One-Phase Batch Update on Sparse Merkle Trees for Rollups

Boqian Ma, Vir Nath Pathak, Lanping Liu, and Sushmita Ruj(✉)

School of Computer Science and Engineering, University of New South Wales,
Kensington, NSW 2052, Australia
{boqian.ma,vir.pathak,sushmita.ruj}@unsw.edu.au,
lanping.liu@unswalumni.com

Abstract. A sparse Merkle tree is a Merkle tree with fixed height and indexed leaves given by a map from indices to leaf values. It allows for both efficient membership and non-membership proofs. It has been widely used as an authenticated data structure in various applications, such as layer-2 rollups for blockchains. zkSync Lite, a popular Ethereum layer-2 rollup solution, uses a sparse Merkle tree to represent the state of the layer-2 blockchain. The account information is recorded in the leaves of the tree. In this paper, we study the sparse Merkle tree algorithms presented in zkSync Lite, and propose an efficient batch update algorithm to calculate a new root hash given a list of account (leaf) operations. Using the construction in zkSync Lite as a benchmark, our algorithm 1) improves the account update time from $\mathcal{O}(\log n)$ to $\mathcal{O}(1)$ and 2) reduces the batch update cost by half using a one-pass traversal. Empirical analysis of real-world block data shows that our algorithm outperforms the benchmark by at most 14%.

Keywords: Blockchain Scalability · Sparse Merkle Trees · Rollups · Layer-2

1 Introduction

Recent advances in distributed ledger technology have introduced a new paradigm of applications called "decentralisation applications" (DApps) with new use cases in areas such as finance [7,20], logistics [29], and Internet-of-Things [26]. However, the increasing number of users and transactions on DApps has also exposed the key limitation of the scalability of their underlying public blockchain infrastructures [17]. Two of the largest public blockchians by market capitalisation[1], Bitcoin [24] and Ethereum [32], can only process 7 and 29 transactions per second (TPS), which is far from their centralised payment provider counterpart, Visa, which claims to have the capacity to process 65,000 TPS [30].

There are many ways to improve blockchain scalability. They can be broadly grouped into two categories: on-chain and off-chain. On-chain research involves

[1] https://coinmarketcap.com/ accessed on 23rd of August 2023.

© The Author(s), under exclusive license to Springer Nature Singapore Pte Ltd. 2024
N. Dong et al. (Eds.): SDLT 2023, CCIS 1975, pp. 1–21, 2024.
https://doi.org/10.1007/978-981-97-0006-6_1

changing the underlying blockchain infrastructure to achieve better scalability. Examples of on-chain research efforts include developing efficient consensus algorithms [18,28], sharding [23,33], and changing block configurations [13]. On the other hand, off-chain research efforts involve changing how we interact with the blockchain (L1). Instead of performing all activities on-chain, we offload the computation- and storage-intensive activities off-chain. Some existing solutions include State Channels [25], Plasma [27], and rollups [11]. These scaling solutions are known as “Layer-2” (L2) solutions.

The recent developments of L2 rollups such as zkSync Lite [21], Aztec Network [9], Loopering [22], and Immutable X [1] has shown prominent results toward increasing transaction throughput on Ethereum. Rollups execute transactions off-chain and bundle the results of many L2 transactions into one L1 transaction. L1 cannot interpret L2 data, it only acts as a *data availability layer* for L2 activity. Such techniques provide a reduction in computation to L1, while also massively decreasing the transaction fees as one L1 transaction fee is shared amongst all transactions bundled within it.

zkSync Lite [21], a widely used and well-documented zero-knowledge rollup technique, has achieved a maximum observed TPS of 110 [2], making it almost 6 times faster than Ethereum. Following the success of rollups, Ethereum has introduced a rollup-centric roadmap [12] specifically directing future scaling efforts on Ethereum to maximise the use of L2 rollups.

In an L2 rollup, there are generally *operators* keeping the L2 state, processing L2 transactions and communicating with L1 through a smart contract. *Users* have *accounts* and *balances* of tokens. L2 users submit signed transactions to the operators, who then collect those transactions and form L2 blocks.

Sparse Merkle trees (SMT) are widely used as authenticated data structures to keep state information in rollups because of their simplicity and effectiveness. The leaves of SMTs represent account-related information, such as balances and nonce. The root hash of SMTs is a succinct representation of the state of all account balances. Given a block of L2 transactions, the operators will calculate a new root hash based on the result of these transactions. Generally, the process of finding the root hash involves two parts: first, the account leaves need to be updated. Then, the new root hash is calculated by updating the paths from the updated leaves to the root.

The current implementation of this in zkSync Lite is to first go through the transactions in a block sequentially to update the leaves individually and then calculate the root hash. This solution involves traversing the SMTs twice for every updated leaf, which is inefficient. We denote this as a two-phase algorithm.

To build on the above solution, this paper introduces the notion of `BatchUpdate` on SMTs. The action of *batching* is defined as processing transactions in a block all at once instead of individually. All accounts involved in transactions in a block are updated together in a batch. Instead of traversing the SMTs twice, we propose a new algorithm to update the leaves and intermediate hashes at the same time by traversing the SMTs only once. We name this approach the one-phase batch update (OBU).

Our Contributions

1. We introduce an efficient SMT leaf update algorithm, `SMT.UpdateLeaf`, that improves account update time from $\mathcal{O}(\log n)$ to $\mathcal{O}(1)$.
2. Building on this, an SMT batch update algorithm, `SMT.BatchUpate`, is proposed to calculate the root hash of an SMT, reducing the total number of traversals by 50% from $\mathcal{O}(k \log n) + \mathcal{O}(k \log n)H$ to $\mathcal{O}(k \log n)H$, where k is the number of updates in a batch, n is the total number of leaves in the SMT, and H is a hash operation.[2]
3. Performance analysis of our proposed algorithm was conducted using both micro- and macro-benchmarks in single and multi-threaded scenarios.
4. In real-world macro-benchmark data, our algorithm outperformed the benchmark by up to 14%.

Organisation. The rest of the paper is organised as follows. Section 2 introduces the preliminary information. Next, Sect. 3 discusses some related work. In Sect. 4, we introduce the batch update algorithm. Section 6 outlines our experimental results, followed by the conclusion and discussion in Sect. 7.

2 Preliminaries

2.1 Leaf Operation

Definition 1 (Leaf Operation). *Given a Merkle tree (MT), T, with n leaf nodes $L = \{\text{leaf}_0, \cdots, \text{leaf}_{n-1}\}$ and their corresponding data items $D = \{d_0, \cdots, d_{n-1}\}$ where $\text{leaf}_j = H(d_j)$, a leaf operation $o^j \in \{$`InsertLeaf, UpdateLeaf, RemoveLeaf`$\}$ where $0 \leq j < |D|$, is a function that modifies the value of leaf_j. `InsertLeaf` inserts a new leaf, given by $\text{leaf}_j = H(d_j)$, into the tree, `UpdateLeaf` updates the value of leaf_j, and `RemoveLeaf` removes leaf_j and d_j from the tree and D respectively.*

2.2 Sparse Merkle Tree

Definition 2 (Sparse Merkle Tree). *An SMT is an MT with a fixed depth of N, and indexed leaves. Data items $D = \{d_0, \cdots, d_{n-1}\}$ where $n < 2^N$ are stored in a map, $M : \{0,1\}^{2^N} \rightarrow D$ mapping from leaf indices to data items. An SMT is defined by the following set of algorithms on M:*

1. *`Gen(N)` $\rightarrow$ `SMT`: Algorithm that generates an empty SMT given a depth N.*
2. *`SMT.Commit(M)` $\rightarrow$ `R'`. Deterministic algorithm that inserts every key-value pair in M into the tree and returns the new root hash.*
3. *`SMT.ApplyOp(`o^i`)` $\rightarrow$ `R'` Deterministic algorithm that applies the leaf operation o^i and returns a new root hash R'. `SMT.ApplyOp(`o^i`)` can be further categorised into three methods depending on the operation type. They are `SMT.InsertLeaf(`o^i`)`, `SMT.UpdateLeaf(`o^i`)`, and `SMT.RemoveLeaf(`o^i`)`. A description of each of these operations can be found in Sect. 2.1.*

[2] Code at: https://github.com/Boqian-Ma/one-phase-batch-update-SMT.

4. *`SMT.MemberWitnessCreate(i)` $\rightarrow w_i$: Deterministic algorithm that returns the Merkle proof of $M(i)$ consisting of a list of siblings nodes from* leaf_i *to the root.*
5. *`SMT.MemberVerify(`w_i`,` d_i`)` $\rightarrow \{true, false\}$: Deterministic algorithm that verifies whether d_i is a member of M.*

SMTs have the same membership-proof construction as regular Merkle trees. However, proving non-membership is more efficient on SMTs than on Merkle trees, since a non-membership for a key k in an SMT is the membership proof of the default value.

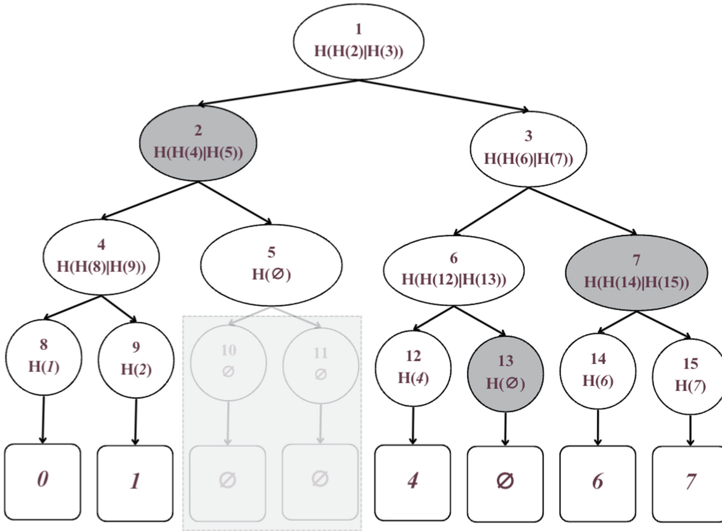

Fig. 1. A SMT of 3 levels. The ovals represent internal nodes. The squares represents its value mapping M, where the numbers are the keys of M and the leaf indices. The default value is represented as $\emptyset$. The highlighted nodes form leaf_4's membership proof. Since leaf_2 and leaf_3 are empty, everything below their highest common parent, node_5, are pruned to increase storage efficiency.

Space Optimisation. Instead of storing the full SMT of $2^{N+1} - 1$ nodes, Bauer [10] presents a memory efficient way of storing an SMT by pruning empty sub-trees. Referring to node 5, Fig. 1, following Bauer's proposal, the subtree of node 5 is replaced with the default hash. As such, the space can be greatly reduced.

3 Related Work

This section introduces zkSync Lite [21] and its relevant SMT root hash update algorithm, which we use as our benchmark.

ZkSync Lite. zkSync Lite [21] is an L2 rollup solution developed by Matter Labs [3]. It supports simple transaction types including transfer or swap of ERC-20 [31] tokens, and ERC-721 [15] token minting. Like most L2 solutions, zkSync Lite has two main components: on-chain and off-chain. The on-chain component includes several Solidity Smart Contracts deployed[3] on Ethereum L1. The off-chain component includes several micro-services that facilitate L2 transaction executions and SNARK [16] generation. Detailed descriptions of the zkSync Lite design are given in the Appendix A.1

Account Tree Construction. SMTs are used in three places in zkSync Lite: account tree[4], circuit account tree, and balance tree. The account tree is the main data structure that keeps track of the account balances of its users. The circuit account tree and the balance tree are derived from the account tree and are used to build zero-knowledge block proofs. Here, we give descriptions of the account tree in zkSync Lite.

The account tree is an SMT of depth $N = 24$. As such, it can store up to $2^{25} - 1$ accounts. The accounts are stored in a map M, mapping from leaf indices to accounts. Each internal node, node_j, where $1 \leq j \leq 2^{N+1}$, node_j's direct children are node_j's children are given by node_{2j} and node_{2j+1} and $\text{node}_j = H(\text{node}_{2j} \| \text{node}_{2j+1})$. node_j is also known as node_{2j} and node_{2j+1}'s parent node. The root of the tree is node_1, which also corresponds to the digest of T.

Each leaf node leaf_k where $0 \leq k < 2^N$, corresponds to a key k and is labelled with the value associated with that key if it exists or the hash of a default value otherwise. Formally, if $v = M(k)$ exists, $\text{leaf}_k = v$, else $\text{leaf}_k = \textit{default}$, where *default* is a predefined default value.

On the Nth level of SMT (i.e. the leaf level), given by the set of 2^N nodes $\{\text{node}_q\}^{2^N}$ where $2^N \leq q < 2^{N+1}$, each node_j corresponds to a key $k = (1 << N) + q$ and is labelled with the hash of the value associated with that key if it exists, or the hash of a default value otherwise. Formally, if $v = M(k)$ exists, $\text{node}_q = H(v)$, otherwise $\text{node}_q = H(\textit{default})$.

For simplicity, we denote the nodes at the leaf level by $L = \{\text{leaf}_0, \cdots, \text{leaf}_k\}$ where $0 \leq k < 2^N$.

Root Hash Update Algorithm. Here we outline the root hash update algorithm implemented in zkSync Lite given a list of leaf operations. This algorithm is divided into two phases. Consider an account tree T and a list of k operations $O = \{o^j\}^k_{j \in [0, 2^N)}$. The first phase updates the leaves to their new values. For each operation $o^j \in O$, the algorithm traverses T from the root to leaf_j and performs the operation. For example, if o^j was an `update` balance operation, then the balance of leaf_j is updated accordingly. At the end of this phase, all accounts affected by O are updated. Note that when a leaf is updated to a new

[3] https://etherscan.io/address/0xaBEA9132b05A70803a4E85094fD0e1800777fBEF.

[4] https://github.com/matter-labs/zksync/blob/master/core/lib/types/src/lib.rs#L84.

value, all nodes in its parent path need to be recomputed. This phase does not concern the hash calculation and takes $\mathcal{O}(k \log n)$ running time to perform k updates.

The second phase re-computes the hashes of affected paths and returns the new root hash. To compute the root hash, the algorithm traverses left and right recursively from T's root to retrieve or compute the child hashes. Recursion terminates when 1) an updated leaf is reached or 2) when all the child leaves of the current nodes are unchanged from the first phase. In case 1), the leaf hash is calculated and returned. In case 2), the current node hash is returned. As a result of this recursive algorithm, the new root hash is calculated. This phase takes $\mathcal{O}(k \log n)H$ running time, where H is the running time of the chosen hash function. Together, the root hash calculation process takes $\mathcal{O}(k \log n) + \mathcal{O}(k \log n)H$.

The first phase occurs in the block producer module, while the second phase occurs in the root hash calculator module. In the actual implementation, these two phases are completed in two separate micro-services. The first phase occurs in the "block producer" module, where the leaves are updates. Then, the second phase happens in the "root hash calculator" module, where the new root hash is computed. This separation takes the hash calculation computation overhead away from the main service.

Inefficiencies. Above we described a two-phase algorithm implemented in zkSync Lite to update the root state of the account tree given a list of k leaf operations. As stated in the zkSync Lite code base[5], there exists a bottleneck that constrains the speed of the block producer producing blocks. If the block producer's speed exceeds the speed of root hash calculation, then the job queue for the root hash calculator will increase indefinitely. Furthermore, we observe that for each operation $o^j \in O$, the path between the updated leaf_j and root is traversed twice. The first traversal occurs when updating the account values and the second time occurs when calculating the root hash.

4 One-Phase Batch Update on Sparse Merkle Tree

In this section, we first outline the basic functionalities of the three leaf operations, `SMT.InsertLeaf`, `SMT.UpdateLeaf` and `SMT.RemoveLeaf`. Then, we introduce a more efficient algorithm, `SMT.BatchUpdate(O)`$\rightarrow R'$ that takes in a list of operations and returns the new SMT root. The pseudocode is outlined in Algorithm 1.

4.1 Leaf Operation Algorithms

1. `SMT.InsertLeaf`(leaf_j) is a deterministic algorithm that inserts leaf_j into the SMT by traversing from the root. It has a runtime of $\mathcal{O}(\log n)$.

[5] https://github.com/matter-labs/zksync/blob/master/core/bin/zksync_core/src/state_keeper/root_hash_calculator/mod.rs#L21.

2. `SMT.UpdateLeaf(leaf'_j)` is a deterministic algorithm that updates the value of leaf_j to leaf'_j. This algorithm assumes the existence of $v = M(j)$. As such, we can complete this algorithm in $\mathcal{O}(1)$.
3. `SMT.RemoveLeaf(j)` is a deterministic algorithm that updates the value of leaf_j to *default*. Similar to `SMT.UpdateLeaf`, it assumes the existence of $v = M(j)$ and can be completed in $\mathcal{O}(1)$.

4.2 Batch Update Algorithm

`SMT.BatchUpdate` is based on bottom-up binary tree level-order traversal using a queue data structure. It is broken down into two parts. In the first part (lines 4–9), we update the leaf nodes. In the second part (lines 10–19), we re-calculate the hashes of nodes in the affected paths in a bottom-up fashion and eventually return the new root hash. **T.cache** is a list of nodes that make up the tree.

Referring to lines 4 to 9, we first initialise an empty set *parent_set*, which we will use to store the indices of the direct parent nodes of the leaves that we updated. We use a set data structure to avoid duplicated parents (i.e. if we update both node_4 and node_5, then the parent node of both nodes, node_2, will only be added to the parent set once). Next, for each $o^j \in O$, we apply o^j to the value $M(j)$, calculate the new hash of $\text{leaf}_j = H(M(j))$ and add leaf_j's parent node's index to *parent_set*. As a result of performing all operations, *parent_set* is filled with a set of node indices at a level above the leaf level (i.e. $N - 1$).

Referring to lines 10–19, given parent_set, we first empty them into a queue current_level, which represents the indexes of the nodes we are updating. Next, for each $i \in$ current_level we calculate and update $H(\text{node}_i)$ by retrieving i's children hashes $H(\text{node}_{2i})$ and $H(\text{node}_{2i+1})$ from T. We are guaranteed to retrieve the most recently updated children's hashes because when we process indexes at level n where $0 \leq n \leq N$, nodes in $n + 1$ have already been updated. Then, we add node_i's parent index $\text{node}_{\lfloor i/2 \rfloor}$ to *parent_set*. We repeat this process until we reach the root level of T. As a result, node_1 (i.e. the root) will be updated and returned.

Example. To illustrate the above algorithm, consider an SMT of depth 2 and a list of operations $O = \{o^0, o^3, o^1\}$. Figure 2 (A) shows the leaf level nodes that are affected by O, they are $L_2 = \{\text{node}_4, \text{node}_5, \text{node}_7\}$ and their corresponding values in M (i.e. $M(0), M(1), M(3)$).

As a result of updating M and re-hashing L_2 nodes, 2.B shows the updated leaf nodes and M, and the parent nodes of L_2 which are $L_1 = \{\text{node}_2, \text{node}_3\}$ as dotted borders. Now, to re-hash node_2, we retrieve node_2's children nodes which are node'_4 and node'_5. The same can be done for node_3. Figure 2 (C) shows the result of re-hashing L_1, and the parent nodes of L_1, which is node_1. In the end, Fig. 2 (D) shows the final result of the algorithm and a new root hash.

Algorithm 1. Sparse Merkle Tree Batch Update

```
1: Input: Sparse Merkle Tree T of depth N, List of leaf operations O = {o^j}^k_{j∈[0,2^N)}
   of size k.
2: Output: Root Hash H
3: procedure SMTBatchUpdate(T, O)
4:     parent_set ← Set()
5:     for all o^j ∈ O do
6:         perform operation o^j on leaf_j
7:         calculate the new hash of leaf_j and update the value in T
8:         parent_set.add(leaf_j.parent)
9:     end for
10:    while parent_set is non-empty do
11:        current_level = empty(parent_set)
12:        for parent p_i in current_level do
13:            left_child_hash = get_child_hash(p_i.left)
14:            right_child_hash = get_child_hash(p_i.right)
15:            calculate the new hash of p_i by using left_child_hash and right_child_hash
   p_i and update the value in T.
16:            parent_set.add(p_i.parent)
17:        end for
18:    end while
         return T.cache[ROOT_index]
19: end procedure
```

4.3 Comparison

Table 1 compares the performance of the baseline and OBU for different types of leaf operations, `SMT.Commit`, and `SMT.BatchUpdate`. The table assumes an SMT of n leaves and a list of k operations. Although our `SMT.BatchUpdate` has the same asymptotic time complexity, it is more efficient because the improvement in `SMT.UpdateLeaf` and `SMT.RemoveLeaf`. Furthermore, the space complexity of our algorithm remained the same as the baseline algorithm, which is $\mathcal{O}(2^N)$.

Table 1. Asymptotic complexity comparison between OBU and the baseline. n is the number of leaves, k is the number of operations in a block, and H is a hash operation.

Method	zkSync Lite [21]	OBU
`SMT.InsertLeaf`	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
`SMT.UpdateLeaf`	$\mathcal{O}(\log n)$	$\boldsymbol{\mathcal{O}(1)}$
`SMT.RemoveLeaf`	$\mathcal{O}(\log n)$	$\boldsymbol{\mathcal{O}(1)}$
$\|w_i\|$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
`SMT.Commit`	$\mathcal{O}(k \log n)H$	$\mathcal{O}(k \log n)H$
`SMT.BatchUpdate`	$\mathcal{O}(k \log n) + \mathcal{O}(k \log n)H$	$\mathcal{O}(k \log n)H$

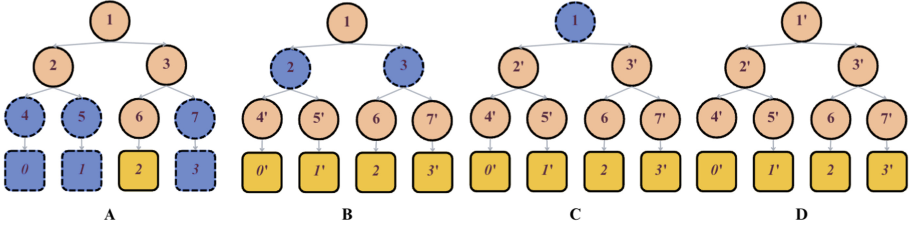

Fig. 2. An illustration of the one-phase batch update example provided in Sect. 4.2. Circle nodes are internal nodes, square nodes are data items with leaf indices, dotted boarders represent the nodes that are currently in the queue, and an apostrophe on a number represents the updated state of a node.

5 Experimental Analysis

We performed both micro- and macro-benchmarks to compare our algorithm with the benchmark. The micro-benchmarks consisted of simple leaf operations in single-threaded and multi-threaded settings. The macro-benchmark compared the performance of the algorithms on real-world block data from zkSync Lite. This section describes the experimental setup, the dataset used for the macro-benchmark, and the multi-threading optimisation for `SMT.BatchUpdate`.

5.1 Experimental Setup

zkSync Lite is implemented in the Rust programming language [19] as an open source project on Github[6]. We implemented Algorithm 1 on top of the existing repository. Further, we also optimised our implementation for multi-threading computation using the Rayon [4] library in rust.

The experiments are performed on an *AWS c5.12xlarge Debian, 48 CPU, 96 GiB memory* virtual machine. The SMT we used for our experiments has a depth of 24, which is the same depth as the one in zkSync Lite. For each experiment, we performed 10 runs and reported the average run time. The main metric we use to compare performance is the percentage decrease in running time given by

$$\%\text{decrease in running-time} = \frac{\text{new running-time} - \text{old running-time}}{\text{old running-time}}.$$

5.2 Dataset Collection

The macro-benchmark dataset contains 100 (block #299246- #299346) recent blocks and their transactions which are collected through the zkSync Lite API [5] and the zkSync Lite block explorer [6].

Of the 8376 transactions collected, 3971 are swap transactions, 1897 are transfer transactions, 1428 are MintNFT transactions, 766 are ChangePubKey

[6] https://github.com/matter-labs/zksync.

transactions, 266 are deposit transactions, 47 are withdraw transactions, and only 1 is a WithdrawNFT transaction. Details of these transaction types can be found in Appendix C.

More than 70% of the transactions are dominated by ERC-20 token transactions. To keep the experiments simple, we only considered the Transfer and Swap transaction.

We also noticed that the transaction count for each block is inconsistent. The maximum number of transactions observed was 133 while the minimum was 74. This is the result of a combination of the gas limit reached and the appearance of *Priority Transactions* such as Deposit and Withdraw during transaction processing, which will cause the current block to be sealed and committed as soon as it is processed (see Table 2).

We observed that there are many highly active accounts. In block # 299273, out of 92 transactions, one leaf was included in 48 transactions, taking up more than half of the block space. On average, each account produced 2.5 transactions in our dataset (Fig. 3).

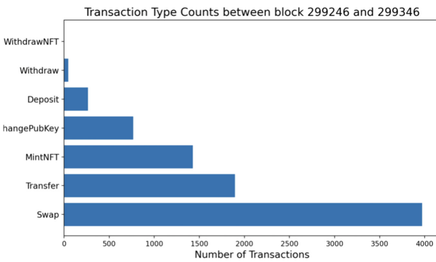

Fig. 3. Transaction count by type between blocks 299264 and 299364

Table 2. Macro-benchmark dataset information of zkSync Lite blocks 299246 - 299346

Statistic	Value
Total # txs	8376
Max tx count in a block	133
min tx count in a block	74
Average tx count in a block	83
Unique accounts	3322
Average tx per account	2.5

5.3 Multi-threading Optimisation

Both the baseline and OBU can be optimised for multi-threading. In the baseline, threads can be created in the recursive stage by visiting the child nodes. In OBU, a thread can be created for every node that requires re-hashing in a level. Note that in the baseline, the threads are nested as the tree is traversed deeper, whereas in OBU, there are no nested threads.

6 Evaluation

6.1 Micro-benchmarks

In Sect. 2.1 we gave three categories of leaf operations: `SMT.UpdateLeaf`, `SMT.InsertLeaf`, and `SMT.RemoveLeaf`. In the micro-benchmarks, we performed simple leaf operations to demonstrate the effectiveness of our pro-

posed algorithm. Without losing generality, we did not include experiments for `SMT.RemoveLeaf` operation as the implementation is similar to `SMT.UpdateLeaf`.

With Multi-threading. Figure 4 shows the performance comparison when multi-threading is enabled. In Fig. 4 (A1), when the update operations are applied to leaves with sequential IDs, we see that OBU outperforms the baseline. We also note that the gap in runtime is increasing by an increasing factor. This is expected because given k update operations, the baseline spends $\mathcal{O}(k \log n)$ on traversal and update, while with OBU, the update time is linear with respect to k (i.e. $\mathcal{O}(k)$ update time).

In Fig. 4 (A2), we see that when the number of operations is small, we see a larger percentage decrease in running time and as the number of operations increases (10% decreases for 1000 updates), % decrease in running time shows exponential decay. The initial large percentage decrease relates to how the two algorithms use multi-threading. In the baseline, threads can be nested as deep as 24 levels, which can cause high computation overhead, whereas in OBU, there is no such problem because threads end when the currently traversed level is finished. Furthermore, the diminishing trend in Fig. 4 (A2) can be explained by hardware limitations. In OBU, as the number of nodes we process on each level increases, the number of concurrent threads becomes insignificant compared to the number of nodes we need to process.

Figure 4 (B1) shows the runtime difference when the update operations are applied to random leaf IDs taken from a uniform distribution. We note that the improvement in runtime is worse visually compared to Fig. 4 (A1). This is because when leaf IDs are randomly assigned, there are fewer common parents. As such, the amount of computation of OBU approaches the baseline. However, we also note that the trend shown in Fig. 4 (B2) is consistent with Fig. 4 (A2) when it comes to the percentage of decrease in running time.

Figure 4 (C1) shows the runtime difference for when insert operations are applied to leaves with sequential IDs. Both Fig. 4 (C1) and Fig. 4 (C2) show consistent trends as Fig. 4 (A1) and Fig. 4 (A2) respectively.

Without Multi-threading. Figure 5 (A) shows the running time comparison between the benchmark and OBU when running on a single thread. We note that there is no visible performance improvement because the tree traversal time $\mathcal{O}(k \log n)$ is insignificant compared to the hashing time. This is further demonstrated in Fig. 5 (B) when we only observe a slight improvement in the percentage decrease in running time.

6.2 Macro-benchmark

We macro-benchmark the performance of OBU with the baseline using zkSync Lite block data. As shown in Fig. 6, OBU almost always outperforms the baseline. Overall, OBU performed, on average, 5.12% faster than the baseline, with the

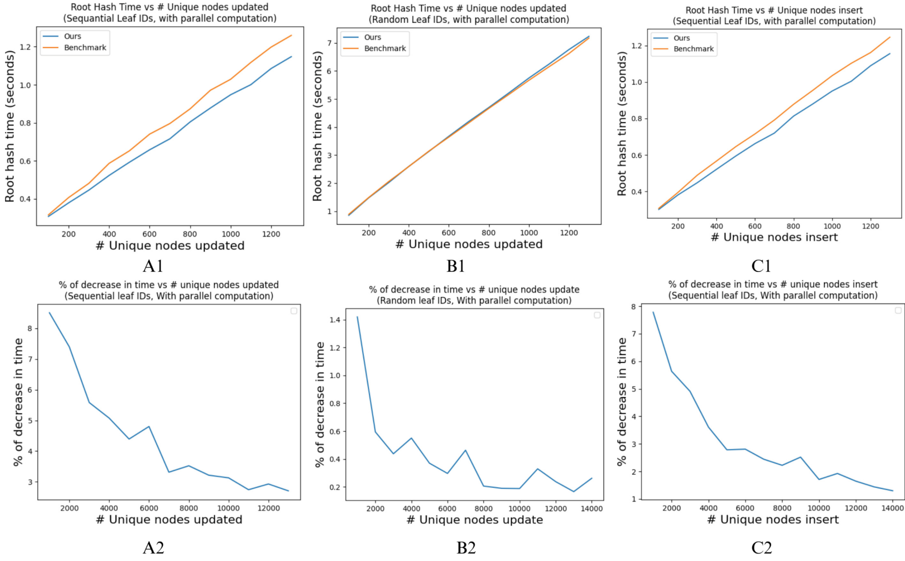

Fig. 4. Top row: root hash time in seconds comparison between benchmark and OBU with various operation types. Bottom row: percentage decrease in root hash time with various operation types. (with multi-threading)

highest percentage of decrease in time being 14%. Next, we analysed the blocks that exhibited the highest/lowest performance improvement. Our observations are as follows:

1. In the block with large percentage of decrease in time (i.e. in solid circles in Fig. 6) we notice that most transactions in the block affected very few accounts. This corresponds to a faster running time because OBU does not repeatedly traverse the same account
2. In the blocks with negative percentage of decrease in time (i.e. in dotted circles in Fig. 6), updates are spread across multiple accounts instead of just a few accounts.

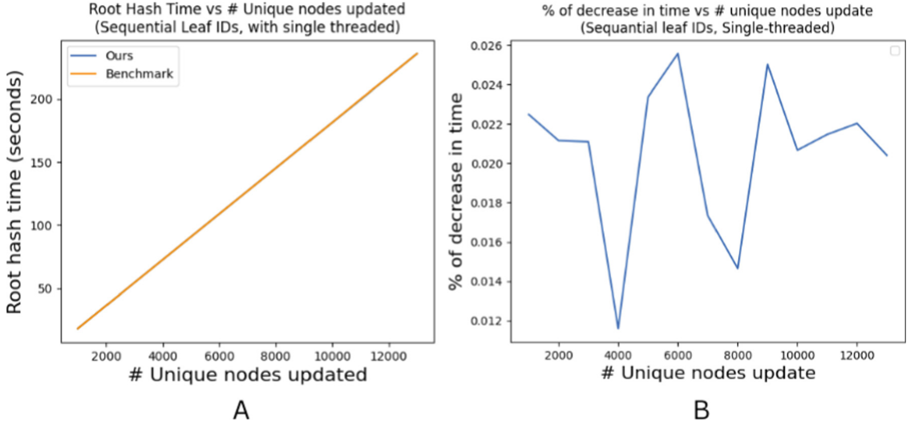

Fig. 5. A) Update operations on leaves with sequential leaf IDs performance comparison, no visible difference. B) Percentage decrease in running time with OBU compared with baseline. (Single-threaded)

Table 3. Result statistics from the macro-benchmark.

Statistic	Runtime Reduction (%, ms)
Mean	5.12%, 25.56
Medium	5.24%, 26.73
Standard Deviation	4.39%, 21.55
Variance	19.24%, 464.68
Minimum	−3.81%, −20.67
Maximum	14.99%, 69.79
Range	18.80%, 90.46

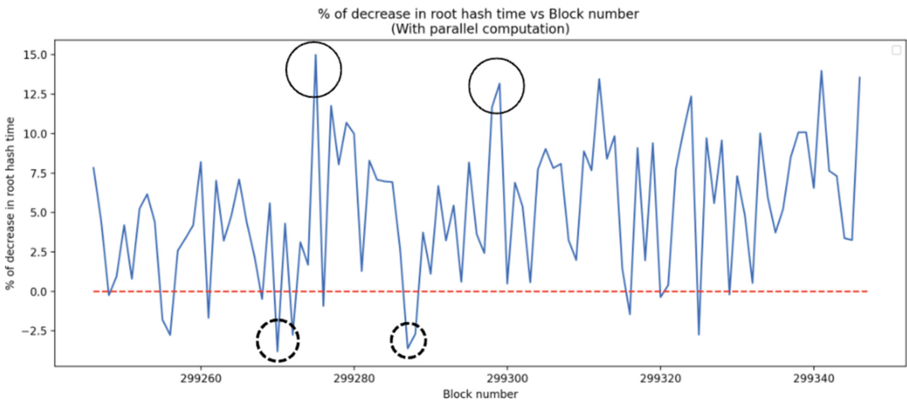

Fig. 6. Percentage decrease in running time on zkSync Lite block data. The dotted horizontal line is when the percentage of decrease is 0.

7 Conclusion and Discussion

In this paper, we presented and evaluated OBU, a batch update algorithm on sparse Merkle trees. The improvement can be summarised as follows. OBU achieved a 50% decrease in run time by traversing the tree once instead of twice. OBU uses threads more efficiently compared to the implementation presented in zkSync Lite. This could reduce the hardware requirement to run an L2 operator. More specifically, OBU reduced the run time by 50% for the `SMT.InsertLeaf` operation. For `SMT.UpdateLeaf` and `SMT.RemoveLeaf` operations, the running time is reduced from $\mathcal{O}(\log n)$ to $\mathcal{O}(1)$ (see Table 1).

High Frequency Transaction Applications. The second improvement will directly benefit applications with a higher frequency of transactions. Suppose that a block has k transactions that affect a single account. Instead of traversing the SMT k times in $\mathcal{O}(k \log n)$ runtime, OBU will complete the operations in $\mathcal{O}(k)$ runtime. This is evident in block #299275[7] where 29 of the 47 transfer/swap transactions in block. In this case, OBU achieved a 14.9% decrease in running time.

7.1 Future Work

For future work, we first want to perform more integration tests in zkSync Lite to better understand the advantages and drawbacks of OBU. Next, we wish to see how our research can improve zkEVM, which is another prominent blockchain scaling direction. Then, we want to see how our research may be used for batch update in other authenticated data structures, such as Vector Commitment schemes [14].

Acknowledgement. The authors extend their thanks to Sean Morota Chu, Ziyu Liu, Nhi Nguyen, and Tim Yang for invaluable feedback on the manuscript, Barak Saini for helping us understand zkSync Lite, and Hao Ren for LaTeX formatting advice.

A zkSync Lite Details

A.1 Design

Like most L2 solutions, zkSync Lite has two main components: on-chain and off-chain. The on-chain component includes several Solidity Smart Contracts deployed[8] on the Ethereum mainnet. The off-chain component includes several microservices that facilitate L2 transaction executions and SNARK generation.

[7] https://zkscan.io/explorer/blocks/299275.
[8] https://etherscan.io/address/0xaBEA9132b05A70803a4E85094fD0e1800777fBEF.

A.2 On-Chain

The on-chain component has three main contracts.

The first one is the zkSync main contract. It stores L1 user funds, bridges funds between L1 and L2 with `Deposit` and `Withdraw` transactions, accepts committed blocks and block proofs from the operator, verifies block proofs, and process withdrawal transactions by executing blocks. Users can deposit $ETH or ERC-20 tokens. However, the allowed ERC-20 tokens are determined by the Security Council.

The second Smart Contract is Verifier. Given a committed block and a proof, the Verifier contract verifies the proof to determine the validity of the state transition caused by the transactions in the block.

The third Smart Contract is Governance. It has the functionalities to add (but not remove) ERC-20 tokens to the whitelisted tokens, change the set of operators, and initiate the upgrade of the contracts.

When L1 users wish to deposit/withdraw their funds to/from L2, they can interact directly with the zkSync main contract.

A.3 Off-Chain

The off-chain component is divided into two main sub-components. The server and the prover. An operator needs to run both sub-components in order to create L2 blocks.

Server. The Server has the following modules [21]:

1. Ethereum Watcher: module to monitor on-chain operations.
2. State Keeper: module to execute and seal blocks.
3. Memory Pool: module to organise incoming transactions.
4. Block Proposer: module to create block proposals for state keeper
5. Committer: module to store pending and completed blocks into the database
6. API: module to allow users to interect with zkSync Lite to query block data or submit transactions.
7. Ethereum Sender: module to sync the operations on zkSync Lite with the Ethereum blockchain. It makes sure that the L1 transactions zkSync Lite created (such as committing a block on-chain) are executed on-chain in the correct order.

Prover. The Prover's only job is to create block proofs given a block's transaction witnesses. It regularly polls the Server for blocks that do not have a corresponding SNARK. When a new block is available, Server sends the block's witnesses so the Prover can begin creating the block proof. Once finished, the Prover returns the SNARK to the Server and the server sends it to the on-chain Smart Contract to be verified.

B zkSync Lite Transaction Flow

Below we describe the transaction flow on zkSync. First, we provide an end-to-end description from L2 transaction submission to L2 block finalisation on-chain. Then, we zoom in on the Server to describe the flow within the Server in details.

B.1 Overall Transaction Flow

Referring to Fig. 7 for a simplified representation of zkSync Lite. When a user submits a transaction, it is placed into the memory pool (mempool) waiting to be collected by the Server. The server periodically collects a queue of transactions from the mempool, in submission order, and puts them into blocks. After the blocks are formed, they are committed to the L1 Smart Contract and stored in the database. At this moment, although the block information is on-chain, they are not finalised. These blocks in this state are known as the "committed block".

At the same time, available Provers poll the Operator for proof generation jobs. When there are blocks without a proof, the Operator will generate and send the block witnesses to the Prover, who will use the witnesses to generate and return the block proof. Once the operator receives the block proof, it will send it to the L1 Smart Contract for verification.

The Verifier contract verifies the block proof along with the committed block data. The L2 Smart Contracts updates the block's from committed to finalised when the proof is validated.

For priority transactions (listed in Sect. C) that are submitted directly to the L1 Smart Contract, they are tracked by the Operator and added to the mempool into a priority queue.

B.2 Transaction Flow Within Server

Looking specifically into the Server shown in Fig. 8, as blocks are created by the block producer, they are sent to the State Keeper. The State Keeper processes the transactions in the blocks and update the accounts' balances accordingly. Although it stores the Account Tree, it does not update the Account Tree's root hash. It delegates the computation intensive job to the Root Hash Calculator, where the re-hashing of the tree is done. Once a block is completed with a root hash, it is committed to the database.

As the Prover polls for committed blocks, the Witness Generator will generate transaction witnesses and send to the Prover.

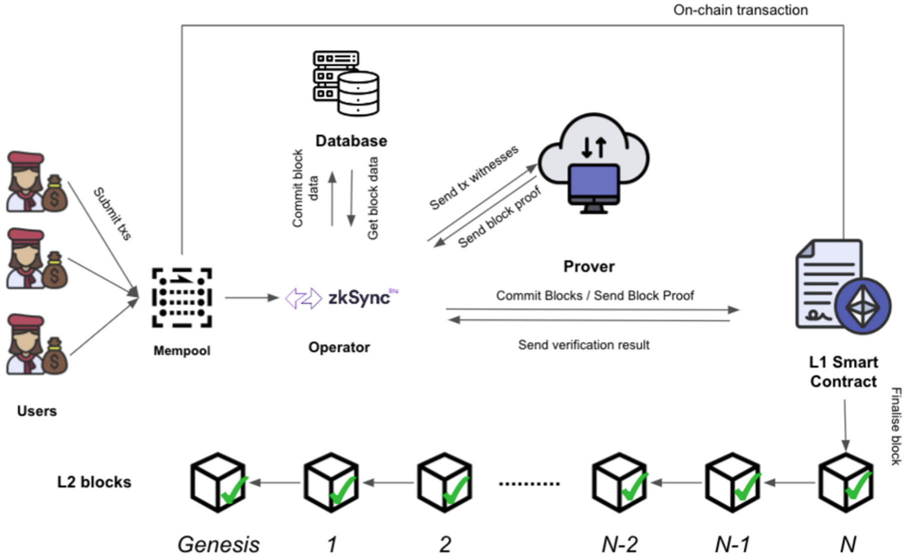

Fig. 7. An illustration of transaction flow within zkSync Lite from transaction submission to L2 block finalisation.

C zkSync Transaction Types

As mentioned above, zkSync Lite supports a number of transaction types. Here, we give a brief description of these transaction types. Full descriptions can be found in [21]. There are two main categories of transactions on zkSync: normal and priority transactions. Priority transactions are handled by the operator differently during the L2 block creation process. Given a queue of transactions from the mempool, and an operator continually placing transactions into blocks, as soon as a priority transaction is processed, the current block is sealed and committed regardless of remaining gas in the block.

C.1 From Transactions to Leaf Operations

One or more accounts can be affected as a result of a transaction. For example, a transfer transaction adds to the receiver's balance, as well as deducting from the sender's balance. To make the account leaf updates atomic, zkSync breaks down each type of transactions into their a number of *leaf operations*. Each operation only affects one account leaf at a time.

In the following sections, as we describe the transaction types, we include the number of operations to which they can be broken down.

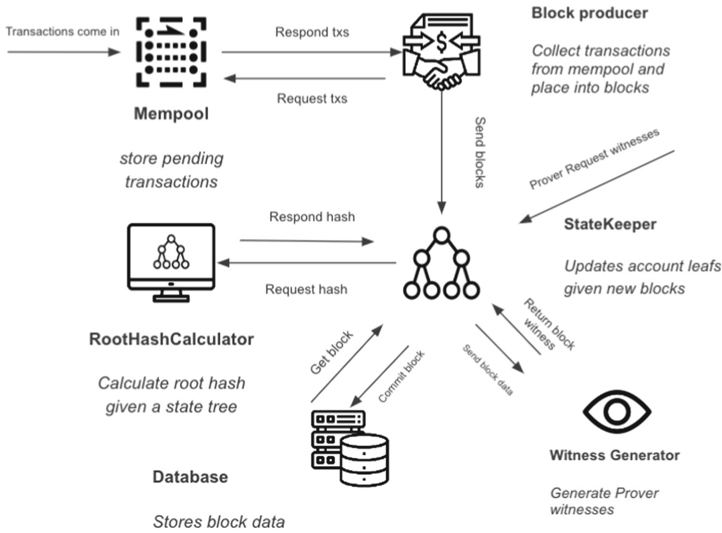

Fig. 8. An illustration of transaction flow zkSync Lite server.

C.2 Normal Transaction Types

1. `Transfer`: Transfer funds between rollup accounts. It translates to two `SMT.UpdateLeaf` operations. The first decreases the sender balance, and the second increases the receiver balance.
2. `Transfer to new`: Transfer funds to a new account. This transaction type is derived from `Transfer` and happens when the `to_account` doesn't exist in the `AccountTree`. Before the transfer of funds, a new account will be created for `to_account`. It translates to an *update* and an `SMT.InsertLeaf` operation. The first one decreases sender balance, and the second one inserts a new account leaf.
3. `Withdraw`: Withdraw funds from the L2 account to the indicated Ethereum address. It translates to an `SMT.UpdateLeaf` operation where the balance of the withdrawal account is decreased.
4. `Withdraw NFT`: Withdraw NFT from the L2 account to the indicated Ethereum address. It translates to two `SMT.UpdateLeaf` operations. The first removes the NFT from the owner account and the second removes the NFT from the creator's account.
5. `Mint NFT`: Mint an NFT token inside L2. It translates to two `SMT.UpdateLeaf` operations. The first adds the NFT to the receiver's account, and the second updates the creator's account.
6. `Change pubkey`: Change the public key used to authorise transactions for an account. This can be useful when a user wishes to delegate the account to another user or Smart Contract wallet with a different Ethereum address without the need to expose their own private key. It translates to an `SMT.UpdateLeaf` operation on the sender's account where the public key is updated.

7. `Forced Exit`: Withdraw funds from L2 accounts without the signing key to the appropriate L1 address. These accounts are known as *unowned* accounts. It translates to `SMT.UpdateLeaf` and `SMT.RemoveLeaf` operations. The first up decreases the sender's balance, and the second one removes the account leaf and replaces it with a default node.
8. `Swap`: Perform an atomic swap of ERC-20 tokens between two L2 accounts at a defined ratio. Its operations are similar to the `transfer` transaction type.

C.3 Priority Transaction Types

1. `Deposit`: Deposit funds from Ethereum to L2. The funds are sent to the zkSync Lite Smart Contract, which informs the operator to include a deposit transaction in the next block. A new account is created if necessary. It may translate to an `SMT.UpdateLeaf` operation, or an `SMT.InsertLeaf` operations. The operation is `SMT.UpdateLeaf` when the account already exists. On the other hand, the operation is `SMT.InsertLeaf` when a new account needs to be created.
2. `Full exit`: In the event that a user thinks the operator has censored their transactions, they can submit a `Full exit` transaction directly to the Smart Contract. The operator will process the transaction accordingly. Its operations are the same as `Forced exit`

In the event that a priority transaction has not been processed for more than a week, the system will enter the `exodus mode` and the operators will stop working, and every user can use an exit tool[9] to withdraw their asset by submitting a proof of balance to the L1 smart contract.

D zkSync Lite Sparse Merkle Tree Usage

zkSync Lite uses the SMT in three separate places as a data accumulator. They are the account tree, the circuit account tree, and the balance tree.

Account Tree. The Account Tree[10] is a binary SMT of depth 24. It is the main data structure that stores the state of the zkSync Lite accounts. Its leaves are the accounts on zkSync. The leaf hash is the rescue hash [8] of an account's fields concatenated in their respective little-endian bit representation.

The leaf indices are the same as the account IDs, which are mapped to account addresses. Empty leaves are replaced with a default hash.

[9] https://github.com/matter-labs/zksync/tree/master/infrastructure/exit-tool.
[10] https://github.com/matter-labs/zksync/blob/master/core/lib/types/src/lib.rs#L84.

Circuit Account Tree. The purpose of the Circuit AccountTree is to generate compatible transaction witnesses so that the Prover can create the block proof. The Circuit Account Tree is structured similar to the AccountTree except for two main differences: 1) account data are encoded as field elements and 2) each account uses an SMT to track balances for each type of token (Balance Tree) instead of using a simple hash map. The Circuit AccountTree is derived from AccountTree.

Balance Tree. As mentioned above, the Balance Tree is a part of the account leaves in the Circuit Account Tree. It is an SMT of depth 8. Each leaf in the Balance Tree represents the balance of the token with the id the same as the leaf index.

References

1. https://www.immutable.com/products/immutable-x
2. https://ethtps.info/
3. https://matter-labs.io/
4. https://github.com/rayon-rs/rayon
5. https://docs.zksync.io/api/
6. https://explorer.zksync.io/
7. Adams, H., Zinsmeister, N., Salem, M., Keefer, R., Robinson, D.: Uniswap v3 core. Technical report, Uniswap, Technical Report (2021)
8. Aly, A., Ashur, T., Ben-Sasson, E., Dhooghe, S., Szepieniec, A.: Design of symmetric-key primitives for advanced cryptographic protocols. IACR Trans. Symmetric Cryptol., 1–45 (2020)
9. Aztec: https://aztec.network/
10. Bauer, M.: Proofs of zero knowledge. arXiv preprint: cs/0406058 (2004)
11. Buterin, V.: An incomplete guide to rollups (2020). https://vitalik.ca/general/2021/01/05/rollup.html
12. Buterin, V.: A rollup-centric Ethereum roadmap (2020). https://ethereum-magicians.org/t/a-rollup-centric-ethereum-roadmap/4698/1
13. Buterin, V.: The limits to blockchain scalability (2021). https://vitalik.ca/general/2021/05/23/scaling.html
14. Catalano, D., Fiore, D.: Vector commitments and their applications. In: Kurosawa, K., Hanaoka, G. (eds.) PKC 2013. LNCS, vol. 7778, pp. 55–72. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36362-7_5
15. Entriken, W., Shirley, D., Sachs, N.: ERC-721: non-fungible token standard. Ethereum Improvement Proposals, no. 721 (2018)
16. Gabizon, A., Williamson, Z.J., Ciobotaru, O.: PLONK: permutations over Lagrange-bases for Oecumenical noninteractive arguments of knowledge. Cryptol. ePrint Arch. (2019)
17. Khan, D., Jung, L.T., Hashmani, M.A.: Systematic literature review of challenges in blockchain scalability. Appl. Sci. **11**(20), 9372 (2021)
18. Kiayias, A., Russell, A., David, B., Oliynykov, R.: Ouroboros: a provably secure proof-of-stake blockchain protocol. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017. LNCS, vol. 10401, pp. 357–388. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63688-7_12

19. Klabnik, S., Nichols, C.: The Rust Programming Language. No Starch Press, San Francisco (2023)
20. Kumar, M., Nikhil, N., Singh, R.: Decentralising finance using decentralised blockchain oracles. In: 2020 International Conference for Emerging Technology (INCET), pp. 1–4. IEEE (2020)
21. Labs, M.: zkSync: scaling and privacy engine for Ethereum (2020). https://github.com/matter-labs/zksync
22. Loopring: https://loopring.org/
23. Luu, L., Narayanan, V., Zheng, C., Baweja, K., Gilbert, S., Saxena, P.: A secure sharding protocol for open blockchains. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 17–30 (2016)
24. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system. Decentralized Business Review, p. 21260 (2008)
25. Negka, L.D., Spathoulas, G.P.: Blockchain state channels: a state of the art. IEEE Access **9**, 160277–160298 (2021)
26. Panarello, A., Tapas, N., Merlino, G., Longo, F., Puliafito, A.: Blockchain and IoT integration: a systematic survey. Sensors **18**(8), 2575 (2018)
27. Poon, J., Buterin, V.: Plasma: scalable autonomous smart contracts. White pap., 1–47 (2017)
28. Rocket, T., Yin, M., Sekniqi, K., van Renesse, R., Sirer, E.G.: Scalable and probabilistic leaderless BFT consensus through metastability. arXiv preprint: arXiv:1906.08936 (2019)
29. Tijan, E., Aksentijević, S., Ivanić, K., Jardas, M.: Blockchain technology implementation in logistics. Sustainability **11**(4), 1185 (2019)
30. Visa: https://www.visa.co.uk/dam/VCOM/download/corporate/media/visanet-technology/aboutvisafactsheet.pdf
31. Vogelsteller, F., Buterin, V.: ERC-20: token standard. Ethereum Improvement Proposals, no. 20 (2015)
32. Wood, G., et al.: Ethereum: a secure decentralised generalised transaction ledger
33. Zamani, M., Movahedi, M., Raykova, M.: RapidChain: scaling blockchain via full sharding. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 931–948 (2018)