vBTC.b — Technical Specification

VerifiedX Bridge: Bitcoin Liquidity on Base L2

Table of Contents

---

1. Executive Summary

vBTC.b is the Base L2 representation of vBTC — tokenized Bitcoin issued on the VerifiedX (VFX) blockchain. It is an ERC-20 token deployed on Base (Chain ID 8453) whose minting is controlled by a decentralized validator network and protected by strict Byzantine Fault Tolerant (BFT) thresholds. Burning vBTC.b initiates a validator-threshold-secured exit back to either the VFX chain or native Bitcoin, coordinated by the same validator set through FROST threshold signatures.

Trust model: The system is non-custodial at the protocol layer — there is no single custodian, admin key, or privileged operator that can unilaterally mint, burn, or redirect funds. All privileged operations require multi-validator threshold agreement enforced on-chain. Liveness depends on a sufficient subset of validators remaining online and reachable; the protocol includes adaptive threshold mechanisms and caster failover to maintain availability under partial validator failure.

This document describes Version 2 of the bridge, which introduces:

- **No single owner or admin key** — upgrades and mints require multi-validator consensus
- **UUPS upgradeable proxy** — implementation swappable by validator threshold, not a privileged EOA
- **Pool-based FIFO unlock** — burns aggregate against all available VFX locks, not a 1:1 mapping
- **Adaptive FROST threshold** — withdrawal threshold relaxes toward 60% to prevent permanent lockouts
- **Block-scan validator registry** — replaces database-persisted validator state with a live on-chain scan window

Threshold Quick-Reference

The bridge 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	Context	Rationale
Base mint (`mintWithProof`)	⌈2n/3⌉ ≈ 67% of Base validator set	vBTCb.sol on-chain (§4.2)	Classical BFT — tolerates ⌊n/3⌋ Byzantine validators
Base upgrade (UUPS)	⌈2n/3⌉ ≈ 67% of Base validator set	vBTCb.sol on-chain (§4.7)	Same safety guarantee as minting
Base validator removal (n > 5)	⌈51n/100⌉ = 51%	vBTCb.sol on-chain (§4.6)	Prioritizes liveness — prevents minority blocking removal
Base validator removal (n ≤ 5)	⌈(n+1)/2⌉ = simple majority	vBTCb.sol on-chain (§4.6)	Preserves liveness in small validator sets
Caster exit consensus	max(2, activeCasters/2 + 1) = simple majority	VFX caster layer (§9)	Byzantine safety for caster coordination
FROST DKG ceremony	75% of active validators	VFX validator layer (see vBTC V2 spec)	Strong supermajority for irreversible key generation
FROST withdrawal signing	51% of DKG snapshot validators	VFX validator layer (§7.2)	Balances liveness against safety; adaptive up to 60%
FROST adaptive ceiling	60% max after inactivity relaxation	VFX validator layer (§7.2)	Prevents permanent lockout while preserving safety bounds
Withdrawal cancellation vote	75% of active validators	VFX governance (see vBTC V2 spec §10)	Supermajority prevents minority griefing
2-of-3 recovery floor	66.67% (2 of 3)	VFX validator layer (see vBTC V2 spec §9)	Mathematical minimum when exactly 3 validators remain
Minimum signature count	2 (absolute floor)	All on-chain operations	Prevents single-validator attack in any set size

> ***Note on validator sets:*** *"Base validator set" refers to the addresses registered in the vBTCb.sol contract on Base L2. "DKG snapshot validators" refers to the validators who participated in the FROST key generation ceremony for a specific vBTC V2 contract. "Active validators" refers to validators with a recent heartbeat on the VFX chain. These sets may differ — validators join and leave over time — and each threshold applies to its specific set.*

---

2. vBTC — The Underlying Vessel (Overview)

vBTC is a smart-contract-native representation of Bitcoin on the VerifiedX blockchain. Each vBTC token is backed 1:1 by real BTC held in a Taproot address whose signing key is generated by a distributed key generation (DKG) ceremony using the FROST protocol. Validators collectively control the signing key; no individual can unilaterally spend the underlying BTC.

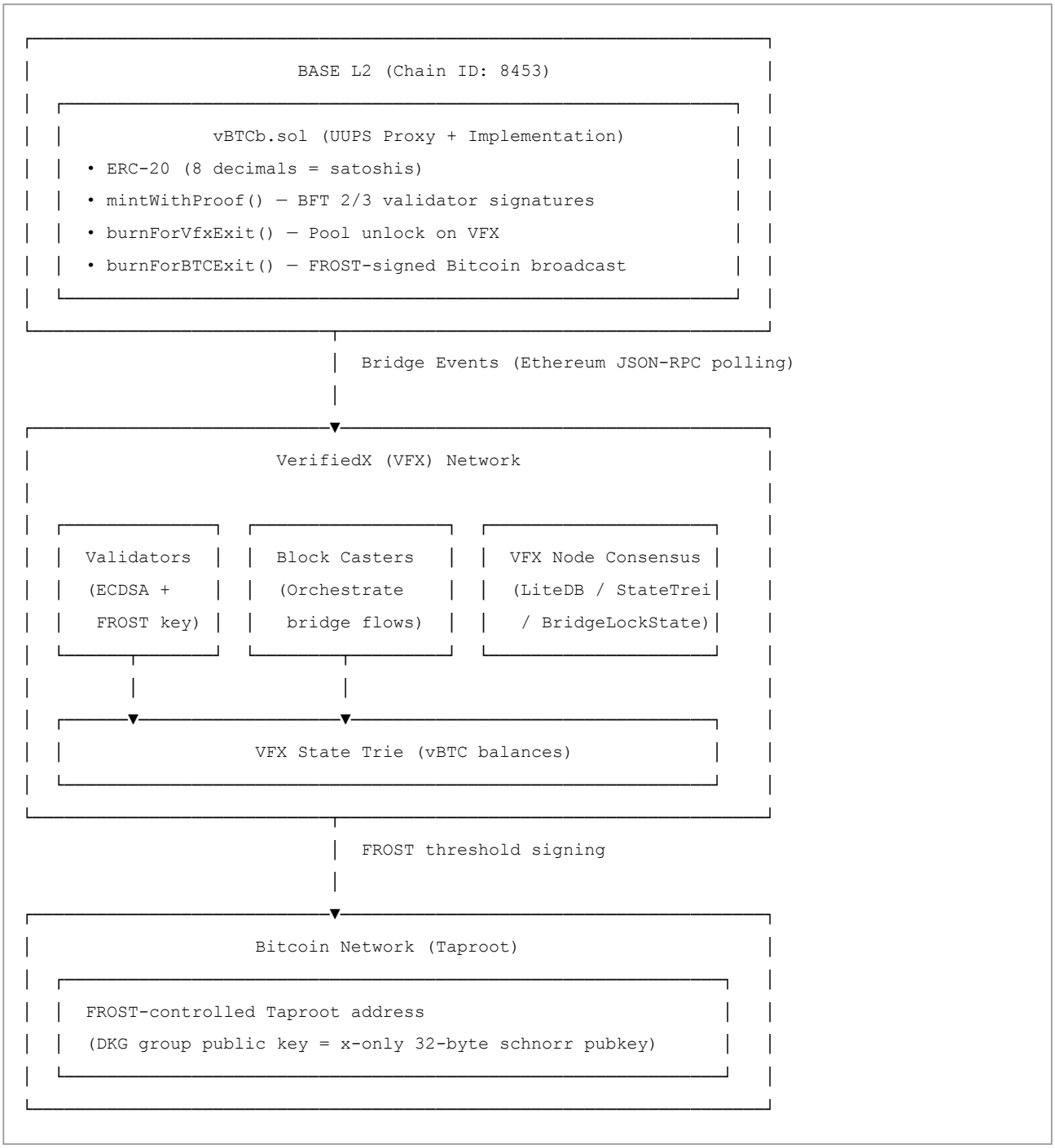
Two contract types exist on VFX:

Type	Description
vBTC V1	Single-owner, single deposit address. Now deprecated for new issuances.
vBTC V2	FROST MPC multi-party custody. Active issuance standard.

vBTC V2 tokens are what may be bridged to Base as vBTC.b. The bridge lock and burn flows described in this document operate exclusively against vBTC V2 balances.

---

3. System Architecture Overview



The bridge is **non-custodial at the L2 layer**: the vBTCb.sol contract has no owner or privileged admin. Minting requires ⌈`2n/3`⌉ of `n` active validators to sign independently, and the contract enforces this on-chain. Exiting burns vBTC.b irrevocably before any VFX or BTC is released.

---

4. vBTCb.sol — The Base L2 Smart Contract

4.1 Contract Design & Standards

vBTCb.sol inherits from OpenZeppelin's upgradeable contract suite:

```
contract VBTCb is
    Initializable,
    ERC20Upgradeable,       // Standard fungible token
    UUPSUpgradeable         // UUPS upgrade pattern
{
    uint8 private constant DECIMALS = 8; // Satoshi precision
    mapping(string => bool) public usedLockIds;
    mapping(address => bool) public validators;
    address[] public validatorList;
    uint256 public adminNonce;
}
```

Key design choices:

- **8 decimals** — matches satoshi precision (1 BTC = 100,000,000 units), eliminating lossy decimal conversion when moving between Bitcoin and EVM environments
- **No `owner`** — OpenZeppelin's `Ownable` is intentionally absent; all privileged operations require threshold signatures
- **UUPS over Transparent Proxy** — upgrade authorization is gated by the same validator multi-sig, not a deployer EOA

4.2 Validator Set & Byzantine Fault Tolerance

The contract maintains an on-chain validator set:

```
mapping(address => bool) public validators;
address[] public validatorList;
```

Signature thresholds are computed at runtime using BFT conventions:

Operation	Threshold Formula	Rationale
Mint	$\lceil 2n/3 \rceil$ (strict BFT)	Tolerates $\lfloor n/3 \rfloor$ Byzantine validators
Validator removal ($n \leq 5$)	$\lceil (n+1)/2 \rceil$ (simple majority)	Preserves liveness in small sets
Validator removal ($n > 5$)	$\lceil 51n/100 \rceil$ (51%)	Prioritizes liveness over safety
Upgrade	$\lceil 2n/3 \rceil$ (strict BFT)	Same safety guarantee as minting
Minimum always	2	Prevents single-validator attack even in tiny sets

Why 2/3 for minting?

In classical BFT consensus (Castro & Liskov PBFT, 1999), a system of n replicas tolerates $f = \lfloor (n-1)/3 \rfloor$ Byzantine faults with $n \geq 3f+1$ correct replicas required for agreement. Applied here: if 10 validators exist, up to 3 can be compromised without enabling an unauthorized mint. An adversary who controls fewer than 1/3 of the validator set cannot forge `mintWithProof` signatures because the contract rejects any call with `validSignatures < requiredSignatures`:

```
uint256 required = (validatorCount * 2 + 2) / 3; // ceiling division
required = required < 2 ? 2 : required;           // minimum 2
require(validSignatures >= required, "Insufficient valid signatures");
```

4.3 Signature Scheme & Message Hash Construction

Each validator independently signs a deterministic EIP-191-compatible message. The hash construction binds signatures to a specific mint operation and prevents cross-chain replay:

```
bytes32 messageHash = keccak256(
    abi.encodePacked(
        to,            // address: EVM recipient on Base
        amount,        // uint256: token amount (satoshis)
        lockId,        // string: GUID from VFX lock TX
        nonce,         // uint256: VFX block height of lock
        block.chainid, // uint256: 8453 for Base Mainnet
        address(this)  // address: proxy contract address
    )
);
bytes32 ethSignedHash = MessageHashUtils.toEthSignedMessageHash(messageHash);
```

Including `block.chainid` and `address(this)` is critical: the same signature set is invalid on Base Sepolia, Base Goerli, or any contract deployment other than the specific proxy. This prevents signature portability attacks where a valid attestation set from one environment is replayed on another.

Signature recovery uses ECDSA:

```
address recovered = ECDSA.recover(ethSignedHash, signatures[i]);
require(validators[recovered], "Invalid validator signature");
require(!seen[recovered], "Duplicate signature");
```

4.4 Mint Flow — `mintWithProof`

```
function mintWithProof(
    address to,
    uint256 amount,
    string calldata lockId,
    uint256 nonce,
    bytes[] calldata signatures
) external
```

Execution steps:

1. Verify `!usedLockIds[lockId]` — rejects double-mint for same VFX lock
2. Reconstruct `ethSignedHash` from `(to, amount, lockId, nonce, chainId, contract)`
3. Recover each signature, verify recovered address is an active validator, reject duplicates
4. Count valid signatures; require $\geq$ `required` ($\lceil 2n/3 \rceil$)
5. Mark `usedLockIds[lockId] = true` (prevents future replay even if validator set changes)
6. Call `_mint(to, amount)` — standard ERC-20 mint
7. Emit `MintExecuted(to, amount, lockId, nonce)`

The `usedLockIds` guard combined with the nonce ensures idempotency: if the caster submits the transaction twice, the second call reverts at step 1.

4.5 Burn Flows — VFX Exit & BTC Exit

vBTC.b supports two exit modes, both of which irrevocably destroy tokens before any off-chain action is taken:

VFX Pool Unlock:

```
function burnForVfxExit(
    uint256 amount,
    string calldata vfxDestinationAddress
) external {
    require(amount > 0, "Amount must be positive");
    _burn(msg.sender, amount);
    emit VfxExitBurned(msg.sender, amount, vfxDestinationAddress, block.chainid);
}
```

Direct BTC Exit:

```
function burnForBTCExit(
    uint256 amount,
    string calldata btcDestination
) external {
    require(amount > 0, "Amount must be positive");
    require(bytes(btcDestination).length >= 26, "Invalid BTC address");
    _burn(msg.sender, amount);
    emit BTCExitBurned(msg.sender, amount, btcDestination, block.chainid);
}
```

The BTC address length check (`≥ 26`) rejects obviously malformed destinations. Valid legacy addresses (`1...`) are 26-34 characters; Bech32 (`bc1...`) are 42-62 characters; Taproot (`bc1p...`) are 62 characters.

Both burns are **fire-and-forget at the EVM layer** — the contract has no callback or confirmation mechanism. The VFX network's `BaseBridgeExitWatchService` detects the emitted events and initiates the corresponding off-chain release flow.

4.6 Validator Management

Validators are managed by the existing validator set using the same multi-sig threshold pattern:

```
function addValidator(
    address newValidator,
    uint256 vfxBlockHeight,
    bytes[] calldata signatures
) external

function removeValidator(
    address oldValidator,
    uint256 vfxBlockHeight,
    bytes[] calldata signatures
) external
```

The `vfxBlockHeight` parameter is included in the message hash to bind validator change proposals to a specific point in VFX chain history, preventing stale proposals from being replayed after validator set churn.

Batch operations are supported (`addValidatorBatch`, `removeValidatorBatch`) with a maximum of 100 validators per call, enabling efficient bootstrapping of the initial validator set without O(n) individual transactions.

Removal uses a lower threshold (51% for n > 5) than minting (2/3 BFT) to prevent a minority of validators from permanently blocking removal of a misbehaving peer — a known liveness concern in high-threshold systems.

4.7 Upgradability (UUPS)

```
function upgradeWithValidatorApproval(
    address newImplementation,
    bytes[] calldata signatures
) external {
    // Requires 2/3 threshold
    _verifyValidatorSignatures(
        keccak256(abi.encodePacked(newImplementation, adminNonce, block.chainid, address(this))),
        signatures
    );
    adminNonce++;
    upgradeToAndCall(newImplementation, "");
}

function _authorizeUpgrade(address) internal override {
    // Intentionally empty — authorization is performed in upgradeWithValidatorApproval
}
```

The `adminNonce` prevents replay of a previously approved upgrade proposal against a new implementation address. The UUPS pattern (EIP-1822) places upgrade logic in the implementation contract rather than the proxy, reducing proxy complexity and gas cost for normal operations.

---

5. Validator Registry — On-Chain Block Scan

The validator registry does not rely on a centralized database. Instead, `VBTCValidatorRegistry` performs a rolling scan of the last **1,000 VFX blocks** (~2.8 hours at 10 seconds/block) to derive the active validator set from on-chain transaction types:

VFX Transaction Type	Effect
`VBTC_V2_VALIDATOR_REGISTER`	Add validator with FROST public key and Base address
`VBTC_V2_VALIDATOR_HEARTBEAT`	Refresh `LastHeartbeatBlock`; must occur every ~500 blocks
`VBTC_V2_VALIDATOR_EXIT`	Mark validator inactive

A validator is considered **active** if:

- A `REGISTER` or `HEARTBEAT` transaction exists within the scan window
- No `EXIT` transaction supersedes it

```
// Scan window: 1,000 blocks = ≥ 2 full heartbeat cycles
const int ScanWindow = 1000;
// Heartbeat interval: ~500 blocks
const int HeartbeatInterval = 500;
```

The result set is cached per block height and invalidated on each new block, ensuring the caster always operates against the current active set without stale reads.

VBTCValidator data model (from scan):

```
public class VBTCValidator
{
    public string ValidatorAddress { get; set; }    // VFX address (base58)
    public string IPAddress { get; set; }           // For P2P attestation
    public string FrostPublicKey { get; set; }      // x-only 32-byte schnorr key (hex)
    public long RegistrationBlockHeight { get; set; }
    public long LastHeartbeatBlock { get; set; }
    public bool IsActive { get; set; }
    public string BaseAddress { get; set; }         // 0x... EVM address on Base
    public bool IsRegisteredOnBase { get; set; }    // Confirmed in vBTCb.sol
}
```

The Base address is either explicitly provided during registration or deterministically derived from the validator's VFX private key using the same secp256k1 curve (VFX uses secp256k1 for signing, same as Ethereum), enabling validators to use the same underlying key material across chains without an explicit derivation step.

---

6. Bridge Lock Lifecycle (VFX → Base)

6.1 Lock Phase

A user initiates the bridge by calling `VBTCService.CreateBridgeLockTx()`, which broadcasts a `VBTC_V2_BRIDGE_LOCK` transaction on the VFX chain:

```
VBTC_V2_BRIDGE_LOCK TX Data:
{
  "LockId": "a3f7e2b1-...",      // GUID, globally unique
  "Amount": 0.01,                // BTC (decimal)
  "AmountSats": 1000000,         // Integer satoshis
  "EvmDestination": "0xABC..."   // Base recipient
}
```

When this transaction is confirmed in a VFX block, two state updates occur atomically:

1. **`VBTCBridgeLockState`** is inserted into the consensus DB (replicated to all nodes):

```
new VBTCBridgeLockState {
    LockId = "a3f7e2b1-...",
    OwnerAddress = "RVfxAddress...",
    Amount = 0.01m,
    AmountSats = 1000000,
    UnlockedAmount = 0,          // Initialized to 0
    IsUnlocked = false,
    IsBlacklisted = false
}
```

2. The vBTC balance is debited from the owner's tokenization ledger (state trie). The token is now in limbo — it no longer exists on VFX but has not yet been minted on Base.

6.2 Attestation Phase

The caster node monitors VFX blocks for confirmed `VBTC_V2_BRIDGE_LOCK` transactions. For each, `BaseBridgeAttestationService` initiates multi-validator signature collection.

Mint nonce: The VFX lock block height is used directly as the `nonce` parameter in `mintWithProof`. This provides:

- Uniqueness (block heights are monotonically increasing)
- VFX chain anchoring (the signature references an observable, immutable chain state)
- Replay prevention (reusing a nonce would require reusing a block height, which is impossible)

Each validator is contacted at its registered IP address:

```
POST /valapi/Validator/SignMintAttestation
Body: {
  "LockId": "a3f7e2b1-...",
  "EvmDestination": "0xABC...",
  "AmountSats": 1000000,
  "MintNonce": 847293,           // VFX lock block height
  "ContractAddress": "0xvBTCbProxy..."
}
```

Before signing, each validator independently verifies:

1. The `LockId` exists on-chain in `VBTCBridgeLockState`
2. The `EvmDestination`, `AmountSats`, and `ContractAddress` match on-chain data
3. The lock is not blacklisted or already unlocked

If verification passes, the validator signs:

```
keccak256(abi.encodePacked(evmDest, amountSats, lockId, nonce, chainId, contract))
```

and returns the ECDSA signature hex. The validator uses its VFX-derived private key for signing, which corresponds to its registered Base address in the smart contract.

The caster collects signatures until `validSignatures ≥ ⌈2n/3⌉`, then marks the lock record as `AttestationReady`.

6.3 Mint Submission Phase

`BaseBridgeMintSubmissionService` processes `AttestationReady` records in a background loop:

```
await contract.mintWithProof.SendTransactionAsync(
    fromAddress: validatorBaseAddress,  // Validator's Base EOA (pays gas)
    to: evmDestination,
    amount: amountSats,
    lockId: lockId,
    nonce: vfxBlockHeight,
    signatures: collectedSignatures
);
```

The calling validator's Base EOA must hold ETH to cover Base L2 gas. On transaction confirmation, the contract emits:

```
MintExecuted(
    address indexed to,       // EVM recipient
    uint256 amount,           // Satoshis minted
    string lockId,            // VFX lock GUID
    uint256 nonce             // VFX block height
)
```

The `BridgeLockRecord` status advances to `ProofSubmitted`, and the Base transaction hash is stored for auditing.

RPC redundancy: Three configurable RPC endpoints are tried in sequence (`BaseBridgeRpcUrl`, `BaseBridgeRpcUrl2`, `BaseBridgeRpcUrl3`) to tolerate node outages during submission.

---

7. Exit Flows (Base → VFX / BTC)

Exit begins on Base when a user calls one of the burn functions. The contract irrevocably destroys their vBTC.b, and the emitted event becomes the authoritative proof of exit intent.

7.1 VFX Pool-Based Unlock

Trigger: `burnForVfxExit(amount, vfxDestinationAddress)` on Base

`BaseBridgeExitWatchService` polls Base via Ethereum JSON-RPC (`eth_getLogs`) every 12 seconds, filtering for `VfxExitBurned` events from the vBTCb.sol contract address. Polling interval and start block are configurable via environment variable `BASE_BRIDGE_EXIT_FROM_BLOCK`.

Caster consensus round:

Once a burn event is detected, `BurnExitConsensusService` coordinates casters:

1. **Proposal:** Each caster computes `SHA256(burnTxHash + casterAddress)`. The caster with the **lowest hash value** becomes the designated handler — a deterministic, leaderless selection that requires no coordination overhead and produces a consistent result across all honest casters.
2. **Confirmation:** The designated caster broadcasts a signed `BurnAlert` to peers. Other casters independently verify the burn event on-chain and counter-sign agreement.
3. **Threshold:** Confirmation requires `max(2, activeCasterCount / 2 + 1)` signatures — a simple majority.

Pool unlock allocation (FIFO):

The designated caster calls `BridgePoolUnlockService.ComputeAllocationPlan()`, which selects from available `VBTCBridgeLockState` records in FIFO order (by VFX confirmation block height), excluding blacklisted locks:

```
List<PoolUnlockAllocation> plan = ComputeAllocationPlan(burnAmountSats);
// Example: burn of 2.5 BTC matched against:
// Lock A: 1.0 BTC (fully consumed)
// Lock B: 1.5 BTC (partially consumed — V3 partial unlock)
```

Partial unlock is a V3 feature: `VBTCBridgeLockState` tracks `UnlockedAmount` separately from `Amount`, allowing a single lock to satisfy multiple partial burns over time:

```
public decimal RemainingAmount => Amount - UnlockedAmount;
public bool IsUnlocked => UnlockedAmount >= Amount;
```

The caster broadcasts a `VBTC_V2_BRIDGE_POOL_UNLOCK` transaction containing the full allocation plan. When applied in a VFX block, all nodes atomically:

- Apply partial unlocks to each consumed `VBTCBridgeLockState`
- Credit vBTC to `vfxDestinationAddress` via the tokenization ledger

7.2 Direct BTC Exit with FROST Signing

Trigger: `burnForBTCExit(amount, btcDestination)` on Base

The same event detection and caster consensus applies. Once a handler is confirmed, the flow diverges into FROST threshold signing:

Resolving the signing set:

The caster identifies which vBTC V2 contract holds the BTC for this exit and reads its `ValidatorAddressesSnapshot` — the list of validators who participated in the DKG ceremony and hold FROST key shares. This snapshot may differ from the current active validator set (validators join and leave over time).

`FrostMPCService.ProbeValidatorReachability()` pings each snapshot validator over P2P and builds a list of reachable participants.

Adaptive threshold calculation:

```
public static int CalculateAdjustedThreshold(
    int snapshotTotal,
    long lastActivityBlock,
    long currentBlock)
{
    int baseThreshold = (snapshotTotal * 51 + 99) / 100; // ceiling: 51%

    long blocksSinceActivity = currentBlock - lastActivityBlock;
    if (blocksSinceActivity > 50)
    {
        // Relax threshold toward 60% to recover from validator attrition
        int delta = Math.Min(9, (int)(blocksSinceActivity / 50));
        return Math.Min(60, baseThreshold + delta);
    }
    return baseThreshold;
}
```

The adaptive threshold addresses a fundamental liveness problem in threshold cryptography: if the signing threshold is too strict and validators drop offline, funds become permanently inaccessible. The relaxation mechanism allows recovery when the network has not seen validator activity for >50 blocks, capping at 60% to preserve safety bounds.

FROST signing and Bitcoin broadcast:

`BitcoinTransactionService.ExecuteFROSTWithdrawal()`:

1. Constructs the Bitcoin transaction spending the Taproot UTXO to `btcDestination`
2. Coordinates a FROST signing round among reachable snapshot validators
3. Each validator contributes a partial signature using their FROST share
4. Aggregates partial signatures into a single Schnorr signature valid under the group public key
5. Broadcasts the signed transaction to the Bitcoin network

On success, `VBTCBridgeBtcExitState` is marked complete and a `VBTC_V2_BRIDGE_EXIT_TO_BTC_COMPLETE` TX is broadcast on VFX carrying the Bitcoin transaction hash.

Failure handling:

If FROST signing fails (e.g., insufficient reachable validators, UTXO problem), the caster broadcasts `VBTC_V2_BRIDGE_EXIT_TO_BTC_FAIL`. This:

- Blacklists the affected `VBTCBridgeLockState` records with a reason string
- Reverses partial unlock state (restores `UnlockedAmount`)

- Does **not** re-mint vBTC.b on Base (the burn is irrevocable at the EVM contract level)

Resolution follows the bridge governance recovery process described in §7.3 below.

7.3 Exit Failure Recovery

The burn-before-release design is a deliberate double-spend prevention measure: if vBTC.b were not burned prior to releasing BTC/VFX, a user could race the release with a transfer, creating unbacked tokens. However, this design means a failed exit leaves the user temporarily without tokens on either chain. The protocol addresses this through four layered recovery mechanisms:

Layer 1 — Stuck exit detection and caster failover (automatic, ~8 minutes)

Every caster monitors pending exits. If a burn event has no corresponding `VBTC_V2_BRIDGE_EXIT_TO_BTC_COMPLETE` or `VBTC_V2_BRIDGE_EXIT_TO_BTC_FAIL` transaction after **50 VFX blocks (~8 minutes)**, the exit is flagged as stuck. The next-ranked caster (determined by the same deterministic `SHA256(burnTxHash || casterAddress)` scoring) automatically takes over and re-attempts the FROST signing ceremony. This provides recovery from single-caster failure with no human intervention.

Layer 2 — Adaptive FROST threshold (automatic, adjusts over time)

If the initial signing attempt failed because too few validators were reachable, the adaptive threshold mechanism (§7.2) gradually relaxes the signing requirement from 51% toward 60% of the *original* DKG snapshot set. For contracts on the VFX-native withdrawal path, the 24-hour safety gate (see vBTC V2 spec §9) provides even deeper threshold relaxation down to a 2-of-3 floor, ensuring funds remain recoverable under severe validator attrition.

Layer 3 — Bridge governance resolution (validator vote, target SLA: ≤24 hours)

If automatic recovery fails (e.g., the UTXO state is incompatible, or all casters are offline), resolution escalates to the bridge governance process:

1. Any party may submit a `VBTC_V2_WITHDRAWAL_CANCEL` transaction on the VFX chain to initiate re-evaluation of the stuck exit.
2. Validators review the exit state and cast `VBTC_V2_WITHDRAWAL_VOTE` transactions. Cancellation requires **75% validator approval** — a supermajority that prevents minority manipulation.
3. On reaching the 75% threshold, the blacklisted lock records are cleared, the user's vBTC balance is re-credited on the VFX ledger, and the exit state is reset.
4. The user may then initiate a fresh bridge exit (with corrected parameters if needed, e.g., a different BTC destination or fee rate).

Layer 4 — Audit trail and accountability

Every state transition — the original burn, the failure, the blacklist, the governance votes, and the resolution — is recorded as a VFX chain transaction with a corresponding transaction hash. This provides:

- A complete, immutable audit trail queryable by any node
- Publicly verifiable proof that governance acted correctly
- The ability for third-party monitors to detect and alert on stuck exits in real-time

Why vBTC.b is not re-minted on failure:

The vBTCb.sol contract on Base has no `re-mint` or `refund` function. This is intentional: adding such a function would create an attack surface where a compromised validator set could mint unbacked tokens by fabricating failure events. Instead, recovery happens at the VFX layer where the lock state is restored, allowing the user to re-bridge or hold their vBTC on VFX. The Base burn event serves as a permanent, auditable record that the exit was attempted.

8. FROST Threshold Signing

FROST (Flexible Round-Optimized Schnorr Threshold Signatures, Komlo & Goldberg 2021) is used for all Bitcoin operations in the VerifiedX system. It produces a standard Schnorr signature indistinguishable from a single-party signature, enabling Taproot (BIP-340/341) integration.

Key generation (DKG ceremony):

1. `VBTCService.InitiateMPCCeremony()` starts a DKG round among the current active validator set
2. Each validator generates a secret polynomial and broadcasts commitments
3. Shares are distributed peer-to-peer (encrypted to each recipient's key)
4. The ceremony produces:
    - A **group public key** — an x-only 32-byte schnorr public key (the Taproot internal key)
    - **Individual key shares** — each validator holds one share; no full key is ever reconstructed
5. The group public key is stored in `VBTCContractV2.FrostGroupPublicKey` and committed on-chain in VFX

```
public class VBTCContractV2
{
    public string FrostGroupPublicKey { get; set; }    // x-only 32-byte hex
    public string? FrostPubkeyPackage { get; set; }    // Serialized for aggregation
    public List<string> ValidatorAddressesSnapshot { get; set; }
    public int RequiredThreshold { get; set; }          // Initially 51 (percent)
    public long LastValidatorActivityBlock { get; set; }
}
```

Signing round (2-round FROST):

Round 1 — Commitment: Each participating validator generates a nonce commitment and broadcasts it. Round 2 — Signature: Each validator receives all commitments, computes its partial signature, and returns it to the coordinator. Aggregation: The coordinator combines all partial signatures into a single Schnorr signature using the `FrostPubkeyPackage`.

The resulting signature is submitted as the Taproot key-path spend witness, which is a single 64-byte signature — identical in structure to a single-party key spend. On-chain Bitcoin observers cannot distinguish a FROST aggregate from a regular signature.

DKG proof:

The DKG ceremony produces a `DKGProof` (base64-encoded, compressed) stored in the VFX contract state. This proof is a cryptographic commitment to the ceremony output, enabling any node to verify that a given `FrostGroupPublicKey` was correctly generated by the claimed validator set without replaying the ceremony.

9. Caster Consensus Protocol

Block casters are VFX nodes designated to orchestrate bridge operations. They do not produce VFX blocks but are trusted to submit bridge transactions on behalf of the network.

Leader election (per exit event):

For each detected Base burn event, casters independently compute:

```
handlerScore = SHA256(burnTxHash || casterVfxAddress)
```

The caster with the lowest score wins handler designation. This is a **deterministic, permissionless leader election** requiring no communication round: all honest casters agree on the winner independently. The VFX transaction submitted by the non-winning casters is rejected by consensus if a valid handler TX already exists.

Confirmation threshold:

```
int required = Math.Max(2, activeCasterCount / 2 + 1);
```

This provides Byzantine safety for the caster layer. Casters who fail to confirm within a timeout are assumed offline; the confirmation window is generous to handle network partitions.

Stuck exit detection:

If an exit event has no associated VFX completion TX after 50 VFX blocks, the system flags the exit as stuck and the next-ranked caster may attempt to take over. This prevents a single offline caster from permanently blocking a user's exit.

---

10. Security Model

10.1 Trust Assumptions

Component	Trust Required
vBTCb.sol	No single custodian or admin key — enforced by Base L2 EVM; all privileged operations require validator threshold consensus
Validator signatures	1/3 Byzantine tolerance (BFT 2/3 threshold)
Caster consensus	Simple majority of active casters; liveness depends on ≥2 casters being online
FROST signing	51% of DKG snapshot validators (adaptive up to 60%); liveness depends on threshold validators being reachable
Bitcoin network	Standard Bitcoin finality assumptions

10.2 Attack Vectors & Mitigations

Replay attack (cross-chain): Message hash includes `block.chainid` and `address(this)`. A signature collected for Base Mainnet is cryptographically invalid on Base Sepolia or any other deployment.

Replay attack (same chain): `usedLockIds[lockId]` is permanently set to `true` after the first successful mint. Re-submitting identical signatures with the same `lockId` reverts at the guard check.

Validator collusion (unauthorized mint): Requires compromising ⌈`2n/3`⌉ validators. With 10 validators, this means compromising 7. Each validator holds an independent key; there is no master key or HSM that, if breached, enables minting.

Validator collusion (BTC theft): Requires ≥ `requiredThreshold` validators from the DKG snapshot. The FROST protocol never reconstructs the full private key — shares are combined only during a signing session and the key material lives only in-memory during that session. An adversary who compromises a share storage system gets a share, not the key.

Sybil attack on validator set: New validators are added only by existing validator multi-sig (`addValidator` requires BFT threshold). An attacker cannot self-register new validators to dilute the set; they must compromise existing validators to approve additions.

Burn without release (exit processing failure): The bridge contract burns tokens before release — a deliberate design choice to prevent double-spend (see §7.3 for rationale). If the VFX network fails to process the exit, vBTC.b is destroyed at the contract level with no EVM-layer recourse. However, the protocol provides four layers of recovery: (1) automatic stuck-exit detection and caster failover after ~8 minutes, (2) adaptive FROST threshold relaxation for validator attrition, (3) bridge governance resolution via 75% validator vote that re-credits vBTC on the VFX ledger, and (4) a complete on-chain audit trail of all state transitions. See §7.3 for the full recovery procedure and SLA targets.

Front-running `mintWithProof`: Since `mintWithProof` specifies the `to` address in both the message hash and the function arguments, any front-running transaction that changes `to` would produce a signature mismatch and revert.

10.3 Signature Deduplication

The contract enforces that each validator can contribute at most one signature per mint:

```
mapping(address => bool) seen;
require(!seen[recovered], "Duplicate signature");
seen[recovered] = true;
```

This prevents a single compromised validator from submitting the same signature multiple times to artificially inflate the valid signature count.

---

11. Configuration & Deployment

config.txt entries:

```
BaseBridgeRpcUrl=https://mainnet.base.org
BaseBridgeRpcUrl2=https://base-mainnet.g.alchemy.com/v2/<key>
BaseBridgeRpcUrl3=https://base.llamarpc.com
BaseBridgeContract=0x<vBTCb_proxy_address>
BaseBridgeChainId=8453
```

Runtime globals (set at startup):

```
Globals.VBTCbContractAddress    // Loaded from BaseBridgeContract
Globals.BaseEvmChainId          // 8453 (mainnet) or 84532 (Sepolia)
Globals.ValidatorBaseAddress    // Derived from validator VFX private key
Globals.ValidatorAddress        // VFX address from config
Globals.IsBlockCaster           // Node role flag
```

Environment variables:

```
BASE_BRIDGE_EXIT_FROM_BLOCK=<n>   // Override exit event scan start block
```

Network parameters:

Parameter	Value	Notes
Base Mainnet Chain ID	8453	Used in all message hashes
Base Sepolia Chain ID	84532	Testnet only
vBTC.b decimals	8	Satoshi precision
Validator heartbeat interval	~500 VFX blocks	~83 minutes
Validator scan window	1,000 VFX blocks	~167 minutes
Exit watch poll interval	12 seconds	Matches Base L2 block time
Stuck exit threshold	50 VFX blocks	~8 minutes
FROST adaptive threshold window	50 VFX blocks	Before relaxation begins

---

12. API Reference

Route base: `/vbtcapi/VBTC/` and `/vbtcapi/Validator/`

Bridge Operations

Method	Endpoint	Description
POST	`/vbtcapi/VBTC/CreateBridgeLock`	Lock vBTC on VFX for bridging to Base
GET	`/vbtcapi/VBTC/GetBridgeLockStatus/{lockId}`	Poll lock status
POST	`/vbtcapi/VBTC/WaitForBridgeLockConfirmed`	Long-poll until VFX lock confirmed

CreateBridgeLock request:

```
{
  "OwnerAddress": "RVfxAddress...",
  "Amount": 0.01,
  "EvmDestination": "0xABCDEF..."
}
```

GetBridgeLockStatus response:

```
{
  "LockId": "a3f7e2b1-...",
  "Status": "AttestationReady",
  "Amount": 0.01,
  "AmountSats": 1000000,
  "EvmDestination": "0xABCDEF...",
  "VfxLockConfirmedOnChain": true,
  "VfxLockBlockHeight": 847293,
  "BaseTxHash": "0xMintTxHash...",
  "ValidatorSignatures": { "0xValidator1": "0xSig1...", "0xValidator2": "0xSig2..." },
  "RequiredSignatures": 7
}
```

Lock status state machine:

```
Locked → AttestationPending → AttestationReady → ProofSubmitted
```

Validator Operations

Method	Endpoint	Description
GET	`/vbtcapi/Validator/GetValidatorInfo`	List active validators from registry
POST	`/vbtcapi/Validator/RegisterValidator`	Broadcast REGISTER TX to VFX
POST	`/vbtcapi/Validator/HeartbeatValidator`	Broadcast HEARTBEAT TX
POST	`/vbtcapi/Validator/ExitValidator`	Broadcast EXIT TX
POST	`/vbtcapi/Validator/SignMintAttestation`	Sign a mint request (called by caster)

SignMintAttestation request/response:

```
// Request
{
  "LockId": "a3f7e2b1-...",
  "EvmDestination": "0xABCDEF...",
  "AmountSats": 1000000,
  "MintNonce": 847293,
  "ContractAddress": "0xvBTCbProxy..."
}

// Response
{
  "Signature": "0x1b7f3a...",   // 65-byte ECDSA signature hex
  "ValidatorAddress": "0xVal..." // For verification
}
```

vBTC V2 Contract Operations

Method	Endpoint	Description
POST	`/vbtcapi/VBTC/InitiateMPCCeremony`	Start FROST DKG ceremony

Method	Endpoint	Description
GET	`/vbtcapi/VBTC/GetCeremonyStatus/{id}`	Poll DKG ceremony status
POST	`/vbtcapi/VBTC/CreateContract`	Deploy new vBTC V2 contract
POST	`/vbtcapi/VBTC/TransferVBTC`	Transfer vBTC tokens on VFX
POST	`/vbtcapi/VBTC/RequestWithdrawal`	Request BTC withdrawal
POST	`/vbtcapi/VBTC/CompleteWithdrawal`	Execute FROST-signed Bitcoin TX
POST	`/vbtcapi/VBTC/CancelWithdrawal`	Cancel pending withdrawal

---

13. Transaction Type Registry

All bridge state transitions are anchored to VFX chain transactions, providing a complete auditable ledger:

Transaction Type	Direction	Description
`VBTC_V2_VALIDATOR_REGISTER`	VFX	Join validator pool with FROST key + Base address
`VBTC_V2_VALIDATOR_HEARTBEAT`	VFX	Liveness signal; required every ~500 blocks
`VBTC_V2_VALIDATOR_EXIT`	VFX	Leave validator pool
`VBTC_V2_CONTRACT_CREATE`	VFX	Create new vBTC V2 smart contract on VFX
`VBTC_V2_TRANSFER`	VFX	Transfer vBTC tokens between VFX addresses
`VBTC_V2_WITHDRAWAL_REQUEST`	VFX	Request BTC withdrawal from vBTC V2 contract
`VBTC_V2_WITHDRAWAL_COMPLETE`	VFX	FROST signing complete; BTC broadcast confirmed
`VBTC_V2_WITHDRAWAL_CANCEL`	VFX	Cancel pending withdrawal request
`VBTC_V2_BRIDGE_LOCK`	VFX	Lock vBTC on VFX; debit from tokenization ledger
`VBTC_V2_BRIDGE_POOL_UNLOCK`	VFX	Release locked vBTC after VFX pool exit burn
`VBTC_V2_BRIDGE_EXIT_TO_BTC_COMPLETE`	VFX	Record successful Bitcoin broadcast post-burn
`VBTC_V2_BRIDGE_EXIT_TO_BTC_FAIL`	VFX	Record failure; blacklist locks; reverse state

---

14. Data Models

VBTCBridgeLockState (Consensus DB — all nodes)

```
public class VBTCBridgeLockState
{
    [BsonId]
    public string LockId { get; set; }           // GUID from BRIDGE_LOCK TX
    public string SmartContractUID { get; set; } // vBTC V2 contract UID on VFX
    public string OwnerAddress { get; set; }     // VFX address of lock creator

    public decimal Amount { get; set; }          // Original lock amount (BTC)
    public long AmountSats { get; set; }         // Original lock amount (satoshis)

    // V3 partial unlock tracking
    public decimal UnlockedAmount { get; set; }
    public long UnlockedAmountSats { get; set; }
    public decimal RemainingAmount => Amount - UnlockedAmount;
    public long RemainingAmountSats => AmountSats - UnlockedAmountSats;

    public bool IsUnlocked { get; set; }         // True when UnlockedAmount >= Amount
    public bool IsBlacklisted { get; set; }
    public string? BlacklistReason { get; set; }
}
```

BridgeLockRecord (Local DB — caster node only)

```
public class BridgeLockRecord
{
    public string LockId { get; set; }
    public string SmartContractUID { get; set; }
    public string OwnerAddress { get; set; }
    public decimal Amount { get; set; }
    public long AmountSats { get; set; }
    public string EvmDestination { get; set; }          // Base recipient
    public string? VfxLockTxHash { get; set; }
    public bool VfxLockConfirmedOnChain { get; set; }
    public long VfxLockBlockHeight { get; set; }        // Used as mint nonce
    public string? BaseTxHash { get; set; }
    public BridgeLockStatus Status { get; set; }        // Locked → ProofSubmitted
    public Dictionary<string, string>? ValidatorSignatures { get; set; }
    public int RequiredSignatures { get; set; }
    public long MintNonce { get; set; }
}
```

VBTCContractV2 (VFX Contract State)

```
public class VBTCContractV2
{
    public string SmartContractUID { get; set; }
    public string OwnerAddress { get; set; }
    public string DepositAddress { get; set; }             // Taproot address (bc1p...)
    public decimal Balance { get; set; }                   // BTC (from Electrum)

    // FROST key material
    public List<string> ValidatorAddressesSnapshot { get; set; }
    public string FrostGroupPublicKey { get; set; }        // x-only 32-byte hex
    public string? FrostPubkeyPackage { get; set; }        // Aggregation package
    public int RequiredThreshold { get; set; }              // Percent (51 initially)

    // DKG provenance
    public string DKGProof { get; set; }
    public long ProofBlockHeight { get; set; }

    // Adaptive threshold state
    public long LastValidatorActivityBlock { get; set; }
    public int TotalRegisteredValidators { get; set; }

    // Active withdrawal state
    public VBTCWithdrawalStatus WithdrawalStatus { get; set; }
    public string? ActiveWithdrawalBTCDestination { get; set; }
    public decimal? ActiveWithdrawalAmount { get; set; }
}
```