RollStore: Hybrid Onchain-Offchain Data Indexing for Blockchain Applications

Qi Lin, Binbin Gu , and Faisal Nawab

Abstract—The interest in building blockchain Decentralized Applications (DApps) has been growing over the past few years. DApps are implemented as smart contracts which are programs that are maintained by a blockchain network. Building DApps, however, faces many challenges—most notably the performance and monetary overhead of writing to blockchain smart contracts. To overcome this challenge, many DApp developers have explored utilizing *off-chain* resources—nodes outside of the blockchain network—to offload part of the processing and storage. In this paper, we propose RollStore, a data indexing solution for hybrid onchain-offchain DApps. RollStore provides efficiency in terms of reduced cost and latency, as well as security in terms of tolerating Byzantine (i.e., malicious) off-chain nodes. RollStore achieves this by: (1) a three-stage commitment strategy where each stage represents a point in a performance-security trade-off—i.e., the first stage is fast but less secure while the last stage is slower but more secure. (2) utilizing zero-knowledge (zk) proofs to enable the on-chain smart contract to verify off-chain operations with a small cost. (3) Combining Log-Structured Merge (LSM) trees and Merkle Mountain Range (MMR) trees to efficiently enable both access and verification of indexed data. We experimentally evaluate the cost and performance benefits of RollStore while comparing with BlockchainDB and BigChainDB.

Index Terms—Blockchain, decentralized applications, indexing.

I. INTRODUCTION

DECENTRALIZED Applications (DApps) are applications that are implemented as smart contracts. A smart contract is a program where its state and logic are maintained by a blockchain network.[1] This makes DApps inherit blockchain features such as decentralization, transparency, and tamper-freedom [6]. (However, this does not mean that DApps are perfectly decentralized or tamper-free. This is because of the possibility of designing smart contracts in an extendable way either for legitimate or malicious reasons.) Recently, there has been a lot of interest in DApps. Various DApps have amassed hundreds of thousands of users and hundreds of millions of dollars in assets [12]. DApps span many areas such as decentralized finance [11], gaming and metaverses [18], and supply-chain [49].

DApp developers face many challenges, including the performance overhead, security concerns, and monetary cost of writing to blockchain smart contracts. Writing to blockchain smart contracts often involves significant transaction finalization times, with some operations taking tens of minutes to complete. This delay can hinder the real-time responsiveness expected in conventional applications [24], DApps also struggle with scalability issues, especially as user bases grow. Blockchain networks, on which many DApps are built, face limitations in terms of transaction throughput [7]. Additionally, Security is a paramount concern in DApps. Smart contract vulnerabilities and consensus algorithm weaknesses are areas of focus [44]. Notably, the average cost of a single smart contract operation is estimated to be around $3 [43]. This monetary factor directly influences the economic feasibility of DApps development and usage. To overcome these challenges, many DApps are developed using the *hybrid onchain-offchain* model [3], [20], [34], [57] which makes them directly centralized (unless the off-chain part is also decentralized), non-transparent and open to any tampering, and we call it the *hybrid model* for short. In this model, part of the DApp logic is maintained in the on-chain contract–where "on-chain" refers to its implementation in the blockchain smart contract, ensuring strong security. The rest of the processing and storage of the DApp is maintained by off-chain nodes—where off-chain nodes are nodes that are outside of the blockchain network. By delegating part of the processing and storage to off-chain nodes, both the monetary cost and performance overhead are significantly reduced.

However, building a complex DApp is still a significant challenge in the hybrid model, particularly when constructing a data management system for DApps that meets diverse performance and security requirements. In this work, we aim to build a data indexing solution for DApps in the hybrid model that can flexibly balance the performance and security of the system to meet different requirements. Data indexing is a fundamental problem in data management and a building block for more complex data management functionality. Therefore, the development of an efficient and secure data indexing solution for DApps can have an impact on a wide-range of decentralized data management systems. Our solution aims to extend and support the space of blockchain-based data management systems [4], [14], [16], [20], [32], [36], [37], [38], [40], [48]. Currently, existing blockchain-based databases (BBDBs) fall under one or more of the following categories: (1) BBDBs that do not utilize off-chain nodes efficiently (i.e., by writing all data and/or operations on-chain) [4], [16], making them inefficient in terms of monetary

The authors are with the Institute of Computer Science Department, University of California, Irvine, CA 92697 USA (e-mail: linq11@uci.edu; binging@uci.edu; nawabf@uci.edu).

[1]In this paper, we consider permissionless blockchain technologies such as Ethereum as they are the ones used predominately by DApps [12].

cost and performance overhead. (2) BBDBs that utilize off-chain nodes that are assumed to be trusted/permissioned (i.e., with closed membership) [32] or utilize trusted execution environments [14]. These are strong assumptions for decentralized environments and limit their practicality for DApps which are widely implemented in permissionless environments. (3) BBDBs that utilize authenticated data structures and verification methods to verify query results using on-chain digests or verifiers [37], [55]. However, these BBDBs only consider querying immutable data and suffer from the two limitations above if data is mutable.

We propose RollStore, a data indexing solution for hybrid blockchain DApps that overcomes the challenges of prior BBDBs. RollStore has the following properties: (1) it utilizes off-chain nodes efficiently by not needing to send raw data or operations to the smart contract. Smart contracts are only used for performing low-overhead operations (i.e., lightweight verification of data digests). This makes RollStore efficient in terms of reducing the monetary cost of writing to the blockchain. (2) RollStore does not assume that any off-chain node is trusted. (3) RollStore has a key-value interface where users can both read and write data. RollStore is the first data indexing solution for hybrid DApps that can achieve all these three properties. We envision that RollStore will be augmented with existing BBDBs as their indexing component to help transform them to enjoy the aforementioned RollStore properties.

RollStore can achieve the three properties above by bringing together and innovating in the areas of zero-knowledge (zk) proofs [53], optimistic and zk rollups [41], [47], Log-Structured Merge (LSM) trees [31], and Merkle Mountain Range (MMR) trees [45]. We utilize Zero-Knowledge Succinct Non-interactive Argument of Knowledge (zk-SNARK) [8], [19]. Zk-SNARK allows an untrusted node to perform a computation that changes the state of the data and produce the new state with a proof of the new state's correctness (i.e., that the new state is the result of applying a correct mutating operation on the previous state). The proof can be verified with low overhead, thus allowing cheap verification on-chain. We also utilize the concept of optimistic rollups (o-rollups) [41]. O-rollups were proposed as a layer-2 scaling solution for blockchain, where off-chain nodes perform compute functions on behalf of the layer-1 blockchain. Then, clients can interact with the blockchain in a challenge period to challenge the correctness of the off-chain outcomes. Finally, we integrate LSM and MMR tree structures in the design. LSM's append-only nature makes it a good candidate to manage the movement of data from one stage to another (e.g., data in each LSM layer corresponds to a different processing/validation stage). MMR trees allow deconstructing a single MMR tree into smaller Merkle trees. This allows better modularity and integration with LSM trees.

RollStore combines the aforementioned technologies in a new design for hybrid DApp data indexing. RollStore consists of three types of nodes: (1) an *updater* node that manages clients requests, (2) a *prover* node that is responsible for generating proofs for operations, and (3) a *backup* node that maintains the verified data and associated proofs. In addition, RollStore includes a smart contract that performs lightweight verification of digests and proofs.

One of RollStore's key proposals is a three-stage commitment process to manage the performance-security trade-off of verification methods. We support three kinds of verification methods for updates to off-chain data: (1) zk-SNARK proofs: this is the most trusted verification as it verifies the correctness of the data operation and new off-chain data state. However, it has high computational complexity requiring a long time to generate proofs. (2) Optimistic rollups (o-rollups): this method relies on a simple data digest that is written on-chain. Users agree on the digest and the data represented by that digest; however, it is not guaranteed that the corresponding operations and new state are processed correctly. (3) Off-chain response proof: this is the weakest guarantee which is for the client to receive a signed response from the off-chain node before any digest or proof is written on-chain.

RollStore addresses the challenge of accommodating diverse user requirements in DApps. Our system is designed to cater to varied needs in terms of security and performance. For instance, in the context of bank transactions, scenarios involving small amounts and high-frequency transactions demand lower latency and higher throughput, aligning with our first-stage commitments. Conversely, situations involving large amounts and low-frequency transactions prioritize strong security and can tolerate higher latency, aspects provided by our last-stage commitments. Our system exhibits flexibility by offering distinct performance-security trade-off services tailored to different user requirements.

The signed response for both o-rollups and the off-chain response represents a promise to include and process the request on a specific LSM page. The client can use the signed response to punish the off-chain node in case it lies or acts maliciously—by not honoring its promise to process the request correctly. (We discuss the punishment smart contract in the paper. This contract withdraws a penalty from the off-chain node's escrow fund if malicious behavior is proven). RollStore ensures that any falsehood by an off-chain node will eventually be detected by a client. This is done by processing transactions through three stages of commitment, starting from the off-chain response (fastest commitment but with weaker guarantee), to o-rollups (slower but with stronger guarantee), and finally to zk-SNARK (slowest but with strongest guarantee). In the final step, RollStore checks if the zk-SNARK result matches the response sent to the client. If they did not match, this is a sign that the off-chain node lied. This process guarantees the detection of such malicious acts. This guarantee of detecting malicious acts—with severe monetary punishments through the punishment smart contract—is a deterrent for off-chain nodes to act maliciously in the first stages of commitment using off-chain response and o-rollups.

The contribution of the paper is summarized as follows:

- RollStore is the first *dynamic indexing solution* for hybrid DApps, featuring a three-stage design that represents a novel commitment path.
- RollStore is the first solution to incorporate both zk proofs and o-rollups verification in the problem of indexing by a novel design that builds on LSM and MMR trees.
- RollStore addresses the multi-level trade-off challenge between security and performance, effectively tackling
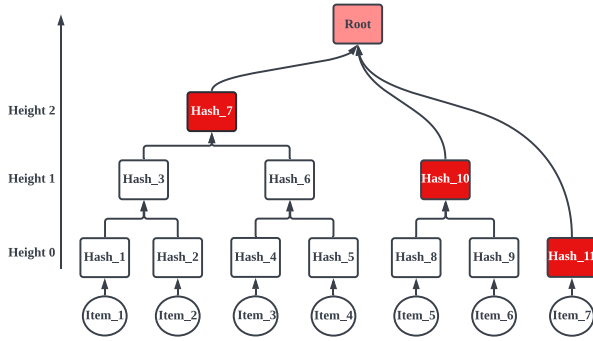
Fig. 1.   An example of the Merkle mountain range tree.

Fig. 2.   An example of blockchain rollups.

the task of accommodating diverse user requirements in DApps.

In the rest of the paper, we present background and related work in Section II. Then, we present the design of RollStore in Section III. An experimental evaluation is presented in Section IV. We conclude in Section VI.

II. BACKGROUND

A. LSM Trees

Log-Structured Merge (LSM) Trees are widely used for data indexing [31]. LSM trees are designed to support fast data ingestion by appending entries for write operations instead of updating the corresponding old entry in-place. Periodically, appended data is merged with the rest of the LSM tree. This append-only nature of ingestion makes LSM trees a suitable candidate for write-intensive workloads.

There are many LSM tree variants [31]. Here, we provide a description of the common and typical design aspects of LSM trees. Generally, LSM trees contain several levels, L_0, L_1, $\cdots$, L_k. Level L_0 is maintained in main memory while other levels are persisted on disk. Incoming write operations are appended to an in-memory mutable table. When the mutable table is full, the data in it—represented as key-value pairs—is ordered and inserted to L_0 as a new page. L_0—as well as other levels—has a threshold on the number of pages. Once this threshold is met, the data in the L_0 pages is merged with the pages in the next level. This continues until data reaches the final level L_k. When two levels are merged, one of two widely used techniques is used: tiering and leveling [10].

B. MMR Trees

The Merkle Mountain Range (MMR) tree [45] is a variant of the Merkle tree [33] which is structured as a group of underlying Merkle trees (Fig. 1). A MMR tree is an append-only tree where elements are added as leaf nodes from left to right (Fig. 1 shows the case of adding items 1 to 7 from left to right.) Once there are two children nodes at a level with no parent node, a parent node for the two children nodes is generated at the higher level. For example, consider the MMR tree in Fig. 1 where internal node numbers represent the generation order (e.g., Hash_i is the ith generated hash node). Hash_6, for example, is generated
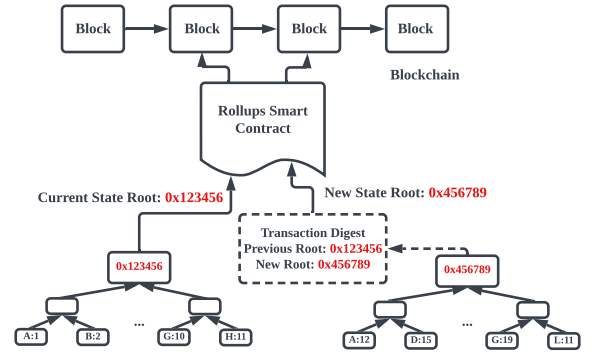
after Hash_4 and Hash_5 are added (for items 3 and 4). Adding Hash_6 in turn leads to creating Hash_7. In the figure, there are three underlying Merkle trees with roots Hash_7, Hash_10 and Hash_11. These roots are also called peak nodes, and each underlying Merkle tree is called a *mountain*. The MMR root is calculated as the hash of the peak nodes.

An MMR tree can provide an inclusion proof of a data item in a similar way to Merkle trees. The inclusion proof includes the sibling node of every node in the path from the data item to the MMR root. For example, in Fig. 1, the inclusion proof of item Item_3 contains Hash_5, Hash_3, Hash_10, and Hash_11. A client receiving the proof calculates the MMR root using the provided hashes. If the calculated MMR root matches the original MMR root, then the client knows that the received item is correct. A malicious server cannot generate a false inclusion proof of an item that is not in the MMR tree. This is because any change to leaf nodes alters the MMR root. Additionally, a strong one-way hash function would likely (with very high probability) not lead to a collision of the hashes of two different input texts [35].

C. Blockchain Rollups

Rollups is a layer 2 solution to enhance blockchain scalability, which aims to reduce the performance overhead and monetary cost of operations on-chain [41]. In rollups, transactions are aggregated and executed in off-chain nodes, then the on-chain smart contract maintains the root value (e.g., Merkle root hash value) which corresponds to the current state (Fig. 2). The off-chain node publishes a digest for batched transactions, which contains the previous Merkle root MMR_Pre (0x123456 in the figure) and the computed new Merkle root MMR_New (0x456789 in the figure). When such digest is written on-chain, the smart contract checks whether the previous Merkle root MMR_Pre in this digest matches the current Merkle root stored in the smart contract; if it does, the smart contract updates its state root to the new MMR root, MMR_New.

The main challenge with rollups solutions is that the off-chain node might act maliciously and provide a new digest, MMR_New, that corresponds to an incorrect new state, i.e., the transactions that lead to the new state with digest MMR_New are incorrect or malicious transactions. To overcome this challenge, two types of
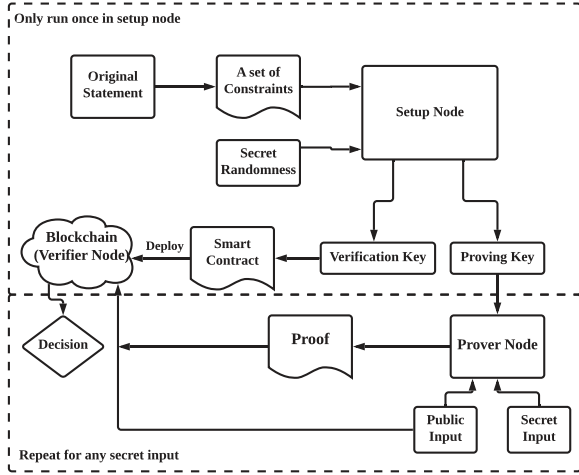
Fig. 3. Components and flow of zk-SNARK.

rollups variants are used: optimistic rollups and zero-knowledge rollups.

1) Optimistic Rollups: O-rollups ensures that the new hash that is written on-chain, MMR_New, is based on correct computation by using an *interactive fraud-proof mechanism* [1]. In this approach, the new digest is written on-chain before verification (optimistically). Then, off-chain nodes have an opportunity to challenge the correctness of the state that is represented by the new digest, MMR_New. This opportunity remains for a pre-defined *challenge period*. After this period expires, if no successful challenges are raised, then the new digest is assumed to be correct. Otherwise, if a client challenges the correctness of MMR_New, then a special smart contract verifies the correctness of the challenge. If the new state turns out to be incorrect, the challenge succeeds, and the smart contract reverts the state to a previous correct state. The first problem with o-rollups is that the challenge period needs to be long–several days to a week [41]–to provide an opportunity for challengers. Another issue is that it requires an incentive mechanism to encourage active participants to challenge and compensate them for their efforts.

2) Zero-Knowledge Rollups: Zk-rollups is a non-interactive solution based on a zero-knowledge proof mechanism [22], [53]. In zk-rollups, a digest includes a validity proof. The validity proof proves that the generated new digest MMR_New corresponds to a state of the data that is correct, i.e., the new state with digest MMR_New is the outcome of processing transactions on the previous state with digest MMR_Pre. The zk-SNARK protocol is one of the methods used to implement zk-rollups [8], [17], [19], [21], [50]. The zk-SNARK protocol is used in the following way by utilizing three components: a setup node, a prover node, and a verifier node (Fig. 3):

- The setup node generates a proving key Pk_s and a verification key Vk_s that will be used to generate and verify proofs. For zk-SNARK, the setup—which is a one-time process before operation—must be performed by a trusted node. After setup, there is no need for trusted nodes. The generation of the two keys is influenced by the type of computation that needs to be proven. The user provides the program to be proven/verified as well as the inputs to such computation. The user assigns which parts of the inputs are public and which parts are secret. In RollStore, for example, the program to prove/verify is the one that updates the LSM tree and produces a new state represented by MMR_New; and the inputs to the program are the previous state and its digest MMR_Pre as well as the operations that are applied to the previous state to generate the new state.

- The prover node is responsible for generating the proof of the computation outcome. It needs three parameters, the proving key Pk_s, the public information, Inf_Pub and the secret information, Inf_Secret. After collecting these parameters, the prover node generates a proof π_s of the computation.

- The verifier node needs three parameters: the verification key Vk_s, the public information, Inf_Pub, and the proof π_s. After collecting these parameters, the verifier node generates a decision (True or False). In hybrid blockchains, the verifier can be a smart contract. Typical zk-SNARK protocols are designed so that verification is fast at the expense of a more lengthy proof generation process. This is suitable for hybrid blockchains, since generating proofs is performed by off-chain nodes that do not have the constraints of smart contracts, while verification is performed on-chain.

It is important to note that zk-SNARKs offers a different functionality compared to digital signatures. Digital signatures can be used to verify the authenticity of data that is signed by a trusted node. However, digital signatures cannot be used to verify computation that modifies data from one state to another state. Also, digital signatures rely on a trusted node signing the data. zk-SNARKs, on the other hand, can be utilized to verify computation and it can generate zero-knowledge proofs on an untrusted node.

D. Related Work

Blockchain-Based Databases (BBDBs): BBDBs are databases that utilize blockchains in various ways to utilize blockchain's features such as transparency and immutability [4], [14], [16], [32], [36], [40]. Most of this work targets *permissioned* blockchain settings, where the blockchain network has a closed-membership assumption, i.e., all the participants in the blockchain network are authenticated and known. This permissioned setting allows faster and cheaper processing which makes it suitable for enterprise and consortium (multi-organization) applications. However, the closed-membership assumption of permissioned blockchain prevents their use in DApps that require open-membership and not rely on a single or group of fixed members. We target supporting these DApps which is now a large market with hundreds of thousands of users and hundreds of millions of dollars in assets [12], [39], [51]. For this reason, we tackle the unique challenges that are faced when building a BBDB over permissionless blockchains. Due to their focus on permissioned blockchains, prior BBDBs [5], [25] do not factor in the monetary cost and latency challenges of using permissionless

blockchain. This led to them being unsuitable for DApps due to high costs and latency from writing raw data directly to the blockchain [4], [14], [16], [32].

Blockchain Rollups: Blockchain rollups was proposed as a layer-2 scaling solution for blockchains [41] (see Section II-C for an overview). Prior work utilizes either one of the two rollups strategies—suffering from the disadvantages of the chosen method. RollStore combines the two in a manner that allows benefiting from their advantages while masking their disadvantages. In particular, o-rollups digests can be written faster on-chain but their challenge period takes a long time up to days [41]. On the other hand, zk rollups' time to generate the digest/proof to be written on-chain is longer than o-rollups time to write the digest; but, that proof is sufficient to finalize the commitment of the operation without having to wait for days in a challenge period. RollStore's design allows enjoying the benefits of fast o-rollups digest writing (stage 1) as well as the finality of zk rollups (stage 2). Finally, RollStore introduces a new kind of rollups that we utilize in stage 0 that is much faster than other kinds of rollups as it does not require writing on-chain. This is possible via a penalty strategy using a penalty smart contract.

Secure and Authenticated Off-Chain Processing: There have been a lot of recent work on utilizing off-chain nodes to perform compute and storage tasks for blockchain applications [2], [23], [29], [34]. This is because utilizing off-chain nodes can reduce the monetary cost and performance overhead of blockchain applications. The challenge that is faced by many works in this category is how to utilize off-chain nodes that might be untrusted. For this reason, trusted and authenticated data structures were used to provide trust on the outcome of off-chain nodes' processing [46], [52], [54]. These solutions focus on querying and storing data securely off-chain, but do not support operations that mutate the state of data, unlike RollStore and blockchain rollups.

Related to this category is the plethora of work in authenticated data and query processing [27], [56], [58]. These methods can be inherited and utilized in the context of querying and processing data in hybrid onchain-offchain applications [4], [46].

III. ROLLSTORE DESIGN

In this section, we present the design of RollStore.

A. System Model and Interface

System Components: RollStore consists of the following components (Fig. 4):
- *Updater node:* the updater receives the write and read requests from clients. It maintains a mutable table T_{mut}, Level L_0 of the LSM tree, and a MMR tree for data in L_0, called MMR_0. Data in L_0 represents stage 0 committed data.
- *Backup node:* the backup maintains LSM levels (L_1 and L_2), and two MMR trees MMR_1 and MMR_2, each corresponding to an LSM level. L_1 contains data that is stage 1 committed (o-rollups) and L_2 contains data that is stage 2 committed (zk-SNARK).
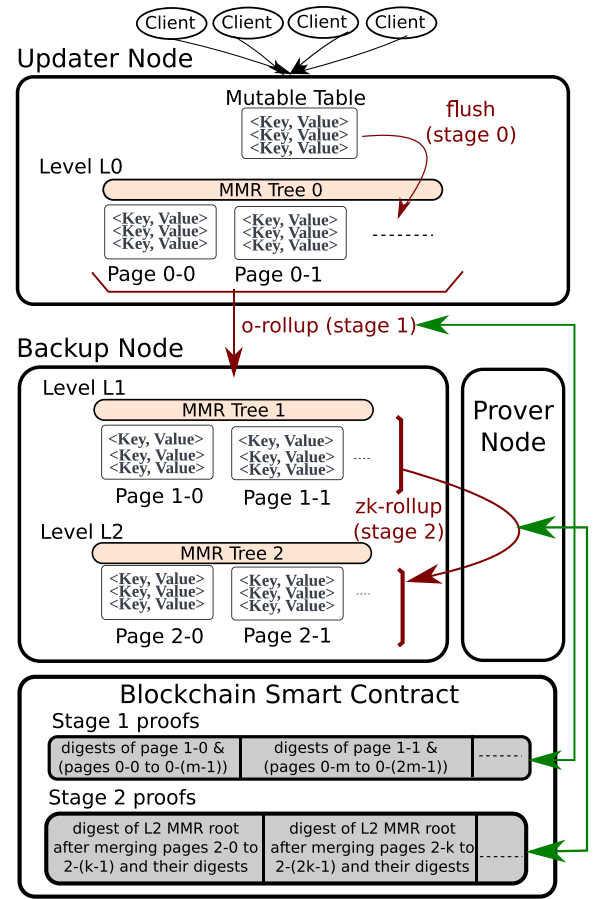

Fig. 4.   Data architecture of RollStore.

- *Prover node:* the prover performs zk-SNARK computation to generate proofs of L_2 pages.
- *Smart contracts:* on-chain smart contracts handle the verification and maintenance of digests related to stage 1 and 2 committed data. Also, the smart contract handles the punishment strategy by verifying whether an off-chain node is malicious if a *challenge* is raised during stage 0 or 1 commitment. If the challenge indicates malicious activity, then the smart contract punishes the off-chain node by withdrawing funds from its escrow account.

The three types of off-chain nodes can be co-located or placed across different machines. Also, the three types of nodes can be elastically scaled, where more nodes of a node type are added to scale its computation, e.g., prover nodes can be added to speed up zk proof generation. We discuss scaling node types in Section III-C.

System Interface: RollStore provides a read, and write operation interface for users to read and write data.
1. *Write:*   (In: key − value pair, Out : inclusion proof, sequence number): this call takes a key-value pair as the input, the output of this call is the inclusion proof for the key-value pair and the sequence number where it is added. Clients use the write interface to submit write requests to the updater node.

2. *Read:* ($\mathsf{In : key}$, $\mathsf{Out : value}$, $\mathsf{inclusion proof}$): this function takes a key as the input, the output of the read operation is the corresponding value, and the inclusion proof for that value. The inclusion proof might be (1) local (stage 0), (2) global without full verification (stage 1), or (3) global with full verification (stage 2). Clients use the read interface to submit the read requests to the updater node.

Security Model: Off-chain nodes (updaters, backups, and provers) are not trusted. They can deviate from the protocol in arbitrary ways, similar to Byzantine failures [30]. Off-chain nodes can collude together and with clients. The smart contract logic executes correctly—without deviating from the protocol—due to running on blockchain. Write requests are assumed to be authenticated by a client, which prevents off-chain nodes from fabricating clients requests.

Data Model: The following are the main data structures maintained in RollStore (Fig. 4):

1. *Distributed LSM tree:* The LSM Tree maintains the key-value pairs appended to RollStore. It has a mutable table and three levels. The mutable table T_{mut} is at the updater node. T_{mut} holds the most recently appended entries that are being staged to be pushed to level L_0 of the LSM tree. Level L_0 maintains batches of appended data objects and is stored in the updater node. A page is added to L_0 only after a signed response is sent back to the clients with operations corresponding to the page's data objects. Pages are assigned a monotonically increasing sequence number Seq. We denote the ith appended page to L_0 as page $P0_i$. Level L_1 represents pages that are consolidated from level L_0. Before a page is written to L_1, its digest must be written on-chain as part of stage 1 o-rollups. Pages in L_1 are also assigned monotonically increasing sequence numbers, where the ith page to be added to L_1 is denoted $P1_i$. Each page in L_1 represents a consolidation of pages in L_0. Therefore, page $P1_i$ represents the consolidation of pages $P0_{i*m}$ to $P0_{(i*m)+m-1}$ from L_0, where m is the threshold of the number of pages in L_0 to trigger a consolidation of pages in L_0 and creating a new page in L_1. Note that a new page in L_1 is added to the set of pages in L_1 and not merged with existing pages. Therefore, pages in L_1 may have overlapping key ranges.

   Level L_2 represents pages that are merged from level L_1 after a successful zk proof is generated for them. Pages from L_1 are merged into L_2 in the order of their sequence numbers. Specifically, when the next k pages at L_1 are zk proven, then they are merged with the pages that already exist in L_2. Therefore, after i merge steps to L_2 (where $i = 0$ corresponds to the first step), the pages in L_2 contain the merged key-value pairs that represent pages $P1_{(i*k)}$ to $P1_{(i*k)+k-1}$, which correspond to pages $P0_{i*k*m}$ to $P0_{(i*k*m)+(m*k)-1}$. Pages are merged from L_1 into L_2 which means that key-value pairs in L_2 are ordered across pages and each page has a unique range of key-value pairs that do not overlap with other pages in L_2.

2. *MMR trees:* The MMR trees are used to create compact digests of the data in the LSM tree. There are three MMR trees, each one corresponding to an LSM level; MMR_0 for L_0 in the updater, and MMR_1 and MMR_2 for L_1 and L_2 in the backup. The digests of this MMR tree are used for the verification process of stage 1 and stage 2 committed data.

3. *On-chain digests:* The smart contract maintains a map of digests and proofs that are related to stage 1 and 2 committed data. Users query these digests to verify the authenticity of responses from off-chain nodes. There are two sets of digests/proofs. The first set is for stage 1 digests. In this set, each smart contract digest $\mathsf{SC\_Digest}^1{}_i$ corresponds to page $P1_i$, which is a consolidation of pages $P0_{i*m}$ to $P0_{(i*m)+m-1}$, where m is the threshold for the number of pages in L_0 before consolidation. The second set is for stage 2 digests/proofs. In this set, each smart contract digest/proof $\mathsf{SC\_Digest}^2{}_i$ corresponds to the ith merge operation on L_2. The ith merge operation in L_2 corresponds to merging the key-value pairs that are consolidated in pages $P1_{i*k}$ to $P1_{(i*k)+k-1}$, where k is the threshold of the number of L_1 pages to merge into L_2.

Commitment Model: A write operation W of client c goes through three stages of commitment:

1. *Stage 0:* When the updater sends a signed response back to c, W is considered stage 0 committed. This signed response includes acknowledging the operation is received and promising to add it to page $P0_i$. This stage of commitment is the fastest as blockchain smart contracts are not involved. Client c can use the signed response later to prove maliciousness if an updater has lied (e.g., operation W is not included in $P0_i$ and later not included in $P1_{\lfloor i/m \rfloor}$, where m is the page threshold at L_0). A penalty smart contract receives punishment requests from clients that wish to prove maliciousness and punish malicious off-chain nodes.

2. *Stage 1:* Consider the page $P0_i$ that includes W and the page $P1_{\lfloor i/m \rfloor}$ that is the consolidated L_1 page that includes $P0_i$. When the digest of $P0_i$ and $P1_{\lfloor i/m \rfloor}$ are written as $\mathsf{SC\_Digest}^1{}_i$ to the smart contract, the operation W is considered stage 1 committed. This takes longer than stage 0 commitment since the digests need to be written on-chain. However, it provides a stronger consistency guarantee—if two clients observe the state of a page $P0_i$ that is stage 1 committed, then they agree on the state of the page. However, this stage of commitment does not guarantee that the page itself is the result of correct computations. This is because the off-chain node can create a digest of arbitrary data. The client has to wait for the next stage of commitment to ensure that the derivation of the page is correct. However, if the off-chain nodes lie about stage 1 committed pages, they won't be able to perform stage 2 commitment. This consequently leads to clients sending a challenge request to the penalty smart contract that punishes the off-chain nodes.

3. *Stage 2:* Operation W is considered stage 2 committed when the following is true: the zk proof of a merge that includes page $P1_{\lfloor i/m \rfloor}$ is verified by the smart contract and written as $\mathsf{SC\_Digest}^2{}_{\lfloor i/(m\cdot k) \rfloor}$, where k is the page threshold at L_1. This is the strongest correctness guarantee
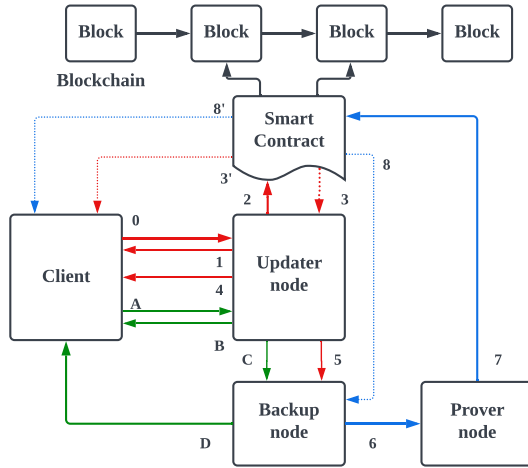
Fig. 5. Lifecycle of RollStore requests. Red arrows represent stage 0 and 1 steps of write operations; blue arrows represent stage 2 steps of write operations; and green arrows represent steps of read operations.

as a page that is stage 2 committed is guaranteed to have been computed correctly by zk-SNARK. However, generating such a proof is a complex and time-consuming process.

B. RollStore Core Design and Protocol

We now provide a description of RollStore's core design and protocols. This includes the protocols for read and write operations with a deployment of one updater, one backup, and one prover. We will describe the protocol as we follow the end-to-end life-cycle of write and read requests (Fig. 5 shows the flow of operations that we refer to as *steps* in the rest of this section).

Stage 0 Commitment: A client c creates a signed write request W_i that has a key-value pair, $[K_i, V_i]$, and signature, S_c, as payload; $W_i = (S_c, [K_i, V_i])$. The signed write request is sent to the updater node (step 0 in the figure). The updater node, after receiving the signed request for W_i, adds W_i to the mutable table T_{mut} of the LSM tree. Once T_{mut} is full, the key-value pairs in T_{mut} are reordered by their key and written as a new page $P0_i$ in L_0 of the LSM tree located in the updater node. Each page in L_0 is assigned a monotonically increasing sequence number. This sequence number will be used by clients to track their operations and ensure that they are eventually stage 1 and 2 committed. Page $P0_i$'s sequence number is denoted Seq_i (if not mentioned otherwise, assume that $Seq_i = i$).

The MMR tree in the updater node is updated to include data in $P0_i$. At this point, a signed response, Ack_i, is sent back to client c for stage 0 commitment of W_i (step 1). This response includes: a stage-0 proof of inclusion of W_i in $P0_i$ (using the MMR tree) denoted $Prf_{W_i}^0$; also, Ack_i includes $P0_i$'s sequence number Seq_i and the updater's signature S_u; $Ack_i = (S_u, Seq_i, Prf_{W_i}^0)$. At this point, client c considers the operation stage 0 committed and has a signed response that the updater node promised to include W_i as part of page $P0_i$ with sequence number Seq_i in the LSM tree. If the updater node does

not honor this promise, then client c can use this signed response to trigger a punishment smart contract.

Stage 1 Commitment: The updater node continues adding pages to L_0 until the threshold of the number of pages, m, is exceeded. At this point, Stage 1 commitment of pages in L_0 starts. The pages in L_0—including $P0_i$—are now mapped by the updater's MMR tree, MMR_0. A consolidated page $P1_{\lfloor i/m \rfloor}$ that consolidates the key-value pairs in pages in L_0 is created. The hashes and sequence numbers of the pages in L_0 and the hash of $P1_{\lfloor i/m \rfloor}$ are sent to the smart contract (step 2). The smart contract records this root hash as the stage 1 commitment o-rollups hash for the pages with the corresponding sequence numbers. This hash is recorded as SC_Digest$^1_{\lfloor i/m \rfloor}$. Then, the smart contract emits an event to the updater node and clients about the new written hash and the corresponding page sequence numbers[2] (step 3 and 3'). The updater node sends a signed response to the client for stage 1 commitment (step 4). This response includes the digest of page $P1_{\lfloor i/m \rfloor}$ and the operation's inclusion proof.

After stage 1 commitment is performed for pages in L_0, all pages in L_0 and the consolidated page $P1_{\lfloor i/m \rfloor}$ are sent to the backup node to be inserted to L_1 (step 5). The original pages in L_0 are sent with $P1_{\lfloor i/m \rfloor}$ to the backup node as they will be used to provide inclusion proofs for read request as well as used to generate the zk proofs in stage 2 commitment. After sending $P1_{\lfloor i/m \rfloor}$ and the L_0 pages to the backup node, the pages in L_0 are cleared in the updater node. The page $P1_{\lfloor i/m \rfloor}$ will not be merged with pages in L_1, rather it will be inserted as a new page. This means that the key-value pairs range of one page in L_1 may overlap with the ranges of other pages in L_1.

Stage 2 Commitment: After page $P1_{\lfloor i/m \rfloor}$ is added to L_1, the backup node checks if the page threshold for L_1, denoted k, is met. If it did, the backup node starts the stage 2 commitment process using zk-SNARK for pages in L_1. This process merges the pages in L_1 with the pages in L_2. In the jth merge operation, the pages to be merged from L_1 are from $P1_{j*k}$ to $P1_{(j*k)+k-1}$.

The merge is performed in the backup node. Then, the merge information is sent to the prover to generate a proof of the correctness of the merge. The information to prove the jth merge includes: (1) pages $P1_{j*k}$ to $P1_{(j*k)+k-1}$, (2) pages $P0_{j*k*m}$ to $P0_{(j*k*m)+(k*m)-1}$, (3) pages in L_2, (4) the MMR root of L_2, MMR_2-Pre, before the merge, and (5) the MMR root of L_2, MMR_2-New, after the merge (step 6). The prover node takes all this information to generate a proof that: (1) each page in pages $P1_{j*k}$ to $P1_{(j*k)+k-1}$ is generated correctly from the corresponding L_0 pages, and (2) the merge of pages in $P1_{j*k}$ to $P1_{(j*k)+k-1}$ with pages in L_2 (with MMR root MMR_2-Pre) yields a new state with MMR root MMR_2-New.

After the zk-SNARK proof is generated, it is sent to the smart contract to be validated (step 7). The smart contract performs the following: (1) it validates the proof, (2) verifies that the hashes used for L_0 and L_1 pages match the ones in stage 1 commit

---

[2] A smart contract in permissionless blockchain cannot communicate directly to off-chain nodes. Here, we use the Ethereum emit operation that allows off-chain nodes to filter and pull emitted data of interest from the smart contract. Emit events in Fig. 5 are shown as dotted arrows.

for the pages with the same sequence numbers, (3) verifies that `MMR_2-Pre` corresponds to the previous verification, (4) record the new proof digest on-chain as `SC_Digest`^2_j for future access by clients, and (5) an event is emitted to the backup node and clients with operations in pages $P1_{j*k}$ to $P1_{(j*k)+k-1}$ (step 8 and 8'). The writes in $P1_{j*k}$ to $P1_{(j*k)+k-1}$ are now considered stage 2 committed. The backup node—once the proof is verified by the smart contract—writes the merged pages to L_2 and clear pages $P1_{j*k}$ to $P1_{(j*k)+k-1}$ from L_1.

Read Operations: A client reading a key x specifies the level of the read request: stage 0, stage 1, or stage 2 committed. We now show the process for a stage 0 committed read. (Stage 1 and stage 2 committed read follow the same process but starting from the backup node at level L_1 for stage 1, and L_2 for stage 2). First, the read request r is sent to the updater node (step A). When the updater node receives a read request, it responds with a signed response with the corresponding key-value pair and MMR inclusion proof (step B). The implications of this commitment is similar to stage 0 commitment for write operations where a read client can use the signed response as a proof of a lie by the updater node in the future.

If the requested key was not in L_0, then the read request moves to L_1 (this is also the start point of a stage 1 committed read). The client reads the most recent written stage 1 and 2 digests from the smart contract to match them with the response once received. The updater node forwards the request to the backup node (step C) that responds with the corresponding key-value pair from a page in level L_1 and the inclusion proof. The guarantee of this read request is similar to a stage 1 committed write where any two read requests would agree on the result but the result is not yet verified by a zk proof. If the requested key is not in L_1, then the read request moves to L_2 (this is also the start point of a stage 2 committed read). The backup node returns the requested key-value pair from L_2 if it exists (step D). The client can check the inclusion proof against the smart contract and verify that the read data object has been verified with a zk proof.

In both the stage 1 and stage 2 reads, the client reads the proof/digest from the smart contract prior to the beginning of the operation (note that unlike writing to blockchain smart contracts, reading data from a smart contract is a fast operation). Consider a read request that goes to a level L_i; if the data object does not exist in that level and the read is forwarded to L_{i+1}, then a proof of non-existence is also returned from L_i. This can be done by returning the pages with the ranges that overlap the requested key so that the client can verify that the key does not exist.

C. Scaling Off-Chain Nodes

In this section, we discuss the scaling strategies for the three node types, updaters, backups, and provers. This allows each node type to utilize multiple nodes—instead of one node—to service requests and improve performance and/or resilience.

Scaling Updater and Backup Nodes: The updater and backup nodes maintain LSM and MMR data. We discuss scaling these two types of nodes—which is increasing the number of updater and backup nodes to achieve higher throughput through distributing the workload.

RollStore offers a single-key operation interface (Section III-A). The simplicity and efficiency of single-key operations make them a suitable approach in DApps. Notably, applications such as decentralized marketplaces utilize single-key operations for activities like minting, allocation, and transfer of items; these operations are performed by changing (or creating) a single data record that corresponds to the item being managed. Another set of examples include identity and access management where single-key operations are used for authentication and access control. Other than applications where it is sufficient to perform single-key operations, the strategy we adopt is to store relevant data together within one shard. This approach—given data locality—enables ordering more complex operations even before stage 2 commitment. Our scaling strategy centers around single-key operations, involving the sharding of data into n shards. Each shard is maintained by a separate set of two nodes, one for the updater and the other for the backup. The smart contract is also deployed as n independent instances, one for each shard. Each shard operates independently, enabling parallel processing of data, thereby allowing the system to scale effectively. This means that different shards can handle different parts of a complex write operation simultaneously. This isolation enables complex write operations to be performed on one shard without affecting the data in other shards. However, it is crucial to note that while each shard functions autonomously, our system incorporates a coordination mechanism to address transactions that involve data across multiple instances. This coordination is essential for processing complex multi-data write operations. The global ordering mechanism realized through a blockchain-based smart contract (RollStore stage 2), plays a pivotal role in orchestrating these operations. This ensures that transactions touching data on multiple instances are executed in a coordinated and deterministic order (see Section III-F, Theorem 4).

Scaling Prover Nodes: The prover node is tasked with generating the zk proofs of stage 2 commitment. Scaling the prove operation is important as it is a lengthy process. To scale proving tasks, we maintain n provers and distribute the zk proving workload across the n provers. Specifically, each zk proving task `Task_i` is divided into n subtasks, `Task`$_{sub\_1}$ ...`Task`$_{sub\_n}$. Each subtask is responsible for proving N/n data items, where N is the total number of data items in the prove task.

Resilience and Availability: Increasing the numbers of nodes can also serve the purpose of increasing the crash resilience and availability of RollStore. Specifically, for stateful node types—updaters and backups—the state of each node can be maintained by a replication cluster [9]. Therefore, the failure of one node can be tolerated by the rest of the nodes in the cluster. For stateless nodes—provers—adding and replacing provers is straight-forward, since the proving task is stateless. Therefore, in the case of a prover failure, it can be replaced by another node that takes over processing the requests from the backup node.

D. DApp-Indexing-as-a-Service Model

In this section, we discuss the payment model to enable DApp-indexing-as-a-service. In this model, each off-chain node
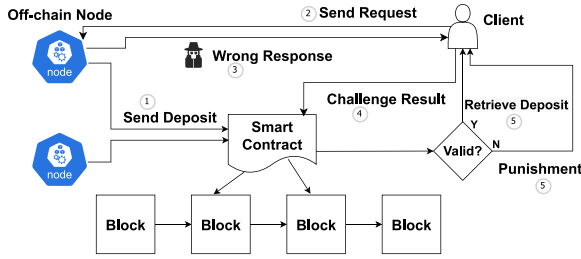
Fig. 6.   Incentive mechanism.

deposits an amount of cryptocurrency to an escrow fund in the penalty smart contract in the setup stage. the addresses (Ethereum addresses and IP addresses) of off-chain nodes that successfully deposited the fund in the smart contract would be stored in the penalty smart contract.

The penalty smart contract is initialized with the following variables: `OC_Address`$_E$, `OC_Address`$_{IP}$, `OC_Deposit`, and `OC_Signature`. The first two variables store the Ethereum address and IP address of the off-chain node that signed up as a server node. The `OC_Deposit` variable sets an amount of how much cryptocurrency (Ether) should be deposited to successfully sign up. The variable `OC_Signature` stores the digital signature of the off-chain node that provides the service. Users can send their requests (writes or reads) to valid server nodes that successfully stored their addresses in the penalty smart contract. The valid server nodes process these requests and interact with other nodes and the blockchain network.

E. Failure Examples

In this section, we briefly present the new threats and discuss how RollStore addresses them. RollStore allows servers to act maliciously, but it guarantees the detection of the malicious act and the punishment of dishonest servers. RollStore handles these threats using a three-stage security protocol, ensuring that the committed state (stage 2) is consistent across all honest nodes, and any malicious act can be detected and punished in the three-stage commitment process

Threat in Stage 0 Commitment: An adversary might deliberately respond incorrectly during stage 0 commitment. In this scenario, if the adversary is an updater node, given a writes request W_i, the malicious updater can return a wrong sequence number Seq_w, or wrong inclusion proofs $Prf^w_{W_i}$. RollStore guarantees that any incorrect response will be detected and punished.

In stage 0 commitment, the updater provides a signed response back to the client that its write request W_i is part of a page $P0_i$ in L_0 with sequence number Seq_i. $Prf^0_{W_i}$ is the signed inclusion proof of the write in page $P0_i$. An updater must use this page $P0_i$ during the o-rollups operation of stage 1 commitment. The client can verify that this is the case by observing the hashes that were written on-chain for stage 1 commitment.

As shown in Fig. 6, we also designed an incentive mechanism to encourage clients to verify and challenge such malicious behavior. For example, if the hash that corresponds to sequence number Seq_i is the same as the one received in the stage 0 response, then the promise is honored. Otherwise, the client starts the penalty process. The client sends a request to the penalty smart contract with the following input: the received stage 0 response received from the updater node, i.e., $Ack_i = (S_u, Seq_i, Prf^0_{W_i})$. The penalty smart contract verifies whether the penalty should be applied by verifying the authenticity of Ack_i (that it is signed by the updater signature S_u), and checking whether the MMR root hash in $Prf^0_{W_i}$ equals the o-rollups hash in the smart contract for the page with sequence number Seq_i. If the hash is different, the penalty is applied. The client can retrieve the deposit from that adversary node. However, if clients submit an invalid challenge request to the penalty smart contract, the smart contract will impose an additional payment that clients are required to pay. As a consequence of an invalid challenge, clients will also face a temporary restriction from connecting to our system.

Threat in Stage 1 Commitment: An adversary can upload the wrong digest of the stage 1 commitment or send the wrong pages used in the proof generation. In this scenario, if the adversary is an updater node, for a write operation W_i in page $P0_i$ with sequence number Seq_i, the stage 1 commitment hashes `Hash_o` in the smart contract include the hash for $P0_i$ as well as $P1_i$ which is the merge result of page $P0_i$ and other L_0 pages. The malicious node can upload a wrong digest `Hash_w` instead of `Hash_o` or send the wrong pages $P1_w$ used in the later zk proof generation. RollStore guarantees that no incorrect response can pass through the rest of the system and will be detected, thus not harming the system.

In stage 1 commitment, the client observes the stage 1 commitment hashes that are written to the smart contract for stage 1 commitment. RollStore protocols ensure that these hashes are the same ones that will be used in stage 2 commitment. This is done because the smart contract—when verifying the proof in stage 2 commitment—verifies that the hashes used to generate the zk proof are identical to the ones used in stage 1 commitment by o-rollups. This is performed by checking which hashes were written to the smart contract during stage 1 for the corresponding pages used in the proof generation. In the case of W_i, this includes the hashes for pages $P0_i$ and $P1_i$. Since this is guaranteed by the verification process in stage 2 commitment, the off-chain nodes must keep their promise in using the stage 1 pages in stage 2 commitment. If they commit a false digest $Hash_w$, then they would have to indefinitely delay the stage 2 commitment, and if they send the wrong pages $P1_w$, they cannot generate the correct zk proof. Thus, they cannot pass through to the final commitment. Clients can then send requests to the penalty smart contract to penalize such incorrect behaviors.

Delay Threat: In both stage 0 and stage 1 commitment, another type of malicious act that the off-chain nodes may do is to delay the next stages of commitment indefinitely. In this case, we designed a two-step process to prove and punish the off-chain nodes.

Consider the case of a client—with operation W_i in $P0_i$—that received a stage 0 or stage 1 response r at time t. If the user suspects that the off-chain nodes are not continuing the processing of the request and future stages of $P0_i$. The first step

is to send a delay $-$ notification request to the penalty smart contract. The input to this notification is a proof that a signed response is received from the off-chain node for page $P0_i$. The smart contract records this notification with the blockchain block number that it was written in, b_i^1. Now, the off-chain nodes have an opportunity to finalize the commitment of $P0_i$ before the second step. The initiation of the second step is contingent on reaching a future block b_i^2, satisfying the condition $b_i^2 - b_i^1 > bt$, where bt represents the threshold specifying the minimum number of blockchain blocks that must have elapsed before the second step can be triggered. This is a predefined number that is agreed on by the off-chain nodes and should be sufficiently large to allow for processing requests. If bt blocks passed and the client still observes that $P0_i$ is not committed, it starts the second step by sending a delay $-$ followup request. This request references the first step. The penalty smart contract checks that bt blocks have been committed since the previous notification and if $P0_i$ is still not committed. If both conditions are true, then the penalty logic is applied and funds are withdrawn from the off-chain node escrow fund.

This strategy can be applied separately for stage 0 and 1 commitment delays where there is a threshold bt for each type of commitment. We use a block number threshold as it is a standard practice in smart contract development. The reason for using block numbers between requests is that it is predictable since the commitment of a block typically takes a predefined amount of time.

F. Safety

In this section, we formally prove the safety of read and write operations in RollStore. Specifically, we first demonstrate that the guarantees of each level of commitment are met, and then we discuss the linearizability of RollStore. With these theorems established, RollStore ensures that any security violations will be detected and punished eventually, and the final commitment stage (stage 2) remains linearizable across all honest nodes.

Theorem 1. (Stage 0 safety guarantee): For a write w that is stage 0 committed in page $P0_i$ with sequence number i, either (1) the write w is going to be part of page $P0_i$ that is committed in stage 1 as part of the o-rollups in the consolidated page $P1_{\lfloor i/m \rfloor}$, where m is the threshold of the number of pages in L_0; or (2) the client can prove that the updater provided a false promise to include w in page $P0_i$.

Proof: We prove this statement by contradiction. Assume to the contrary to the defined guarantee that there is a write w that is stage 0 committed as part of page $P0_i$, however, (1) the off-chain node used another page $P0_i'$ (with the same sequence number as $P0_i$) during stage 1 commitment for page $P1_{\lfloor i/m \rfloor}$, and (2) the client cannot prove the fake promise about w.

If page $P0_i'$ was used in stage 1 o-rollups of $P1_{\lfloor i/m \rfloor}$ instead of $P0_i$, this means that the stage 1 digest written on-chain, SC_Digest$^1_{\lfloor i/m \rfloor}$ for page $P0_i$ is different from the one returned to the client during the response (step 1 in Fig. 5). This is because any change to the contents of the page would lead to a different digest. Therefore, the client knows that the off-chain node lied by detecting the different digests. The client

can then prove that the off-chain node promised to include w as part of $P0_i$ by showing the signed response received in step 1. This is a contradiction, which proves the guarantee. $\square$

Theorem 2. (Stage 1 safety guarantee): For a write w that is stage 1 committed in page $P1_j$ with sequence number j, the following is guaranteed: the write w in $P1_j$ is going to be part of the $\frac{j}{k}^{th}$ merge to L_2, where k is the threshold of the number of pages in L_1.

Proof: We prove this statement by contradiction. Assume to the contrary to the defined guarantee that another page $P1_j'$ with sequence number j—that does not include w—was included in the merge to L_2. This means that the digest SC_Digest^1_j (which corresponds to $P1_j$) in the smart contract is different than the digest of the page $P1_j'$. However, during the smart contract verification of the merge proof, part of the verification is that the digest of L_1 pages used in the merge are equivalent to the ones that were written to the smart contract during stage 1 commitment; this includes SC_Digest^1_j. This means that the proof verification in the smart contract will fail, which is a contradiction, which proves the guarantee. $\square$

Theorem 3. (Stage 2 safety guarantee): For a write w that is stage 2 committed (i.e., the corresponding L_1 page $P1_i$ is stage 2 committed as part of merge number j), the following is true: any stage 2 read operation will receive the key-value pair of w if it reads from any merge starting from merge j to merge $j' - 1$ where the first write w' that overwrites w is in merge j'.

Proof: We prove this by contradiction. Assume to the contrary that a stage 2 read operation that reads from merge $j*$, where $j < j* < j'$, observes a value written by $w*$ that is different from the value written by w.[3] As part of the assumption, w is part of the state of L_2 as of merge j. Therefore, returning another write value $w*$ after merge $j*$, but before merge j' can happen in one of two ways: (1) a write $w*$ is introduced in a merge $\mathcal{J}$ between j and $j*$. This means that $w*$ is part of the L_0 pages that correspond to merge $\mathcal{J}$. This is a contradiction since we assume that the first write to overwrite w, w', is performed as part of merge j' that is after $j*$. (2) the updater returns the value of $w*$ that is not part of any merges between j and $j*$. However, to be returned and verified by the reader, the write $w*$ must be part of L_2. Being incorporated in L_2 necessitates that a zk proof was obtained for it in some merge $\mathcal{J}$ between j and $j*$. This is a contradiction since we assume that the first write to overwrite w, w', is performed as part of merge j' that is after $j*$. $\square$

Isolation Guarantee: RollStore guarantees linearizability [26] of operations that are stage 2 committed. We focus our isolation guarantee discussion on stage 2 commitment since it represents the point of final commitment and verification of operations.

Theorem 4. (Consistency of stage 2 operations): Any history H of stage 2 operations is linearizable.

Proof: A history H of read and write operations is linearizable [26] if (1) H is equivalent to some sequential history S, and (2) the partial time order $<_H$ is a subset of the equivalent sequential history order $<_S$.

---

[3] We ignore the trivial case when there are multiple writes to the same key of w in the merge j. In such a case, the most recent write—the one in the highest sequence numbered L_0 page—overwrites the others.

First, we prove the first property—H is equivalent to some sequential history S. RollStore performs stage 2 merge operations one-by-one in the order of the pages in L_1 which in turn are consolidations of ordered L_0 pages. In particular, the jth stage 2 merge operation commits the operations consolidated in pages $P1_{j*k}$ to $P1_{(j*k)+k-1}$, where k is the threshold of the number of pages in L_1. Now, we construct the equivalent sequential history S. Consider the commit point for write operations in $P1_{j*k}$ to $P1_{(j*k)+k-1}$ to be the time when the verification is performed and the proof is written on-chain. A read operation that reads a value written by w that is committed as part of the jth merge is ordered in the sequential history to be between the jth and $(j+1)$th merge. The history H is equivalent to this constructed sequential history S.

Second, we show that $<_H$ is a subset of $<_S$. This is trivial for write operations as the commit points are ordered by the smart contract so that the values committed in the jth merge precedes the values committed in the $(j+1)$th merge. For read operations, consider a read r that reads a value committed in the jth merge. Consider the following partial time order in H. The read r starts at time t_r^{start} and terminates at time t_r^{end}. The read algorithm checks the smart contract first to inquire about the most recent successful stage 2 merge in L_2. It receives the proof and digest for the jth merge. Then, the read operation is serviced from the backup node. From this timeline, we deduce the following about the partial time ordering in H: (1) $t_j^{commit} < t_r^{end}$, where t_j^{commit} is the commit time of the jth merge in the smart contract. This is true because the read observes the commit digest/proof. (2) $t_{j+1}^{commit} > t_r^{start}$. This is true because the read observed the jth merge in the smart contract, which is a point after t_r^{start}, therefore, the next merge must have happened after the start of the read operation. Therefore, r can be assigned a commit time in the history in any point between t_r^{start} and $minimum(t_r^{end}, t_{j+1}^{commit})$. This partial time ordering is part of the constructed sequential ordering in S. $\square$

IV. EVALUATION

In this section, we experimentally evaluate the performance of RollStore in comparison to two blockchain-based databases (BBDBs): BlockchainDB [16] and BigchainDB [32], as well as compare our system with an oracle-based logging system, WedgeBlock [42]. We perform our experiments by deploying off-chain nodes on Chameleon cloud machines [28]. Each machine has two 64-bit Skylake CPUs with 192 GB of RAM and 300 GB of storage. We used the Zokrates [15] framework to implement the zk-SNARK proof mechanism. The underlying blockchain network we utilize for our experiments is the Ropsten network, a widely used Ethereum test network.

Default Configuration: For each experiment, we use the following default configuration. The threshold of the mutable table T_{mut}, level L_0, and level L_1 are set to 64 writes, 7 pages, and 3 pages, respectively. The default batch size is 512. The main variables we vary are the batch size and the number of server nodes.

Benchmark: We use the Yahoo! Cloud Serving Benchmark (YCSB) to generate the workload for experiments [13]. YCSB is a key-value store benchmark that offers various workloads. In our experiments, we utilize: (1) Workload A: 50% write operations and 50% read operations, and (2) Workload C: read-only workload. We use a uniform distribution to choose access keys.

Evaluation Objectives: Our evaluation addresses:
- What are the performance characteristics of RollStore in terms of throughput, transaction cost, and latency?
- What is the impact of the batch size on performance?
- How does the performance of our system compare to other hybrid blockchain-based database systems?
- How does the performance of our system compare to that of an oracle-based system?

Metrics: The metrics we measure are:
- *Throughput:* This metric represents the throughput in terms of operations per second. We measure and report the throughput for each stage of commitment.
- *Latency:* The average latency to perform the three stages of commitments for writes and the average time to serve read requests. The latency of zk-SNARKs proof generation for Rollup-2 includes the time of setup and proof generation. Since the time of setup is much less than that of the proof generation, we show both latencies together.
- *Transaction cost:* The transaction fee cost incurred by optimistic rollups (stage 1 commit) and zk proving (stage 2 commit) in terms of dollars per thousand operations (we assume that the Ether price is $1500). Although the base gas fee may fluctuate, our experiments were conducted in close time proximity, during which we did not experience significant fluctuation in gas cost. The cost in Ether indicates the resource consumption on blockchain for methods accurately.

Comparisons: We compare the performance of our system on two widely used blockchain networks, Ethereum (permissionless) and Tendermint (permissioned). *It is worth noting that the Ethereum mainnet does not provide direct support for key-value store tasks. As a result, the throughput on the mainnet cannot be directly compared to the throughput of RollStore:* To evaluate the performance of RollStore, we select two database systems that are deployed on these two networks, namely BlockchainDB (built on Ethereum) and BigChainDB (built on Tendermint). For an oracle-based system, while WedgeBlock does not support a key-value store service, it does provide a secure logging service that closely resembles a key-value store service. Hence, the comparison between our system and Wedgeblock is justified for the purpose of performance evaluation
- *BlockchainDB [16]:* BlockchainDB is a hybrid BBDB that utilizes a blockchain layer as a storage layer and builds a database layer on top of it. We focus on the performance of read and write operations in BlockchainDB when a permissionless blockchain is used.[4] BlockchainDB stores all data on-chain. This leads to high monetary costs and latency overhead for write operations. For both reads and

---

[4]BlockchainDB is designed for permissioned settings. We make it with permissionless settings here as it is the closest BBDB that can be adapted to utilize permissionless settings. We also compare with BigChainDB while maintaining its permissioned settings.

writes, the operation is first performed on off-chain nodes (BlockchainDB-1), which is done fast, and then performed on-chain (BlockchainDB-2), which is full on-chain execution. When evaluating BlockchainDB, we utilize the Ethereum testnet network called Ropsten.

- *BigChainDB [32]:* BigChainDB is a BBDB that is implemented as a permissioned blockchain. A Byzantine agreement protocol, Tendermint, is used to implement a blockchain ledger and a database layer runs on top of this blockchain. This makes BigChainDB not suitable for DApps that require decentralization and open membership (permissionless) blockchains. However, we include it in our evaluation to understand the differences in performance characteristics compared to RollStore. Being on a permissioned blockchain, BigChainDB does not incur monetary costs. Also, because the database layer is integrated with the permissioned blockchain layer, the performance of operations is dependent on the performance of the underlying consensus mechanism.

- *WedgeBlock [42]:* WedgeBlock, a data logging platform for DApps, introduces the Lazy-Minimum Trust (LMT) concept to address high latency challenges. In LMT, the off-chain node adopts a lazy approach by asynchronously writing the log entry digest on-chain, promptly responding to user requests before the actual write occurs. This strategy is supported by a robust trust-proof and penalty mechanism, reminiscent of an oracle solution. We include WedgeBlock in our comparison to explore variations in the performance and cost of RollStore compared to an oracle-based solution.

We utilize and adapt available implementations of both BlockchainDB [16] and BigChainDB [32] that are made as a part of a study of hybrid BBDBs performance [20]. For the oracle-based system, we employ an available implementation of WedgeBlock [42].

A. Baseline Performance

In baseline experiments, we configure RollStore to have one updater node, one prover node and one backup node. The three nodes are located on three different machines and we use YCSB's workload A. The following experiments are performed while varying the batch size from 1 to 32 operations per batch (small) and from 64 to 2048 operations per batch (large).

Throughput: The left side of Fig. 7 shows the throughput results for small batch sizes. RollStore stage 0 commitment (RollStore-0) achieves the highest throughput. This is because all processes in stage 0 are performed locally and do not need to coordinate with the smart contract. Both stage 0 and stage 1 (RollStore-1) throughputs increase with the increase in the batch size. Batching amortizes the cost of committing operations. In the case of RollStore-1, when batches are bigger, this means that the number of writes to the blockchain is lower, which increases performance. Stage 2 (RollStore-2) achieves a lower throughput compared to stage 1, primarily due to the added overhead of performing a compute-intensive proof generation process. Unlike RollStore-0 and RollStore-1, the performance
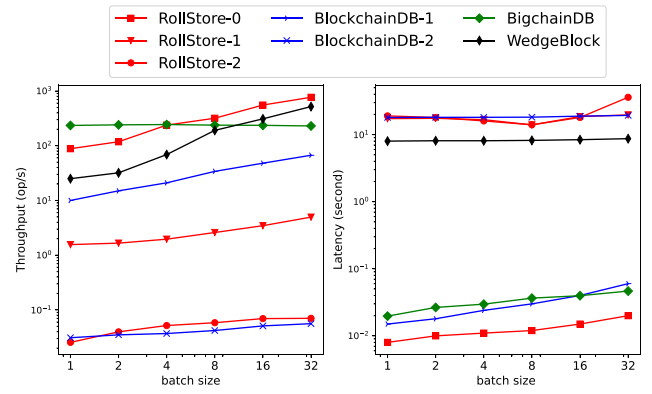

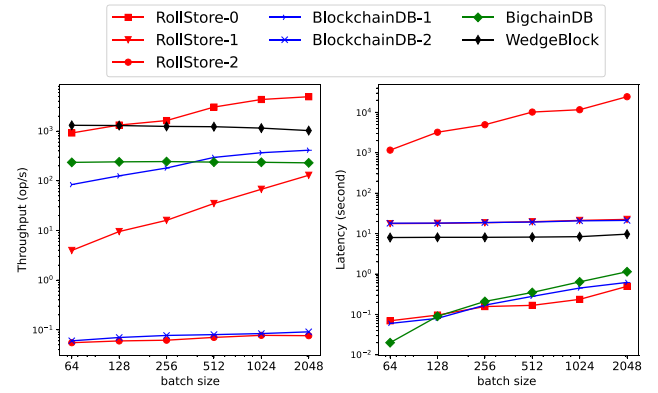Fig. 7. Throughput and latency in small batch sizes.

Fig. 8. Throughput and latency in large batch sizes.

of RollStore-2 does not significantly improve as the batch size increases. This is because the performance of RollStore-2 is primarily determined by the time required for proof generation, which becomes increasingly challenging as the batch size increases.

The local (off-chain) throughput of BlockchainDB (BlockchainDB-1) achieves a higher performance compared to RollStore-1 because it does not require interacting with the blockchain. However, when it comes to committing operations on-chain in Ethereum, BlockchainDB-2 exhibits poorer performance compared to RollStore-1 and even underperforms RollStore-2. This is because BlockchainDB-2 writes raw data on-chain which increases the overhead of interacting with blockchain. BigChainDB performance is between RollStore-0 and RollStore-1. This is because it does not utilize permissionless blockchain, which means that it does not suffer from the high overhead associated with it. However, BigChainDB incurs overhead from the underlying permissioned blockchain, Tendermint, and the consensus mechanism, leading to worse performance than RollStore-0. WedgeBlock outperformed BigChainDB as the batch size increased, but it lagged behind RollStore-0. However, the gap between these two systems narrowed when the batch size reached 32 operations per batch.

The left side of Fig. 8 displays the throughput results for large batch sizes. The findings are similar to those for small sizes: our system achieved the best performance in both the off-chain process (RollStore-0) and on-chain process (RollStore-1). Furthermore, it demonstrated improved performance compared to its results with small sizes. The performance trend remains consistent, particularly in Stage 1 (RollStore-1), where throughput increases rapidly as batch sizes become larger. While Wedge-Block achieved better performance than our system in specific batch sizes (64 and 128), our off-chain process (RollStore-0) outperforms WedgeBlock as batch sizes become larger.

Latency: The right part of Fig. 7 shows the latency results for small batch sizes. The latency of off-chain operations—RollStore-0, BlockchainDB-1, and BigChainDB—are the lowest as they do not need to write to a permissionless blockchain smart contract. The latency of RollStore-1, BlockchainDB-2, and WedgeBlock—both requiring a write to the smart contract—is similar at around 20 seconds, which is proportional to the time to write to the smart contract. Although the compute-intensive proof generation process in RollStore-2 gradually increases its latency, it outperforms Ethereum (BlockchainDB-2) in certain batch sizes where the advantage of batching outweighs the time required for proof generation.

The right part of Fig. 8 shows the latency results for large batch sizes. The latency of RollStore-2 increases very rapidly as it requires more time for proof generation in larger batches. The latency of off-chain operations also increases as more processing time is needed for larger batches. However, it's worth noting that the latency of RollStore-2 can be reduced by adding more server nodes, as discussed in Section III-C. We will introduce the scalability performance in Section IV-B. In the case of WedgeBlock, it demonstrated lower latency compared to our on-chain process (RollStore-1), although it lagged behind our off-chain process. This can be attributed to its reliance on an oracle-based design, which depends on security guarantees provided by the oracle design. While this design effectively reduces communication latency between the off-chain and on-chain elements, it still results in higher latency compared to our dedicated off-chain process and sacrifices the security guarantees provided by the blockchain mainnet.

Transaction cost: Fig. 9 shows the monetary cost results. In RollStore-1, each batch requires sending one transaction only—that writes a simple set of digests—to the blockchain. Therefore, the transaction cost in stage 1 will decrease when the batch size becomes larger. This is not the case in stage 2. Since we need to send the proof parameters to the blockchain, the size of these parameters also increases with the increase in the batch size; this increases the cost. For this reason, the transaction cost per thousand operations in stage 2 does not change significantly when the batch size becomes larger. For Ethereum (BlockchainDB-2), the monetary cost is the largest (around $122 per 1000 operations). This is because raw operations are written on-chain, unlike RollStore that only writes digests and verifies proofs. The cost in WedgeBlock also decreased with the increase in batch size. Both stage 1 (RollStore-1) and stage 2 (RollStore-2) can reduce the cost of interacting with the permissionless blockchain. (RollStore-0, BlockchainDB-1, and BigChainDB
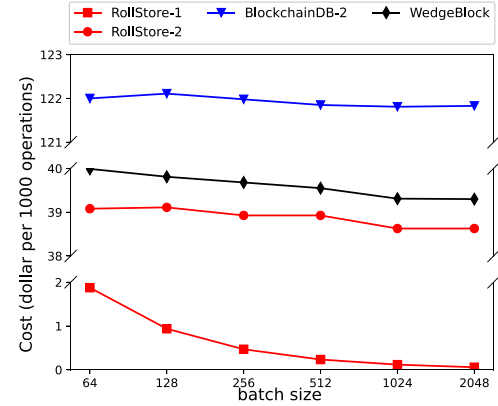

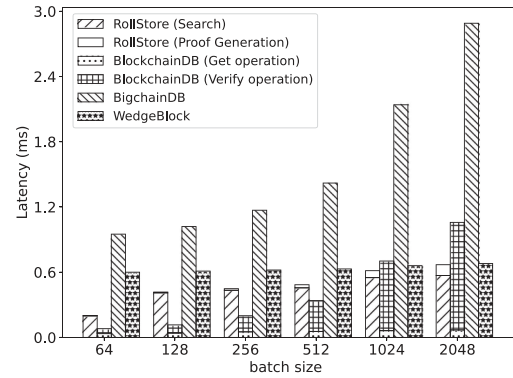Fig. 9.   Cost in different batch sizes.

Fig. 10.   Read latency in different batch sizes.

are not included in the figure as they do not utilize permissionless blockchain that requires fees).

Read latency: Fig. 10 shows the read latency while varying the batch sizes. The average read latency of RollStore becomes larger when increasing the batch size of reading requests. As batch size increases, the backup node requires more time to search and generate the related proofs. The read latency in BlockchainDB is higher than RollStore when the batch size is larger than 512; we attribute this to the *Verify* operation in BlockchainDB. This operation spends more time to verify the read result when the batch size becomes larger. The read latency in BigchainDB is the longest and becomes longer when the batch size increases; this is because—although it does not need to perform a consensus round for reads—BigchainDB needs to build the block to record the read request; this process increases the read latency. The read latency in WedgeBlock increases slightly, and the batch size does not significantly affect the read latency. This is because it processes these reads locally without interacting with on-chain nodes.

B. Scalability Performance

In this section, we present a set of experiments to test the scalability performance of our system. In this configuration, multiple updater nodes, multiple prover nodes, and multiple backup nodes are located on three different machines. Each
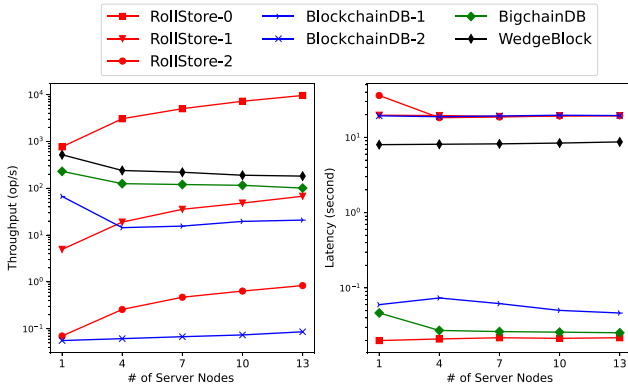
Fig. 11.   Throughput and latency in multiple server nodes.

Fig. 12.   Read throughput in multiple server nodes.

machine contains multiple instances of a type of node. We evaluate scalability by changing the number of server nodes and fixing the batch size of requests at 32. We vary the number of server nodes from 4 to 13.

Throughput: This set of experiments focused on measuring the throughput, as depicted on the left side of Fig. 11. As the number of server nodes increased, the throughput of all three stages increased. This is because more updater and backup nodes were able to work in parallel, resulting in higher throughput. This observation highlights that even smaller batch sizes can achieve higher throughput by adding more server nodes, which also leads to reduced latency in stage 2. The throughput of stage 2 commitment (RollStore-2) increased by a factor of 11.9X when the number of server nodes was increased from 1 to 13. This is due to the leveraging of computation resources from multiple prover nodes to accelerate the proof generation process (see Section III-C). Compared to the baseline performance, the throughput of stage 2 in the scalability configuration was much higher than that of Ethereum (BlockchainDB-2). While this throughput was not as high as that of stage 0 and stage 1, it still played a significant role in reducing the waiting time for verifying the results of stage 1 commitment. Specifically, the waiting time was reduced from several days to hours, highlighting the importance of stage 2 in the overall performance of the system. (see Section II-C1). The throughput of WedgeBlock decreases when adding server nodes up to 4 nodes, then fluctuates slightly with the addition of more nodes.

Another observation is that the local (off-chain) throughput, as seen in BlockchainDB-1 and BigchainDB, decreases when adding more server nodes compared to a single server node. We attribute this to the cost of the underlying consensus mechanism, where a larger number of nodes causes the overhead of coordination to increase.

Latency: As shown in the right side of Fig. 11, the reduction of latency in RollStore-0, RollStore-1, and WedgeBlock is not significant. This is because the latency is mainly determined by the updater node processing for RollStore-0 and the blockchain confirmation time for RollStore-1 and WedgeBlock. The latency of RollStore-2 is significantly reduced because multiple prover nodes work in parallel to generate the proof. It is important to note that although the latency of RollStore-2 is significantly
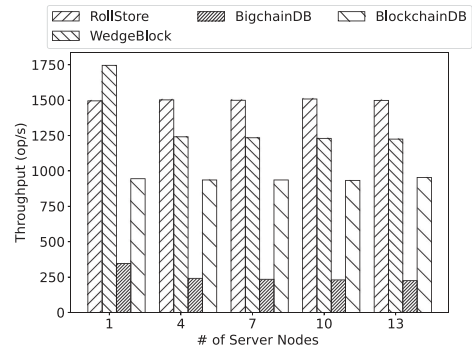
reduced due to multiple prover nodes working in parallel to generate the proof of one task, this reduction is limited. It can only bring the latency to a slightly higher level than the blockchain confirmation time. Nevertheless, RollStore benefits from batching and can achieve higher throughput and lower cost.

The addition of more server nodes does not significantly benefit BlockchainDB and BigchainDB due to coordination overhead in their consensus mechanism.

Transaction cost: Since the content of transactions and smart contracts do not change when we add multiple server nodes, the change of transaction fee (Ether cost) is negligible and is only due to the fluctuation of gas fees.

Read throughput: Fig. 12 shows the read throughput while varying the number of server nodes. RollStore and BlockchainDB are not impacted by the increase in the number of server nodes. This is because the throughput is determined by the overhead of assembling the read responses and verifying reads. BlockchainDB achieves lower performance than RollStore due to its verification step that takes more latency than RollStore for large batch sizes. WedgeBlock achieves better performance than RollStore in a single-server configuration; however, its performance worsens when additional nodes are added. BigchainDB achieves the lowest throughput due to the added overhead to synchronize the response to the read operations. As the number of nodes increases, this overhead increases and lowers the throughput of BigChainDB.

V. Discussion and Future Work

Investigating Cross-Chain Interoperability: RollStore currently focuses on providing a secure and efficient data indexing solution for hybrid onchain-offchain DApps within a single blockchain ecosystem, specifically Ethereum. Future work could explore the challenges and opportunities of enabling cross-chain interoperability. This would involve developing mechanisms to index and manage data across multiple blockchain networks, allowing DApps to leverage the strengths of different blockchains while maintaining consistency and security. Addressing cross-chain data management could significantly expand the applicability and flexibility of RollStore in the rapidly evolving blockchain landscape.

Enhancing Flexibility in Security and Performance Trade-offs: RollStore currently uses a three-stage commit protocol that balances security and performance. However, the implementation does not allow users to actively control the degree of security based on their specific needs. For example, users might want to partially guarantee security for some higher-priority operations while accepting lower security for less critical ones. Exploring this flexibility could lead to broader usage of RollStore in various DApps, enabling them to customize the balance between security and performance according to their unique requirements.

VI. Conclusion

The authors propose RollStore, a data indexing solution for hybrid onchain-offchain DApps. RollStore builds on advances in blockchain scaling solutions such as rollups, as well as indexing and authenticated data structures. The outcome is a three-stage commit protocol that allows balancing the trade-off between security and performance for hybrid blockchain methods. Our evaluations demonstrate the advantages of RollStore in terms of cost and performance while comparing with two blockchain-based databases, BlockchainDB and BigChainDB.

References

[1] J. Adler and M. Quintyne-Collins, "Building scalable decentralized payment systems," 2019, *arXiv: 1904.06441*.

[2] H. Al-Breiki, M. H. U. Rehman, K. Salah, and D. Svetinovic, "Trustworthy blockchain oracles: Review, comparison, and open research challenges," *IEEE Access*, vol. 8, pp. 85675–85685, 2020.

[3] A. Alkhateeb, C. Catal, G. Kar, and A. Mishra, "Hybrid blockchain platforms for the Internet of Things (IoT): A systematic literature review," *Sensors*, vol. 22, no. 4, 2022, Art. no. 1304.

[4] L. Allen et al., "Veritas: Shared verifiable databases and tables in the cloud," in *Proc. 9th Biennial Conf. Innov. Data Syst. Res.*, 2019, pp. 1–9.

[5] M. J. Amiri, D. Agrawal, and A. El Abbadi, "Sharper: Sharding permissioned blockchains over network clusters," in *Proc. Int. Conf. Manage. Data*, 2021, pp. 76–88.

[6] A. M. Antonopoulos and G. Wood, *Mastering Ethereum: Building Smart Contracts and Dapps*. Sebastopol, CA, USA: O'Reilly Media, 2018.

[7] M. Belotti, N. Božić, G. Pujolle, and S. Secci, "A vademecum on blockchain technologies: When, which, and how," *IEEE Commun. Surv. Tut.*, vol. 21, no. 4, pp. 3796–3838, Fourth Quarter, 2019.

[8] P. Biel, S. Zhang, and H.-A. Jacobsen, "A zero-knowledge proof system for openlibra," in *Proc. 22nd Int. Middleware Conf. Demos Posters*, 2021, pp. 3–4.

[9] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li, "Paxos replicated state machines as the basis of a high-performance data store," in *Proc. 8th USENIX Symp. Netw. Syst. Des. Implementation*, 2011, pp. 141–154.

[10] Z. Cao, S. Dong, S. Vemuri, and D. H. Du, "Characterizing, modeling, and benchmarking {RocksDB }{ Key-Value} workloads at facebook," in *Proc. 18th USENIX Conf. File Storage Technol.*, 2020, pp. 209–223.

[11] Y. Chen and C. Bellavitis, "Blockchain disruption and decentralized finance: The rise of decentralized business models," *J. Bus. Venturing Insights*, vol. 13, 2020, Art. no. e00151.

[12] D. Company, "Dapp radar rankings," 2022. [Online]. Available: https://dappradar.com/rankings

[13] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. 1st ACM Symp. Cloud Comput.*, 2010, pp. 143–154.

[14] H. Desai, M. Kantarcioglu, and L. Kagal, "A hybrid blockchain architecture for privacy-enabled and accountable auctions," in *Proc. IEEE Int. Conf. Blockchain (Blockchain)*, 2019, pp. 34–43.

[15] J. Eberhardt and S. Tai, "Zokrates-scalable privacy-preserving off-chain computations," in *Proc. IEEE Int. Conf. Internet Things IEEE Green Comput. Commun. IEEE Cyber Phys. Social Comput. IEEE Smart Data*, 2018, pp. 1084–1091.

[16] M. El-Hindi, C. Binnig, A. Arasu, D. Kossmann, and R. Ramamurthy, "Blockchaindb: A shared database on blockchains," in *Proc. VLDB Endowment*, vol. 12, no. 11, pp. 1597–1609, 2019.

[17] U. Feige, A. Fiat, and A. Shamir, "Zero-knowledge proofs of identity," *J. Cryptol.*, vol. 1, no. 2, pp. 77–94, 1988.

[18] D. Foundation, "decentraland," 2020. [Online]. Available: https://decentraland.org/

[19] A. Garoffolo, D. Kaidalov, and R. Oliynykov, "Zendoo: A ZK-Snark verifiable cross-chain transfer protocol enabling decoupled and decentralized sidechains," in *Proc. IEEE 40th Int. Conf. Distrib. Comput. Syst.*, 2020, pp. 1257–1262.

[20] Z. Ge, D. Loghin, B. C. Ooi, P. Ruan, and T. Wang, "Hybrid blockchain database systems: Design and performance," in *Proc. VLDB Endowment*, vol. 15, no. 5, pp. 1092–1104, 2022.

[21] O. Goldreich and H. Krawczyk, "On the composition of zero-knowledge proof systems," in *Proc. Int. Colloq. Automata Lang. Program.*, Springer, 1990, pp. 268–282.

[22] O. Goldreich and Y. Oren, "Definitions and properties of zero-knowledge proof systems," *J. Cryptol.*, vol. 7, no. 1, pp. 1–32, 1994.

[23] L. Gudgeon, P. Moreno-Sanchez, S. Roos, P. McCorry, and A. Gervais, "SoK: Off the chain transactions," *IACR Cryptol. ePrint Arch.*, vol. 2019, 2019, Art. no. 360.

[24] H. Guo and X. Yu, "A survey on blockchain technology and its security," *Blockchain Res. Appl.*, vol. 3, no. 2, 2022, Art. no. 100067.

[25] S. Gupta, S. Rahnama, J. Hellings, and M. Sadoghi, "ResilientDB: Global scale resilient blockchain fabric," in *Proc. VLDB Endowment*, vol. 13, pp. 868–883, 2020.

[26] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, 1990.

[27] R. Jain and S. Prabhakar, "Trustworthy data from untrusted databases," in *Proc. IEEE 29th Int. Conf. Data Eng.*, 2013, pp. 529–540.

[28] K. Keahey et al., "Lessons learned from the chameleon testbed," in *Proc. USENIX Annu. Tech. Conf.*, 2020, pp. 219–233.

[29] R. Kumar, N. Marchang, and R. Tripathi, "Distributed off-chain storage of patient diagnostic reports in healthcare system using IPFs and blockchain," in *Proc. IEEE Int. Conf. Commun. Syst. Netw.*, 2020, pp. 1–5.

[30] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, 1982.

[31] C. Luo and M. J. Carey, "LSM-based storage techniques: A survey," *VLDB J.*, vol. 29, no. 1, pp. 393–418, 2020.

[32] T. McConaghy et al., "BigchainDB: A scalable blockchain database," *White Paper BigChainDB*, 2016.

[33] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Proc. Conf. Theory Appl. Cryptographic Techn.*, 1987, pp. 369–378.

[34] A. Miller, I. Bentov, S. Bakshi, R. Kumaresan, and P. McCorry, "Sprites and state channels: Payment networks that go faster than lightning," in *Proc. Int. Conf. Financial Cryptogr. Data Secur.*, Springer, 2019, pp. 508–526.

[35] M. Naor and M. Yung, "Universal one-way hash functions and their cryptographic applications," in *Proc. 21st Annu. ACM Symp. Theory Comput.*, 1989, pp. 33–43.

[36] S. Nathan et al., "Blockchain meets database: Design and implementation of a blockchain relational database," 2019, *arXiv: 1903.01919*.

[37] Q. Pei, E. Zhou, Y. Xiao, D. Zhang, and D. Zhao, "An efficient query scheme for hybrid storage blockchains based on merkle semantic trie," in *Proc. Int. Symp. Reliable Distrib. Syst.*, 2020, pp. 51–60.

[38] Y. Peng, M. Du, F. Li, R. Cheng, and D. Song, "Falcondb: Blockchain-based collaborative database," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2020, pp. 637–652.

[39] C. Pop, T. Cioara, I. Anghel, M. Antal, and I. Salomie, "Blockchain based decentralized applications: Technology review and development guidelines," 2020, *arXiv: 2003.07131*.

[40] F. M. Schuhknecht, A. Sharma, J. Dittrich, and D. Agrawal, "Chainifydb: How to blockchainify any data management system," 2019, *arXiv: 1912.04820*.

[41] C. Sguanci, R. Spatafora, and A. M. Vergani, "Layer 2 blockchain scaling: A survey," 2021, *arXiv:2107.10881*.

[42] A. Singh, Y. Zhou, S. Mehrotra, M. Sadoghi, S. Sharma, and F. Nawab, "WedgeBlock: An off-chain secure logging platform for blockchain applications," *Adv. Database Technol.*, vol. 26, pp. 526–539, 2023.

[43] M. Tan, "Ethereum charts and statistics," 2015. [Online]. Available: https://etherscan.io/charts

[44] P. J. Taylor, T. Dargahi, A. Dehghantanha, R. M. Parizi, and K.-K. R. Choo, "A systematic literature review of blockchain cyber security," *Digit. Commun. Netw.*, vol. 6, no. 2, pp. 147–156, 2020.

[45] P. Todd, Making UTXO set growth irrelevant with low-latency delayed TXO commitments, 2016.

[46] H. Wang, C. Xu, C. Zhang, J. Xu, Z. Peng, and J. Pei, "VChain+: Optimizing verifiable blockchain boolean range queries," in *Proc. IEEE 38th Int. Conf. Data Eng.*, 2022, pp. 1927–1940.

[47] Y. Wang, Z. Tu, Y. Bai, H. Yuan, X. Xu, and Z. Wang, "A blockchain-based infrastructure for distributed Internet of Services," in *Proc. IEEE World Congr. Serv.*, 2021, pp. 108–114.

[48] Q. Wei, B. Li, W. Chang, Z. Jia, Z. Shen, and Z. Shao, "A survey of blockchain data management systems," *ACM Trans. Embedded Comput. Syst.*, vol. 21, no. 3, pp. 1–28, 2022.

[49] H. Wu et al., "Data management in supply chain using blockchain: Challenges and a case study," in *Proc. IEEE 28th Int. Conf. Comput. Commun. Netw.*, 2019, pp. 1–8.

[50] H. Wu, W. Zheng, A. Chiesa, R. A. Popa, and I. Stoica, "{DIZK}: A distributed zero knowledge proof system," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 675–692.

[51] K. Wu, "An empirical study of blockchain-based decentralized applications," 2019, *arXiv: 1902.04969*.

[52] C. Xu, C. Zhang, J. Xu, and J. Pei, "Slimchain: Scaling blockchain transactions through off-chain storage and parallel processing," in *Proc. VLDB Endowment*, vol. 14, no. 11, pp. 2314–2326, 2021.

[53] K. Yang, P. Sarkar, C. Weng, and X. Wang, "Quicksilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2021, pp. 2986–3001.

[54] C. Zhang, C. Xu, J. Xu, Y. Tang, and B. Choi, "Gem^2-tree: A gas-efficient structure for authenticated range queries in blockchain," in *Proc. IEEE 35th Int. Conf. Data Eng.*, 2019, pp. 842–853.

[55] Q. Zhang, Y. He, R. Lai, Z. Hou, and G. Zhao, "A survey on the efficiency, reliability, and security of data query in blockchain systems," *Future Gener. Comput. Syst.*, vol. 145, pp. 303–320, 2023.

[56] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou, "VSQL: Verifying arbitrary SQL queries over dynamic outsourced databases," in *Proc. IEEE Symp. Secur. Privacy*, 2017, pp. 863–880.

[57] Q. Zhou, H. Huang, Z. Zheng, and J. Bian, "Solutions to scalability of blockchain: A survey," *IEEE Access*, vol. 8, pp. 16440–16455, 2020.

[58] W. Zhou, Y. Cai, Y. Peng, S. Wang, K. Ma, and F. Li, "VeriDB: An SGX-based verifiable database," in *Proc. Int. Conf. Manage. Data*, 2021, pp. 2182–2194.

Qi Lin received the master's degree in computer science from the University of California, Irvine. He is currently working towards the PhD degree in computer science and engineering with Arizona State University. His research interests lie in the areas of zero-knowledge proofs, blockchain, and query compilation.
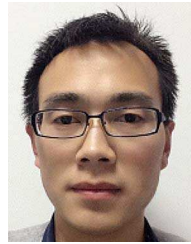
Binbin Gu is currently working towards the PhD degree in computer science and engineering with the University of California, Irvine. His research interests include machine learning, blockchain, zero-knowledge proofs, and Natural Language Processing (NLP). He has published several papers in *IEEE Transactions on Knowledge and Data Engineering*, ICDE, EDBT, DASFAA, etc.

Faisal Nawab is an assistant professor with the University of California, Irvine (UCI). He leads EdgeLab which tackles research problems in the intersection of data management and distributed systems with a focus on decentralized and Internet of Things (IoT) applications. He has published papers in VLDB, SIGMOD, ICDE, EDBT, *IEEE Internet of Things Journal*, and other data management and systems conferences and journals.