vBTC V2 — Technical Specification

Protocol: VerifiedX Core (VFX) · vBTC Version 2

Cryptographic Core: FROST (Flexible Round-Optimized Schnorr Threshold Signatures) over secp256k1

Primary Layers: VFX (smart contracts, ledger, consensus) · Bitcoin (custody via Taproot MPC)

Status: Implementation complete, Mainnet & Testnet Live

Table of Contents

1. Introduction and Design Philosophy
2. System Architecture
3. Cryptographic Foundation: FROST
4. Distributed Key Generation (DKG)
5. Validator Registry and Byzantine Fault Tolerance
6. Contract Creation Flow
7. Token Transfer Protocol
8. Withdrawal Protocol
9. Dynamic Threshold and Recovery Hardening
10. Withdrawal Cancellation Governance
11. FROST Ceremony Network Protocol
12. Consensus and State Machine Integration
13. Security Properties and Attack Surface Analysis
14. Protocol Parameters Reference
15. API Reference

---

1. Introduction and Design Philosophy

1.1 What vBTC V2 Solves

vBTC V1 used arbiters — a small, privileged set of trusted parties that co-signed withdrawal transactions. This is a weak security model: it collapses to n-of-n multi-sig in the adversarial case, and any arbiter collusion or compromise can drain the entire vault. V2 eliminates arbiters entirely.

vBTC V2 is a **non-custodial, validator-threshold-secured tokenized Bitcoin** system built on four hard guarantees:

1. **The full Bitcoin private key never exists.** Not in memory, not on disk, not even transiently during signing. FROST threshold signatures mean the key is only ever held in distributed shares across N validators; no coalition smaller than the quorum can reconstruct or impersonate the key.

2. **The contract owner cannot unilaterally withdraw funds.** The Taproot address is controlled by the FROST group key — a mathematical aggregate of validator key shares. Even the person who created the contract cannot bypass the validator quorum.

3. **Funds are always recoverable.** The dynamic threshold system with a 24-hour safety gate ensures that if validators go offline en masse, the threshold adjusts down to match reality after 24 hours — preventing permanent fund lockup.

4. **No single custodian or admin key.** Validator admission and heartbeating are tracked on-chain. The registry is derived from blockchain transactions, not a centralized database. All privileged operations require multi-validator threshold agreement.

Trust model: The system eliminates single-custodian risk and admin keys. All privileged operations — withdrawals, key generation, validator management — require multi-validator threshold agreement enforced by the FROST protocol and on-chain transaction validation. Liveness depends on a sufficient subset of validators remaining online and reachable; the protocol includes adaptive threshold mechanisms and a 24-hour safety gate to maintain availability under partial validator failure.

1.2 Design Decisions That Differ from V1

Property	V1	V2
Key management	Arbiters hold key shares (Shamir)	FROST threshold signatures (no reconstruction)
Signing	n-of-n arbiter multi-sig	(t,n) FROST Schnorr — 51% threshold
Bitcoin address type	Legacy or P2SH	Taproot (BIP341, key-path spend)
Privacy on Bitcoin	Multi-sig visible on-chain	Indistinguishable from single-sig
Validator set	Static, privileged	Dynamic, derived from on-chain TXs
Recovery	Manual arbitration	Automated threshold adjustment (24h gate)
Cross-chain bridge	N/A	Base L2 via ERC-20 vBTCb contract

1.3 Threshold Quick-Reference

The vBTC V2 protocol uses different threshold values for different operations. Each threshold is chosen for its specific security/liveness trade-off. This table summarizes all thresholds in one place; detailed rationale follows in subsequent sections.

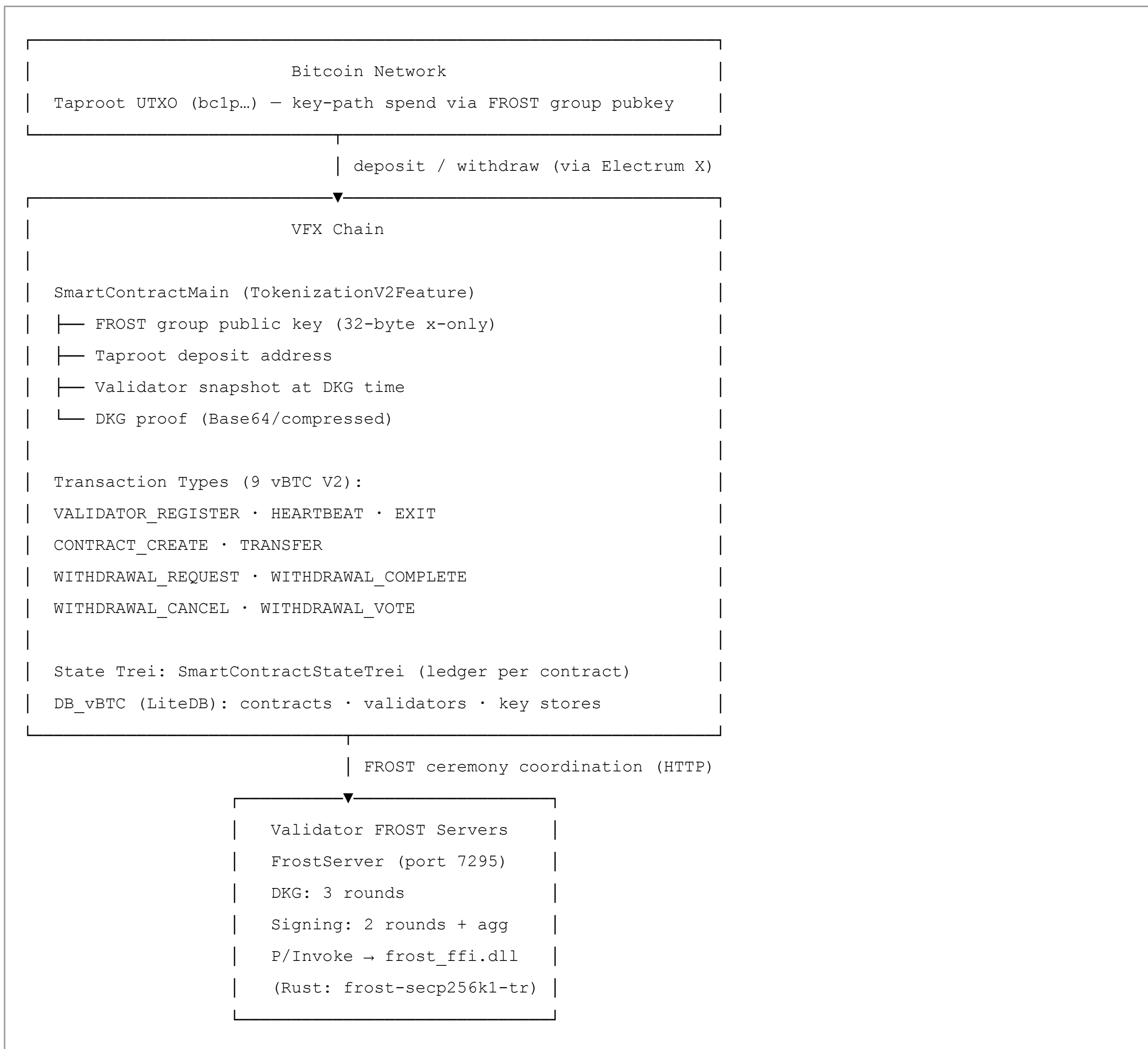
Operation	Threshold	Applies To	Rationale
FROST DKG ceremony	75% of active validators	Validators at DKG time (§4.6, §5.2)	Strong supermajority for irreversible key generation
FROST withdrawal signing	51% of DKG snapshot validators	DKG snapshot set (§5.2, §8)	Balances liveness against safety
Withdrawal cancellation vote	75% of active validators	Current active set (§5.2, §10)	Supermajority prevents minority griefing
24-hour safety gate	Delays threshold relaxation for 7,200 blocks	Threshold adjustment (§9)	Prevents instant exploitation during validator outage
2-of-3 recovery floor	66.67% (2 of 3)	When exactly 3 validators remain (§9)	Mathematical minimum for recovery
Base mint (`mintWithProof`)	$\lceil 2n/3 \rceil \approx 67\%$	Base validator set (see vBTCb spec §4.2)	Classical BFT — tolerates $\lfloor n/3 \rfloor$ Byzantine validators
Base upgrade (UUPS)	$\lceil 2n/3 \rceil \approx 67\%$	Base validator set (see vBTCb spec §4.7)	Same safety guarantee as minting
Base validator removal (n > 5)	51%	Base validator set (see vBTCb spec §4.6)	Prioritizes liveness over safety

> ***Note on validator sets:*** *The term "validators" appears throughout this document in several contexts.* ***DKG snapshot validators*** *are the validators who participated in the FROST key generation ceremony for a specific contract — this set is immutable after DKG.* ***Active validators*** *are validators with a recent heartbeat on the VFX chain (scanned from the last 1,000 blocks).* ***Base validators*** *are the Ethereum addresses registered in the vBTCb.sol contract on Base L2. These sets may differ as validators join and leave over time, and each threshold applies to its specific set.*

---

2. System Architecture

2.1 Layer Map



2.2 Software Stack

The implementation spans three layers:

C# Application (ReserveBlockCore)

- `VBTCService.cs` — orchestrates all vBTC operations
- `FrostMPCService.cs` — DKG and signing ceremony coordinator
- `BitcoinTransactionService.cs` — unsigned TX construction and broadcast via Electrum X
- `VBTCThresholdCalculator.cs` — dynamic threshold math
- `VBTCValidatorRegistry.cs` — decentralized, block-scan-based registry
- `FrostNative.cs` — P/Invoke bindings to the Rust FFI

Rust FFI (`frost_ffi.dll`) Compiled from the `plonk-ffi` crate. Wraps the ZCash Foundation's `frost-secp256k1-tr` crate and exposes a C ABI callable from .NET via P/Invoke.

Solidity (`vBTCb.sol`) ERC-20 deployed behind an ERC1967 UUPS proxy on Base L2. Validator-governed mint and burn functions.

---

3. Cryptographic Foundation: FROST

3.1 Why FROST

FROST (Flexible Round-Optimized Schnorr Threshold Signatures) was published by Komlo & Goldberg (2020, revised 2021) and is maintained by the ZCash Foundation. It achieves a (t,n) threshold Schnorr scheme with two key properties not present in earlier threshold signature designs:

- **Concurrency safety:** Earlier Schnorr threshold protocols (e.g., MuSig2 round-1) were vulnerable to Wagner's attack when multiple signing sessions ran concurrently. FROST's two-round design is provably secure under concurrent session assumptions.
- **Optimal round complexity:** 2 rounds for signing versus 6–9 for threshold ECDSA, and the key never needs to be reconstructed — even for signing.

3.2 Why Not Threshold ECDSA

Bitcoin historically required ECDSA. Taproot (BIP340/BIP341, activated November 2021) introduced native Schnorr signatures on mainnet. FROST outputs a standard 64-byte Schnorr signature that is valid as a Bitcoin key-path Taproot witness — it looks identical to a single-signer transaction on-chain. This means:

- Lower transaction fees (Taproot inputs are ~57.5 vBytes vs ~148 vBytes for P2PKH)
- No script is published on-chain (key-path spend), preserving privacy
- No trusted dealer required at key generation

3.3 Cryptographic Primitives

Primitive	Instance
Elliptic curve	secp256k1
Hash function	SHA-256 (via Taproot tagged hashes per BIP340)
Commitment scheme	Pedersen commitments over secp256k1
Secret sharing	Shamir Secret Sharing over Zq (share distribution only — key never reconstructed)
Signature scheme	Schnorr (64-byte `(R, s)`) — BIP340 compatible
Bitcoin address	Taproot P2TR, x-only public key

3.4 FROST FFI Interface

The C# layer communicates with FROST via six P/Invoke calls. All memory allocated in Rust is freed by the Rust allocator via `frost_free_string`, preventing cross-runtime heap corruption:

```csharp
// ReserveBlockCore/Bitcoin/FROST/FrostNative.cs

[DllImport("frost_ffi", CallingConvention = CallingConvention.Cdecl)]
public static extern int frost_dkg_round1_generate(
    ushort participantId,
    ushort maxSigners,
    ushort minSigners,
    out IntPtr outCommitment,
    out IntPtr outSecretPackage);

[DllImport("frost_ffi", CallingConvention = CallingConvention.Cdecl, CharSet = CharSet.Ansi)]
public static extern int frost_dkg_round2_generate_shares(
    [MarshalAs(UnmanagedType.LPStr)] string secretPackage,
    [MarshalAs(UnmanagedType.LPStr)] string commitmentsJson,
    out IntPtr outSharesJson,
    out IntPtr outRound2Secret);

[DllImport("frost_ffi", CallingConvention = CallingConvention.Cdecl, CharSet = CharSet.Ansi)]
public static extern int frost_dkg_round3_finalize(
    [MarshalAs(UnmanagedType.LPStr)] string round2SecretPackage,
    [MarshalAs(UnmanagedType.LPStr)] string round1PackagesJson,
    [MarshalAs(UnmanagedType.LPStr)] string round2PackagesJson,
    out IntPtr outGroupPubkey,
    out IntPtr outKeyPackage,
    out IntPtr outPubkeyPackage);

[DllImport("frost_ffi", CallingConvention = CallingConvention.Cdecl, CharSet = CharSet.Ansi)]
public static extern int frost_sign_round1_nonces(
    [MarshalAs(UnmanagedType.LPStr)] string keyPackageJson,
    out IntPtr outNonceCommitment,
    out IntPtr outNonceSecret);

[DllImport("frost_ffi", CallingConvention = CallingConvention.Cdecl, CharSet = CharSet.Ansi)]
public static extern int frost_sign_round2_signature(
    [MarshalAs(UnmanagedType.LPStr)] string keyPackageJson,
    [MarshalAs(UnmanagedType.LPStr)] string nonceSecret,
    [MarshalAs(UnmanagedType.LPStr)] string nonceCommitmentsJson,
    [MarshalAs(UnmanagedType.LPStr)] string messageHashHex,
    out IntPtr outSignatureShare);

[DllImport("frost_ffi", CallingConvention = CallingConvention.Cdecl, CharSet = CharSet.Ansi)]
public static extern int frost_sign_aggregate(
    [MarshalAs(UnmanagedType.LPStr)] string signatureSharesJson,
    [MarshalAs(UnmanagedType.LPStr)] string nonceCommitmentsJson,
    [MarshalAs(UnmanagedType.LPStr)] string messageHashHex,
    [MarshalAs(UnmanagedType.LPStr)] string pubkeyPackageJson,
    out IntPtr outSchnorrSignature);
```

Error codes are typed constants:

```
public const int SUCCESS               =  0;
public const int ERROR_NULL_POINTER    = -1;
public const int ERROR_INVALID_UTF8    = -2;
public const int ERROR_SERIALIZATION   = -3;
public const int ERROR_CRYPTO_ERROR    = -4;
public const int ERROR_INVALID_PARAMETER = -5;
```

---

4. Distributed Key Generation (DKG)

4.1 DKG Overview

FROST DKG is a three-round protocol that, given N participants and threshold t, produces:

- **N key packages** — one per validator; contains that validator's secret signing share
- **One group public key** — the aggregate Taproot key, from which the Bitcoin deposit address is derived
- **N verification shares** — used to verify that each signing participant contributed honestly during signing

No party ever possesses the full private key. The private key effectively exists only as an implicit mathematical aggregate that can be exercised only through the threshold protocol.

4.2 DKG Round 1: Commitment Exchange

Each validator i independently:

1. Samples a random secret polynomial $f_i(x)$ of degree `t-1` over Zq (the secp256k1 scalar field)
2. Computes a vector of Pedersen commitments: $C_i = \{f_i(0) \cdot G, a_1 \cdot G, \ldots, a_{t-1} \cdot G\}$ where G is the secp256k1 generator
3. Returns: $(C_i,$ `secretPackage_i`$)$

The commitment vector proves, without revealing any secret, what polynomial was chosen. The coordinator aggregates all N commitment vectors and broadcasts them to all validators.

```
Validator call:
  frost_dkg_round1_generate(participantId=i, maxSigners=N, minSigners=t)
  → (commitment_i_json, secretPackage_i)

Coordinator:
  Broadcasts {commitment_1, commitment_2, …, commitment_N} to all validators
```

4.3 DKG Round 2: Secret Share Distribution

Each validator i:

1. For each other validator j ≠ i, evaluates its polynomial: $s_{ij} = f_i(j)$
2. Sends the secret share s_{ij} to validator j (encrypted channel — HTTPS peer communication via FrostServer)

Each validator j collects shares from all other validators. After this round, each validator j holds:

- $\{s_{1j}, s_{2j}, \ldots, s_{nj}\}$ — shares from all other validators
- Its own round-2 secret package

```
// FrostNative.cs helper wrapper
var (sharesJson, round2Secret, err) = FrostNative.DKGRound2GenerateShares(
    secretPackage: secretPackage_i,
    commitmentsJson: allCommitmentsJson  // broadcast from Round 1
);
```

4.4 DKG Round 3: Finalization and Group Key Derivation

Each validator j:

1. Verifies each received share s_{ij} against the Round 1 commitment C_i:

   ```
   s_ij · G == Σ_k (j^k · C_i[k])   for k = 0..t-1
   ```

   This is the standard Pedersen commitment verification. If any share fails, the ceremony aborts.

2. Computes its final key share: $\sigma_j = \Sigma_i s_{ij} \bmod q$

3. Derives the group public key: $Y = \Sigma_i f_i(0) \cdot G = \Sigma_i C_i[0]$

The group public key Y is identical for all validators (it's computed from the publicly broadcast commitments). This is the secp256k1 point from which the Bitcoin Taproot address is derived.

```
var (groupPubkey, keyPackage, pubkeyPackage, err) = FrostNative.DKGRound3Finalize(
    round2SecretPackage: round2Secret_j,
    round1PackagesJson: allRound1Commitments,
    round2PackagesJson: allRound2Shares
);
// groupPubkey: 32-byte hex x-only public key (BIP340 format)
// keyPackage: validator j's private signing share (stored encrypted)
// pubkeyPackage: full public key package (needed for signature aggregation)
```

4.5 Taproot Address Derivation

From the FROST group public key (32-byte x-only), the Bitcoin Taproot address is:

```
internalKey = groupPubkey  (x-only, 32 bytes)
outputKey   = internalKey + hash_taptweak(internalKey) · G   (BIP341 tweak)
address     = bech32m("bc", witness_version=1, outputKey)
```

For key-path spend (no script), the tweak is `hash_taptweak(internalKey || "")`. The resulting `bc1p`… address is indistinguishable on-chain from any ordinary single-sig Taproot address.

4.6 DKG Quorum and Ceremony Safety

- **Quorum required:** 75% of active validators must complete all 3 rounds
- **Ceremony TTL:** 1 hour — prevents memory leaks from abandoned ceremonies
- **Max concurrent ceremonies:** 100 globally, 1 per owner address (anti-spam)
- **Any verification failure:** entire ceremony aborts; no partial state persists

---

5. Validator Registry and Byzantine Fault Tolerance

5.1 Decentralized Registry via Block Scanning

There is no central database of validators. The validator set is derived on-demand from the last 1,000 blocks of the VFX chain by scanning for three transaction types:

Transaction Type	Effect
`VBTC_V2_VALIDATOR_REGISTER`	Mark validator as active
`VBTC_V2_VALIDATOR_HEARTBEAT`	Confirm validator is online; update IP
`VBTC_V2_VALIDATOR_EXIT`	Mark validator as inactive

At ~12 seconds per block, 1,000 blocks spans approximately 3.3 hours — covering two full heartbeat cycles (heartbeat interval: 500 blocks ≈ 1.7 hours). Any validator that fails to heartbeat within two cycles falls out of the active set automatically.

```
// ReserveBlockCore/Bitcoin/Services/VBTCValidatorRegistry.cs
public const int SCAN_WINDOW = 1000;

public static List<VBTCValidator> GetActiveValidators()
{
    var currentHeight = Globals.LastBlock.Height;

    lock (_cacheLock)
    {
        if (_cachedValidators != null && _cachedAtHeight == currentHeight)
            return _cachedValidators;  // Cached per block — at most one scan per new block
    }

    var validators = ScanBlocks(currentHeight);

    lock (_cacheLock)
    {
        _cachedValidators = validators;
        _cachedAtHeight = currentHeight;
    }

    return validators;
}
```

The scan processes each block's transactions:

```
for (long h = scanFrom; h <= currentHeight; h++)
{
    var block = BlockchainData.GetBlockByHeight(h);
    foreach (var tx in block.Transactions)
    {
        if (tx.TransactionType == TransactionType.VBTC_V2_VALIDATOR_REGISTER)
            ProcessRegister(tx, h, validatorMap);
        else if (tx.TransactionType == TransactionType.VBTC_V2_VALIDATOR_HEARTBEAT)
            ProcessHeartbeat(tx, h, validatorMap);
        else if (tx.TransactionType == TransactionType.VBTC_V2_VALIDATOR_EXIT)
            ProcessExit(tx, h, validatorMap);
    }
}
```

State reconstruction is deterministic — any node scanning the same block range produces the same validator set.

5.2 Byzantine Fault Tolerance

The vBTC V2 system operates at two distinct Byzantine fault tolerance levels:

DKG (Key Generation) — 75% quorum

FROST DKG requires a strict supermajority. With N validators and $t = \lceil 0.75N \rceil$ required participants, the system can tolerate $f = N - t = \lfloor 0.25N \rfloor$ Byzantine failures (validators that are offline, malicious, or partition-ed) during key generation. This is consistent with classic BFT guarantees: a system tolerating f faults requires $n \geq 3f + 1$ participants in full Byzantine settings, but since DKG is a setup protocol rather than a live consensus protocol, the 75% supermajority is appropriate for the stronger security property.

Signing (Withdrawal) — 51% threshold

FROST signing requires $t = \lceil 0.51N \rceil$ validators. This allows the system to survive up to $f \approx 49\%$ validator failure rate during withdrawals. The 51% threshold balances liveness (funds are not locked by a minority going offline) against safety (an adversary controlling 50% minus one validator cannot steal funds).

Withdrawal Cancellation Governance — 75% vote

For a disputed or stuck withdrawal to be cancelled, 75% of validators must cast `VBTC_V2_WITHDRAWAL_VOTE` transactions in favor of cancellation. This supermajority prevents a minority coalition from maliciously cancelling legitimate withdrawals.

5.3 Validator Data Model

Each `VBTCValidator` record carries:

```
public string ValidatorAddress { get; set; }       // VFX address (ed25519 pubkey hash)
public string IPAddress { get; set; }              // IP for FROST ceremony HTTP calls
public string FrostPublicKey { get; set; }         // Validator's contribution pubkey
public string BaseAddress { get; set; }            // Derived Ethereum address (Base L2)
public long RegistrationBlockHeight { get; set; }
public long LastHeartbeatBlock { get; set; }
public bool IsActive { get; set; }
public string RegisterTransactionHash { get; set; }
public string? ExitTransactionHash { get; set; }
public long? ExitBlockHeight { get; set; }
```

The `BaseAddress` field is derived from the validator's VFX public key using `ValidatorEthKeyService.DeriveBaseAddressFromVfxPublicKey()`. This allows the same validator identity to participate in both VFX FROST ceremonies and Ethereum ECDSA attestation on Base.

---

6. Contract Creation Flow

6.1 End-to-End Flow

```
User
  |
  ├─ POST /vbtcapi/vbtc/InitiateMPCCeremony/{ownerAddress}
  |    Returns: ceremonyId
  |
  ├─ System queries VBTCValidatorRegistry.GetActiveValidators()
  |
  ├─ FROST DKG — 3 rounds across active validators
  |    Round 1: Each validator calls frost_dkg_round1_generate()
  |    Round 2: Each validator calls frost_dkg_round2_generate_shares()
  |    Round 3: Each validator calls frost_dkg_round3_finalize()
  |    Result: groupPubkey (x-only), keyPackage per validator, pubkeyPackage
  |
  ├─ Taproot address derived from groupPubkey
  |
  ├─ POST /vbtcapi/vbtc/CreateVBTCContract (or CreateVBTCContractRaw)
  |    Builds SmartContractMain with TokenizationV2Feature:
  |      DepositAddress, FrostGroupPublicKey, FrostPubkeyPackage,
  |      ValidatorAddressesSnapshot, DKGProof, RequiredThreshold=51
  |
  ├─ VBTC_V2_CONTRACT_CREATE transaction broadcast to VFX network
  |
  └─ After block confirmation:
       SmartContractStateTrei created
       VBTCContractV2 local record saved
       Deposit address ready to receive BTC
```

6.2 TokenizationV2Feature Contract State

The `TokenizationV2Feature` stored in the smart contract code (immutable after creation) contains:

```
DepositAddress             — bc1p…  (Taproot address)
FrostGroupPublicKey        — 32-byte hex (x-only, BIP340)
FrostPubkeyPackage         — JSON (needed for signature aggregation in future signing ceremonies)
ValidatorAddressesSnapshot — list of VFX addresses that participated in DKG
RequiredThreshold          — 51 (percent)
DKGProof                   — Base64-encoded, compressed proof of successful DKG
ProofBlockHeight           — Block at which DKG completed
```

6.3 Local VBTCContractV2 Record

Stored in `DB_vBTC` (LiteDB). The mutable withdrawal state fields are updated as operations progress:

```
public class VBTCContractV2
{
    public string SmartContractUID { get; set; }   // "txhash:unix_timestamp"
    public string OwnerAddress { get; set; }
    public string DepositAddress { get; set; }
    public decimal Balance { get; set; }           // Refreshed from Electrum every 5 min

    // FROST Data (immutable — copied from TokenizationV2Feature)
    public List<string> ValidatorAddressesSnapshot { get; set; }
    public string FrostGroupPublicKey { get; set; }
    public string? FrostPubkeyPackage { get; set; }
    public int RequiredThreshold { get; set; }      // 51
    public string DKGProof { get; set; }
    public long ProofBlockHeight { get; set; }

    // Threshold Adjustment Tracking
    public long LastValidatorActivityBlock { get; set; }
    public int TotalRegisteredValidators { get; set; }
    public int OriginalThreshold { get; set; }      // Always 51

    // Withdrawal State Machine (mutable)
    public VBTCWithdrawalStatus WithdrawalStatus { get; set; }
    public string? ActiveWithdrawalBTCDestination { get; set; }
    public decimal? ActiveWithdrawalAmount { get; set; }
    public string? ActiveWithdrawalRequestHash { get; set; }
    public long? WithdrawalRequestBlock { get; set; }
    public int ActiveWithdrawalFeeRate { get; set; }
    public List<VBTCWithdrawalHistory> WithdrawalHistory { get; set; }
}

public enum VBTCWithdrawalStatus
{
    None,
    Requested,
    Pending_BTC,
    Completed,
    Cancelled,
    Cancellation_Requested
}
```

---

7. Token Transfer Protocol

7.1 Two-Layer Balance Model

vBTC V2 uses a hybrid balance model because BTC deposits happen off-chain (on the Bitcoin network) while transfers happen on-chain (on the VFX ledger):

Contract Owner:

```
spendable = (BTC confirmed in Taproot deposit address via Electrum)
          + (total vBTC received via VFX transfers)
          - (total vBTC sent via VFX transfers)
          - (active withdrawal amount, if any)
```

Non-Owner Token Holders:

```
spendable = (total vBTC received via VFX transfers)
          - (total vBTC sent via VFX transfers)
```

Non-owners cannot spend from the Bitcoin deposit directly — they hold only ledger credit. This separation is important: the owner's Bitcoin balance is the authoritative source of truth for the total supply; the ledger tracks internal vBTC distribution.

7.2 Transfer Transaction Structure

A `VBTC_V2_TRANSFER` transaction is a zero-VFX-value, zero-BTC-movement transaction whose semantic payload lives in the `Data` field:

```
{
  "Function": "Transfer",
  "ContractUID": "abc123:1773285700",
  "FromAddress": "xAliceVFXAddress...",
  "ToAddress": "xBobVFXAddress...",
  "Amount": 0.5
}
```

The transaction is self-referential (from == to at the VFX level, or rather the sender signs it and the recipient is encoded in Data). The fee is paid in VFX by the sender. State is applied in `StateData.HandleVBTCV2Transfer()`, which appends a record to `SmartContractStateTrei.SCStateTreiTokenizationTXes`.

7.3 State Integrity

The State Trei is the canonical ledger. It is updated exactly once per confirmed block, deterministically, by all full nodes. A double-spend attempt (sending more than the ledger balance) is rejected at both the mempool level (by `TransactionValidatorService`) and the block-level validation (by `BlockTransactionValidatorService`).

---

8. Withdrawal Protocol

Withdrawal is a two-step operation: a user first declares their intent on-chain (creating a reservation), and then triggers the FROST signing ceremony to produce the Bitcoin transaction.

8.1 Step 1 — Withdrawal Request

Transaction type: `VBTC_V2_WITHDRAWAL_REQUEST`

Guard: Only one active withdrawal is permitted per (user address, contract) pair at a time. `VBTCContractV2.HasActiveWithdrawal()` enforces this:

```
public static bool HasActiveWithdrawal(string smartContractUID)
{
    var contract = GetContract(smartContractUID);
    return contract?.WithdrawalStatus == VBTCWithdrawalStatus.Requested ||
           contract?.WithdrawalStatus == VBTCWithdrawalStatus.Pending_BTC;
}
```

Transaction Data payload:

```
{
  "Function": "WithdrawalRequest",
  "ContractUID": "abc123:1773285700",
  "RequestorAddress": "xAliceVFXAddress...",
  "BTCAddress": "tb1q...",
  "Amount": 0.5,
  "FeeRate": 10
}
```

After the request transaction confirms, the contract state transitions:

```
WithdrawalStatus = Requested
ActiveWithdrawalBTCDestination = "tb1q..."
ActiveWithdrawalAmount = 0.5
ActiveWithdrawalRequestHash = <VFX transaction hash>
WithdrawalRequestBlock = <current block height>
ActiveWithdrawalFeeRate = 10
```

8.2 Step 2 — FROST Signing and BTC Broadcast

Transaction type: `VBTC_V2_WITHDRAWAL_COMPLETE`

This step coordinates a full FROST signing ceremony, builds and signs a Bitcoin transaction, and broadcasts it.

8.2.1 Unsigned Bitcoin Transaction Construction

The `BitcoinTransactionService.BuildUnsignedTaprootTransaction()` method:

1. Queries Electrum X for confirmed UTXOs at the Taproot deposit address
2. Selects UTXOs sufficient to cover `amount + fee`
3. Estimates the transaction virtual size (vBytes):

   ```
   vBytes ≈ (57.5 × inputCount) + (43 × outputCount) + 10.5
   fee     = vBytes × feeRateSatsPerVByte
   ```

4. Constructs:
   - **Input(s):** Selected Taproot UTXOs (no witness yet)
   - **Output 1:** `(amount - fee)` BTC to the user's Bitcoin destination address
   - **Output 2:** Any change returns to the deposit address (vault)
5. Returns the unsigned `NBitcoin.Transaction` object

The fee is deducted from the withdrawal amount, not added on top, so the user receives an exact net amount.

8.2.2 FROST Signing — Round 1 (Nonce Generation)

The coordinator (`FrostMPCService`) contacts each qualifying validator via HTTP (port 7295):

Each validator i:

1. Calls `frost_sign_round1_nonces(keyPackageJson)` to generate a fresh random nonce pair (d_i, e_i) — binding and hiding nonces per the FROST protocol
2. Returns the nonce commitment $(D_i, E_i) = (d_i \cdot G, e_i \cdot G)$

The coordinator aggregates all nonce commitments and broadcasts the full set to all participants.

Why fresh nonces matter: Reusing a nonce in Schnorr signatures reveals the private key. FROST's two-nonce commitment scheme ((D, E)) and the binding computation in Round 2 prevent a rogue-key attack even when nonces might be reused across sessions by a malicious participant.

8.2.3 FROST Signing — Round 2 (Signature Share Generation)

Each validator i:

1. Receives the aggregated nonce commitments from all other participants
2. Computes the transaction sighash per BIP341:

```
sighash = TapSighash(tx, inputIndex, SIGHASH_DEFAULT)
```

3. Calls `frost_sign_round2_signature(keyPackageJson, nonceSecret, nonceCommitmentsJson, sighash)` to produce a partial signature share z_i
4. Returns z_i to the coordinator

Internally, FROST computes:

```
ρᵢ = H(i, message, {(D₁,E₁), …, (Dₙ,Eₙ)})   — binding factor
Rᵢ = Dᵢ + ρᵢ · Eᵢ                            — per-participant nonce
R   = Σᵢ Rᵢ                                    — group nonce (coordinator aggregates)
c   = H(R, Y, message)                           — Schnorr challenge (Fiat-Shamir)
zᵢ  = dᵢ + ρᵢ·eᵢ + λᵢ·σᵢ·c                    — partial signature
```

where λ_i is the Lagrange interpolation coefficient for participant i, and σ_i is participant i's key share.

8.2.4 Signature Aggregation

The coordinator calls `frost_sign_aggregate()`:

```
var (schnorrSignature, err) = FrostNative.SignAggregate(
    signatureSharesJson: allSharesJson,
    nonceCommitmentsJson: allNonceCommitmentsJson,
    messageHash: sighashHex,
    groupPubkey: pubkeyPackageJson
);
```

Aggregation computes:

```
z   = Σᵢ zᵢ   (sum of all partial signatures)
sig = (R, z)  — 64-byte Schnorr signature
```

The aggregator verifies: $z \cdot G == R + c \cdot Y$, which holds if and only if all z_i were honestly computed. A dishonest participant producing an invalid share causes aggregation to fail.

8.2.5 Taproot Witness Attachment and Broadcast

The aggregated 64-byte Schnorr signature is attached to the Bitcoin transaction as a Taproot key-path witness (a witness stack with exactly one item):

```
witness[0] = schnorrSignature (64 bytes)
```

No public key appears in the witness — the verifying node derives the output key from the previous transaction's scriptPubKey. This is what makes Taproot key-path transactions indistinguishable from single-sig on-chain.

The signed transaction is broadcast to Bitcoin via Electrum X. The resulting transaction ID is recorded in the `VBTC_V2_WITHDRAWAL_COMPLETE` VFX transaction.

State transition after completion:

```
WithdrawalStatus  = Pending_BTC       (awaiting Bitcoin confirmations)
LastValidatorActivityBlock = currentBlock
```

Once BTC confirmation is detected:

```
WithdrawalStatus  = Completed
```

8.3 Withdrawal Timeline Example

```
Block N:     User sends VBTC_V2_WITHDRAWAL_REQUEST
             Status: Requested

Block N+1:   Request confirmed on VFX
             Validators query active set

Blocks N+1..N+K: FROST signing ceremony (2 rounds + aggregation)
             ~seconds to minutes depending on validator latency

After signing: Bitcoin TX broadcast to network
             VFX VBTC_V2_WITHDRAWAL_COMPLETE transaction broadcast
             Status: Pending_BTC

~10-60 min:  Bitcoin TX confirms (1+ blocks at chosen fee rate)
             Status: Completed
```

---

9. Dynamic Threshold and Recovery Hardening

9.1 The Liveness Problem

Any system where N parties jointly control funds faces a liveness risk: if enough parties go offline, funds become inaccessible. The naive solution (reduce the threshold immediately) creates an attack surface: an adversary can temporarily take validators offline, wait for the threshold to drop, and then conduct a hostile signing with fewer validators.

vBTC V2's solution is a **24-hour safety gate** before any threshold reduction takes effect.

9.2 The Algorithm

`VBTCThresholdCalculator.CalculateAdjustedThreshold()` takes four inputs: the count of validators registered at DKG time, the count currently active, the block height of the last successful activity, and the current block height.

```
public static int CalculateAdjustedThreshold(
    int totalRegisteredValidators,   // Snapshot at DKG time
    int activeValidators,            // Currently online
    long lastActivityBlock,          // Last successful withdrawal
    long currentBlock)
{
    // SAFETY GATE: 24 hours = 7,200 VFX blocks (300 blocks/hour × 24)
    long blocksSinceActivity = currentBlock - lastActivityBlock;
    decimal hoursSinceActivity = (decimal)blocksSinceActivity / BLOCKS_PER_HOUR;

    if (hoursSinceActivity < SAFETY_GATE_HOURS)
        return ORIGINAL_THRESHOLD;   // 67% until gate expires

    // Gate has expired — adjust proportionally
    decimal availablePercentage = ((decimal)activeValidators / totalRegisteredValidators) * 100m;
    decimal adjustedThreshold = Math.Min(ORIGINAL_THRESHOLD, availablePercentage + SAFETY_BUFFER_PERCENT);

    // SPECIAL CASE: 2-of-3 rule
    if (activeValidators == MINIMUM_VALIDATORS_ABSOLUTE)  // 3 validators
    {
        decimal twoOfThree = ((decimal)MINIMUM_REQUIRED_OF_THREE / MINIMUM_VALIDATORS_ABSOLUTE) * 100m;
        adjustedThreshold = Math.Max(adjustedThreshold, twoOfThree);
    }

    return (int)Math.Ceiling(adjustedThreshold);
}
```

Protocol constants:

```
private const int ORIGINAL_THRESHOLD       = 67;   // 67% — BFT supermajority; held until safety gate expires
private const int SAFETY_BUFFER_PERCENT    = 10;   // +10% above availability
private const int SAFETY_GATE_HOURS        = 24;   // Hours before adjustment
private const int BLOCKS_PER_HOUR         = 300;   // 12-second VFX blocks
private const int BLOCKS_PER_DAY          = 7200;
private const int MINIMUM_VALIDATORS_ABSOLUTE = 3;
private const int MINIMUM_REQUIRED_OF_THREE   = 2; // 2-of-3 if only 3 remain
```

9.3 Recovery Scenario Walkthrough

Scenario: A contract was created with 300 validators. Due to a network event, all but 3 go offline.

```
T=0:    300 validators → 3 remain
        ORIGINAL_THRESHOLD = 67% ≈ 201 of 300 required
        Result: withdrawals blocked (only 3 available)

T=0 to T=24h (0 to 7,200 blocks):
        Safety gate ACTIVE
        Any withdrawal attempt: "Insufficient validators for threshold"
        This window prevents instant exploitation

T=24h (block 7,201 past last activity):
        Safety gate EXPIRED
        availablePercentage = (3 / 300) × 100 = 1%
        adjustedThreshold   = min(67%, 1% + 10%) = 11%
        2-of-3 rule applies: max(11%, 66.67%) = 66.67%
        required            = ceil(66.67% × 3) = 2
        Result: 2-of-3 validators can authorize withdrawal □
```

After the 24-hour window, funds are recoverable as long as at least 2 of the 3 remaining validators are online. This is a mathematical floor: the system guarantees recovery as long as any 2 validators survive.

9.4 Required Validator Count Computation

```
public static int CalculateRequiredValidators(int threshold, int availableValidators)
{
    if (availableValidators == MINIMUM_VALIDATORS_ABSOLUTE)
        return MINIMUM_REQUIRED_OF_THREE;  // Hard 2-of-3 rule

    int required = (int)Math.Ceiling(availableValidators * (threshold / 100.0));
    return Math.Max(1, required);  // At least 1 if any are online
}
```

The `LastValidatorActivityBlock` field on `VBTCContractV2` is updated every time a FROST signing completes, resetting the 24-hour safety gate countdown.

---

10. Withdrawal Cancellation Governance

If a withdrawal gets stuck — for example, the Bitcoin fee rate was too low and the transaction is never mined — the system provides a governance mechanism to cancel it.

10.1 Cancellation Flow

1. **Request:** Any party may submit `VBTC_V2_WITHDRAWAL_CANCEL`:

   ```
   WithdrawalStatus → Cancellation_Requested
   ```

2. **Vote:** Each validator submits `VBTC_V2_WITHDRAWAL_VOTE` with their judgment. Votes are stored in `VBTCWithdrawalCancellation`.

3. **Threshold:** 75% of validators must vote to cancel. This supermajority prevents minority manipulation.

4. **Resolution:** On reaching 75%, `StateData.HandleVBTCV2WithdrawalVote()` applies:

```
WithdrawalStatus → Cancelled
ActiveWithdrawal* fields → cleared
```

The user may then initiate a new withdrawal with a higher fee rate.

The strict 75% threshold mirrors the DKG quorum, ensuring that cancellation requires the same level of consensus as key generation. A 51%-quorum cancellation would allow a bare majority to grief users; a 100% requirement would allow a single validator to veto forever.

---

11. FROST Ceremony Network Protocol

11.1 FrostServer Architecture

Every validator node runs a dedicated HTTP server (`FrostServer`) that listens on two ports:

Port	Protocol	Purpose
7295 (mainnet)	HTTP	FROST ceremony coordination (DKG start, share exchange, nonce collection)
7296 (mainnet)	HTTPS	Same endpoints over TLS (self-signed certificate, 2048-bit RSA, SHA-256)

The server is started on process startup if `Globals.IsFrostValidator == true`. A background cleanup loop runs every 5 minutes to evict expired sessions from in-memory storage, preventing unbounded memory growth.

```
// FrostServer.cs — background session cleanup
private static async Task SessionCleanupLoop()
{
    while (true)
    {
        await Task.Delay(TimeSpan.FromMinutes(5));
        FrostSessionStorage.CleanupOldSessions();
    }
}
```

The coordinator role (the node that orchestrates a ceremony) can be any VFX node — it does not need to be a validator. The coordinator only makes HTTP calls and orchestrates data exchange; it never touches private key material. Private signing happens exclusively inside each validator's local FROST native library invocation.

11.2 Validator Reachability Probing

Before initiating a DKG or signing ceremony, the coordinator probes all registered validators to determine which ones are actually reachable on the FROST port. This is critical: without probing, the coordinator might attempt a ceremony with N validators, fail, and leave the user confused — when in reality only a subset are online.

The probe uses a dedicated `HttpClient` with a 5-second timeout (vs. the ceremony client's 30-second timeout):

```
private static readonly HttpClient _probeHttpClient = new HttpClient
{
    Timeout = TimeSpan.FromSeconds(5)
};

public static async Task<List<VBTCValidator>> ProbeValidatorReachability(
    List<VBTCValidator> validators)
{
    var reachable = new ConcurrentBag<VBTCValidator>();

    var tasks = validators.Select(async validator =>
    {
        // Defensive IP validation before any HTTP call
        if (!InputValidationHelper.ValidateValidatorIPAddress(validator.IPAddress, out _))
            return;

        var url = $"http://{validator.IPAddress}:{Globals.FrostValidatorPort}/health";
        var response = await _probeHttpClient.GetAsync(url);

        if (response.IsSuccessStatusCode)
        {
            var body = await response.Content.ReadAsStringAsync();
            if (body.Contains("\"Success\":true"))
                reachable.Add(validator);
        }
    });

    await Task.WhenAll(tasks);
    return reachable.ToList();
}
```

All probes run in parallel. Only validators that respond with `{"Success":true}` to the `/health` endpoint are included in the ceremony participant list. This ensures that the threshold check is computed against actually-reachable validators, not the theoretical active set.

11.3 DKG Ceremony Coordination — HTTP Flow

The coordinator orchestrates DKG across five phases, with progress reporting (0–100%):

```
Phase 1 — DKG Start Broadcast (0→10%)
  POST http://{validatorIP}:7295/frost/dkg/start
  Body: { SessionId, CeremonyId, LeaderAddress, Timestamp, LeaderSignature,
          ParticipantAddresses[], RequiredThreshold }
  Guard: LeaderSignature = Sign(leaderKey, "$sessionId.$leaderAddr.$timestamp")
  Quorum: ≥ requiredCount validators must accept before proceeding

Phase 2 — Round 1 Collection (10→40%)
  GET http://{validatorIP}:7295/frost/dkg/round1/{sessionId}
  Response: { Success, SessionId, Commitments: { "vfxAddress": "commitmentData" } }
  Result: dictionary of {validatorAddress → commitmentJSON}

Phase 3 — Round 2 Share Distribution (40→65%)
  POST http://{validatorIP}:7295/frost/dkg/round2/{sessionId}
  Body: all Round 1 commitments (JSON)
  Response: { Success, GeneratedShares: {...} }

  Coordinator then redistributes all shares to every validator:
  POST http://{validatorIP}:7295/frost/dkg/shares/{sessionId}
  Body: { LeaderAddress, Timestamp, LeaderSignature, AllGeneratedShares: {addr→shares} }

  Each validator extracts the shares meant for itself and calls DKGRound3Finalize locally.

Phase 4 — Round 3 Verification (65→85%)
  GET http://{validatorIP}:7295/frost/dkg/round3/{sessionId}
  Response: { Success, SessionId, Verifications: { "vfxAddress": true/false } }

  Any false means that validator's share verification failed — ceremony aborts.

Phase 5 — Result Collection (85→100%)
  GET http://{validatorIP}:7295/frost/dkg/result/{sessionId}
  Response: { Success, IsCompleted, GroupPublicKey, TaprootAddress, DKGProof }

  Coordinator collects from all validators and cross-checks GroupPublicKey agreement.
  If any two validators produce different group public keys — the ceremony is aborted.
```

Group key agreement check: This is a critical safety property. Because FROST DKG is deterministic given the same shared inputs, all validators should compute the same group public key. A mismatch indicates either a network attack or a protocol bug:

```
if (groupPublicKey != gpk)
{
    ErrorLogUtility.LogError(
        $"FROST DKG: Validators disagree on group public key! " +
        $"Expected: {groupPublicKey[..16]}..., Got: {gpk[..16]}...",
        "FrostMPCService.AggregateDKGResult");
    return null;  // Fail closed
}
```

After collecting the result, the coordinator validates the Taproot address format using NBitcoin:

```
var parsedAddress = BitcoinAddress.Create(taprootAddress, Globals.BTCNetwork);
if (parsedAddress is not TaprootAddress)
    return null;  // Reject non-Taproot addresses
```

11.4 Signing Ceremony Coordination — HTTP Flow

```
Phase 1 — Signing Start Broadcast
  POST http://{validatorIP}:7295/frost/sign/start
  Body: { SessionId, MessageHash (BIP341 sighash hex), SmartContractUID,
          LeaderAddress, Timestamp, LeaderSignature, SignerAddresses[],
          RequiredThreshold, WithdrawalRequestHash }

  WithdrawalRequestHash is included so validators can deduplicate signing
  requests for the same withdrawal (FIND-028 dedup guard).

Phase 2 — Round 1 Nonce Collection
  GET http://{validatorIP}:7295/frost/sign/round1/{sessionId}
  Response: { Success, SessionId, Nonces: { "vfxAddress": "nonceCommitmentJSON" } }

Phase 3 — Round 2 Share Collection
  POST http://{validatorIP}:7295/frost/sign/round2/{sessionId}
  Body: all nonce commitments (JSON)
  Response: { Success, ShareGenerated, SignatureShare, SessionId }

  The signature share is returned INLINE in the POST response to avoid
  a round-trip. The coordinator also maintains a fallback polling loop
  (GET /frost/sign/share/{sessionId}) with up to 5 retries and
  increasing delays (2s, 2s, 3s, 3s, 5s — 15 seconds total).

Phase 4 — Aggregation
  Coordinator calls FrostNative.SignAggregate() locally.
  No network call — the aggregation happens in the coordinator process.
```

11.5 FROST Identifier Mapping

FROST's native library identifies participants by `Identifier` — a secp256k1 scalar serialized as 32 bytes big-endian (64 hex characters). VFX identifies validators by their VFX address strings. The ceremony coordination layer maintains a deterministic mapping between these two namespaces.

Mapping rule: Sort validator VFX addresses alphabetically (ordinal/binary comparison), then assign 1-based sequential FROST Identifiers:

```
private static Dictionary<string, string> BuildAddressToFrostIdentifierMap(
    List<string> signerAddresses)
{
    // Sort MUST be identical between coordinator and each validator node
    var sorted = signerAddresses.OrderBy(a => a, StringComparer.Ordinal).ToList();
    var map = new Dictionary<string, string>();
    for (int i = 0; i < sorted.Count; i++)
    {
        // Participant index i+1 → 32-byte BE scalar → 64 hex chars
        map[sorted[i]] = (i + 1).ToString("x").PadLeft(64, '0');
    }
    return map;
}
```

Examples:

```
Participant 1 → "0000000000000000000000000000000000000000000000000000000000000001"
Participant 2 → "0000000000000000000000000000000000000000000000000000000000000002"
...
```

This mapping must be **identical** between the coordinator (`FrostMPCService.BuildAddressToFrostIdentifierMap`) and each validator (`FrostStartup.BuildAddressToIdentifierMap`). A mismatch causes aggregation to fail because the FROST library expects matching Identifier keys in the signature share and nonce commitment maps. The stored participant order from DKG is persisted in `FrostValidatorKeyStore.ParticipantOrderJson` and replayed during signing to ensure consistency even if the signer set changes between DKG and signing.

11.6 Pubkey Package Lookup (5-Tier Fallback Chain)

The coordinator needs the FROST pubkey package (which encodes all validators' public verification shares) to call `FrostNative.SignAggregate()`. Because different nodes have different local state, the coordinator tries five resolution strategies in order:

```
Tier 1: FrostValidatorKeyStore by SCUID
  — Works after SignStart has auto-updated the record with the real SC UID.

Tier 2: FrostValidatorKeyStore by ceremonyId (original DKG session GUID)
  — Fallback if the SCUID update hasn't propagated yet.

Tier 3: State Trei decompile → TokenizationV2Feature.FrostGroupPublicKey
          → FrostValidatorKeyStore.GetKeyPackageByGroupPublicKey()
  — Works on any full node; the FROST group public key is in the on-chain
    smart contract and can be used to index the local key store.

Tier 4: VBTCContractV2.FrostPubkeyPackage (local DB)
  — For non-validator wallet nodes that store the pubkey package
    from the DKG result in their local contract record.

Tier 5: Fetch from a live validator via /frost/key/pubkey/{id}
  — Last resort. Fetches from any responsive validator, then caches
    the result locally in VBTCContractV2 for future use.

If all five tiers fail → aggregation aborts (fail-closed).
```

---

12. Consensus and State Machine Integration

12.1 Transaction Lifecycle on VFX

Every vBTC V2 operation produces a VFX chain transaction. These transactions flow through the standard VFX mempool and block pipeline:

```
User/Validator
  |
  ├─ Build transaction (type, data payload, fee)
  ├─ Sign with VFX private key (ed25519)
  ├─ Submit to local node → TransactionPool
  |
Mempool Validation (TransactionValidatorService)
  ├─ Signature verification
  ├─ Nonce/sequence check
  ├─ VFX fee balance check
  ├─ Type-specific validation (balance, status guards, etc.)
  |
P2P Broadcast (SendTXMempool)
  |
Block Production
  ├─ Block-level re-validation (BlockTransactionValidatorService)
  ├─ State Trei application (StateData.cs)
  └─ Confirmed on-chain
```

12.2 Mempool Validation Rules by Transaction Type

`TransactionValidatorService` applies type-specific rules before accepting a transaction into the mempool:

Transaction Type	Key Validation Rules
`VBTC_V2_VALIDATOR_REGISTER`	Valid IP address; VFX address not already registered in scan window
`VBTC_V2_VALIDATOR_HEARTBEAT`	Sender is an active registered validator
`VBTC_V2_VALIDATOR_EXIT`	Sender is an active validator
`VBTC_V2_CONTRACT_CREATE`	Valid DKG proof; valid Taproot address; threshold ≥ 1
`VBTC_V2_TRANSFER`	Sender has sufficient ledger balance; amount > 0
`VBTC_V2_WITHDRAWAL_REQUEST`	No active withdrawal for this (user, contract) pair; valid BTC address; sufficient balance
`VBTC_V2_WITHDRAWAL_COMPLETE`	Request exists in `Requested` state; caller is a validator
`VBTC_V2_WITHDRAWAL_CANCEL`	Active withdrawal exists in `Requested` or `Pending_BTC`
`VBTC_V2_WITHDRAWAL_VOTE`	Caller is a validator; not already voted; cancellation request exists

12.3 State Trei Handlers

When a block is confirmed, `StateData.cs` applies state transitions in transaction order. Each vBTC V2 type has a dedicated handler:

`HandleVBTCV2Transfer` Appends a `SCStateTreiTokenizationTX` record to `SmartContractStateTrei.SCStateTreiTokenizationTXes`. The record holds `FromAddress`, `ToAddress`, `Amount`, and the block height. Balance queries replay this list to compute the current ledger balance.

`HandleVBTCV2WithdrawalRequest` Transitions contract state:

```
WithdrawalStatus          = Requested
ActiveWithdrawalBTCDestination = btcAddress
ActiveWithdrawalAmount    = amount
ActiveWithdrawalRequestHash = txHash
WithdrawalRequestBlock    = blockHeight
ActiveWithdrawalFeeRate   = feeRate
```

`HandleVBTCV2WithdrawalComplete`

```
WithdrawalStatus = Pending_BTC
LastValidatorActivityBlock = blockHeight   ← resets 24h safety gate countdown
```

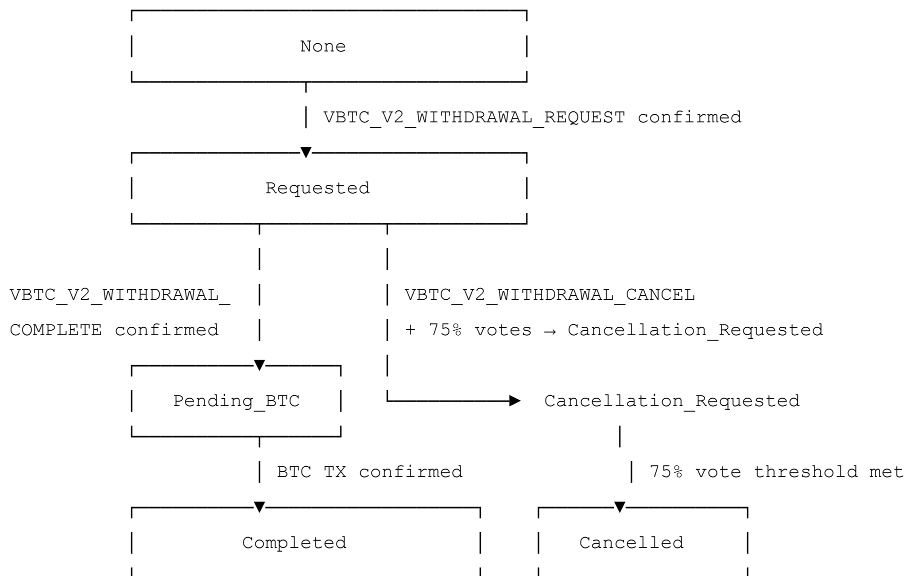
Appends a `VBTCWithdrawalHistory` record. After Bitcoin confirmation is detected via Electrum, an additional state update sets `Completed`.

`HandleVBTCV2WithdrawalVote` Tallies the vote in `VBTCWithdrawalCancellation`. When the cumulative count reaches ≥ 75% of active validators:

```
WithdrawalStatus                = Cancelled
ActiveWithdrawal* fields        = cleared
```

`HandleVBTCV2ValidatorRegister / Heartbeat / Exit` These do not modify the State Trei directly — they are recorded in block data and processed by `VBTCValidatorRegistry.ScanBlocks()` on demand.

12.4 Withdrawal Status State Machine



The state machine is enforced at two levels: the mempool validator rejects transactions that attempt illegal state transitions, and the block-level validator re-checks before applying state in `StateData.cs`.

12.5 Node Startup Sequence

On process start, the vBTC V2 subsystem initializes in a fixed order:

```
// 1. Initialize database collections (LiteDB)
DbContext.InitializeVBTCDatabases();

// 2. Start balance scan loop (queries Electrum every 5 minutes)
VBTCService.VBTCV2BalanceScanLoop();

// 3. Validator-only initialization
if (!string.IsNullOrEmpty(Globals.ValidatorAddress))
{
    // Derive Base Ethereum address from VFX key pair (for cross-network identity)
    ValidatorEthKeyService.TryInitializeGlobalsValidatorBaseAddress();

    // Start FROST HTTP server (ports 7295 / 7296)
    FrostServer.Start();

    // Start heartbeat loop (sends VBTC_V2_VALIDATOR_HEARTBEAT every ~500 blocks)
    VBTCValidatorHeartbeatService.VBTCValidatorHeartbeatLoop();
}
```

The balance scan loop calls `VBTCService.ScanVBTCV2Balances()` which queries Electrum X for the confirmed BTC balance of every known Taproot deposit address and updates `VBTCContractV2.Balance`. This is the only source of truth for the on-chain BTC balance; the VFX ledger tracks intra-chain transfers but does not independently verify Bitcoin state.

12.6 Database Layout

All vBTC V2 data is persisted in two LiteDB databases:

`DB_vBTC` — Core contract and validator state:

Collection	Type	Description
`vbtc_v2_contracts`	`VBTCContractV2`	Contract state, FROST keys, withdrawal status
`vbtc_validators`	`VBTCValidator`	Cached validator records (supplemented by block scan)
`vbtc_withdrawal_cancellations`	`VBTCWithdrawalCancellation`	Cancellation vote tracking
`vbtc_frost_key_store`	`FrostValidatorKeyStore`	Validator key shares (encrypted at rest)

`DB_VBTCWithdrawalRequests` — Withdrawal audit trail:

Collection	Type	Description
`vbtc_withdrawal_requests`	`VBTCWithdrawalRequest`	All withdrawal requests; used for replay attack prevention

The `FrostValidatorKeyStore` entries store each validator's FROST key package (the private signing share), the associated pubkey package, and the participant order JSON used for FROST Identifier mapping. These are validator-only records; non-validator wallet nodes do not have them.

---

13. Security Properties and Attack Surface Analysis

13.1 Key Security Guarantees

Private key non-existence. The FROST protocol's core property is that no coalition of fewer than t validators can compute any information about the group private key. This is guaranteed by the information-theoretic properties of t-degree polynomial secret sharing and the discrete logarithm problem over secp256k1. The owner of a vBTC V2 contract cannot exit funds unilaterally — the deposit address is controlled by the FROST group key, and only a threshold of validators can authorize a spend.

Equivocation resistance. A validator cannot produce two valid signing shares for different messages in the same signing session without being detected. The FROST binding factor ρ_i `= H(i, message, {all nonce commitments})` ties each partial signature to the specific message and the complete nonce commitment set. If a validator broadcasts different nonce commitments to different participants, the aggregation step will fail (the partial signatures won't sum to a valid Schnorr signature).

Replay attack prevention. Three independent layers:

- VFX layer: `VBTC_V2_WITHDRAWAL_REQUEST` creates a unique `RequestHash`; only one active withdrawal per (address, contract) pair
- Bitcoin layer: Each withdrawal consumes a specific UTXO, which cannot be double-spent
- Base layer: `usedLockIds` mapping prevents `lockId` reuse; `adminNonce` monotonically increments for all admin operations

Domain separation on Base. The mint message hash includes `block.chainid` and `address(this)`, so signatures from the VFX attestation process cannot be replayed on a different chain or at a different contract address.

13.2 Threat Model

Threat	Defense
Validator compromise (< t validators)	Threshold cryptography: quorum of t required; minority cannot sign
Contract owner exit scam	Owner cannot reconstruct full key; needs validator quorum
Rogue-key attack during DKG	FROST DKG commit-before-reveal prevents rogue key injection
Group key disagreement	Coordinator cross-checks all validators produce identical group public key; aborts on mismatch
Validator DoS (temporary)	Heartbeat-based liveness; 1,000-block scan window retains validators after transient offline periods
Mass validator exodus	24-hour safety gate prevents instant threshold reduction; 2-of-3 floor ensures recovery
Replay attack on withdrawal	One active withdrawal per (user, contract) pair; unique request hashes; UTXO consumed on spend
Withdrawal replay via different signing session	`WithdrawalRequestHash` included in signing start; validators dedup on this field (FIND-028)
Ceremony hijack	Leader signature (VFX key) required on all ceremony broadcasts; validators verify before processing
Invalid FROST aggregate accepted	FROST native library verifies aggregated signature internally before returning; fail-closed on mismatch
Withdrawal cancellation griefing	75% supermajority required; minority cannot maliciously cancel
Balance inflation (false BTC credit)	Balance derived exclusively from Electrum X UTXO query; ledger tracks only transfers, not deposits

13.3 Known Limitations

- **FROST key reuse.** If the FROST group public key is reused across different vBTC contracts (same DKG participants generating the same key), a compromise of key shares becomes correlated. The current design creates a fresh DKG ceremony per contract, mitigating this.
- **Off-chain signing coordination.** The FROST signing ceremony is coordinated via HTTP among validator peers. A network partition between the coordinator and a subset of validators can cause a ceremony to time out. The ceremony TTL (1 hour) bounds recovery time. The retry logic (up to 5 polls, 15 seconds total) provides resilience against brief network hiccups.
- **Electrum dependency.** Bitcoin balance lookups and UTXO queries depend on Electrum X. A compromised Electrum server could feed false UTXO data, blocking legitimate withdrawals or causing incorrect balance displays. Production deployments should use multiple Electrum endpoints with cross-validation.
- **HTTP ceremony channel.** Validator peer communication during ceremonies is over plaintext HTTP (with an optional HTTPS port). Secret shares are never sent over this channel — only commitments, nonce commitments, and public outputs of FROST operations — but traffic analysis of the ceremony could reveal signing activity. Future hardening should enforce TLS with certificate pinning.

---

14. Protocol Parameters Reference

Parameter	Value	Derivation
VFX block time	~12 seconds	Network consensus parameter
FROST DKG quorum	75% of active validators	Strong BFT supermajority for key generation
FROST signing threshold	51% of registered validators	Majority rule; recovery after validator exit
Withdrawal cancellation votes	75% of validators	Supermajority to prevent griefing
Validator heartbeat interval	500 blocks (~1.7 hours)	Liveness signal
Validator scan window	1,000 blocks (~3.3 hours)	Covers 2 full heartbeat cycles

Parameter	Value	Derivation
Threshold safety gate	24 hours = 7,200 blocks	Prevents instant exploitation during outage
Safety buffer (threshold)	+10% above availability	Maintains margin above minimum viable set
Minimum validators (absolute)	3	Below this, 2-of-3 rule applies
2-of-3 threshold	66.67%	When exactly 3 validators remain
MPC ceremony TTL	1 hour	In-memory session expiry
Max concurrent ceremonies	100 (global), 1 (per owner)	Anti-spam / DoS prevention
Balance scan interval	5 minutes	BTC balance refresh from Electrum X
Session cleanup interval	5 minutes	FrostServer background GC
Taproot input size estimate	57.5 vBytes	BIP341 key-path spend
FROST validator port (HTTP)	7295 (mainnet) / 17295 (testnet)	Ceremony coordination
FROST validator port (HTTPS)	7296 (mainnet) / 17296 (testnet)	TLS ceremony coordination
Signing Round 2 probe timeout	30 seconds (ceremony client)	Per HTTP call
Signing Round 2 retry attempts	5 (delays: 2s, 2s, 3s, 3s, 5s)	Backoff polling for share collection
Validator probe timeout	5 seconds	/health check before ceremony
FROST Identifier format	32-byte big-endian scalar (64 hex)	secp256k1 Scalar as per frost-secp256k1-tr
Participant ordering	Alphabetical (Ordinal) by VFX address	Deterministic; coordinator and validator must agree

15. API Reference

All endpoints use base URL `http://localhost:{APIPort}/vbtcapi/vbtc/`.

Ceremony Management

Method	Endpoint	Description
POST	`InitiateMPCCeremony/{ownerAddress}`	Start a new FROST DKG ceremony
GET	`GetCeremonyStatus/{ceremonyId}`	Check DKG progress
POST	`CancelCeremony/{ceremonyId}/{ownerAddress}`	Abort DKG

Contract Operations

Method	Endpoint	Description
POST	`CreateVBTCContract`	Create contract with embedded MPC ceremony
POST	`CreateVBTCContractRaw`	Create contract from pre-signed payload
GET	`GetMPCDepositAddress/{scUID}`	Get Taproot address and FROST group key
GET	`GetContractDetails/{scUID}`	Full contract state
GET	`GetContractList/{address?}`	List contracts (optionally by owner)

Validator Management

Method	Endpoint	Description
POST	`RegisterValidator/{address}/{ip}`	Register as FROST validator
POST	`ValidatorHeartbeat/{address}`	Emit heartbeat transaction
GET	`GetActiveValidators`	List active validators from block scan
GET	`GetValidatorStatus/{address}`	Single validator details

Token Operations

Method	Endpoint	Description
POST	`TransferVBTC`	Transfer to single recipient

Method	Endpoint	Description
POST	`TransferVBTCMulti`	Transfer to multiple recipients
GET	`GetVBTCBalance/{address}/{scUID}`	Balance in one contract
GET	`GetAllVBTCBalances/{address}`	All balances across contracts

Withdrawal Operations

Method	Endpoint	Description
POST	`RequestWithdrawal`	Step 1: declare withdrawal intent
POST	`CompleteWithdrawal`	Step 2: FROST sign + broadcast BTC TX
POST	`CancelWithdrawal`	Request cancellation governance
POST	`VoteOnCancellation`	Validator votes on cancellation
GET	`GetWithdrawalStatus/{scUID}`	Current withdrawal state
GET	`GetWithdrawalHistory/{scUID}`	All past withdrawals

FROST Ceremony (Validator HTTP Server, port 7295)

Method	Endpoint	Description
GET	`/health`	Validator reachability probe (returns `{"Success":true}`)
POST	`/frost/dkg/start`	Broadcast DKG session start to validator
GET	`/frost/dkg/round1/{sessionId}`	Collect Round 1 commitments
POST	`/frost/dkg/round2/{sessionId}`	Trigger Round 2 share generation
POST	`/frost/dkg/shares/{sessionId}`	Redistribute all shares for Round 3 finalization
GET	`/frost/dkg/round3/{sessionId}`	Collect Round 3 verification results
GET	`/frost/dkg/result/{sessionId}`	Collect finalized DKG result (group public key, Taproot address)
POST	`/frost/sign/start`	Broadcast signing session start
GET	`/frost/sign/round1/{sessionId}`	Collect nonce commitments
POST	`/frost/sign/round2/{sessionId}`	Broadcast nonces; receive signature share inline
GET	`/frost/sign/share/{sessionId}`	Fallback poll for signature share
GET	`/frost/key/pubkey/{id}`	Fetch pubkey package by SCUID or ceremony ID

Example: Complete Withdrawal Request/Response

Request (Step 1 — Withdrawal Request):

```
POST /vbtcapi/vbtc/RequestWithdrawal
{
  "SmartContractUID": "abc123:1773285700",
  "RequestorAddress": "xAlice...",
  "BTCAddress": "bc1qxxx...",
  "Amount": 0.5,
  "FeeRate": 15
}
```

Response:

```
{
  "Success": true,
  "TransactionHash": "0x4d5e6f...",
  "Status": "Requested"
}
```

Request (Step 2 — Complete Withdrawal):

```
POST /vbtcapi/vbtc/CompleteWithdrawal
{
  "SmartContractUID": "abc123:1773285700",
  "WithdrawalRequestHash": "0x4d5e6f..."
}
```

Response:

```
{
  "Success": true,
  "VFXTransactionHash": "0x7a8b9c...",
  "BTCTransactionHash": "a1b2c3d4e5f6...",
  "Status": "Pending_BTC",
  "ValidatorsUsed": 7,
  "ThresholdUsed": 51,
  "AdjustedThreshold": 51,
  "FeeRateSatsPerVByte": 15
}
```

This document describes the vBTC V2 implementation on VFX as found in VerifiedX-Core at commit `856b9827`. Protocol parameters and specific thresholds may be tuned during testnet operation. All cryptographic claims are grounded in the FROST paper (Komlo & Goldberg, 2021) and the ZCash Foundation's `frost-secp256k1-tr` Rust crate. The Base L2 bridge layer (vBTCb) is documented separately.