Satyabit Protocol 副本

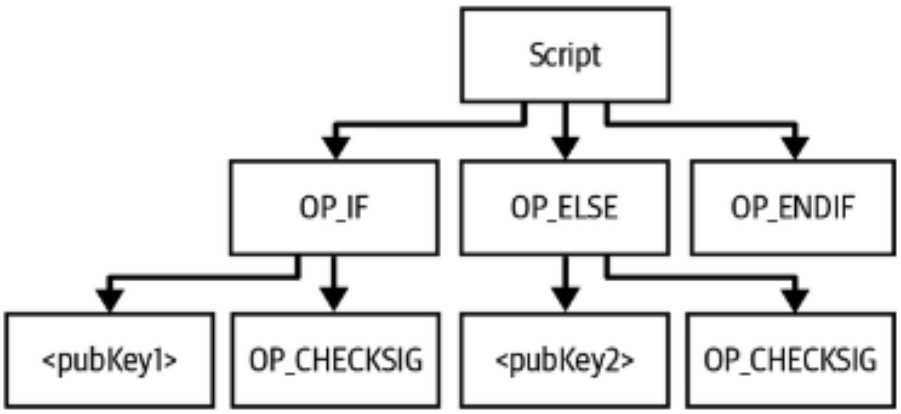
Satyabit Protocol, driven by Bitcoin's MAST (Merkelized Abstract Syntax Trees) contracts, is an innovative data protocol specifically designed for the Bitcoin network. Through the Satyabit Protocol, users can issue and manage UTXO assets on the Bitcoin network, introducing new application scenarios and value transfer methods. Combined with the Sat DA node verification system, the Satyabit Protocol can help other assets on the Bitcoin network expand their applications, improve the scalability and interoperability of the entire ecosystem.

The Satyabit Protocol will become the infrastructure for opening up new application areas and diversified scenarios on the Bitcoin network.

What's MAST Contracts?

Bitcoin has already been using a data structure called a Merkle tree, which allows verifying that an element is a member of a set without identifying all the other members of the set.

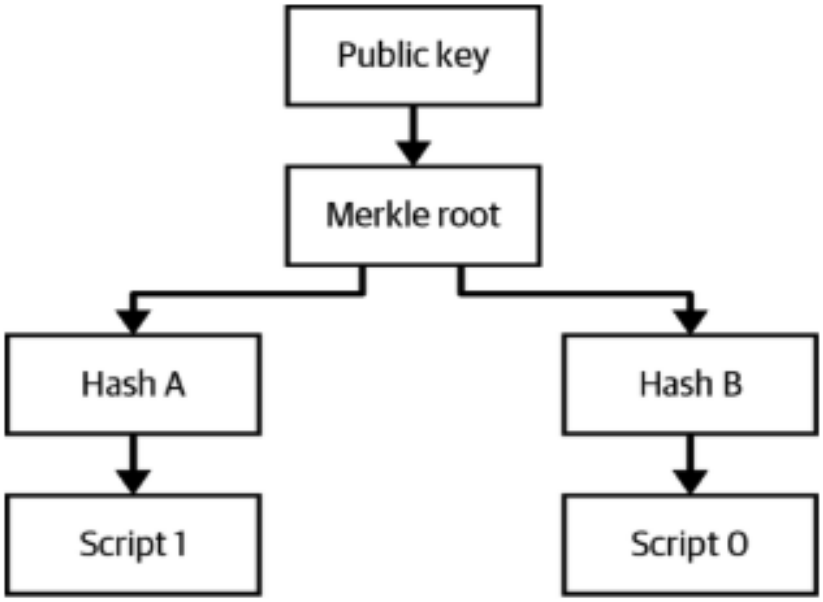
In Bitcoin, what people commonly refer to as MAST stands for Merkelized Abstract Syntax Trees. A Merkelized Abstract Syntax Tree is a set of scripts, each of which is complete in itself, but only one can be chosen - they are interchangeable with each other.



For the abstract syntax tree, only the 32-byte digests from the spender's chosen script to the root of the tree need to be revealed. For most scripts, this way is more space-efficient on Bitcoin blocks.

The program code of the contract is committed by committing to the root of the MAST. By spending through the script path, the on-chain data contains the public key (in the witness program) and the signature (on the witness stack). New key signatures can be committed,

meaning it can satisfy the Taproot key generated by the contract by revealing the MAST branch we want to use for transaction commitments.



A Taproot commits its public key to a Merkle root.

Taproot transactions

Taproot is a new transaction type defined in BIP 341, which has been fully activated on the Bitcoin mainnet since November 2021. The main difference between Taproot transactions and traditional Bitcoin transactions is that the scripts controlling the funds are contained within a tree-like structure called a "TapScript branch," which is privately committed to the transaction. If funds are moved using the KeySpend path, there is no need to disclose these scripts publicly.

Traditional Bitcoin transactions require the public disclosure of the entire script, whereas Taproot transactions allow for payments using a key without disclosing the script. If the KeySpend path is not viable, only the script parts executed on the blockchain will be disclosed. All other script paths can remain private or be disclosed off-chain at the discretion of the parties involved.

This enables the creation of more complex scripts under the KeySpend path without submitting additional data to the blockchain, while still efficiently verifying pruned script data. In the context of the Satyabit Protocol, it allows for the attachment of verifiable arbitrary data to UTXO transactions without leaking this data on-chain.

Tapetching

The Satyabit Protocol, any changes to the state of arbitrary Taproot script tree data are committed as UTXO transactions. A UTXO that contains Satyabit data (Satyabit Fractal Hash Trees) within a Bitcoin UTXO is known as an etched UTXO transaction. Once this transaction is included in a Bitcoin block, the data is etched, making them immutable and unchangeable.

To etch data, we use a method called "Tapetching" to adjust our Taproot payment key's public key. It allows us to selectively reveal data without disclosing the private key, or spend outputs without revealing the etched actual data. The Taproot script tree used in Tapetching is known as script path payment. For script path payments, the on-chain data includes a public key placed in a witness program, referred to as the Taproot output key in this context.

The witness structure is the core of etched data.

It includes the following information:

- Version number;
- The underlying key - a key that exists before being adjusted by the Merkle root to generate the taproot output key. This underlying key is referred to as the taproot internal key;
- The script to be executed, known as a leaf script;
- A 32-byte hash value at each fork along the path that connects the leaf to the Merkle root;
- Any data (Satyabit Fractal Hash Trees) required to satisfy the script.

Tapetching is used in UTXO transaction outputs to commit to the Taproot script tree and can be used to commit to arbitrary data.

Tapetching Commitments

Construction of Tapetching commitments is based on the OP_RETURN opcode in UTXO transaction outputs and is placed within an unspendable script path in the Taproot script tree. Therefore, this script/commitment is not exposed to the outside world in Bitcoin transactions or blockchain data. It is a commitment to the actual Tapetching data contained within the UTXO transaction output ScriptPubKey, created using the standard Taproot process defined in BIP-341.

Tapetching （Satyabit Fractal Hash Trees）

The length of Tapetching commitment scripts (Taproot leaf scripts) is always 64 bytes, allowing them to be distinguished from the data used to generate Taproot branch hashes. The proof of the absence of alternative Taproot commitments can be verified by a simple comparison of the first child node hash with the Taproot leaf script prefix.

Commitment Position

Leaves with Taproot leaf scripts are always located at depth 1 (with an index starting from 0) in the Taproot script tree. If there is another Taproot node (either a leaf script or a branch) at the depth 1 position on the far right of the tree, an additional branch node is created, whose children include the Tapetching leaf script and the parent of the far right depth 1 node.

The commitment is always placed in the rightmost node at this depth, using the node consensus order defined in the Merkle path construction of BIP-341.

If the original Taproot script tree does not have a node at depth 1 on its right side, a Tapetching subtree composed of duplicated Tapetching script leaf nodes is created and inserted at the depth of the last node on the right side Merkle path of the tree. The subtree's height is selected to place the Taproot script leaf node at depth 1.

The deterministic nature of the Tapetching commitment requires the ability to prove that there are no alternative commitments within the same tree, thus two components are used to maintain the determinacy of the commitment position and to prove its uniqueness.

Firstly, a special one-byte variable (a nonce) is added to the Taproot leaf script, allowing for the hash value to be "mined" in such a way that the added subtree will appear on the right side of the tree. At depth 1, there are only 2 possible tree positions, and iterating each nonce value 256 times is sufficient to address this issue.

Secondly, if the issue cannot be resolved through nonce, a special uniqueness proof is generated to ensure that any nodes at depth 1 on the right side of the tree do not contain alternative commitments.

Commitment Output and Proof

Under the OP_RETURN opcode mechanism, Tapetching commitments are placed in the ScriptPubKey of a UTXO transaction output, generated by the new Merkle root of a Taproot script tree embedded with the Tapetching commitment. If multiple Taproot outputs exist in the same transaction, the transaction output number containing the commitment must be deterministically defined by a higher-level protocol using the current Tapetching commitment scheme. Combining the internal key value, Merkle proof, and uniqueness proof, a Taproot proof is constructed. As UTXO etched data, it is passed to the Satyabit DA Node for verification before sending transaction information to Bitcoin.

Fractal Hash Trees

Fractal Hash Trees (F-Hash Trees) are an advanced data structure that leverages the self-similarity and recursive characteristics of fractal theory to optimize the storage and verification process of key-value pairs. In Fractal Hash Trees, the fundamental relationship between the keys or positions of the leaves and their contents is preserved and enhanced through fractal self-

similarity and recursive properties. This approach offers a method for the efficient, scalable, and verifiable management of key-value pairs in distributed systems.

Structure and Attributes

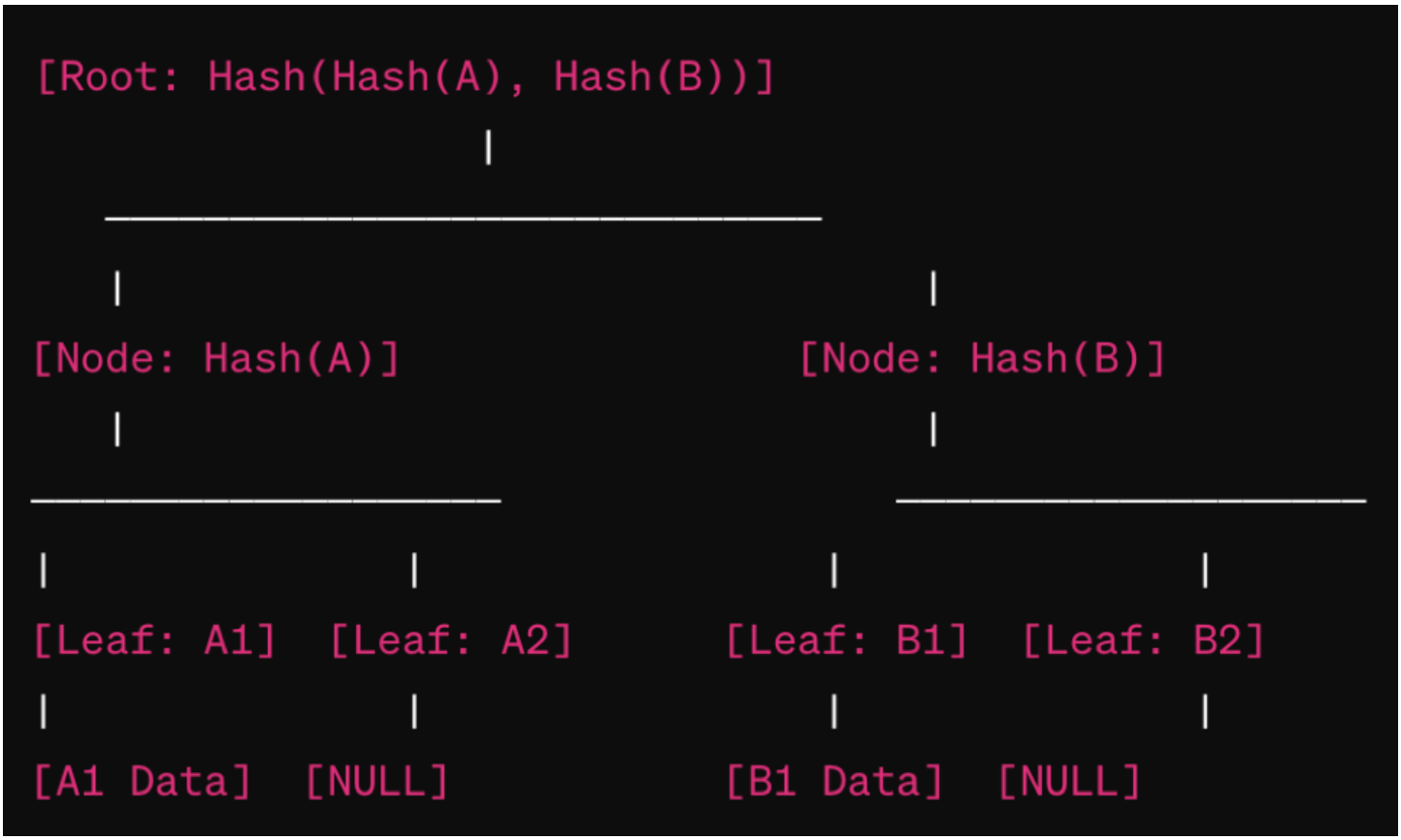
Fractal Hash Trees (F-Hash Trees) integrate fractal properties into the architecture of hash trees. F-Hash Trees use hashing algorithms to bind the content of the leaves with their position in the tree, ensuring that the structure can serve as an authenticated key-value store. However, F-Hash Trees optimize the construction and navigation of these trees through the use of fractal self-similarity concepts.

Fractal Design
Each node within the Fractal Hash Trees is designed to mimic the structure of the entire tree on a smaller scale. This self-similarity is recursively applied down to the leaf nodes, making the organization of data more efficient and reducing the number of hash calculations needed for data verification.

Dynamic Layering
The tree dynamically adjusts its structure based on the distribution of data, using a fractal-inspired method to minimize storage space by merging or splitting nodes. This results in an efficient tree structure that can adapt to varying data sizes and densities without compromising performance.

Self Similar Structure
Each node is structurally a miniature of the whole hash tree. This design not only optimizes the storage layout but also simplifies the data verification path. In this way, efficient access and verification speeds are maintained even when handling large datasets.

Hash Strategy
By utilizing a specific hashing function to bind the content of leaf nodes with their position in the tree, data consistency and integrity are ensured. The fractal structure of F-Hash Trees further optimizes the storage and verification processes.



Root
Contains references to the hash values of two main child nodes, A and B, which represent the two main branches of the tree.

Node
Each internal node contains references to the hash values of its leaf nodes. In the fractal-optimized structure, the architecture of these internal nodes can be seen as a miniature of the entire tree, where each node could further expand into its own subtree, possessing fractal characteristics.

Leaf
Leaf nodes contain the actual data (e.g., A1 Data and B1 Data). In this structure, some leaf nodes might be empty (null), indicating they contain no data, similar to the empty leaf nodes in sparse Merkle trees.

A1 Data, B1 Data
Nodes that actually store information, where “NULL” indicates that the data node at that position is currently empty.
In F-Hash Trees, leaf nodes are typically located at the bottom layer of the tree, containing the actual data values (such as key-value pairs or transaction information).

F-Hash Tree Data Relationships

In F-Hash Trees, leaf nodes are typically located at the bottom layer of the tree, containing the actual data values (such as key-value pairs or transaction information).

No Child Nodes
Leaf nodes do not have references to other nodes. If a node has child nodes, then it is considered an internal node.

Hash Values Correspond to Actual Data
The hash values contained in leaf nodes are the result of hashing the actual data stored within them (such as file contents, UTXO data, etc.).

Position Encoding
In F-Hash Trees, leaf nodes may have a unique fractal encoding that reflects their precise position in the tree, often related to their hash value or key.

Data Structure Manifestation
If the complete data structure of the tree were visible, leaf nodes would be the endpoints without further branches in this structure.

Storage and Verification Process

Implementation of Self Similar Design
In the data structure definition, each node is designed not only to store traditional hash values and data/child node pointers but also includes a fractal encoding that represents its position within the tree. This encoding reflects the path from the root node to the current node, used for quick localization and verification.

```
1 class FHashNode: class FHashNode:
2 def __init__(self, data=None, children=None, fractal_code=""):
3 self.hash = hash(data) self.data = data # 仅叶节点存储实际数据 Only leaf nodes
  store actual data.
4 self.children = children or [] self.fractal_code = fractal_code # 分形编码
  Fractal Encoding
```

Implementation of Dynamic Layering Adjustment
Data distribution monitoring dynamically determines node splitting or merging by monitoring the amount of data in nodes and their access frequency. A split operation is performed when the data volume in a node exceeds a preset threshold; a merge operation is performed when the data volumes in multiple adjacent nodes fall below another threshold.

```
1 def adjust_node(node):
2 if len(node.data) > MAX_THRESHOLD:
3 split_node(node) elif len(node.data) < MIN_THRESHOLD and can_merge(node):
  merge_nodes(node.parent)
```

Splitting and Merging Strategies

The splitting operation creates new child nodes based on the node's fractal encoding and reallocates the data. The merging operation combines the data of adjacent nodes into one node and updates the parent node's list of child nodes.

```
def split_node(node):
# Split the node, create new child nodes, and reallocate the data according to fractal encoding.
# ...
def merge_nodes(parent_node):
# Merge adjacent nodes and update the parent node's list of child nodes.
# ...
```

Implementation of Efficient Data Verification

Path calculation utilizes fractal encoding to directly compute the verification path, instead of traversing the entire tree structure. This reduces the time complexity required to verify data in large datasets.

```
def calculate_path(data_hash):
# Calculate the path to the leaf node based on data hash and fractal encoding #
... return path
```

Hash computation optimization involves pre-calculating and storing hash paths from the current node to the root node during node creation and updates. When verifying data, these pre-stored hash values are directly utilized, reducing the need for real-time calculations.

```
def precompute_hashes(node):
# Pre-calculate the hash path from the current node to the root node
# ...
```

Through the implementation steps mentioned above, F-Hash Trees effectively utilize the self-similarity and recursive characteristics of fractal theory to achieve dynamic optimization of the data structure, enhancing the efficiency of data storage and verification. This approach provides an efficient and flexible solution for handling large-scale, dynamically changing datasets.

F-Hash Tree Construction and Verification

Constructing an F-Hash Tree involves hashing the contents of each leaf and organizing these leaves according to a binary representation of hash digests, similar to Merkle trees. However, F-Hash Trees introduce a hierarchical system that allows for multiple levels of nodes, each representing different scales of the dataset, akin to observing fractals at different magnifications.
Example: Imagine constructing an F-Hash Tree using a simplified hash function, resulting in a hash space of 0 to 15. If only the leaf corresponding to the binary representation 1010 is populated, the F-Hash Tree structure allows for efficient navigation and verification by following the tree's inherent fractal pattern.

Data Verification

The process of verifying the presence or absence of data in F-Hash Trees is significantly optimized through the fractal structure. To verify a piece of data, it's only necessary to traverse the tree following the fractal pattern leading to the specific leaf, without the need to rebuild or traverse the entire tree.

Efficient Path Finding

The binary representation of each leaf's position not only guides navigation through the tree but does so in a way that leverages the efficiency of the fractal structure. This means that verifying data or proving its absence requires fewer computational steps compared to traditional Merkle trees.

Applications and Advantages

The fractal optimization of F-Hash Trees makes them particularly suited to applications that require efficient and dynamic management of large datasets.

Data Association and Deletion

F-Hash Trees can associate data with public keys and provide a verifiable way to prove data deletion. However, fractal optimization enhances this capability, allowing for more efficient data association and easier verification of deletions without compromising the integrity of the tree.
F-Hash Trees offer a dynamic, scalable, and efficient structure for managing key-value pairs in distributed systems. Its fractal optimization not only improves storage efficiency and verification speed but also introduces a new level of flexibility in how data is organized and accessed within such trees.

Bitcoin UTXO Container Abstraction

MAST Contracts Integration with Fractal Hash Trees

The root of the F-Hash Tree is added to the main root Tapscript, and a main root address is created together. Using the Satyabit Protocol to create UTXO assets, the F-Hash Tree within the protocol offers proof to asset holders through Bitcoin UTXO Tapetching commitments. This enables asset owners to independently verify whether their accounts are included in the tree,

whether the appropriate amounts have been filled, and whether the corresponding Taproot transactions exist and have been confirmed on the Bitcoin blockchain.

In Taproot, MAST contracts allow committing many different execution paths via a single hash. Combined with F-Hash Trees, we can encode the state of an entire F-Hash Tree in one hash, allowing each UTXO to be verified without exposing the contents of the entire tree.

```go
func CreateTaprootAddressWithFHashTreeRoot(fTree *FHashTree) (string, error) {
    // Calculating the Root Hash of a Fractal Hash Tree
    fTreeRootHash := fTree.CalculateRootHash()

    // Creating a Tapscript Containing the Root Hash of an F-Hash Tree
    tapscript := CreateTapscript(fTreeRootHash)

    // Creating and Returning the Main Root Address
    return GenerateTaprootAddress(tapscript), nil
}

func CreateTapscript(fTreeRootHash string) []byte {
    // Creating Tapscript with the Root Hash of an F-Hash Tree
    // ...
}

func GenerateTaprootAddress(tapscript []byte) string {
    // The Logic of Creating the Actual Root Address


    // ...
}
```

Asset Issuance

Fractal Assets on Bitcoin UTXO

Satyabit Protocol proposes a method called "Bitcoin UTXO Fractal Assets", through which UTXOs are defined as containers for mounting MAST contract assets on the Bitcoin chain. This protocol uses UTXO outputs with OP_RETURN that record Taproot commitments, making these UTXOs a special data output. By borrowing concepts from fractal geometry, Satyabit Protocol can define asset states within UTXOs and use them as protocol asset containers.

In Satyabit Protocol, UTXO outputs that change the asset state would place the F-hash tree from the MAST contract in an unspendable script path within the Taproot script tree, indicating that each fractalization of the UTXO asset state would generate at least two different UTXOs as transaction outputs. One for data record, the other for mounting the Satyabit Protocol fractal

assets. When assets need to be transferred, the owner of the first UTXO can specify which UTXO will hold the asset by changing the state.

Additionally, Satyabit Protocol adopts "330 Satoshi" as the minimum unit for UTXO containers, allowing UTXO fractal assets to be integrated into any UTXO output. This approach avoids the meaningless bloat of the network UTXO set due to Bitcoin UTXO dust outputs, while also ensuring fractal UTXO assets have a certain amount of Bitcoin anchoring to enhance stability and reliability.

By utilizing the F-hash tree structure and Bitcoin's Taproot functionality combined with Satyabit Protocol, assets can be issued. The fractal hash tree root would be used as part of a Tapscript and jointly constitute the master root address. In this way, both the issuance and validation of assets will rely on the collaborative work of the fractal hash tree and Taproot structure.

Issuing UTXO Fractal Assets Using Satyabit Protocol

Through the Satyabit Protocol, a UTXO fractal asset is generated and a valid asset proof is provided using the structure of an F-hash tree.
The process for releasing the UTXO Fractal Asset:

```
1  func IssueFractalAsset(amount int, fTree *FHashTree, taprootAddress string)
   (string, error) {
2      // Create asset
3      asset := CreateAsset(amount)
4
5      // Record assets in a fractal hash tree
6      leafNode := NewLeafNode([]byte(asset.Metadata), asset.ID)
7      fTree.Insert(leafNode)
8
9      // Broadcast UTXOs with new assets to the Bitcoin network using Satyabit
   Protocol
10     utxoTxHash, err := BroadcastUTXOWithAsset(taprootAddress, asset, fTree)
11     if err != nil {
12         return "", err
13     }
14
15     // Returns the hash of assets offerings
16     return utxoTxHash, nil
17 }
18
19 func CreateAsset(amount int) *Asset {
20     // Creating a New Asset Instance
21     // ...
22 }
23
```

```
24 func BroadcastUTXOWithAsset(taprootAddress string, asset *Asset, fTree
   *FHashTree) (string, error) {
25     // Broadcasting UTXOs Containing Asset Data in F-Hash Tree
26     // ...
27 }
```

Asset holders can independently verify that their assets are correctly included in the F-hash tree using its proof mechanism, while also ensuring the related Taproot transactions have been confirmed by the blockchain.

Transferring Assets

Asset Definition

First, define the asset and its representation in the F-hash tree. Each asset is identified by a unique identifier (such as a hash value or token ID) and is associated with a specific leaf node.

Verify Asset for Transfer

Confirm the asset to be transferred exists in the current F-hash tree. The asset identifier can locate its leaf node and validate the balance it represents.

Create Transfer Transaction

Create a transaction that describes the process of transferring the asset from one address to another.

This typically involves:

- Reducing the balance of the original owner's leaf node
- Increasing the balance of the new owner's leaf node
- Creating the new owner's leaf node in the tree if it doesn't exist

Update F-hash Tree Structure Data

After the asset transfer, update the affected leaf nodes and recursively update the hash values of all internal nodes along the path to the root.

This may involve:

- Splitting or merging nodes to maintain tree balance
- Updating the fractal encoding of nodes along the path

Generate Proof

Create proof to validate this transfer.

The proof should include:

- The pre-transfer and post-transfer states of the affected leaf nodes

- Hash values of nodes along the path to prove their position in the tree
- The updated root hash as an aggregate verification of all changes

Submit Transaction

Submit the constructed transaction and generated proof to the network for other nodes to verify and eventually confirm.

Transaction Confirmation

After consensus is completed, the transaction will be included in a block and finally added to the blockchain. At this point, the transfer of asset ownership is formally completed and the new asset state will be reflected in the updated F-hash tree.

Validation and Storage of Fractal UTXO Assets

资产**状态**更新后的F-哈希树的状态永久化存储在 Bitcoin 链区块中。通常涉及到计算新的树根哈希值，并将其与相关的区块或事务记录一起保存。由于资产证明和交易历史都被记录，资产从发行到当前持有人的整个历史都是透明且可追踪的。

The updated state of the F-hash tree after an asset state change is permanently stored in Bitcoin chain blocks. This typically involves calculating the new root hash value of the tree and saving it together with the related block or transaction record. Since the asset proofs and transaction history are recorded, the entire history of the asset from issuance to the current holder is transparent and traceable.

通过使用分形哈希树和 Taproot 技术，资产转移变得更加高效和安全，同时保持了区块链的去中心化特征和抗审查性。这种机制允许资产在无需信任第三方的情况下进行安全、透明的转移，这对于提升分布式财产和资产管理的效率至关重要。

By utilizing fractal hash trees and Taproot technology, asset transfers become more efficient and secure while maintaining the decentralized characteristics and censorship resistance of the blockchain. This mechanism allows assets to be securely and transparently transferred without needing to trust third parties, which is crucial for improving the efficiency of distributed property and asset management.

Asset Proof

Proof Creation

At time t0, the asset is created at the root of the F-hash tree, constituting the first proof point. Each leaf node represents an asset or asset set, and each asset is assigned a unique identifier and corresponding initial balance.

Existence Proof

Proof is needed to show the asset existed in its original state in the tree at t0. This requires constructing a path from the leaf node containing the asset to the root node to validate its existence.

Zeroing Proof

At time t1, proof is needed to show the balance of the original leaf node was set to zero. This may involve updating the state of the leaf node and creating new nodes at the corresponding position in the fractal hash tree to reflect this change.

New State Proof

The existence of the asset in the new F-hash tree at t1 also needs to be proven, explaining how the asset was transferred from the original tree to the new tree. Again, this requires traversing a path from the leaf node to the root of the new tree.

Auditing and Continuity

A new proof must be generated for the asset after each transfer, forming a complete history record of the asset from its origination output to the current state along with the issuance proof.

The generation and verification of these proofs need to be efficient on the fractal hash tree structure. Each transaction update may result in tree reconstruction, so proof generation involves calculating paths based on the tree's current state. All proofs must unequivocally demonstrate the asset’ s exact state after each transaction and ensure continuity and validity of these proofs. Proofs are only valid as long as the corresponding output of the asset in the F-hash tree remains unspent.

Asset Invalidation

Once the UTXO output of an asset is spent without being submitted to a new F-hash tree, the asset is considered invalid. This does not affect other holders of the asset. However, in some cases, it may be preferable to spend the output on a new empty F-hash tree to demonstrate the asset has been destroyed, voided or "burned".

- When the asset output is spent, it is removed from its original position in the F-hash tree.
- If no new tree is populated with the asset in the spending transaction, it results in an invalid unrecorded state.
- The asset can no longer be verified or transferred going forward.
- By including it in an empty tree instead, one can provably invalidate the asset.
- This "burns" the asset forever without disturbing other valid instances.

Ensuring proper invalidation provides certainty for remaining holders that destroyed assets are permanently removed from circulation. It maintains the integrity of asset representation within the native UTXO framework.

Satyabit Data availability Node

Satyabit, as the Bitcoin UTXO asset protocol, consists of MAST contracts and Satyabit Protocol DA nodes, while Satyabit UTXO assets utilize settlements on Bitcoin.

Therefore, Satyabit Protocol designed the Satyabit DA nodes with a specialized state-consensus mechanism suitable only for data availability consensus, to complete state changes of Fractal Hash Trees UTXO DA data related to the protocol.

The two key functions of Satyabit DA nodes are Data Availability Sampling (DAS) and Fractal Hash Trees state changes.

These are two innovative Satyabit extensions:

DAS enables light clients to verify data availability without downloading entire blocks.

The F-hash tree structure enables Satyabit Protocol fractal UTXO assets related to only download relevant transactions from Bitcoin settlement data.

Satyabit Pos

The Satyabit DA consists of a P2P network of nodes competing via a PoS (Proof of Stake) mechanism.

Satyabit DA operates as a decentralized network to manage aspects like data availability.Network nodes running the Satyabit protocol participate by staking some native crypto.A PoS mechanism is implemented where the chance of a node being selected is proportional to its stake amount.Selected nodes are responsible for maintaining the shared state of F-hash trees and DAS sampling.This incentivizes uptime and honesty from participating nodes.Over time, the PoS selection mechanism ensures all correct and staked nodes have an equal opportunity to contribute to the Satyabit network.

Satyabit's Decentralized Autonomous infrastructure utilizes a staked PoS model for its p2p node network, enabling scalable and decentralized management of protocol data.

Providing Data Availability

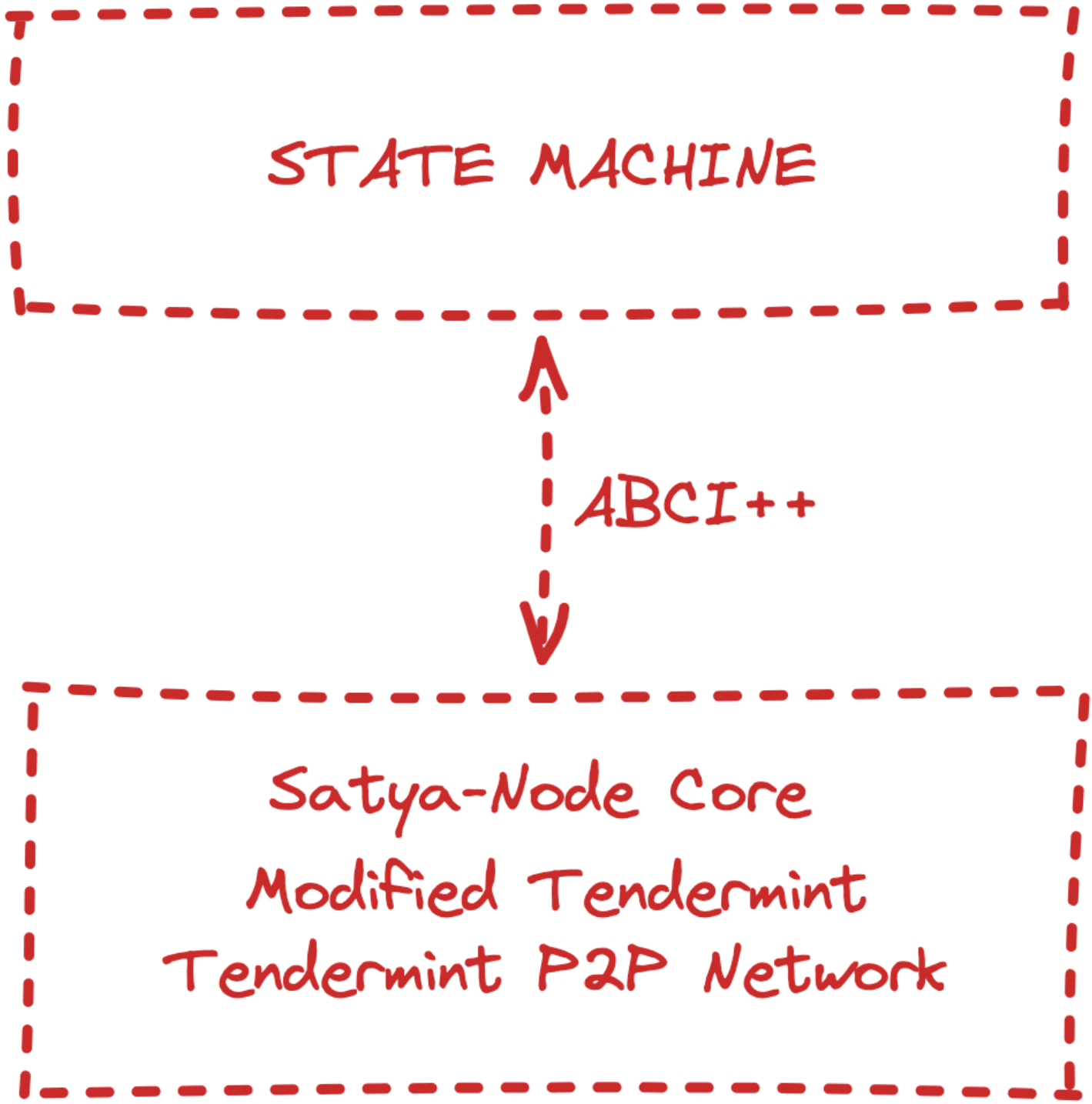
Satyabit refers to this network for data consensus as Satyabit-Node, an application built with Cosmos SDK that facilitates state synchronization for the DA (Decentralized Autonomous organization) and assists with transactions related to fractional assets.

Satyabit-Node is built upon Satyabit-Core, which is a modified version of the Tendermint consensus algorithm. Specifically:

- Satyabit-Node runs on validators participating in the Satyabit Decentralized Autonomous network.

- It uses Cosmos SDK for modular development and Tendermint for securing the network through Byzantine Fault Tolerant consensus.
- Satyabit-Node Core handles the modified Tendermint consensus with extensions for datatypes used in Satyabit like Fractal Hash Trees.
- Its role involves maintaining the shared state through state syncing and assisting queries about asset data availability.
- Data availability sampling performed by Satyabit-Node ensures light clients can validate relevant transaction data existence.

By leveraging technologies like Cosmos SDK and a customized Tendermint variant, Satyabit-Node is able to reliably broadcast state changes and provide data availability - a key requirement for the decentralized Satyabit protocol.



Satyabit DAAPP

Similar to Tendermint, Satyabit-Node Core connects to the application state machine through ABCI++.

Satyabit DA Validation

While Satyabit-core state machine is necessary for executing the PoS mechanism and enabling DA governance, Satyabit-Nodes are agnostic to state changes - the state machine only validates Satyabit Protocol data produced on Bitcoin.

Nodes validating data availability do this by downloading Bitcoin blocks and filtering out relevant Satyabit Protocol transactions. If they can download the related data, they have

validated its availability. In Satyabit, Satyabit-Nodes can use a novel mechanism to validate data availability without needing to download all data in a block.

This new method for validating data availability is called Data Availability Sampling (DAS).

- DAS allows validating transaction inclusion without fully downloading block contents.
- Satyabit-Nodes are sampled to prove certain transactions are verifiably included.
- If the proof can be reproduced, light clients can validate data availability.
- This improves scalability by reducing block propagation overhead for validation.

While Satyabit-core state machine verifies on-chain data, DAS is the innovative solution powering lightweight validation of data availability in Satyabit.

Data availability sampling (DAS)

DA nodes only download block data containing Satyabit Protocol data (i.e. transaction list) commitments (i.e. Tapetching). To enable DAS, Satyabit DA Nodes encode block data using a 2D Reed-Solomon encoding scheme . This is consistent with the solution used in Celestia

DA nodes filter blocks only those containing relevant Satyabit Protocol data commitments.Block data is then encoded using a 2D Reed-Solomon scheme for error correction.This allows encoding block data efficiently into shares distributed to DA nodes.For DAS, light nodes randomly select shares which DA nodes provide proofs for.Proofs validate inclusion of transaction hashes in block without full data download.This is similar to the approach adopted by Celestia for efficient light client validation

For more information on the specific DAS implementation and Reed-Solomon encoding scheme used, please refer to the original specification document.