VerifiedX Privacy Layer — Technical Paper

Abstract

This paper describes the design and implementation of the VerifiedX (VFX) Privacy Layer — an opt-in shielded transaction system that provides confidential transfers for both the native VFX asset and tokenized Bitcoin (vBTC). The privacy layer combines **Pedersen commitments** for value hiding, **PLONK zero-knowledge proofs** for validity without disclosure, **Poseidon-based Merkle trees** for commitment tracking, **ECDH-encrypted notes** for recipient-only decryption, and a **nullifier scheme** for double-spend prevention. All privacy features are optional, preserving full transparency for users who do not require confidentiality while enabling strong privacy guarantees for those who do.

Table of Contents

---

1. Introduction

VerifiedX is an account-based Proof of Stake Layer 1 blockchain. While its default transaction model operates transparently — with sender, recipient, and amount visible on-chain — certain use cases demand confidentiality. The VFX Privacy Layer addresses this by enabling users to move assets into a **shielded pool** where balances and transfers are protected by zero-knowledge cryptography.

The privacy layer supports three fundamental operations:

- **Shield (T→Z):** Move funds from a transparent address into the shielded pool.
- **Private Transfer (Z→Z):** Transfer funds entirely within the shielded pool, hiding sender, recipient, and amount.
- **Unshield (Z→T):** Move funds from the shielded pool back to a transparent address.

These operations are available for both VFX (the native asset) and vBTC (tokenized Bitcoin on the VFX chain), making VerifiedX one of few blockchains to offer privacy for bridged Bitcoin.

2. Design Principles

The privacy layer was designed around five core principles:

1. **Opt-in Privacy:** Privacy features are entirely optional. Users explicitly choose to shield or unshield funds. Standard transparent transactions remain the default and are unaffected.

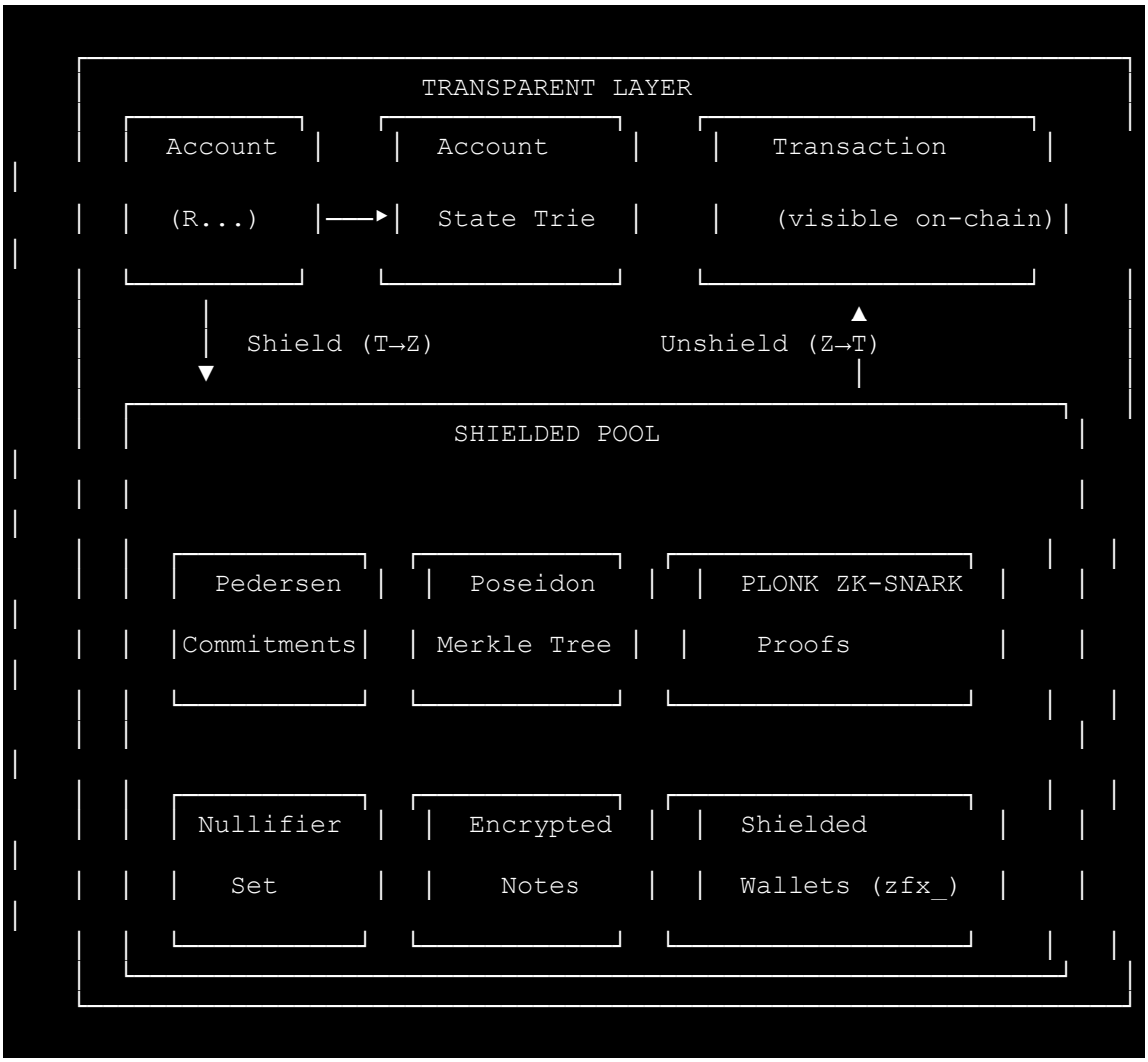
2. **Composability:** The shielded pool coexists with the transparent account model. Users can freely move between transparent and shielded states.

3. **Multi-Asset Support:** A single unified architecture supports privacy for multiple asset types (VFX and vBTC), with per-asset shielded pools and Merkle trees.

4. **Auditability:** View keys allow third-party auditing of shielded balances and transaction history without granting spending authority, enabling regulatory compliance.

5. **Proven Cryptography:** The system relies on well-established cryptographic primitives — Pedersen commitments on BLS12-381, PLONK ZK-SNARKs, Poseidon hashes, and secp256k1 ECDH — rather than novel or experimental constructions.

3. Architecture Overview



The shielded pool is a virtual construct represented by the sentinel address `Shielded_Pool`. When funds are shielded, VFX (or vBTC) is debited from the sender's transparent balance and a Pedersen commitment is added to the asset-specific Merkle tree. When funds are unshielded, nullifiers prove the commitment has not been previously spent, and the transparent recipient is credited.

4. Cryptographic Primitives

Primitive	Curve / Algorithm	Purpose
Pedersen Commitments	BLS12-381 (G1 compressed, 48 bytes)	Hide transaction amounts while enabling homomorphic verification
PLONK ZK-SNARKs	BLS12-381	Prove transaction validity (correct amounts, unspent inputs, valid nullifiers) without revealing private data
Poseidon Hash	BLS12-381 scalar field	SNARK-friendly hash for Merkle tree leaves (`NoteHash = Poseidon(amount_scaled, randomness)`) and nullifier derivation
SHA-256	—	Key derivation (viewing keys, spending key normalization)
secp256k1 ECDH	secp256k1	Ephemeral key exchange for encrypting notes to recipients
AES-256-GCM	—	Symmetric encryption of shielded notes after ECDH key agreement
BIP32 / BIP39	secp256k1	HD key derivation for shielded wallets

4.1 Why BLS12-381?

BLS12-381 is chosen for commitments and proofs because it provides a 128-bit security level, is widely audited, and is the standard curve for PLONK-based proof systems. The Pedersen commitment scheme on BLS12-381 G1 produces 48-byte compressed commitments, offering a compact on-chain footprint.

4.2 Why Poseidon?

Poseidon is an algebraic hash function designed specifically for efficient evaluation inside arithmetic circuits (ZK-SNARKs). Using Poseidon for note hashes and nullifier derivation minimizes the constraint count in PLONK circuits, enabling faster proving and verification times compared to SHA-256 inside a circuit.

5. Shielded Key Hierarchy

Each shielded wallet derives a hierarchical key bundle from the user's existing BIP39 seed or private key, ensuring a single backup (mnemonic phrase) recovers both transparent and shielded funds.

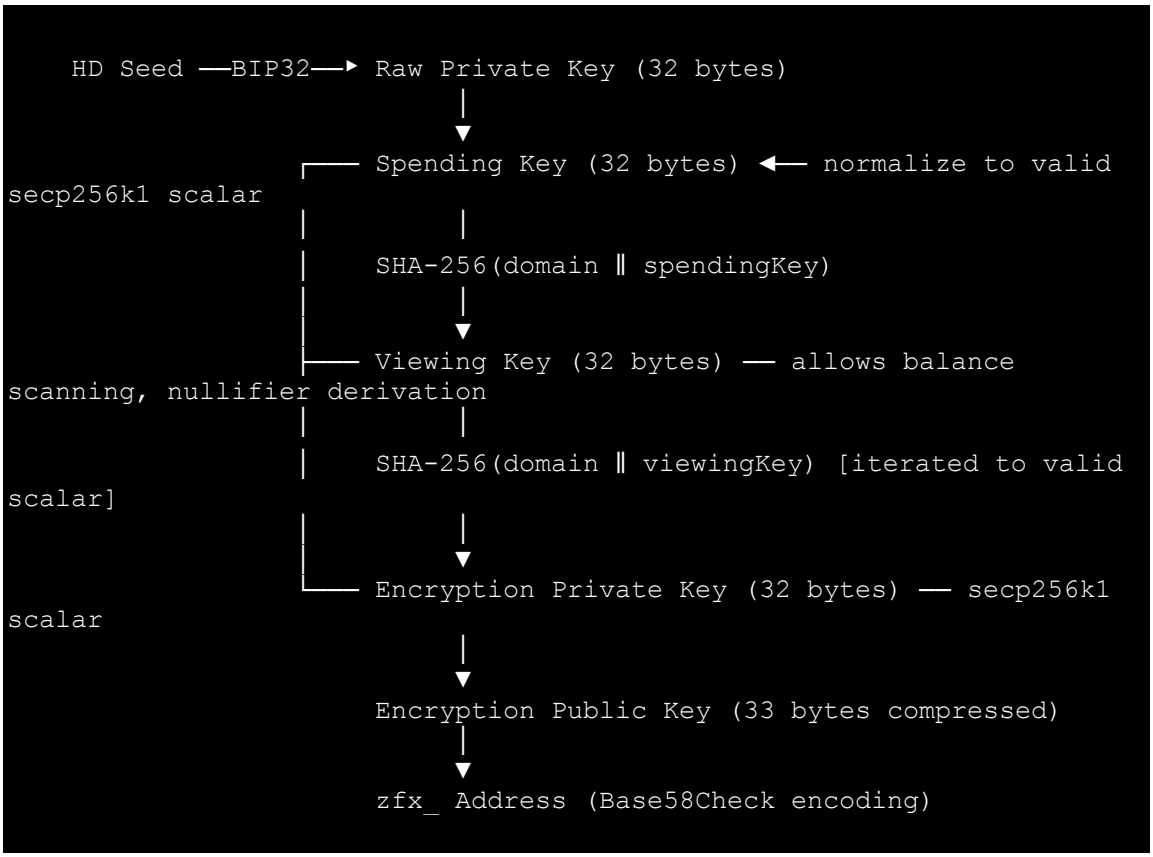
5.1 Key Derivation Path

```
BIP44 Path: m/44'/{coinType}'/0'/1'/{addressIndex}'
                                 ↑
                          Shielded branch (chain index 1')
```

The shielded branch uses chain index `1'` (hardened), parallel to the transparent branch at `0'`.

5.2 Key Bundle

From the HD-derived 32-byte secret, three keys are deterministically derived:



Domain Separation Tags:

- Spending key normalization: `VFX/shielded/spend-scalar/v1`
- Viewing key derivation: `VFX/shielded/viewing/v1`
- Encryption key derivation: `VFX/shielded/enc-priv/v1`

5.3 Shielded Addresses (zfx_)

The user's shielded address (`zfx_` prefix) encodes the 33-byte compressed secp256k1 encryption public key. This address is safe to share publicly — it allows others to encrypt notes to the recipient but does not reveal viewing or spending capability.

5.4 Key Capabilities

Key	Can Scan Balances	Can Spend	Can Decrypt Notes	Can Derive Nullifiers
Spending Key	✓	✓	✓	✓
Viewing Key	✓	✗	✓	✓
Encryption Key	✗	✗	✓	✗
zfx_ Address	✗	✗	✗	✗

6. Transaction Types

The privacy layer introduces six on-chain transaction types:

Transaction Type	Direction	Asset	Description
`VFX_SHIELD`	T → Z	VFX	Shield native VFX into the shielded pool
`VFX_UNSHIELD`	Z → T	VFX	Unshield VFX back to a transparent address
`VFX_PRIVATE_TRANSFER`	Z → Z	VFX	Fully private transfer within the shielded pool
`VBTC_V2_SHIELD`	T → Z	vBTC	Shield tokenized Bitcoin into the shielded pool
`VBTC_V2_UNSHIELD`	Z → T	vBTC	Unshield vBTC back to a transparent address
`VBTC_V2_PRIVATE_TRANSFER`	Z → Z	vBTC	Fully private vBTC transfer within the shielded pool

Shield transactions (T→Z) are authorized by a standard ECDSA signature from the transparent sender. Unshield and private transfer transactions (Z→T, Z→Z) are authorized by PLONK zero-knowledge proofs — no private key signature is exposed on-chain. These ZK-authorized transactions use the sentinel signature `"PLONK"`.

7. Pedersen Commitment Scheme

7.1 Amount Encoding

VFX amounts (up to 8 decimal places) are mapped to unsigned 64-bit integers using a fixed-point scaling factor of 10^8:

```
scaled_amount = amount × 100,000,000
```

This supports values up to approximately 184 billion VFX — far beyond the 200 million total supply — while maintaining full decimal precision.

7.2 Commitment Construction

For each shielded output, a Pedersen commitment is computed on the BLS12-381 G1 curve:

```
C = amount_scaled · G + randomness · H
```

Where:

- `G` and `H` are fixed, independent BLS12-381 G1 generators (nothing-up-my-sleeve points)
- `amount_scaled` is the 64-bit scaled amount
- `randomness` is a cryptographically secure random 32-byte scalar (generated via `System.Security.Cryptography.RandomNumberGenerator`)
- `C` is a 48-byte compressed G1 point

The commitment is **computationally hiding** (an observer cannot determine the amount without the randomness) and **perfectly binding** (the committer cannot open the commitment to a different amount).

7.3 Homomorphic Properties

Pedersen commitments are additively homomorphic: $C(a, r_1) + C(b, r_2) = C(a+b, r_1+r_2)$. This property is leveraged by the PLONK circuits to verify that input

commitments balance against output commitments plus fees — all without revealing the actual values.

8. Note Encryption & Decryption

When a shielded output is created, the sender encrypts a **plain note** containing the amount and randomness so that only the intended recipient can decrypt it and later spend the funds.

8.1 Plain Note Structure

```
{
  "v": 1,
  "amount": 10.5,
  "randomness_b64": "<32 bytes, Base64>",
  "asset": "VFX",
  "memo": "optional message"
}
```

The plain note contains everything the recipient needs to reconstruct the Pedersen commitment and derive the nullifier for future spending.

8.2 Encryption Protocol (ECIES-like)

1. **Ephemeral Key Generation:** Sender generates a fresh secp256k1 keypair `(ek_priv, ek_pub)`.

2. **ECDH Key Agreement:** Sender computes shared secret: `S = ECDH(ek_priv, recipient_pub)` where `recipient_pub` is decoded from the recipient's `zfx_` address.

3. **KDF:** A symmetric AES-256 key is derived: `aes_key = DoubleSHA-256("VFX/shielded/note-aes/v1" ‖ S ‖ ek_pub ‖ recipient_pub)`.

4. **Encryption:** The serialized plain note is encrypted with AES-256-GCM using a random 12-byte nonce.

5. **Sealed Blob:** The on-chain encrypted note is: `version (1 byte) ‖ ek_pub (33 bytes) ‖ nonce (12 bytes) ‖ GCM tag (16 bytes) ‖ ciphertext`

8.3 Decryption

The recipient:

1. Extracts the ephemeral public key from the sealed blob.

2. Computes the same ECDH shared secret using their encryption private key.

3. Derives the same AES key and decrypts the note.
4. Recovers the amount and randomness, enabling future spending.

8.4 Security Properties

- **Forward secrecy:** Each note uses a fresh ephemeral key; compromising the recipient's key does not reveal past notes encrypted with different ephemeral keys.
- **Authenticated encryption:** AES-256-GCM ensures integrity — tampered ciphertexts fail decryption.
- **Recipient-only access:** Only the holder of the encryption private key (derivable from the viewing key) can decrypt the note.

9. Shielded Merkle Tree

9.1 Structure

Each asset (VFX, each vBTC contract) maintains its own append-only Merkle tree of commitments. The tree provides:

- **Inclusion proofs:** Prove that a commitment exists in the tree at a specific position.
- **State root:** A single 32-byte digest representing the current state of all shielded commitments.

9.2 Leaf Construction

Each leaf is derived from the **note hash** (preferred, v2) or the G1 commitment (legacy fallback):

```
leaf = Poseidon(amount_scaled, randomness) // v2 - SNARK-friendly
leaf = G1_compressed_commitment            // v1 legacy fallback
```

The v2 path uses the Poseidon note hash directly as the Merkle leaf, providing in-circuit amount binding and allowing the PLONK circuit to efficiently verify that the amount committed to in G1 matches the leaf in the Merkle tree.

9.3 Internal Nodes

Internal nodes use Poseidon: `node = Poseidon(left ‖ right)`. This keeps the entire Merkle tree within the same algebraic hash family, enabling efficient in-circuit verification of inclusion proofs.

9.4 Merkle Root Recency

To prevent replay attacks with stale state, transactions include the expected Merkle root. Validators reject transactions whose Merkle root is more than **100 blocks** old (`MaxMerkleRootAge`), ensuring that spending proofs reference a reasonably recent view of the shielded pool state.

---

10. Nullifier Scheme

Nullifiers prevent double-spending of shielded commitments. Each commitment can be spent exactly once; the corresponding nullifier is published on-chain when spent.

10.1 Nullifier Derivation

v2 (preferred — note-hash based):

```
    nullifier = Poseidon(viewing_key, note_hash, tree_position)
```

Where `note_hash = Poseidon(amount_scaled, randomness)`.

v1 (legacy — G1-based):

```
    nullifier = plonk_ffi.nullifier_derive(viewing_key, G1_commitment,
tree_position)
```

Using the viewing key ensures that only the owner (holder of the viewing key) can derive the correct nullifier. The tree position binds the nullifier to a specific commitment slot, preventing ambiguity.

10.2 Double-Spend Prevention

- **On-chain:** Validators maintain a persistent nullifier set per asset. If a transaction's nullifiers already exist in the set, the transaction is rejected.

- **Mempool:** A concurrent nullifier tracker prevents conflicting transactions from coexisting in the mempool. If a nullifier is already claimed by a pending transaction, new transactions attempting to spend the same commitment are rejected.
- **Block scope:** During block validation, an in-memory set tracks nullifiers within the current block to catch intra-block double-spends.

10.3 Nullifier Properties

- **Deterministic:** The same commitment always produces the same nullifier (given the same viewing key), ensuring consistent validation.
- **Unlinkable:** Observers cannot link a nullifier back to its source commitment without the viewing key, preserving the privacy of which commitment was spent.

---

11. PLONK Zero-Knowledge Proofs

11.1 Proof System

The privacy layer uses **PLONK** (Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge), a universal ZK-SNARK system with the following properties:

- **Universal trusted setup:** A single set of structured reference parameters supports all circuit sizes, eliminating per-circuit ceremonies.
- **Succinct proofs:** Proofs are compact (hundreds of bytes), enabling efficient on-chain verification.
- **Non-interactive:** Proofs can be verified by any node without interaction with the prover.

The PLONK implementation is provided via a native FFI library (`plonk-ffi`) compiled in Rust and linked to the C#/.NET node runtime.

11.2 Circuit Types

Circuit	Used For	Public Inputs
Shield	T→Z transactions	Output commitment, amount (transparent), Merkle root, asset tag
Transfer	Z→Z transactions	Input nullifiers, output commitments, Merkle root, fee, asset tag
Unshield	Z→T transactions	Input nullifiers, output commitments, transparent amount, recipient, Merkle root, fee, asset tag
Fee	Cross-asset fee payment (vBTC Z→Z/Z→T)	VFX nullifier, VFX change commitment, VFX Merkle root, fee amount

11.3 Public Inputs Structure (v1)

Public inputs are deterministically constructed from the on-chain `PrivateTxPayload` and include:

1. **Circuit header** (1 byte): Identifies the circuit type.
2. **Asset tag** (32 bytes): SHA-256 hash of the asset name, binding the proof to a specific asset pool.
3. **Merkle root** (32 bytes): The expected shielded pool state root.
4. **Nullifiers** (32 bytes each, padded to max inputs): Spent commitment nullifiers.
5. **Output commitments** (48 bytes each, padded to max outputs): New Pedersen commitments.
6. **Note hashes** (32 bytes each): Poseidon hashes binding circuit amounts to Merkle leaves.
7. **Amount / fee fields**: Scaled 64-bit integers for transparent amounts and fees.

11.4 Proving & Verification

Proving (client-side):
The native FFI exposes circuit-specific proving functions (`plonk_prove_shield`, `plonk_prove_transfer`, `plonk_prove_unshield`, `plonk_prove_fee`) that take private witness data (amounts, randomness, Merkle paths) and produce a proof along with public inputs.

Verification (validator-side):
Validators call `plonk_verify(circuit_type, proof, public_inputs)` for each private transaction. The verification is constant-time regardless of circuit complexity. A batch verification API (`PlonkBatchVerifier`) enables efficient validation of multiple proofs in a single block.

11.5 Parameter Management

PLONK structured reference parameters are distributed as a file (~several MB) and loaded at node startup via `plonk_load_params`. The node supports:

- Automatic parameter download from official CDN
- File hash verification before loading
- Environment variable configuration for custom parameter paths

12. Transaction Lifecycle

12.1 Shield (T→Z)

```
    1. User selects transparent address and amount to shield
    2. System generates fresh 32-byte randomness
    3. Pedersen commitment: C = Commit(amount_scaled, randomness)
    4. Note hash: NH = Poseidon(amount_scaled, randomness)
    5. Plain note created: { amount, randomness, asset }
    6. Note encrypted to recipient's zfx_ address via ECDH + AES-256-
GCM
    7. PLONK shield proof generated
    8. Transaction constructed:
       - FromAddress: user's transparent R-address
       - ToAddress: "Shielded_Pool"
       - Amount: shield amount (visible, debited from transparent
balance)
       - Data: PrivateTxPayload { commitment, encrypted note, note
hash, proof, merkle root }
       - Signature: ECDSA signature from transparent key
    9. Broadcast to network
    10. Validators verify: signature, sufficient balance, valid PLONK
proof
    11. On acceptance: transparent balance debited, commitment appended
to Merkle tree
```

12.2 Private Transfer (Z→Z)

```
    1. Wallet selects unspent commitments as inputs (max 2)
    2. For each input: derive nullifier from note hash + viewing key +
tree position
    3. Compute payment commitment and change commitment (if needed)
    4. Encrypt payment note to recipient's zfx_ address
    5. Encrypt change note to sender's own zfx_ address
    6. PLONK transfer proof generated (proves: inputs balance outputs +
fee, nullifiers valid)
    7. Transaction constructed:
       - FromAddress: "Shielded_Pool"
       - ToAddress: "Shielded_Pool"
       - Amount: 0 (hidden)
       - Signature: "PLONK" (sentinel — proof replaces signature)
       - Data: PrivateTxPayload { nullifiers, commitments, encrypted
notes, proof, merkle root }
    8. Broadcast to network
    9. Validators verify: nullifiers not spent, valid PLONK proof,
Merkle root recency
    10. On acceptance: nullifiers recorded as spent, new commitments
appended to tree
```

12.3 Unshield (Z→T)

```
    1. Wallet selects unspent commitments as inputs
    2. Derive nullifiers for all inputs
    3. Compute change commitment (if input sum > unshield amount + fee)
    4. PLONK unshield proof generated
    5. Transaction constructed:
       - FromAddress: "Shielded_Pool"
       - ToAddress: recipient's transparent R-address
       - Amount: unshield amount (visible, credited to transparent
balance)
       - Signature: "PLONK"
       - Data: PrivateTxPayload { nullifiers, change commitment, proof,
merkle root }
    6. On acceptance: nullifiers recorded, transparent recipient
credited, change commitment appended
```

---

13. Consensus Integration & Validation

13.1 Block-Level Limits

- **Maximum private transactions per block:** 50 (`MaxPrivateTxPerBlock`)
- **Maximum payload size:** 8,192 characters (`MaxPrivateTxDataSize`)
- **Maximum inputs per transaction:** 2 (`MaxPrivateTxInputs`)
- **Maximum outputs per transaction:** 2 (`MaxPrivateTxOutputs`)

13.2 Validation Pipeline

Every private transaction undergoes a multi-stage validation:

1. **Payload Decode & Structure:** JSON payload is decoded and structural invariants are checked (version, field lengths, Base64 validity, G1/scalar sizes).
2. **Kind Cross-Check:** The payload's `kind` field must match the transaction type (e.g., `shield` for `VFX_SHIELD`).
3. **Merkle Root Recency:** The payload's Merkle root must match a root from the last 100 blocks.
4. **Nullifier Uniqueness:** All nullifiers must be unseen in both the persistent nullifier set and the current block's pending set.
5. **Transparent Balance (Shield):** For T→Z, the sender must have sufficient transparent balance.
6. **PLONK Proof Verification:** For Z→Z and Z→T, the proof must verify against the deterministically reconstructed public inputs.
7. **Hash Integrity:** The transaction's privacy hash is verified to ensure payload integrity.

13.3 Block Proof Verification

Validators perform batch PLONK proof verification for all private transactions in a block via `PlonkBatchVerifier.TryVerifyAll()`, which returns the index of the first invalid proof (if any) for precise error reporting.

13.4 Ledger Application

Upon block acceptance:

- Shield outputs are added to the commitment Merkle tree
- Nullifiers are recorded in the persistent nullifier database

- Shielded pool supply accounting is updated
- Transparent balances are debited (shield) or credited (unshield)
- The shielded state root is updated

---

14. Multi-Asset Privacy (vBTC)

14.1 Per-Asset Pools

Each vBTC smart contract has its own independent shielded pool, Merkle tree, and nullifier set. The asset key is formatted as `VBTC:{contractUID}`, keeping pools isolated.

14.2 Cross-Asset Fee Payment

vBTC private transactions (Z→Z and Z→T) require fees to be paid in VFX, not vBTC. This is accomplished via a **fee leg** — a separate PLONK proof that spends a VFX shielded commitment to cover the fee:

```
    vBTC Z→Z Transaction:
    ├── Primary proof: vBTC transfer (nullifiers, commitments)
    └── Fee proof: VFX fee payment (VFX nullifier, VFX change
commitment)
```

The fee proof operates on the VFX Merkle tree while the primary proof operates on the vBTC Merkle tree, with both proofs included in the same `PrivateTxPayload`.

14.3 Minimum Amounts

- **VFX minimum shield:** 0.001 VFX (`MinShieldAmountVFX`)
- **vBTC minimum shield:** 0.00001 vBTC (`MinShieldAmountVBTC`)

---

15. Auditability & View Keys

15.1 View Key Export

Users can export their 32-byte viewing key, which enables a third party to:

- **Scan** the blockchain for all shielded notes belonging to that wallet
- **Compute balances** by summing unspent commitments
- **Derive nullifiers** to determine which commitments have been spent
- **Decrypt notes** (the viewing key allows deriving the encryption private key)

The viewing key **cannot** authorize spending. Only the spending key can construct valid PLONK proofs for Z→Z or Z→T transactions.

15.2 API Endpoints

Endpoint	Description
`ExportViewingKey`	Export the viewing key for a shielded wallet
`ImportViewingKey`	Import a viewing key to create a watch-only shielded wallet
`ScanShielded`	Scan blockchain for notes belonging to a shielded wallet
`GetShieldedBalance`	Query the current shielded balance
`GetShieldedPoolState`	Query aggregate pool state (total supply, commitment count, root)

15.3 Auto-Scanner

The node includes a real-time **auto-scanner** that processes each new block for shielded notes belonging to any local wallet. This eliminates the need for manual `ScanShielded` calls — balances update automatically as blocks arrive.

The auto-scanner:

1. Checks if a block contains any private transactions
2. Loads all local shielded wallets and pre-derives viewing key material

3. Attempts trial decryption of each encrypted note with each wallet's encryption key
4. Successfully decrypted notes are recorded as unspent commitments
5. Detects spent notes by matching published nullifiers against known commitments

---

16. On-Chain Data Model

16.1 PrivateTxPayload (stored in Transaction.Data)

```
{
  "v": 1,
  "kind": "shield | unshield | private_transfer",
  "sub_type": "Shield | Unshield | PrivateTransfer",
  "asset": "VFX",
  "outs": [
    {
      "i": 0,
      "c": "<Pedersen commitment, G1 compressed, Base64>",
      "nh": "<Poseidon note hash, 32 bytes, Base64>",
      "note": "<ECDH-encrypted note, Base64>"
    }
  ],
  "nulls": ["<nullifier, 32 bytes, Base64>"],
  "spent_tree_positions": [42],
  "merkle_root": "<current Merkle root, Base64>",
  "proof_b64": "<PLONK proof, Base64>",
  "transparent_input": "<R-address for shield>",
  "transparent_output": "<R-address for unshield>",
  "transparent_amount": 10.5,
  "fee": 0.000003
}
```

16.2 Persistent Storage

Collection	Contents
`CommitmentRecord`	All commitments: G1 bytes, note hash, tree position, block height, asset type, spent status
`NullifierRecord`	All published nullifiers with block height and timestamp
`ShieldedPoolState`	Per-asset pool state: total shielded supply, commitment count, current Merkle root
`ShieldedWallet`	Local wallet metadata: zfx address, encrypted spending key, viewing key, asset type

Collection	Contents
`MerkleTreeNodeRecord`	Persisted Merkle tree nodes for efficient proof generation

---

17. Protocol Parameters

Parameter	Value	Description
`PrivateTxFixedFee`	0.000003 VFX	Fixed fee for all private transactions
`MaxPrivateTxPerBlock`	50	Maximum private transactions per block
`MaxPrivateTxInputs`	2	Maximum input commitments per transaction
`MaxPrivateTxOutputs`	2	Maximum output commitments per transaction
`MaxPrivateTxDataSize`	8,192 chars	Maximum payload JSON size
`MaxMerkleRootAge`	100 blocks	Merkle root recency window (~20 minutes)
`MinShieldAmountVFX`	0.001 VFX	Minimum VFX shield amount
`MinShieldAmountVBTC`	0.00001 vBTC	Minimum vBTC shield amount
`PrivacyAmountScalingFactor`	10^8	Fixed-point scaling for circuit amounts
`G1CompressedSize`	48 bytes	BLS12-381 G1 commitment size
`ScalarSize`	32 bytes	BLS12-381 scalar / nullifier / randomness size

---

18. Security Analysis

18.1 Confidentiality

- **Amount hiding:** Pedersen commitments are computationally hiding under the discrete logarithm assumption on BLS12-381.
- **Sender/recipient hiding (Z→Z):** Private transfers reveal only that a Z→Z transfer occurred; no addresses or amounts are exposed on-chain.
- **Note confidentiality:** ECDH + AES-256-GCM ensures only the intended recipient can decrypt the note.

18.2 Integrity

- **Binding:** Pedersen commitments are perfectly binding — a commitment cannot be opened to two different amounts.
- **Soundness:** PLONK proofs ensure that any transaction accepted by validators correctly balances inputs and outputs (including fees), and that all nullifiers correspond to real, unspent commitments.
- **Replay protection:** Merkle root recency checks (100-block window) prevent stale proof reuse.

18.3 Double-Spend Prevention

- **Nullifier uniqueness:** The persistent nullifier set, mempool tracker, and block-scoped set collectively guarantee that each commitment is spent at most once.
- **Consensus enforcement:** All validators independently verify nullifier uniqueness and PLONK proofs before accepting blocks.

18.4 Key Separation

- The three-level key hierarchy (spending → viewing → encryption) ensures that compromising a lower-privilege key does not grant spending authority.
- View keys enable auditing without risk of fund loss.

18.5 Limitations & Mitigations

Concern	Mitigation
Shield/unshield reveals amounts	Inherent to T↔Z transitions; users can shield in private, then transfer Z→Z
Transaction graph analysis	Fixed fee, max 2 inputs/outputs limits metadata leakage; decoy outputs possible
Timing correlation	Users advised to avoid predictable shield/unshield patterns
Pool size privacy set	As more users shield funds, the anonymity set grows naturally

---

19. Future Work

- **Increased input/output counts:** Expanding beyond 2-in/2-out to support more complex transactions.
- **Recursive proofs:** Aggregating multiple PLONK proofs into a single succinct proof for reduced on-chain verification cost.
- **Cross-chain shielded transfers:** Extending privacy to bridged assets on other chains.
- **Stealth addresses:** One-time destination addresses for enhanced recipient privacy.

---

20. References

1. **PLONK:** Gabizon, A., Williamson, Z.J., Ciobotaru, O. "PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge." *IACR ePrint* 2019/953.

2. **Pedersen Commitments:** Pedersen, T.P. "Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing." *CRYPTO 1991*.

3. **Poseidon Hash:** Grassi, L., et al. "Poseidon: A New Hash Function for Zero-Knowledge Proof Systems." *USENIX Security 2021*.

4. **BLS12-381:** Bowe, S. "BLS12-381: New zk-SNARK Elliptic Curve Construction." *Zcash Blog*, 2017.

5. **BIP32:** Wuille, P. "Hierarchical Deterministic Wallets." *Bitcoin Improvement Proposal 32*.

6. **BIP39:** Palatinus, M., et al. "Mnemonic code for generating deterministic keys." *Bitcoin Improvement Proposal 39*.

7. **AES-GCM:** Dworkin, M.J. "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM)." *NIST SP 800-38D*.

8. **VerifiedX Core Repository:** https://github.com/VerifiedXBlockchain/VerifiedX-Core

---