated
    let loan = stacks_client.get_loan(loan_id).await.unwrap();
    assert_eq!(loan.status, "auction");
}
```

10.3 Chaos Testing

```
/// Test validator failures during signing
#[tokio::test]
async fn test_validator_failure_during_signing() {
    let mut validators = setup_test_validators(15).await;
    let coordinator = Coordinator::new();
```

```rust
    // Start signing
    let session_id = coordinator.initiate_signing(1, message).await.unwrap();

    // 12 validators respond
    for i in 0..12 {
        validators[i].respond(&session_id).await.unwrap();
    }

    // 3 validators fail to respond (simulated)
    // But we have 12 > 10, so signing should succeed

    let result = coordinator.finalize_signing(&session_id).await;
    assert!(result.is_ok());
}

/// Test network partition
#[tokio::test]
async fn test_network_partition() {
    let validators = setup_test_validators(15).await;

    // Create network partition: 8 vs 7 validators
    let (group_a, group_b) = validators.split_at(8);

    // Disconnect groups
    partition_network(group_a, group_b).await;

    // Larger group (8) should continue
    let result_a = group_a[0].initiate_signing(message).await;
    assert!(result_a.is_err()); // 8 < 10, should fail

    // Reconnect
    reconnect_network(group_a, group_b).await;

    // Now all 15 can sign
    let result = validators[0].initiate_signing(message).await;
    assert!(result.is_ok());
}
```

---

11. Deployment Plan

11.1 Pre-Deployment Checklist

```
Infrastructure:
- [ ] 15 validator servers provisioned
- [ ] HSMs installed and initialized
```

```
- [ ] Bitcoin nodes synced (full nodes, txindex enabled)
- [ ] Stacks nodes synced
- [ ] Network connectivity verified
- [ ] Monitoring dashboards configured
- [ ] Alert systems tested

Security:
- [ ] Security audit completed (threshold crypto + validators)
- [ ] Penetration testing completed
- [ ] Bug bounty program active
- [ ] Incident response plan documented
- [ ] Key backup procedures tested
- [ ] Disaster recovery procedures tested

Testing:
- [ ] Unit tests: >95% coverage
- [ ] Integration tests: all scenarios pass
- [ ] Chaos tests: system resilient to failures
- [ ] Load testing: can handle 50+ concurrent signings
- [ ] Testnet deployment successful (3+ months)

Documentation:
- [ ] Validator setup guide complete
- [ ] Operator runbooks finalized
- [ ] API documentation published
- [ ] User guides updated (Phase 2 features)
```

11.2 Deployment Sequence

Week 1: DKG Ceremony (Testnet)

```
Day 1-2: Validator Setup
- All 15 validators deploy software
- Connect to testnet
- Verify connectivity

Day 3: DKG Rehearsal
- Test run of DKG ceremony
- Verify key generation
- Test threshold signing

Day 4-5: Fixes and Adjustments
- Address any issues found
- Re-run DKG if needed
- Finalize configuration
```

```
Day 6-7: Buffer
- Additional testing
- Documentation updates
```

Week 2: Testnet Validation

```
Day 1: Deploy Stacks Contract v2
- Deploy to Stacks testnet
- Initialize with validator registry

Day 2-5: End-to-End Testing
- Create test loans with native BTC
- Test full lifecycle
- Monitor validator performance
- Verify cross-chain sync

Day 6-7: Stress Testing
- Multiple concurrent loans
- Validator failures (intentional)
- Network issues (simulated)
- Bitcoin reorgs (simulated)
```

Week 3: Mainnet DKG Ceremony

```
Day 1: Pre-Ceremony
- All validators ready
- Mainnet nodes synced
- Final configuration check

Day 2: DKG Ceremony (Mainnet)
- All 15 validators participate
- Generate mainnet threshold key
- Verify key generation successful
- Backup key shares

Day 3: Post-Ceremony
- Verify all validators have correct keys
- Test signing on mainnet (small amount)
- Confirm everything working

Day 4-5: Buffer
- Address any issues
- Additional verification
```

Week 4: Mainnet Launch

```
Day 1: Contract Deployment
- Deploy loan-protocol-v2 to Stacks mainnet
- Initialize with validator registry
```

```
- Verify deployment

Day 2: Soft Launch
- Announce Phase 2 availability
- Invite select users (whitelisted)
- Start with small loan amounts (<$50K)
- Monitor closely

Day 3-5: Gradual Rollout
- Increase loan size limits
- Open to more users
- Continue monitoring

Day 6-7: Full Launch
- Remove limits
- Public announcement
- Marketing campaign
```

11.3 Rollback Plan

If Critical Issue Found:

```
Severity 1: Funds at Risk
→ Immediate Actions:
  1. Pause new loan creation (emergency function)
  2. Allow existing loans to complete
  3. Investigate issue
  4. Fix and re-audit
  5. Redeploy

Severity 2: Functionality Broken
→ Measured Response:
  1. Identify affected loans
  2. Provide workarounds if possible
  3. Fix issue
  4. Deploy patch
  5. Migrate affected loans

Severity 3: Minor Issues
→ Standard Process:
  1. Document issue
  2. Add to fix queue
  3. Deploy in next release
```

Phase 1 Fallback: - Phase 1 (sBTC) remains available - Users can choose Phase 1 or Phase 2 - No forced migration - Gradual migration as confidence builds

---

12. Monitoring & Operations

12.1 Monitoring Dashboard

Validator Health:

```
Validator Network Status


Online: 14/15 (93%)
Threshold: 10 required
Current Capacity: 140%

Validators:
[1]        99.8% ↑ 2s
[2]        99.9% ↑ 1s
[3]        99.7% ↑ 3s
[4]        99.6% ↑ 5s
[5]        98.2% ↓ 15m
...

Recent Signatures:
• Loan #157: Released (12 sigs, 2m)
• Loan #155: Released (11 sigs, 3m)
• Loan #154: Claimed (13 sigs, 2m)
```

Signing Performance:

```
Threshold Signing Performance


Last 24 Hours:
• Signatures: 47
• Success Rate: 100%
• Avg Time: 2m 15s
• P95: 3m 45s
• P99: 4m 30s

Current Queue:
• Pending: 2 sessions
• Oldest: 45 seconds
```

Bitcoin/Stacks Sync:

```
Blockchain Synchronization


Bitcoin:
• Height: 825,432  Synced
• Mempool: 2,347 txs
• Avg Fee: 45 sat/vB

Stacks:
• Height: 145,678  Synced
• Anchored to Bitcoin: 825,432

Deposits Pending Confirmation:
• Loan #158: 2/3 confirmations
```

12.2 Alerting Rules

P0 Alerts (Page Immediately):

```
- Validators online < 10 (below threshold)
- Signing failure (unable to generate signature)
- Bitcoin deposit not confirmed after 6 hours
- Security incident detected
- Key share access anomaly
```

P1 Alerts (Response within 1 hour):

```
- Validator down for >15 minutes
- Signing latency >5 minutes (P95)
- Bitcoin reorg detected (>3 blocks)
- Network partition suspected
- Abnormal signing activity
```

P2 Alerts (Response within 4 hours):

```
- Validator slow response (>99% to 95-99%)
- Low peer connectivity (<8 peers)
- Bitcoin/Stacks node behind (>10 blocks)
- High Bitcoin fees (>100 sat/vB)
```

12.3 Runbook: Common Scenarios

Scenario: Validator Down

```
Detection: Heartbeat missing for >5 minutes

Actions:
1. Check validator server status
   - SSH to server
   - Check systemd status: systemctl status validator
   - Review logs: journalctl -u validator -n 100

2. If process crashed:
   - Identify crash reason from logs
   - Restart: systemctl restart validator
   - Monitor for successful startup

3. If server down:
   - Contact hosting provider
   - Bring server back online
   - Verify blockchain sync
   - Restart validator process

4. If >4 validators down (critical):
   - Page all on-call team
   - Escalate to incident commander
   - Activate backup validators if available
   - Consider emergency pause if <10 online

Expected Resolution Time: 15-30 minutes
```

Scenario: Signing Timeout

```
Detection: Signing session >5 minutes without completion

Actions:
1. Check signing session status:
   GET /api/v1/signing/session/:id

2. Identify non-responsive validators:
   - Review nonce_commitments and partial_signatures
   - Identify missing validators

3. Contact non-responsive validator operators
   - Request status check
   - Identify issue (network, process, etc.)

4. If sufficient signatures (10+):
   - Coordinator can proceed with finalization
   - Log non-responsive validators
```

```
5. If insufficient signatures:
   - Retry signing session
   - Exclude non-responsive validators
   - Use different validator subset

6. Post-Incident:
   - Review why validators were non-responsive
   - Update monitoring if needed
   - Consider validator rotation if persistent

Expected Resolution Time: 5-15 minutes
```

Scenario: Bitcoin Reorg

```
Detection: Bitcoin block height decreased or block hash changed

Actions:
1. Determine reorg depth:
   - Compare current chain to our last known
   - Identify affected blocks

2. Identify affected loans:
   - Query loans with deposits in reorged blocks
   - Check if any confirmations were reversed

3. For each affected loan:
   - If deposit was removed:
     * Update Stacks contract status back to "awaiting-deposit"
     * Notify borrower
     * Monitor for deposit re-confirmation

   - If deposit still exists (different block):
     * Update confirmation count
     * Continue normal process

4. Monitor for chain stability:
   - Wait for 6+ confirmations before proceeding
   - Verify no further reorgs

5. Document incident:
   - Reorg depth, duration, affected loans
   - Any funds at risk (should be none)
   - Lessons learned

Expected Resolution Time: 1-2 hours
```

---

13. Disaster Recovery

13.1 Key Recovery Procedures

Scenario: Validator Key Share Lost

```
Prerequisites for Recovery:
- Validator identity verified (multiple factor authentication)
- Backup key share available (from secure storage)
- Recovery ceremony authorized by protocol governance

Recovery Process:
1. Verify validator identity through multiple channels
2. Retrieve backup key share from secure storage
   (requires 3-of-5 Shamir shares)
3. Decrypt backup using validator's master password
4. Load key share into new HSM
5. Verify key share is correct:
   - Compute public key share
   - Compare with on-chain registry
6. Resume validator operations
7. Test signing with other validators

Timeline: 2-4 hours
Risk Level: Medium (requires multiple authorization steps)
```

Scenario: Multiple Validators Lost (6-10)

```
This is a CRITICAL scenario approaching threshold failure.

Immediate Actions (First Hour):
1. Emergency protocol activation
2. Page all validator operators
3. Assess situation:
   - Which validators are affected?
   - Is this coordinated attack or coincidence?
   - Can validators be recovered quickly?

Short-term Mitigation (Hours 1-6):
1. Activate backup validators if available
2. Expedite recovery of lost validators
3. Pause new loan creation if below threshold
4. Allow existing loans to complete
5. Coordinate emergency validator recruitment

Long-term Recovery (Days 1-7):
1. Add emergency validators (if needed):
   - Community members
```

```
   - Partner organizations
   - Temporary commercial validators
2. Perform new DKG ceremony (if threshold broken)
3. Migrate to new validator set (if necessary)
4. Post-mortem and prevention measures

Timeline: 1-7 days depending on severity
Risk Level: CRITICAL
```

13.2 Contract Upgrade Path

Phase 2 Contracts are Immutable: - Smart contracts cannot be upgraded - If critical bug found, must deploy new version - Existing loans continue on old contract - New loans use new contract

Migration Strategy:

```
1. Deploy new contract version
2. Mark old contract as deprecated
3. Allow both versions to run simultaneously
4. Migrate users gradually:
   - Existing loans complete normally on old contract
   - New loans go to new contract
5. After all old loans complete, old contract becomes inactive
```

Emergency Pause Function:

```
;; Emergency pause (only for new loans)
(define-data-var emergency-paused bool false)

(define-public (set-emergency-pause (paused bool))
  (begin
    (asserts! (is-eq tx-sender (var-get contract-owner)) ERR_UNAUTHORIZED)
    (var-set emergency-paused paused)
    (ok true)
  )
)

;; Check before allowing new loans
(asserts! (not (var-get emergency-paused)) ERR_CONTRACT_PAUSED)
```

13.3 Communication Plan

Incident Severity Levels:

SEV-1: Critical

```
Examples:
- Validators below threshold (<10 online)
```

```
- Funds at risk
- Security breach

Communication:
- Immediate: Discord announcement (within 15 minutes)
- Website banner
- Twitter/X announcement
- Email to all users
- Update every hour until resolved

Stakeholders:
- All users
- Validator operators
- Core team
- Investors/partners
- Media (if funds at risk)
```

SEV-2: Major

```
Examples:
- Signing delays (>10 minutes)
- 3-5 validators down (but above threshold)
- Bitcoin reorg affecting loans

Communication:
- Discord announcement (within 1 hour)
- Status page update
- Email to affected users
- Update every 4 hours

Stakeholders:
- Affected users
- Validator operators
- Core team
```

SEV-3: Minor

```
Examples:
- Single validator down
- Planned maintenance
- Non-critical bugs

Communication:
- Status page update
- Discord post
- Regular status updates

Stakeholders:
```

```
- Core team
- Validator operators (if relevant)
```

Template: SEV-1 Incident:

```
INCIDENT ALERT [SEV-1]

Title: [Brief description]
Time Detected: [Timestamp UTC]
Status: INVESTIGATING / IDENTIFIED / MONITORING / RESOLVED

Impact:
- [What is affected]
- [How many users/loans affected]
- [Are funds at risk? YES/NO]

Actions Taken:
- [Step 1]
- [Step 2]

Next Steps:
- [What we're doing now]
- [ETA for next update]

User Action Required:
- [What users should do, if anything]

We apologize for any inconvenience. Updates will be posted every hour
until resolution.

[Link to status page for updates]
```

Appendix A: FROST Algorithm Details

Mathematical Specification

Setup Parameters: - Group: secp256k1 (Bitcoin's elliptic curve) - Generator: G - Order: n (large prime) - Participants: n = 15 - Threshold: t = 10

Key Generation (DKG):

```
Each participant i (i = 1..15):
1. Choose random secret: s_i  Z
2. Create polynomial of degree t-1:
   f_i(x) = s_i + a_{i,1}x + a_{i,2}x² + ... + a_{i,t-1}x^(t-1)
3. Compute commitments: C_{i,j} = a_{i,j} * G for j = 0..t-1
4. Broadcast commitments to all participants
```

```
5. Send share to each participant j: f_i(j) (encrypted)

Each participant j verifies received shares:
For each share f_i(j) from participant i:
  Check: f_i(j) * G =  (C_{i,k} * j^k) for k = 0..t-1

Final key share for participant j:
  s_j =  f_i(j) for i = 1..15

Public key:
  P =  C_{i,0} =  (s_i * G) for i = 1..15
```

Signing:

```
Message: m
Participants: S  {1..15} where |S| >= 10

Nonce Generation (each participant i  S):
1. Choose random: (d_i, e_i)
2. Compute: (D_i, E_i) = (d_i * G, e_i * G)
3. Broadcast: (D_i, E_i)

Signature Generation:
1. Compute binding values:  _i = H(i, m, {D_j, E_j})
2. Compute group commitment: R =  (D_i +  _i * E_i) for i  S
3. Compute challenge: c = H(R || P || m)
4. Compute Lagrange coefficient for each i  S:
    _i =  (j / (j - i)) for j  S, j  i
5. Each participant i creates partial signature:
   z_i = d_i + (e_i *  _i) + ( _i * s_i * c)
6. Combine: z =  z_i for i  S
7. Final signature: (R, z)

Verification:
  z * G = R + c * P
```

Appendix B: Validator Hardware Specifications

Minimum Requirements:

```
CPU: 8 cores (3.0 GHz+)
RAM: 32 GB
Storage: 2 TB NVMe SSD
Network: 100 Mbps symmetric
Uptime: 99.5%
Operating System: Ubuntu 22.04 LTS
```

```
Estimated Cost: $200-300/month (cloud)
                $2,000-3,000 (self-hosted hardware)
```

Recommended Setup:

```
CPU: 16 cores (3.5 GHz+)
RAM: 64 GB
Storage: 4 TB NVMe SSD (RAID 1)
Network: 1 Gbps symmetric, redundant connections
UPS: Battery backup
HSM: YubiHSM 2 or equivalent
Uptime: 99.9%

Estimated Cost: $400-600/month (cloud)
                $5,000-8,000 (self-hosted hardware)
```

Geographic Distribution: - No more than 3 validators in same data center - No more than 5 validators in same country - Minimum 3 continents represented - Diverse cloud providers (AWS, GCP, Azure, OVH, Hetzner, etc.)

Appendix C: Glossary

DKG (Distributed Key Generation): Protocol to generate threshold keys without trusted dealer

FROST: Flexible Round-Optimized Schnorr Threshold signatures

HSM: Hardware Security Module for secure key storage

Partial Signature: Signature fragment created by single validator using key share

Schnorr Signature: Bitcoin's native signature scheme (post-Taproot)

Taproot: Bitcoin upgrade enabling efficient threshold signatures

Threshold Signature: Single signature created by M-of-N parties

Validator: Node operator who holds key share and participates in signing

VSS (Verifiable Secret Sharing): Technique ensuring shares distributed correctly

---

End of Technical Specification Document - Phase 2

Document Version: 1.0
Last Updated: January 12, 2026

This technical specification provides complete implementation details for Phase 2 with Bitcoin-native custody using threshold signatures.