Modular Indexer: Fully User-Verified Execution Layer for Meta-Protocols on Bitcoin

Hongbo Wen[1,2], Hanzhi Liu[1,2], Shuyang Tang[1,3], Shuhan Cao[1], Domo[4], and Yu Feng[1,2]

[1] Riema Labs
{hongbo,hanzhi,htftsy,shuhan,yu}@riema.xyz
[2] University of California, Santa Barbara
[3] Shanghai Jiao Tong University
[4] Layer 1 Foundation
domodata@proton.me

Abstract. Before the emergence of inscriptions and ordinal protocols, Bitcoin was limited in its applications due to the Turing-incompleteness of its script language. Fortunately, with recent advances in techniques, Turing-complete off-chain execution layers are established via Bitcoin indexers. Yet, existing indexers have their data integrity and availability strongly dependent on the honesty of indexers. This violated the trustlessness and decentralization principle of the cryptocurrency literature. To provide an alternative Bitcoin indexer scheme and overcome the above limitations, we have reallocated the roles of committee indexers (for heavy computations), normal indexers, and light indexers (the client end), and established a fully user-verified execution layer based on our modular indexer protocol. For the trustless relay of data, we have adopted Verkle trees to store and prove the states. Thus, data integrity and availability are guaranteed even in the case of a majority of malicious committee indexers. Ideally, our modular indexer would safely bridge the gap between the Bitcoin layer-1 and applications from BRC-20, and contribute to the further prosperity of the Bitcoin ecosystem.

Keywords: Bitcoin, Indexer, Ordinal, BRC-20, Execution Layer, Cryptocurrency.

1 Introduction

Blockchain technology continues to captivate the financial and technological sectors with its promise of decentralization, transparency, and security. The Bitcoin [12] (BTC) ecosystem, a pioneer in this realm, has recently made impressive strides, particularly with the advent of Layer-2 solutions like Stacks [3] and CKB [2], which significantly alleviates scalability concerns by processing transactions off the main blockchain, thus enabling faster and more cost-efficient transactions. Furthermore, the introduction of inscriptions [13] and ordinal protocols [6] has opened new horizons for embedding data and creating unique, indivisible assets directly on the Bitcoin blockchain, fostering a new wave of innovation and utility.

Due to the Turing-incompleteness of the Bitcoin script language, complex program logic cannot be executed directly on Bitcoin, which restricts the functionalities of Bitcoin applications. To overcome this shortage, developers leverage *Bitcoin indexers* to establish an off-chain execution layer that is Turing-complete. Specifically, an indexer first allows users to upload general data and contract codes to Bitcoin, where the contract code is written in some Turing-complete programming languages. Based on these codes and data, the indexers maintain a set of states. For each generation of Bitcoin blocks according to the Bitcoin transaction order, indexers execute the user codes on the data and update the states accordingly. Indexers dramatically enhance the accessibility and usability of blockchain data.

However, despite their utility, Bitcoin indexers face limitations such as data integrity and availability. Specifically, for an off-chain execution layer, the indexer could tamper with the data leading to bogus states for the user. To mitigate the data integrity problem, the user could download and maintain up-to-date data from the Bitcoin blockchain and verify the validity of the output by executing her indexer. However, doing so will include the need for substantial storage capacity and processing power to manage the ever-expanding blockchain. Additionally, maintaining up-to-date and accurate indexing amidst the blockchain's decentralized and immutable nature poses ongoing challenges.

Due to the integrity and cost of maintenance concerns mentioned above, one native solution is to leverage a *decentralized indexer network* to conduct the computation [9]. However, since the network is completely permissionless, the consensus mechanism of the existing decentralized indexer network is vulnerable to Sybil attacks, which enable malicious indexer operators to provide users with false states, such as asset ownership and spendable balance.

To solve this issue, we propose a *modular indexer* architecture that enables a truly decentralized, fully user-verified indexer network. The key challenge here is to design a mechanism that allows users to verify the validity of the states provided by the indexers efficiently and cost-effectively. Our *insight* is based on the following crucial observation: the expensive integrity checking of execution over whole-state transitions can be reduced to checking the validity of a tiny amount of *checkpoints* through the design of proper cryptography protocols.

Architecture of Our Design. In a fully user-verified modular indexer network, Bitcoin remains the most basic trusted consensus and data layer. The indexer obtains the protocol state corresponding to each Bitcoin block by reading data on Bitcoin and performing computations. Specifically, the protocol state $S = \{K_0 : V_0, K_1 : V_1, ..., K_n : V_n\}$ is a hash table composed of many Key-Value pairs. Users can obtain the protocol state S corresponding to the block height H and block hash P through the following two methods:

- Users run a full Bitcoin node and operate an indexer that saves full states. This is the most reliable method, but running a full Bitcoin node with a full-

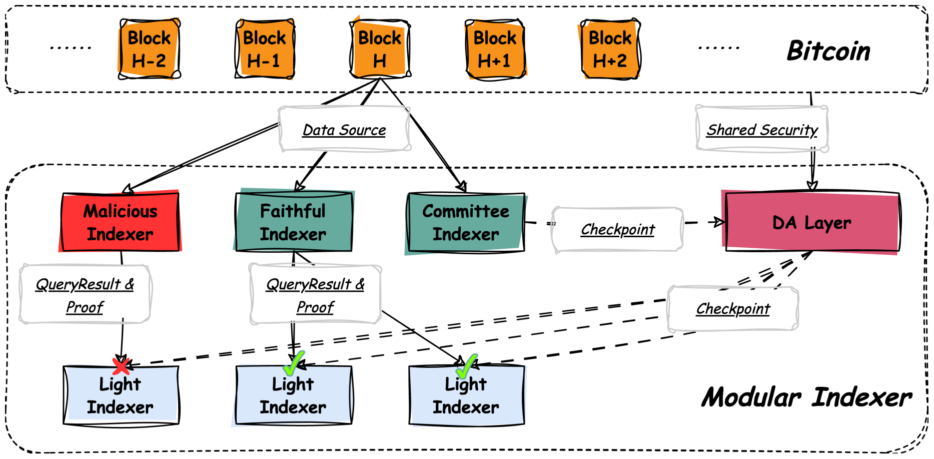

Fig. 1. Basic Architecture of Modular Indexer

state indexer requires significant storage resources, which are unaffordable to normal users.

- Using the key, a user queries the value from the indexer in the decentralized network. The indexer returns the result (*i.e.*, QueryResult) to the user. This is current common practice, but this method cannot prevent scenarios where malicious nodes provide bogus data.

Currently, a decentralized network composed of indexers cannot prevent Sybil attacks because its consensus layer lacks a penalty mechanism and cannot punish these malicious indexers, i.e., it lacks economic security. However, when the indexer network functions solely for the execution of data that has already reached consensus on Bitcoin, based on predefined protocols, the need for an additional consensus layer becomes unnecessary. Instead, it has to provide users with proofs for each execution result, which is inefficient. To address this challenge, we propose the modular indexer architecture. It allows users to efficiently verify whether the states and query results provided by the indexer are trustworthy under the assumption that at least one honest indexer node is present (1-of-N trust assumption).

As shown in Figure 1, the modular indexer architecture involves the following roles:

- The *committee indexer* is responsible for reading each block of Bitcoin, calculating protocol states, and summarizing these states as a polynomial commitment namely *checkpoint*. Whenever the committee indexer obtains a new Bitcoin block, it generates a new checkpoint for the protocol and publishes it to the data availability layer for users to access.
- The *data availability* layer is responsible for data publication and ensuring the availability of data at the checkpoints.

– The *indexer* is responsible for sending the results and the corresponding proofs to the queries of the users. Based on the checkpoint stored in the DA layer, users can verify the validity of these results using the provided proofs.
– The *light indexer* is a lightweight client operated by users, which can query the necessary parts of the protocol state from the indexer, verify the corresponding proofs, and when the checkpoints provided by multiple committee indexers are inconsistent, utilize stateless computation with *minimal critical states* to generate the correct checkpoint, thus identifying the honest committee indexer.

In the modular indexer architecture, Bitcoin is responsible for transaction ordering, consensus, and providing Bitcoin's economic security for the DA layer. The DA layer ensures that users have the availability to access checkpoints.

2 Checkpoints and Data Attestations

In the architecture of the modular indexer, both indexers and committee indexers might act maliciously. Therefore, the difficulty of realizing our modular indexer architecture lies in (1) the verification of checkpoints in the case of inconsistent committee indexers; and (2) the attestation of each data queried by light indexers in case of malicious indexer.

To begin with, we provide a strawman protocol in a simplified scenario in which one ideal prover (to be realized by committee indexers) generates a new checkpoint after one state transition and wishes to convince one verifier (light indexer) that the new checkpoint is valid. We show that the minimal critical states can be viewed as "proofs" for a checkpoint. In fact, the verifier is allowed to locally build a *critical subtree* from the critical states and recover the correct checkpoint based on it. Notably, minimal critical states can be obtained from the Bitcoin network by light indexers. Therefore, the verification of a checkpoint is practical for each light indexer.

2.1 A Strawman Verification of Checkpoints

To simplify the descriptions, we assume that the state transition only involves the modification of one value or the insertion of one key-value pair. In general, since the checkpoint is the hash of a Verkle tree root, the proof is done by unfolding every concerned node of the two trees (before and after the transition). Due to the stateless nature of Verkle, the prover does not have to provide the sibling nodes, and hence the communication complexity is bounded by $O(\log n)$.

Specifically, for the transition from a checkpoint C_i of block B_i to a checkpoint C_{i+1} of block B_{i+1}, ideal provers and a verifier interact as follows.

Prover. Each prover parses block B_{i+1}, and assembles a *critical subtree* T'_i. This is a subtree of the previous (Verkle) state tree T_i that contains the minimum number of nodes such that each path from the root to any critical state (either as a leaf, extension node, or internal node) of B_{i+1} (not B_i) is contained. Then, it assembles the next state tree T_{i+1} according to B_{i+1}. Finally, a tuple $(i+1, b_i, C_i, T'_i, b_{i+1}, C_{i+1}, T'_{i+1})$ is sent to the verifier, where (1) b_i and b_{i+1} are the block hashes of B_i and B_{i+1}; and (2) C_i and C_{i+1} are the commitments of the roots of T_i and T_{i+1}.

Verifier. After receiving the tuple $(i+1, b_i, C_i, T'_i, b_{i+1}, C_{i+1}, T'_{i+1})$, the verifier proceeds as follows.

(1) The verifier verifies that (b_i, C_i) is equal to its local block hash and Verkle root of height i. In addition, it verifies that C_i and C_{i+1} are the hashes of the roots of T_i and T_{i+1}.
(2) If it finds (b_{i+1}, C_{i+1}) in the local dataset, it completes the verification and halts. If C_{i+1} is the first local Verkle root for B_{i+1} of hash b_{i+1}, it locally stores (b_{i+1}, C_{i+1}) and exits the verification.
(3) In the other case, there exists $(b_{i+1}, \widetilde{C_{i+1}})$ locally and $\widetilde{C_{i+1}} \neq C_{i+1}$. To verify that C_{i+1} is correct, the verifier locally constructs T''_{i+1} from T'_i by modifying the leafs (for existing keys) and structures (for data insertion) accordingly. This is the direct application of the Verkle tree specification. However, the only difficulty lies in the commitment of each node. Since the sibling nodes are not provided in the subtree, node commitments could not be calculated by the formulae defining them. Fortunately, the nature of the KZG-based commitment scheme provides an alternative. Recall that we only discuss the case of one modification. For any node $p \in T''_{i+1}$, if any slot of its commitment (say, slot k) is changed by δ, then its commitment should be $c_0 + \delta \cdot \mathsf{L}_i(\tau)$, where c_0 is its node commitment in T'_i. Clearly, this calculation is possible with gp. In the other case, this is an internal node newly generated from an extension node q (by inserting a new value). In this case, this internal has only two children. One of them already exists in T''_{i+1} and the other one is an extension node with the stem and children set equal to q. Hence, the commitment can be calculated since the commitments of both are known.
(4) Compare T'_{i+1} with T''_{i+1}, accept if the two critical subtrees are equal.
(5) Replace $(b_{i+1}, \widetilde{C_{i+1}})$ with (b_{i+1}, C_{i+1}) in the local dataset.

In the event of a chain fork, or falsified block hashes, the verifier is allowed to download the finalized block hash b_{i+1} after the finality of block B_{i+1}.

2.2 Checkpoint Verification

Intuition. For (light) indexers, the verification of a checkpoint after multiple modifications is a simple generalization of the above protocol. (1) Since there is more than one critical state, the critical path is extended to a *critical subtree.* (2) The recovery algorithm of the transited Verkle tree is the same, but the iteration of nodes is extended to all nodes of the critical subtree.

The difficulty lies in the fact that there are multiple committee indexers (provers), and we wish to take merit of the fact that at least one committee

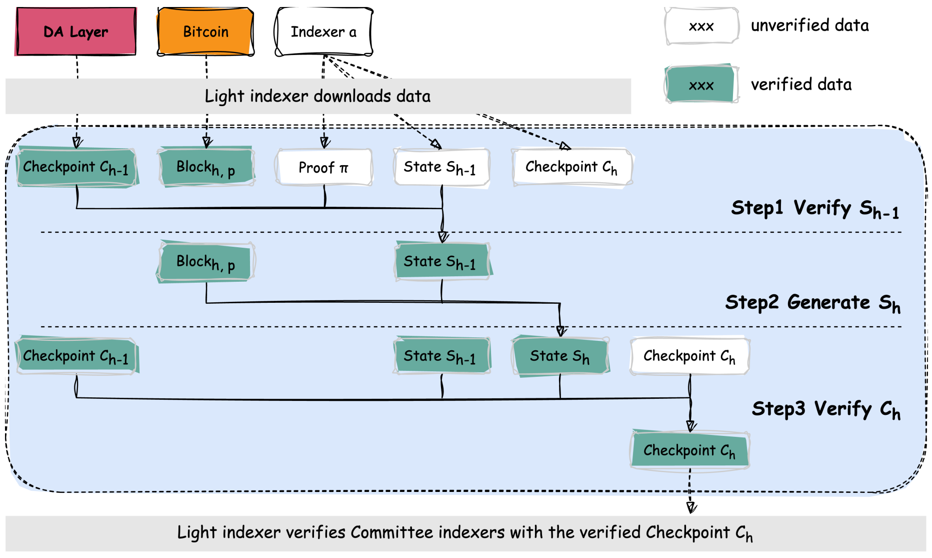

Fig. 2. Generating Checkpoint and Verifying Checkpoint Provided by the Indexer

indexer is honest and lessen the verification overhead. Our goal is that the full verification, which consists of the local reconstruction of a critical subtree, is only triggered in case of inconsistent checkpoints are received from two committee indexers. This is realized by the following method.

(a) For each pair of block height i and block hash t, the first checkpoint received from any committee indexer is directly stored locally without verification.
(b) When a different checkpoint is received for an (i, t) pair, the verification protocol is triggered, the light indexer obtains the critical states, locally recovers the critical subtree T_i'', asks for the corresponding critical subtree T_i' from indexers, and compares them to identify its correctness. If it is correct, replace the local checkpoint for (i, t).

Observably, the final checkpoint is correct if any one of the committee indexers is honest.

The Protocol. When the Bitcoin block height is h, the block hash is p, and the checkpoints at the height $h - 1$ of the committee indexers are consistent, the checkpoint submitted by the committee indexer set selected by users at height h, under the block hash p is inconsistent. In this case, without loss of generality, we assume that there are only two inconsistent checkpoints. A malicious committee indexer provided the false checkpoint C_a, but at least one honest committee indexer provided the true checkpoint C_b. Upon discovering the inconsistency, the light indexer will undergo the following steps for stateless computation to generate the correct checkpoint, thereby identifying the honest committee indexer and removing attackers from the committee indexer set. In a high-level view, the light indexer can request the state of the previous block S_{h-1} from an indexer and verify the legitimacy of the state provided with the known and consistent previous checkpoint C_{h-1}. The light indexer then executes the current block's transactions $B_{h,p}$ using the verified S_{h-1} to derive the current state S_h and its corresponding checkpoint C_h. Given the impracticality of handling the full state due to its massive size, often in the hundreds of GB, the user only needs to request a subset of the full state necessary for transaction execution. Using the properties of Verkle Tree, it is possible to compute the full state's checkpoint from this subset of state changes.

As shown in Figure 2, the light indexer needs to request the following information:

(1) Request from both committee indexers for the parent block's state S_{h-1}. The key of S_{h-1} $Key = \{K_0, K_1, ..., K_m\}$ is a subset of all keys in the state, containing only those keys read and written by transactions in the block. In the parent block state S_{h-1}, these keys correspond to the set of $Values = \{V_0, V_1, ..., V_m\}$, and in the current block's state S_h, these keys correspond to the post-execution state of $Value' = \{V'_0, V'_1, ..., V'_m\}$. Additionally, the indexer also needs to send proof $\pi = Proof(Key, Value)$ that $Keys$ and $Values$ indeed exist in S_{h-1}.
(2) Request the parent block's checkpoint C_{h-1} from the DA layer, as the Committee submitted C_{h-1} is consistent, it can be considered legitimate and honest.
(3) Request the Block content at height h, under block hash p from Bitcoin $Block_{h,p}$.

The light indexer needs to verify the following: Is the proof π correct? Is the parent block state S_{h-1} provided by the indexer legitimate? Among the two checkpoints, C_a and C_b, which is correct? This involves the following steps:

(1) *Verify* S_{h-1}. The light Indexer needs to verify whether the proof π is correct, which can be directly verified through $Verify(C_{h-1}, \pi)$.
(2) *Generate* S_h. The light indexer parses the current block $Block_{h,p}$ to obtain the transactions that need to be executed and $Values$ from the parent block's state S_{h-1} required for executing these transactions. Subsequently, the user executes transactions through $Values$, calculating the post-execution state of $Values'$. Note that the light indexer does not need to request the full state (saving the full state requires hundreds of GB of storage space), but calculates from a subset of the full state, containing only those read and written by transactions in the block. Also, only one honest indexer needs to provide $\underline{S_{h-1}}$ as well as the proof π for the transaction execution.
(3) *Verify Checkpoint* C_h. After verifying the correctness of S_{h-1} and calculating S_h, the light indexer can calculate the polynomial commitment of the complete state, as well as the values of the leaves that need to be updated. This means that the light indexer can recover the current block's Verkle Tree T_h from S_{h-1}, S_h and the parent block's checkpoint C_{h-1}, even though it only knows a tiny part of the leaves of the tree, we can still generate the

entire tree's checkpoint C_h. Thereby, the light indexer can determine the correctness of C_a and C_b by comparing these checkpoints with the generated checkpoint C_h, thus identifying the attacker and removing it from the committee indexer set maintained by itself.

Therefore, in the modular indexer architecture, as long as there is one honest committee indexer, the light indexer can observe the inconsistency of checkpoints and calculate the correct checkpoint, thereby identifying the attacker and avoiding financial losses. It is worth noting that this verification process is different from consensus and is not affected by the "51%" attack.

2.3 Proofs for Data Query

When the Bitcoin block height is h, the block hash is b, and the checkpoints of the committee indexer set selected by users on the DA Layer are consistently honest, the user's query is a set of keys existing in the state $Q = \{K_0, K_1, ..., K_m\}$. A malicious indexer provides incorrect query responses $R = \{V_0, V_1, ..., V_m\}$ Therefore, proofs are expected for any value sending from an indexer to a light indexer. For an indexer to attest a value p in a Verkle tree with root commitment (*i.e.*, a checkpoint) C, KZG openings of commitments for nodes of the path $(p_0, p_1, \ldots, p_\ell)$ $(p \in p_\ell.\mathsf{leaf})$ are sent along with the value. In this case, as shown in Figure 2, the light indexer can verify the proof through the verification procedure by reading the corresponding checkpoint C_h on the DA Layer, *i.e.*, verifying the KZG opening proofs. Since indexers cannot generate proof for incorrect query results for the checkpoint, users can verify the validity of the query results. If the query results are invalid, users can continue to query other indexers until they receive the query results from an honest indexer.

3 Applications

In this section, we introduce how existing meta-protocols can be integrated into the modular indexer architecture:

In this section, we introduce how existing meta-protocols can be integrated into the modular indexer architecture:

- *At the Modular Indexer Level.* The indexer independently stores and indexes each meta-protocol, thus calculating and publishing checkpoints for each meta-protocol. For example, if an indexer chooses to serve both BRC-20 [5] and Bitmap protocols [1], it needs to publish the current state commitment tuple (consisting of a block height h, a block hash p, and a checkpoint C) separately in the DA layer for BRC-20 and Bitmap.
- *At the Meta-Protocol Level.* On each block height h, block hash p, and checkpoint C, the indexer maintains a Verkle Tree storing all the state variables of the meta-protocol. This Verkle Tree uses a 32-byte Storage Identifier as the Key and the current value of the state variable as the value. The meta-protocol is responsible for defining how to generate the Storage Identifier and

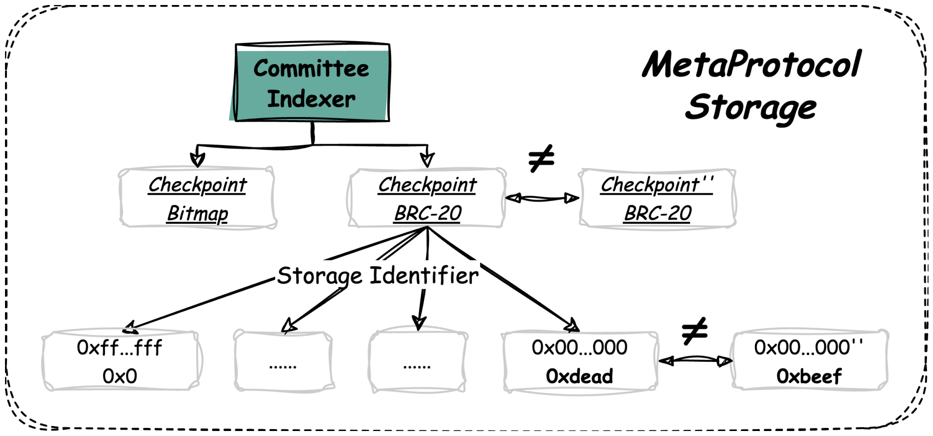

Fig. 3. Any change in the meta-protocol states leads to Checkpoint inconsistency

ensure its uniqueness. For example, in the BRC-20 meta-protocol, a state variable that needs to be maintained is the user's current available balance under a certain instance. Therefore, the meta-protocol needs to define how each user's available balance under each BRC-20 instance is mapped/hashed to a unique Storage Identifier.

In terms of state correctness, as shown in Figure 3, any changes in a state variable will lead to a change in the meta-protocol checkpoint. Upon detecting inconsistency in the meta-protocol checkpoints, the light indexer can verify and determine the trustworthy committee indexer through the verification process described in the previous chapter.

References

1. Bitmap tech. https://bitmap.tech. Last accessed on March 1, 2024.
2. Nervos network. https://www.nervos.org. Last accessed on March 1, 2024.
3. Stacks: Activate the bitcoin economy with the leading bitcoin l2. https://www.stacks.co. Last accessed on March 1, 2024.
4. https://l1f.discourse.group/latest, 2023. Last accessed on March 1, 2024.
5. BRC-20 documentation. https://layer1.gitbook.io/layer1-foundation/protocols/brc-20/documentation, 2023. Last accessed on March 1, 2024.
6. Ordinal theory handbook. https://docs.ordinals.com/, 2023. Last accessed on March 1, 2024.
7. Verkle trees for statelessness. https://verkle.info, 2023. Last accessed on March 1, 2024.
8. V. Buterin. Verkle trees. https://vitalik.eth.limo/general/2021/06/18/verkle.html, 2021. Last accessed on March 1, 2024.
9. JinseFinance. BRC20 indexer war: Will BRC20 be forked at the beginning of the year? what happened. https://www.coinlive.com/news/brc20-indexer-war-will-brc20-be-forked-at-the-beginning, 2024. Last accessed on March 1, 2024.

10. A. Kate, G. M. Zaverucha, and I. Goldberg. Constant-size commitments to polynomials and their applications. In *Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings*, pages 177–194. Springer, 2010.
11. D. R. Morrison. PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968.
12. S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.
13. P. Wuille, J. Nick, and A. Towns. Taproot: SegWit version 1 spending rules. https://github.com/bitcoin/bips/blob/master/bip-0341.mediawiki, 2020. Last accessed on March 1, 2024.

A Frequently Asked Questions

- *Q: Is there a consensus mechanism among committee indexers?*
  A: No, within the committee indexer, only one honest indexer needs to be available in the network to satisfy the 1-of-N trust assumption, allowing the light indexer to detect checkpoint inconsistencies and thus proceed with the verification process.
- *Q: How is the set of committee indexers determined?*
  A: Committee indexers must publish checkpoints to the DA Layer for access by other participants. Users can maintain their list of committee indexers. Since the user's light indexer can verify the correctness of checkpoints, attackers can be removed from the committee indexer set upon detection of malicious behavior; the judgment of malicious behavior is not based on a 51% vote but on a challenge-proof mechanism. Even if the vast majority of committee indexers are malicious, if there is one honest committee indexer, the correct checkpoint can be calculated/verified, allowing the service to continue.
- *Q: Why do users need to verify data through checkpoints instead of looking at the simple majority of the indexer network?*
  A: This would lead to Sybil attacks: joining the indexer network is permissionless, without a staking model or proof of work, so the economic cost of setting up an indexer attacker cluster is very low, requiring only the cost of server resources. This allows attackers to achieve a simple majority at a low economic cost; even by introducing historical reputation proof, without a slashing mechanism, attackers can still achieve a 51% attack at a very low cost.
- *Q: Why are there no attacks like double-spending in the Modular Indexer architecture?*
  A: Bitcoin itself provides transaction ordering and finality for meta-protocols (such as BRC-20). It is only necessary to ensure the correctness of the indexer's state transition rules and execution to avoid double-spending attacks (there might be block reorganizations, but indexers can correctly handle them).

– *Q: Why upload checkpoints to the DA Layer instead of a centralized server or Bitcoin?*
  A: For a centralized server, if checkpoints are stored on a centralized network, the service loses availability in the event of downtime, and there is also the situation where the centralized server withholds checkpoints submitted by honest indexers, invalidating the 1-of-N trust assumption.
  For indexers, checkpoints are frequently updated, time-sensitive data:
  - The state of the Indexer updates with block height and block hash, leading to frequent updates of checkpoints (10 minutes).
  - The cost of publishing data on Bitcoin in terms of transaction fees is very high.
  - The data throughput demand for hundreds or even thousands of meta-protocol indexers storing checkpoints is huge, and the throughput of Bitcoin cannot support it.
– *Q: What are the mainstream meta-protocols on Bitcoin currently?*
  A: The mainstream meta-protocols are all based on the Ordinals protocol, which allows users to store raw data on Bitcoin. BRC-20, Bitmap, SatsNames, etc., are mainstream meta-protocols. More meta-protocols and information can be found in [4].
– *Q: What kind of ecosystem support has this proposal received?*
  A: The proposal is put forward by Nubit as a long-term supporter and builder of the Bitcoin ecosystem. We have also exchanged ideas with many ecosystem partners and hope to jointly promote the progress and improvement of the modular indexer architecture.

B Preliminaries

B.1 Vector Commitment

In the cryptocurrency literature, vector commitments are often realized by interpolating the vector into a polynomial and committing this polynomial by the KZG commitment [10]. It involves the following algorithms.

– $\mathsf{setup}(1^\kappa) \to \mathsf{gp}$. The setup algorithm samples a random τ and returns $\mathsf{gp} = (G, \tau \cdot G, \tau^2 \cdot G, \ldots, \tau^d \cdot G)$.
– $\mathsf{commit}(\mathsf{gp}, \boldsymbol{v} = (v_1, v_2, \ldots, v_\ell)) \to \mathsf{com}_{\boldsymbol{v}}$. For a vector $\boldsymbol{v}$, this algorithm halts if $\ell > 256$. In the other case, the commitment algorithm first interpolates a polynomial as $f(X) = \sum_{i=1}^{\ell} v_i \cdot \mathsf{L}_i(X)$, where $\mathsf{L}_i(X)$ is the Lagrange base of the index i. Then, it returns $\mathsf{com}_{\boldsymbol{v}} := f(\tau) \cdot G$. Notably, although τ is not contained in gp, this value can be calculated since $\tau^i \cdot G$ is contained for each $i \in [\ell]$.
– $\mathsf{open}(\mathsf{gp}, \boldsymbol{v}, i) \to (v_i, \pi_i)$. To open a value of index i, the opening algorithm returns v_i and $\pi_i = \frac{f(\tau)-v_i}{\tau - i} \cdot G$, where f is the interpolation of $\boldsymbol{v}$.
– $\mathsf{verif}(\mathsf{gp}, \mathsf{com}_{\boldsymbol{v}}, i, v_i, \pi_i) \to 0/1$. The verification algorithm succeeds only if $e\left((\tau - i) \cdot G, \pi_i\right) = e\left((f(\tau) - v_i) \cdot G, G\right)$.

B.2 Verkle Tree

The modular indexer architecture utilizes the *Verkle tree* [7, 8] (a variant of *Merkle Patrica tree* [11]) as the data structure for state storage. The Verkle Tree is a variant of the Merkle Tree, featuring more branching and shallower depth. Its nodes use polynomial commitments instead of the results of cryptographic hash functions as summaries of the child nodes. The Verkle Tree treats the root node as the commitment for the entire tree (also referred to as the checkpoint mentioned earlier), meaning the prover, unlike with a Merkle Tree, does not need to provide all the "sibling nodes" hash values from the root to the leaf node. It only needs to provide all the nodes on the path from the root to the leaf node. Thus, the proof size of the Verkle Tree is smaller than that of the Merkle Tree, especially when users query multiple leaf nodes.

Verkle tree contains three types of nodes, *i.e.*, *internal nodes*, *extension nodes*, and *leafs* (*suffixes*). The children of internal nodes can be internal or extensional. However, the children of extension nodes can only be leafs. In other words, for each path from the Verkle tree root to any leaf, it always bypasses several internal nodes, one extension node, and one leaf, in sequence.

We model a Verkle tree as a set T. For each tree node $p \in T$, it is an extension node if $p.\mathsf{type} = 0$. In this case, its key $p.\mathsf{key} = (u_1, u_2, \ldots, u_\ell)$ is a sequence and its commitment is $C_p = \mathsf{commit}(\mathsf{gp}, 1, \mathsf{stem}, C_1, C_2)$. Here, stem is the concatenation of all the keys of its ancestor nodes and its key,

$$C_1 = \mathsf{commit}(\mathsf{gp}, v_0^l, v_0^u, v_1^l, v_1^u, \ldots, v_{127}^l, v_{127}^u),$$

and

$$C_2 = \mathsf{commit}(\mathsf{gp}, v_{128}^l, v_{128}^u, v_{129}^l, v_{129}^u, \ldots, v_{255}^l, v_{255}^u).$$

Here, each value v_i is separated into two values (v_i^l, v_i^u). See details in the Verkle tree documentation. Since each leaf is a value, we do not model them as tree nodes and the values are stored as part of its parent extension node $p.\mathsf{leaf}[0..255]$. For an internal node p, its type is $p.\mathsf{type} = 1$ and its commitment is

$$C_p = \mathsf{commit}(\mathsf{gp}, C_0, C_1, \ldots, C_{255}),$$

where each C_i is the commitment of one child.