The Locality of Memory Checking

Weijie Wang[*]
Yale University

Yujie Lu[†]
Yale University

Charalampos Papamanthou[‡]
Yale University

Fan Zhang[§]
Yale University

ABSTRACT

Motivated by the extended deployment of authenticated data structures (e.g., Merkle Patricia Tries) for verifying massive amounts of data in blockchain systems, we begin a systematic study of the I/O efficiency of such systems. We first explore the fundamental limitations of memory checking, a previously-proposed abstraction for verifiable storage, in terms of its locality—a complexity measure that we introduce for the first time and is defined as the number of non-contiguous memory regions a checker must query to verifiably answer a read or a write query. Our central result is an $\Omega(\log n/\log \log n)$ lower bound for the locality of any memory checker. Then we turn our attention to (dense and sparse) Merkle trees, one of the most celebrated memory checkers, and provide stronger lower bounds for their locality. For example, we show that any dense Merkle tree layout will have average locality at least $\frac{1}{3} \log n$. Furthermore, if we allow node duplication, we show that if any write operation has at most polylog complexity, then the read locality cannot be less than $\log n/\log \log n$. Our lower bounds help us construct two new locality-optimized authenticated data structures (DupTree and PrefixTree) which we implement and evaluate on random operations and real workloads, and which are shown to outperform traditional Merkle trees, especially as the number of leaves increases.

1 INTRODUCTION

Memory checking, introduced by Blum et al. [6], is a primitive that captures the fundamental abstraction of cryptographically verifying *untrusted storage*. In memory checking, an array $\mathbf{A}$ of n elements is stored in an untrusted memory M. A memory checker, with the help of trusted and potentially private sublinear memory s provides algorithms to verifiably read and write $\mathbf{A}[i]$ by querying public memory M and private memory s. There are different implementations of memory checking, with the most widely-used memory checker being the celebrated Merkle tree [35]. In Merkle trees, the small memory corresponds to the Merkle root digest, the public memory corresponds to the Merkle tree itself, and reading/writing $\mathbf{A}[i]$ requires querying $\log n$ public memory locations, which along with the root digest (stored in the private memory s), can be used to verify the correctness of $\mathbf{A}[i]$. Other primitives that can serve as memory checkers are, for example, vector commitments [9, 12, 29, 31, 42] and accumulators [5, 17, 34].

Query complexity of memory checking. One of the basic complexity measures of a memory checker is its *query complexity*, i.e.,

the maximum number (across all index queries $1, \ldots, n$) of public memory locations that need to be queried for either a read or a write. For example, the query complexity of Merkle trees is $\log n$ since $\log n$ hashes must be collected for a verifiable read or a write. Dwork et al. [18] showed that there is a query lower bound of $\log n/\log \log n$ for every memory checker, ruling out the existence of memory checkers with $O(1)$ query complexity (Note that although memory checkers like accumulators can have $O(1)$ query complexity for reads, their write query complexity is almost linear.)

Introducing locality to capture I/O efficiency. Currently various instantiations of memory checkers are being used on massive amounts of data, mainly for blockchain applications. For example, all smart contract states in Ethereum can be accessed efficiently via an authenticated Merkle Patricia Trie, a type of Merkle tree designed for storing key-value pairs. At the time of this writing, the current Ethereum block stores the root digest of a Merkle Patricia Trie on 240 million accounts [19]. The state stored in the Merkle Patricia Trie of an operational full node is about 1.2 terabytes and is stored on SSD via LevelDB [26], a LSM-based database system. Vector commitments, another implementation of memory checking, are also used to commit to large databases in many zero-knowledge proof systems (such as Plonky2 [32], Spartan [39]) in various applications such as zk-rollups [22], sequencers [21] and stateless blockchains [8, 14]. Clearly, since these real-world memory checkers are used on large amounts of data residing in secondary storage mediums such as SSDs and HDs, we must take into consideration their I/O efficiency.

To formally capture the I/O efficiency of memory checking, we introduce and study its *locality*. The definition of locality is quite natural and follows a similar framework with the one used in searchable encryption [4, 10, 15, 40]: We say that a memory checker has locality t, if the maximum number (across all index queries $1, \ldots, n$) of non-contiguous public memory regions queried is t. For example, the "textbook" implementation of Merkle trees results in $\log n$ locality since for every index i, the hash values along the path from i to the root are stored in $\log n$ non-contiguous memory locations (The Merkle tree "textbook" implementation stores the first level in one contiguous memory region, followed by the second level in another contiguous memory region, and so on.) However, one might choose to store a Merkle tree differently (We call this layout "NAIVE" later on.) For every index i, store every path p_i in consecutive memory locations, by duplicating the root n times, its children $n/2$ times, etc. In this case, while the read locality is 1 and read query complexity is $\log n$, the write query complexity increases to linear (This is because all copies of the root node must be updated whenever an index changes.) This is a fundamental trade-off between query

complexity and locality, which is among the things studied in this paper.

In particular this paper poses and answers the following two questions that relate to the locality of memory checking.

What are the fundamental limitations of memory checking in general and (dense and sparse) Merkle trees in particular, in terms of locality, both with and without node duplication?

Can we outperform the "textbook" implementation of Merkle trees for large amounts of data by improving its locality?

We now present our contributions in detail.

1.1 Our contributions

This paper comprises two pairs of contributions (one pair per question above) which we summarize in the following.

Lower bound for the locality of memory checking. Our first contribution is to prove an $\Omega(\log n/\log\log n)$ lower bound on the locality of memory checking. In particular, we show that, given a memory checker with poly-logarithmic query complexity q and locality t, we can construct another checker with query complexity at most $2t$. Since, by Dwork et al. [18], we know that the query complexity of any memory checker is lower-bounded by $\log n/\log\log n$, our locality lower bound follows. The key point in our reduction is to transform each contiguous chunk we need to query (there are at most t such chunks for one index) in the first checker into at most $2t$ "memory slots" in the new checker, by changing the alphabet size, yielding a new checker with query complexity $2t$.

Lower bounds for locality of Merkle trees. Next, we turn our attention to studying the locality of Merkle trees. Our hope is to derive stronger lower bounds given that Merkle trees is a specific memory checker. We consider two cases, one where no duplicate nodes are allowed and one where duplicate nodes are allowed. For the first case, we show that, no matter how we arrange the Merkle tree nodes on disk, the average concrete locality for each index is at least $\frac{1}{3}\log n$. Also, motivated by the use of sparse Merkle trees in blockchain systems like Ethereum, we extend our result to sparse Merkle trees, showing that for any sparse tree with height h and n leaves, the locality is $\Omega(\log_h n)$. Finally, when storing at most K copies of a Merkle node in memory (node duplication), we show that the read locality should be $\Omega(\log n/\log K)$ and the update complexity becomes proportional to K. This result essentially implies a tradeoff between the update complexity and the read locality when duplicate nodes are used, which can be useful to explore depending on the application.

New Merkle tree constructions with low locality. Our lower bound proofs above help us derive Merkle trees with better locality. In particular, for the case of no duplication we provide a Merkle tree layout that has average locality exactly $\frac{1}{3}\log n$, matching our lower bound. This layout could be a fair optimization for practical purposes. When we allow duplicate nodes, we first ask the question whether the "NAIVE" layout presented in the beginning of the introduction can be improved to occupy space $O(n)$, instead of $O(n\log n)$. We answer this question in the affirmative by building DupTree: The basic idea is to copy nodes only of certain height so that the data structure size still stays linear. See Section 4.1 for details of our construction.

However, DupTree has large update complexity ($n/\log n$). To address this, we propose DupTree++ with $O(\log^c n)$ update complexity. Our main idea is to partition the tree of height $\log n$ into layers of height $\log\log n$. There are $\log n/\log\log n$ such layers. In each layer, there are several sub-trees of height $\log\log n$ with $O(2^{\log\log n}) = O(\log n)$ nodes. We then apply DupTree to each subtree to make the locality in each tree almost constant. The resulting locality for reads is about $O(\log n/\log\log n)$ and the write complexity is about $O(\log^2 n)$, while the disk space is still linear.

Our final locality-aware construction is PrefixTree for sparse Merkle trees, like the Ethereum Merkle Patricia Trie (MPT). Our idea is to tweak the MPT implementation used in Ethereum as follows: In Ethereum's implementation of MPT, each node is stored in a key-value database with its hash as the address (index) and the serialized node as the value. As a result, tree nodes are essentially stored in random places in the database, leading to bad read and write locality. Instead, our PrefixTree construction stores nodes with its key prefixes as the index, so that nodes on a Merkle path are stored in adjacent locations.

Evaluation. We implemented DupTree++ and PrefixTree in C++ using LevelDB [26] as the persistent storage. Our evaluation has two components.

Comparison of DupTree++ with "textbook" implementation of Merkle trees. We compare the performance of DupTree++ with the "textbook" implementation a Merkle tree stored in LevelDB, with no duplication. Our experiments show that DupTree++ outperforms the "textbook" Merkle tree when n is large, by up to 1.64×, both for reads and writes.

Evaluation of PrefixTree on real-world workload. We also evaluate PrefixTree on a real-world workload extracted from transactions in the first 4 million Ethereum blocks (from July 2015 to July 2017). We compare the performance of PrefixTree with a simplified implementation of Merkle Patricia Trie (as the baseline). We find that PrefixTree is up to 3× faster after 2.4 million blocks and the advantage is increasing as the tree grows. This experiment shows that the locality of sparse Merkle trees significantly impacts its performance on real-world workloads.

1.2 Related work

Memory checkers. Blum et al. [6] introduced the notion of memory checking, formalizing the algorithm that uses the client's small secret storage to detect faults in the large remote storage. Dwork et al. [18] investigated the lower bound of the query complexity of memory checkers, i.e., the number of queries made to the remote storage per user query. They proved that when the client's secret space is sub-linear to the size of the database, the query complexity has lower bound $\Omega(\log n/\log\log n)$.

Efficient authenticated storage systems. Authenticated storage access has become the performance bottleneck for blockchain systems [33, 37, 38], because each read or write operation in the standard Merkle Patricia Trie structure incurs $O(\log n)$ I/O operations. LVMT [33] uses the authenticated multipoint evaluation tree (AMT) [43] to improve I/O efficiency. Read/write amplification in authenticated key-value storage systems means that each

read or write operation to the authenticated storage will be amplified to multiple disk operations. By leveraging AMT to reduce the read/write amplifications from $O(\log n)$ to $O(1)$ (this is because AMT needs only constant time to update the digest while Merkle trees require log-time), LVMT reduces disk I/O at the cost of a significant increase in the proof generation time due to the use of vector commitments.

mLSM [38] combines LSM [36] with Merkle Patricia Tries and caching techniques to reduce the read/write amplification and it is optimized for write-heavy workloads. Similar to LSM, mLSM maintains multiple independent trees and organizes them with multiple levels. However, mLSM also does not consider the effect of locality on I/O efficiency.

Rainblock [37] proposed a new architecture that transforms local storage I/O into network-distributed storage I/O, benefiting from parallel I/O and in-memory storage. To decrease network storage read latency, RainBlock also implements I/O prefetchers and mandates that miners attach all accessed key-value pairs and witnesses (MPT nodes) when broadcasting blocks.

Jellyfish Merkle tree [23] leverages spatial locality to reduce compactions to zero, thereby reducing the write amplification. More specifically, they add the version number before the hash key, so that nodes with the same version will be stored near each other on the disk. Our work further introduces locality to formalize such optimizations.

Spatial locality. Locality [16] is a common attribute exhibited by all computational processes, whereby they frequently access a specific subset of their resources over long periods. System and cryptography researchers have utilized this attribute to enhance performance through various methods. For example, Cash et al. [11] initiated the study of the locality in searchable symmetric encryption and provided a tradeoff between the locality of memory accesses and the server storage. Demertzis et al. [15] investigated the optimal locality of searchable encryption, reducing disk I/O by proposing a better access pattern of each keyword search. EvenDB [24] is a persistent key-value store that leverages spatial locality by combining spatial data partitioning with LSM-like batch I/O.

1.3 Paper outline

The paper outline is as follows. In Section 2 we give preliminary definitions and notations. In Section 3 we give four lower bound proofs for locality (memory checking, full (dense) Merkle trees, sparse Merkle trees, Merkle trees with duplication). In Section 4 we introduce our new locality-aware constructions. In Section 5 we present an evaluation of our constructions.

2 PRELIMINARIES

2.1 Notation

We use lowercase letters $a, b, c, \ldots$ to denote tree nodes and capital letters $T, L, P, \ldots$ to denote trees. When referring to a tree T, we use r to denote the root node, T_1 the left subtree, T_2 the right subtree, r_1 the root of T_1, and r_2 the root of T_2. Furthermore, by default, $a_1, a_2, \ldots$ are nodes in T_1 and $b_1, b_2, \ldots$ are nodes in T_2. We use $\mathcal{H}(\cdot)$ to denote the *height* of a tree node defined in the standard way where any leaf l has height 0 (i.e., $\mathcal{H}(l) = 0$). The height of a tree $\mathcal{H}(T)$ is defined to be the height of its root r.

2.2 Memory checking

The problem of *memory checking*, introduced by Blum, Evans, Gemmel, Kannan and Naor [7] in 1991, arises in the following setting. A user wants to outsource the storage of a large bit database to a public and untrusted party (called the public memory) and wishes to query (i.e., read and write) entries in the database with correctness guarantees. To enable that, one can use a memory checker. A memory checker receives queries from the user, interacts with the public memory, and returns to the user the query result or an error. A memory checker should guarantee that the returned result is the latest value the user stored in the database with overwhelming probability. To make the problem non-trivial, we require that the memory checker's local memory is much smaller than the database (otherwise the memory checker can store the entire database).

Dwork et al. [18] formalized the security for online memory checkers, i.e., checkers that respond to the user queries immediately after the requests (These are the checkers we will be studying here.) We adjust their definition to have two algorithms Setup and Check.

Definition 2.1 (Memory Checker). A (Σ, n, q, s) deterministic and non-adaptive memory checker consists of the following algorithms (Without loss of generality we assume this checker is for a database of n binary bits):

(1) Setup$(n, \mathrm{Db}) \rightarrow (\mathcal{S}, O)$: Given a database Db of n bits, it outputs the initial secret memory $\mathcal{S}$ and an array O as the public memory. The secret space and public memory are over the alphabet Σ, which is allowed to be non-binary. $\mathcal{S}$ is stored locally and secretly, so we can assume that $\mathcal{S}$ is trusted.

(2) Check$(i, \mathrm{mode}, u, \mathcal{S}, O) \rightarrow (b, v, \mathcal{S}', O')$: Given index $i \in [1, n]$, the access mode mode $\in \{\mathrm{Read}, \mathrm{Write}\}$, the value to be written u ($u = $ null if mode = Read), the secret memory $\mathcal{S}$ and public memory O, return result (b, v) (b is a bit and v is the value corresponding to the i-th index) as well as the updated secret and public memories $\mathcal{S}'$ and O'.

In the above definition, $b = 1$ indicates the checker returns the correct (latest) value of location i, when the operation is a read. Also, we say the checker has query complexity q if the maximum number of public memory locations that Check accesses for any index i is q. We say the checker has secret space s if $|\mathcal{S}| \leq s$.

Definition 2.2 (Completeness of memory checker). We say that a (Σ, n, q, s) memory checker (Setup, Check) has completeness c (2/3 by default) if for all $(\mathcal{S}_0, O_0) \leftarrow$ Setup(n, Db), for all polynomial-sized set of requests $(j_1, \mathrm{mode}_1, u_1), (j_2, \mathrm{mode}_2, u_2), \ldots$ it is

$$\mathrm{Check}(j_i, \mathrm{mode}_i, u_i, \mathcal{S}_{i-1}, O_{i-1}) \rightarrow (1, v_i, \mathcal{S}_i, O_i)$$

with probability at least c. Importantly, when mode$_i$ is Read, we require v_i equal the last value written at index j_i.

Definition 2.3 (Soundness of memory checker). We say that a (Σ, n, q, s) memory checker (Setup, Check) has soundness p (1/3 by default) if for all $(\mathcal{S}_0, O_0) \leftarrow$ Setup(n, Db), for all polynomial-sized set of requests $(j_1, \mathrm{mode}_1, u_1), \ldots$ the probability that

$$\mathrm{Check}(j_i, \mathrm{mode}_i, u_i, \mathcal{S}_{i-1}, O_{i-1}) \rightarrow (1, v_i, \mathcal{S}_i, O_i)$$

and v_i is not the latest value written on index j_i is at most p, even if the contents of the public memory that Check accesses can be maliciously changed.

A query complexity lower bound. In their seminal work, Dwork et al. [18] studied the query efficiency of a memory checker and showed the following query complexity lower bound.

THEOREM 2.4 ([18]). *For a (Σ, n, q, s) deterministic and non-adaptive online memory checker, with secret space $s < n^{1-\epsilon}$ for some $\epsilon > 0$ and alphabet size $|\Sigma| \leq 2^{\text{polylog } n}$, it must be that the query complexity*

$$q = \Omega(\log n / \log \log n).$$

Read-write tradeoffs for memory checking. Dwork et al. [18] provide a trade-off with respect to query complexities of read and write operations. More specifically, for any desired logarithm base T, they show how to build a checker where the frequent operation (read or write) is inexpensive and has query complexity $O(\log_T n)$, and the infrequent operation (write or read respectively) has query complexity $O(T \log_T n)$. If $\log_T n = 1/\epsilon$ where ϵ is some constant, then the resulting complexities are $O(1/\epsilon)$ for the frequent operation and $O(n^\epsilon)$ for the infrequent operation. One of our construction will be manifesting a similar tradeoff.

2.3 Merkle trees

A Merkle tree [35] is an authenticated data structure with many applications, such as efficient state authentication in blockchain systems. In a Merkle tree, every non-leaf node stores the hash of its children, while leaf nodes store the actual values. When the value of a leaf node changes, it is necessary to update the hashes of all nodes along the path from the affected leaf to the root. The Merkle root, which is publicly accessible, can be saved in trusted local storage and be used for verifying read operations.

Merkle trees can be considered as a special case of online memory checker, where the Merkle commitment is stored in the secret memory and all the nodes in the Merkle tree are placed in the public memory. Whenever we need to read a value for index i, we fetch the nodes on the path from leaf i to the root and check if that Merkle proof is valid. Writing values is similar. The query complexities for both reads and writes are exactly $\log n$.

2.4 LevelDB

Our implementation is using LevelDB [26]. We chose LevelDB mainly because it is the backend database of choice in most influential blockchain systems. In particular, Bitcoin Core [1] and go-ethereum [20] store the blockchain metadata using a LevelDB database. LevelDB has also been used for other projects, for instance, as the backend database for Google Chrome's IndexedDB [28].

LevelDB provides a persistent key value store. Keys and values are arbitrary byte arrays. The keys are ordered according to a user-specified comparator function. Benchmarks of read/write show that sequential read/write operations outperform random read/write operations (see Table 1). These initial findings indicate that locality might be an important factor that we can leverage by scheduling the order of the data we read/write in those applications using LevelDB. We will have more discussion in Section 5.

3 LOWER BOUNDS FOR LOCALITY

We start with studying the *locality* of memory checkers. As we mentioned in the introduction, the locality of a memory checker is naturally defined as the maximum number of jumps the checker

Table 1: Benchmarks of different I/O operations in LevelDB. Note that these numbers might depend on caching as well as on additional optimizations implemented in LevelDB.

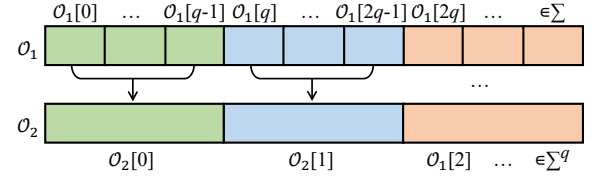| LevelDB operation	Average time (micro seconds)
Random read	16.677
Sequential read	0.476
Random write	2.460
Sequential write	1.765

Figure 1: Illustration for construction of O_2 from O_1 (S_1 and S_2 similarly).

performs on the public memory when querying any single index. We now give the formal definition.

Definition 3.1 (Locality of a memory checker). Fix a (Σ, n, q, s) memory checker C. We say C has locality t if for a query (either read or write) of any index i ($1 \leq i \leq n$), C accesses at most t memory regions in public memory, where a memory region is defined as a set of contiguous indices.

We now prove lower bounds on locality. We first prove the lower bound for general memory checkers and then we prove locality lower bounds for Merkle trees, which is a special memory checker. Finally we conclude with a lower bound on the locality of sparse Merkle trees.

3.1 Bounds for memory checkers

We now prove the lower bound on memory checking locality by giving a reduction. In particular we show that if we can build a memory checker with very small locality, we can also build a memory checker with very small query complexity, violating the lower bound from Theorem 2.4.

THEOREM 3.2. *Suppose we have a (Σ, n, q, s) memory checker C_1 where $q = O(\text{polylog } n)$ and locality is $t \leq q$, then we can construct a new $(\Sigma^q, n, q', s/q)$ memory checker C_2 where $q' \leq 2t$.*

PROOF. We construct C_2 from C_1 with the same database size n:
(1) C_2.Setup$(n, \text{Db}) \rightarrow (S_2, O_2)$:

Call C_1.Setup$(n, \text{Db}) \rightarrow (S_1, O_1)$. Construct S_2, O_2 from S_1, O_1 as follows. Let N be the total size of words in Σ contained in O_1. O_2 will contain exactly the same information as O_1, but organized in N/q words in Σ^q. The same procedure is followed for the secret memory and the size of secret memory becomes s/q. See Fig. 1.
(2) C_2.Check$(i, \text{mode}, u, S_2, O_2) \rightarrow (b, v, S_2', O_2')$:

Follow every step in the code of C_1.Check$(i, \text{mode}, u, S_1, O_1)$, except whenever C_1 sends a "read" request for $O_1[j]$, send a "read" request for $O_2[\lfloor j/q \rfloor]$ and get its "$j \mod q$"-th element; similarly

for a "write" request on $O_1[j]$, send a "read" request for $O_2[\lfloor j/q \rfloor]$, modify its "$j \bmod q$"-th element and send a "write" request for $O_2[\lfloor j/q \rfloor]$. We apply the same process for $\mathcal{S}_1$.

Now we consider complexities in C_2. Any query in C_1 accesses at most q locations in O_1 and therefore can occupy at most two contiguous words in O_2. Since each query in C_1 accesses at most t separate memory regions in O_1 (by definition of locality), the query complexity in C_2 is at most $2t$ for each index. Note also that the alphabet size of C_2, is still $|\Sigma^q| \le 2^{\text{polylog } n}$. □

This result demonstrates that the locality lower bound is almost equivalent to the query complexity lower bound, which leads to the following corollary.

COROLLARY 3.3. *For a* (Σ, n, q, s) *memory checker, with query complexity* $q = O(\text{polylog } n)$, *secret space* $s < n^{1-\epsilon}$ *for some* $\epsilon > 0$ *and alphabet size* $|\Sigma| \le 2^{\text{polylog } n}$, *it must be that the locality*

$$t = \Omega(\log n / \log \log n).$$

Remark. $q = O(\text{polylog } n)$ is a reasonable restriction. Otherwise if q is not $O(\text{polylog } n)$, then there exist memory checker constructions that can achieve constant locality. For example, view a n-element database as a $\sqrt{n} \times \sqrt{n}$ 2-dimensional array and store $\sqrt{n}$ Merkle trees (one for each row) in the public memory and $\sqrt{n}$ roots in the secret memory; querying any element can be done with locality 1 by reading a $\sqrt{n}$-sized Merkle tree.

3.2 Bounds for Merkle trees

As we discussed in Section 2.3, a Merkle tree is a special type memory checker, and, based on our results in the previous section, the locality lower bound $\Omega(\log n / \log \log n)$ applies. In this section we ask the question of whether we can prove an even stronger lower bound just for Merkle trees. For example, the textbook implementation of Merkle trees has locality $\log n$ and there does not seem to be an easy way to to re-arrange the memory locations of the Merkle tree nodes so that to reduce the locality to say, $o(\log n)$. Indeed, we prove in this section that Merkle trees admit an improved $\Omega(\log n)$ locality lower bound.

For sake of simplicity, we consider an equivalent implementation of Merkle trees where each internal node stores the node hash, as well as the left child hash and the right child hash. In this way to build the Merkle proof for a leaf x, we just need to access the nodes on the path from x to the root of the Merkle tree, and not any sibling nodes. Therefore both read and write operations must access exactly the same nodes.

Consider now a Merkle tree with $2n-1$ nodes, where the number of leaves is n and where the height of the tree is $h = \log n$. We define a natural numbering for the nodes of a Merkle tree, where 1 is the Merkle root, 2 is its left child, 3 is its right child, 4 is its leftmost grandchild, etc. Any permutation from $[2n-1]$ to $[2n-1]$ defines a way to store these nodes in memory (For example, without loss of generality, we can assume that the textbook implementation of Merkle trees uses the identity permutation.) If each node can only be stored once (i.e., no copies of the same node are allowed), we are going to show that for any permutation σ of the $2n-1$ Merkle nodes, there must be some leaf x for which accessing the path $path(x)$ from x to the root requires $\Omega(\log n)$ jumps, i.e., in σ, the nodes on $path(x)$ are partitioned into $\Omega(\log n)$ memory regions.

3.2.1 Definitions. For a fixed permutation σ of all $2n-1$ Merkle nodes of a Merkle tree of height h, we denote with $\Lambda(\sigma)$ the sum, over all leaves, of their locality, where the locality of each leaf x is the number of non-consecutive memory regions accessed when traversing the path from x to the root. We also define the following:

(1) With $\Delta(h)$ we denote the minimum value of $\Lambda(\sigma)$ over all σ;
(2) With $\delta(h)$ we denote the minimum value of $\Lambda(\sigma)$ over all σ such that the Merkle root r is stored as one end of σ, i.e., $\sigma(r) = 1$ or $\sigma(r) = 2n-1$.

We also define $\mathcal{N}_\sigma$ as the set of all neighbor pairs in σ, i.e., if σ is the permutation $\sigma = [x_1, x_2, \ldots, x_{2n-1}]$, then

$$\mathcal{N}_\sigma = \{(x_i, x_{i+1}) : 1 \le i \le 2n-2\}.$$

3.2.2 Main result and proof. In this section, we will prove the following theorem, which indicates that the locality for Merkle tree is $\Omega(\log n)$.

THEOREM 3.4. $\Delta(h) = \Theta(2^h \cdot h)$, *i.e., for any* $\sigma, \Lambda(\sigma) = \Omega(n \log n)$.

We will use the following two lemmas to prove Theorem 3.4. In the lemmas, we consider a tree of height $h \ge 1$. Also, please refer to Section 2 for the definition of variables r, r_1, r_2, a_i and b_i that appear in the lemmas below.

LEMMA 3.5. *There is one permutation* σ *such that* σ *is of the form*

$$[a_1, a_2, \ldots, b_1, b_2, \ldots, r_2, r] \text{ or } [r, r_1, a_1, a_2, \ldots, b_1, b_2, \ldots]$$

and such that $\Lambda(\sigma) = \delta(h)$.

LEMMA 3.6. *There is one permutation* σ *such that* σ *is of the form*

$$[a_1, a_2, \ldots, r_1, r, r_2, b_1, b_2, \ldots]$$

and such that $\Lambda(\sigma) = \Delta(h)$.

We will prove these lemmas later. Here, we show the proof of our main result using these lemmas.

PROOF OF THEOREM 3.4. According to Lemma 3.6, there exists a permutation σ for a tree of height h such that $\Lambda(\sigma) = \Delta(h)$ and σ is of the following form

$$[a_1, a_2, \ldots, r_1, r, r_2, b_1, b_2, \ldots]$$

Note that $\sigma_1 = [a_1, a_2, \ldots, r_1]$ is a permutation for the left subtree and $\sigma_2 = [r_2, b_1, b_2, \ldots]$ a permutation for the right subtree. From the definition of $\delta(\cdot)$, we have

$$\Lambda(\sigma_1) \ge \delta(h-1), \ \Lambda(\sigma_2) \ge \delta(h-1).$$

We can observe that it must be that $\Lambda(\sigma_1) = \delta(h-1)$, otherwise if $\Lambda(\sigma_1) > \delta(h-1) = \Lambda(\sigma_1')$ for some σ_1' then we can replace σ_1 with σ_1' in σ to get smaller $\Lambda(\sigma)$, which is a contradiction. Similarly, it must be that $\Lambda(\sigma_2) = \delta(h-1)$. Moreover, since r is next to both r_1 and r_2 in σ, we have

$$\Lambda(\sigma) = \Lambda(\sigma_1) + \Lambda(\sigma_2) \Rightarrow \Delta(h) = \delta(h-1) + \delta(h-1). \quad (1)$$

Now, according to Lemma 3.5, there is a permutation σ for a tree of height h such that $\Lambda(\sigma) = \delta(h)$ and σ is, for example, of the form

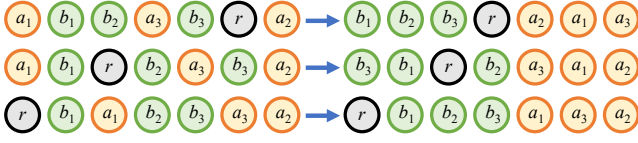$$[a_1, a_2, \ldots, b_1, b_2, \ldots, r_2, r].$$

Figure 2: Cut-and-paste. Cut-and-paste transformation. In these three examples, cut pairs and paste pairs are chosen in such a way so that the permutation is divided into two pieces such that each one contains all the nodes in one subtree.

Note that $\sigma_3 = [a_1, a_2, \ldots]$ is a permutation for the left subtree and $\sigma_4 = [b_1, b_2, \ldots, r_2]$ for the right subtree. From the definitions and based on similar arguments as above, we have

$$\Lambda(\sigma_3) = \Delta(h-1), \ \Lambda(\sigma_4) = \delta(h-1).$$

However note now that none of the nodes in the left subtree is the neighbor of r, so for queries of each leaf node from the left subtree, we need to access the root r with an additional jump. Therefore, we have the following equation

$$\Lambda(\sigma) = \Lambda(\sigma_3) + \Lambda(\sigma_4) + \frac{n}{2}$$

which is equivalent to

$$\delta(h) = \delta(h-1) + \Delta(h-1) + \frac{n}{2} = \delta(h-1) + \Delta(h-1) + 2^{h-1}. \ (2)$$

Now, we can easily compute the values of $\Delta(\cdot)$ and $\delta(\cdot)$ for small h:

$$\Delta(0) = \delta(0) = 1, \ \Delta(1) = 2, \ \delta(1) = 3.$$

We can now solve Eq. (1) and Eq. (2):

$$\Delta(h) = \frac{(h+1)2^h}{3} + \frac{2^{h+2}}{9} + \frac{2}{9}(-1)^h = \Theta(2^h \cdot h),$$

$$\delta(h) = \frac{(h+1)2^h}{3} + \frac{7 \cdot 2^h}{9} + \frac{1}{9}(-1)^{h+1}.$$

Therefore, for any σ, $\Lambda(\sigma) \geq \Delta(h) = \Omega(n \log n)$. □

3.2.3 Proof of Lemma 3.5 and Lemma 3.6.

Cut-and-paste. Before we prove Lemma 3.5 and Lemma 3.6, we introduce the notion of *cut-and-paste* for permutations. In particular, assume we have a permutation σ with a fixed $\Lambda(\sigma)$ value. We will define a cut-and-paste transformation that will output another permutation σ' whose $\Lambda(\sigma')$ value we can control. In particular, the cut-and-paste transformation takes as input k *cut pairs* $(x_1, y_1), \ldots, (x_k, y_k)$ and k *paste pairs* $(z_1, w_1), \ldots, (z_k, w_k)$ and outputs a new permutation σ' by first "cutting" at *cut pairs* and then "pasting" at *paste pairs*. *cut pairs* must be neighbors in σ and *paste pairs* must be valid cut locations from *cut pairs*.

Fig. 2 shows three examples of cut-and-paste, where, for instance, the first cut-and-paste cuts at (a_1, b_1), (b_2, a_3), (a_3, b_3) and pastes at (b_2, b_3), (a_2, a_1), (a_1, a_3). Here we show an important lemma about how locality changes after cut-and-paste.

LEMMA 3.7. *If we cut a permutation σ at $(x_1, y_1), \ldots, (x_k, y_k)$ and paste at $(z_1, w_1), \ldots, (z_k, w_k)$ to get σ', then we have*

$$\Lambda(\sigma') = \Lambda(\sigma) + \sum_{i=1}^{k} P(x_i, y_i) - \sum_{i=1}^{k} P(z_i, w_i),$$

where $P(x, y) = \min\{2^{\mathcal{H}(x)}, 2^{\mathcal{H}(y)}\}$ when x and y have at least one common descendant. Otherwise $P(x, y) = 0$.

PROOF OF LEMMA 3.7. For any leaf node l, suppose its locality in σ is k. When we cut at a pair (x_i, y_i), if x_i and y_i are both ancestors of l, then the locality of l will become $k + 1$, otherwise it stays k. When we paste at pair (z_i, w_i), if z_i and w_i are both ancestors of l the locality of l will become $k - 1$, otherwise it stays k. Clearly for any cut (x_i, y_i) and paste (z_i, w_i) (such that x_i and y_i have at least one common descendant and z_i and w_i have at least one common descendant) there are exactly $\min\{2^{\mathcal{H}(x_i), \mathcal{H}(y_i)}\}$ and $\min\{2^{\mathcal{H}(z_i), \mathcal{H}(w_i)}\}$ common descendants of (x_i, y_i) and (z_i, w_i) respectively, and therefore the result follows. □

LEMMA 3.8. *For any permutation σ, there is a cut-and-paste transformation that outputs a new permutation σ' such that*

(1) $\Lambda(\sigma') \leq \Lambda(\sigma)$;
(2) *The left half part of σ' are all from one subtree and the right half part of σ' are from another subtree (except for the root r);*
(3) *If in σ, r is on one end, then in σ', r is still on one end with the same neighbor.*

PROOF OF LEMMA 3.8. We apply the cut-and-paste transformation as follows.

(1) We cut at pairs $(x_1, y_1), \ldots, (x_k, y_k)$ in σ such that x_i and y_i belong to different subtrees;
(2) We paste at pairs $(z_1, w_1), \ldots, (z_k, w_k)$ so that the new permutation σ' has at most one neighbor pair with nodes belonging to different sub-trees (The output permutation can have zero such neighbor pairs in case the root lies exactly in the middle.)

For example, consider a Merkle tree of 4 leaves, with 3 nodes a_1, a_2, a_3 belonging to the left subtree and 3 nodes b_1, b_2, b_3 belonging to the right subtree. Assume the initial permutation σ is

$$[a_1 \ b_1 \ b_2 \ a_3 \ b_3 \ r \ a_2].$$

The aforementioned cut-and-paste transformation works as follows. First it creates three cuts, i.e.,

$$[a_1 \mid b_1 \ b_2 \mid a_3 \mid b_3 \ r \ a_2]$$

with cut pairs (a_1, b_1), (b_2, a_3) and (a_3, b_3), yielding three chunks (a_1, a_2 and $b_1 \ b_2$) eligible for move and then rearranges the permutation by moving chunks a_1 and a_3 to the right (in any order) and chunk $b_1 \ b_2$ right before b_3. The final permutation is

$$[b_1 \ b_2 \ b_3 \ r \ a_2 \ a_1 \ a_3],$$

with paste pairs (a_2, a_1), (a_1, a_3) and (b_2, b_3). Fig. 2 shows three types of such cut-and-paste transformations.

Note that based on the above transformation, the root will never be a node in a cut pair and therefore the root will never move if it is on one end of the initial permutation. Also, the final step of the transformation ensures that nodes of the same subtree lie in one side of the permutation. Therefore (2) and (3) are true.

For (1), we calculate $\Lambda(\sigma')$ using Lemma 3.7. In particular, for cut pairs $(x_1, y_1), \ldots, (x_k, y_k)$, each (x_i, y_i) contains nodes from different subtrees, so all $P(x_i, y_i) = 0$ and therefore $\sum_{i=1}^{k} P(x_i, y_i) = 0$. For

paste pairs $(z_1, w_1), \ldots, (z_k, w_k)$, it must be that $\sum_{i=1}^{k} P(z_i, w_i) \geq 0$ since $P(\cdot, \cdot)$ is always ≥ 0. Therefore,

$$\Lambda(\sigma') = \Lambda(\sigma) - \sum_{i=1}^{k} P(z_i, w_i) \leq \Lambda(\sigma),$$

and (1) also holds. □

PROOF OF LEMMA 3.5. Pick any permutation σ_0 such that $\Lambda(\sigma_0) = \delta(h)$ and r is on one end of σ_0. According to Lemma 3.8 we can apply cut-and-paste to σ_0 to get σ_1 as described in Lemma 3.8. Note that r is still on one end of σ_1. If we have either $(r, r_1) \in \mathcal{N}_{\sigma_1}$ or $(r, r_2) \in \mathcal{N}_{\sigma_1}$, then the proof is done. Now consider the case that we have neither $(r, r_1) \in \mathcal{N}_{\sigma_1}$ nor $(r, r_2) \in \mathcal{N}_{\sigma_1}$. Assume r_1 is on the right side of r_2 and σ_1 is of the form

$$\sigma_1 = [\ldots, c_1, r_1, c_2, \ldots, c_3, r],$$

where c_2, c_3 are all of height less than $h-1$ and $P(c_1, r_1) \leq 2^{h-2}$. We apply cut-and-paste to σ_1: Cut at $(c_1, r_1), (r_1, c_2), (c_3, r)$ and paste at $(c_1, c_2), (c_3, r_1), (r_1, r)$ to get

$$\sigma_2 = [\ldots, c_1, c_2, \ldots, c_3, r_1, r].$$

According to Lemma 3.7, we have

$$\Lambda(\sigma_2) \leq \Lambda(\sigma_1) + 2^{h-2} + 2^{\mathcal{H}(c_2)} + 2^{\mathcal{H}(c_3)} - 2^{\mathcal{H}(r_1)} - 2^{\mathcal{H}(c_3)} \leq \Lambda(\sigma_1).$$

Since $\Lambda(\sigma_2) \leq \Lambda(\sigma_1) \leq \Lambda(\sigma_0) = \delta(h)$ and also $\Lambda(\sigma_2) \geq \delta(h)$, we have $\Lambda(\sigma_2) = \Lambda(\sigma_0) = \delta(h)$. r_1 is now the only neighbor of r in σ_2 and therefore σ_2 is the desired permutation. □

PROOF OF LEMMA 3.6. Pick any permutation σ_0 such that $\Lambda(\sigma_0) = \Delta(h)$. Apply cut-and-paste as in Lemma 3.8 to produce σ_1. Note that in σ_1, r cannot be on one end of σ_1. If it was, e.g., if

$$\sigma_1 = [a_1, \ldots, a_2, b_1, \ldots, b_2, r],$$

then we could cut at (a_2, b_1) and paste at (r, a_1) to get σ_2. According to Lemma 3.7, it would then hold

$$\Lambda(\sigma_2) < \Lambda(\sigma_1) \leq \Lambda(\sigma_0) = \Delta(h),$$

which is not possible since $\Delta(h)$ is the minimum value. Now we only need to consider the case that r is not on one end of σ_1 and we have neither $(r, r_1) \in \mathcal{N}_{\sigma_1}$ nor $(r, r_2) \in \mathcal{N}_{\sigma_1}$. We consider the following two cases.

Case 1: σ_1 is of the following form (as in the first case in Fig. 2)

$$[a_1, \ldots, r_1, a_2, \ldots, a_3, r, b_1, \ldots, b_2, r_2, \ldots, b_3],$$

where $a_1, a_2, a_3, b_1, b_2, b_3$ are all of height less than $h-1$. Apply cut-and-paste on σ_1: cut at $(r_1, a_2), (a_3, r), (r, b_1), (b_2, r_2)$ and then paste at $(a_3, a_1), (r_1, r), (r, r_2), (b_3, b_1)$ to get

$$\sigma_2 = [a_2, \ldots, a_3, a_1, \ldots, r_1, r, r_2, \ldots, b_3, b_1, \ldots, b_2].$$

According to Lemma 3.7, we have

$$\Lambda(\sigma_2) \leq \Lambda(\sigma_1) + 2^{\mathcal{H}(a_2)} + 2^{\mathcal{H}(a_3)} + 2^{\mathcal{H}(b_1)} + 2^{\mathcal{H}(b_2)}$$
$$- 2^{\mathcal{H}(r_1)} - 2^{\mathcal{H}(r_2)} \leq \Lambda(\sigma_1).$$

Thus $\Lambda(\sigma_2) = \Delta(h)$ and σ_2 is the desired permutation.
Case 2: σ_1 is of the following form

$$[a_1, \ldots, r_1, a_2, \ldots, a_3, b_1, \ldots, b_2, r, b_3, \ldots, b_4, r_2, \ldots, b_5],$$

as in the second case in Fig. 2 (Note we can assume that r_2 is on the right side of r, otherwise we can flip $[b_1, \ldots, b_5]$ to make r_2 on

right side of r.) Here, $a_1, a_2, a_3, b_1, b_2, b_3, b_4, b_5$ are all of height less than $h-1$. Apply cut-and-paste on σ_1:

Cut at: $(r_1, a_2), (a_3, b_1), (b_2, r), (r, b_3), (b_4, r_2)$

Paste at: $(a_3, a_1), (r_1, r), (r, r_2), (b_5, b_1), (b_2, b_3)$

Get: $\sigma_2 = [a_2, \ldots, a_3, a_1, \ldots, r_1, r, r_2, \ldots, b_5, b_1, \ldots, b_2, b_3, \ldots, b_4]$

According to Lemma 3.7, we have

$$\Lambda(\sigma_2) \leq \Lambda(\sigma_1) + 2^{\mathcal{H}(a_2)} + 2^{\mathcal{H}(b_2)} + 2^{\mathcal{H}(b_3)} + 2^{\mathcal{H}(b_4)}$$
$$- 2^{\mathcal{H}(r_1)} - 2^{\mathcal{H}(r_2)} \leq \Lambda(\sigma_1).$$

Thus $\Lambda(\sigma_2) = \Delta(h)$ and σ_2 is the desired permutation. □

Remark. In Theorem 3.4, we proved that $\Delta(h)$ is $\approx \frac{1}{3} n \log n$. Also, we proved that permutation σ of Lemma 3.6 has $\Lambda(\sigma) = \Delta(h)$. In some applications, where locality is significant, arranging the nodes as in Lemma 3.6 could provide around 33% to 67% concrete reduction on the number of non-consecutive accesses, leading to better overall performance.

3.3 Bounds for Merkle trees with duplicates

In light of the lower bounds on locality discussed in the previous subsection, we can conclude that simply changing the order of the nodes cannot asymptotically reduce the locality of the textbook implementation of Merkle trees. But what if we could duplicate Merkle tree nodes? More precisely, if we allow a maximum of K copies for each node, implying that the query complexity for write may increase to $K \log n$, then what is the lower bound for the locality of reads? We now prove the following theorem.

THEOREM 3.9. *For a Merkle tree of height h (with $n = 2^h$ leaves), if we allow at most K copies for each Merkle tree node, then the sum, taken over all leaf nodes v, of v's read locality is $\Omega(h \cdot 2^h/\log K) = \Omega(n \log n/\log K)$.*

PROOF. From the definition of $\Omega(\cdot)$, it is enough to prove that there are constants c and h_0 such that the locality sum is at least

$$c \cdot \frac{h \cdot 2^h}{\log K}$$

for all $h \geq h_0$. Set c to be a fixed value in $(0, \log K/2(\log K + 2))$ and $h_0 = \log K$ (Note $K \geq 2$.) We are going to use induction on h.

For the base case, we prove that the claim holds for $\log K \leq h \leq \log 4K = 2 + \log K$. The locality is at least 2^h, which is

$$2^h > c \cdot 2^h \frac{2 + \log K}{\log K} \geq c \cdot \frac{2^h \cdot h}{\log K}.$$

Now suppose the theorem is correct for any tree of height $< h$, e.g., for a tree of height $h - \log(4K)$, the locality sum is at least

$$c \cdot \frac{(h - \log(4K)) \cdot 2^{h-\log(4K)}}{\log K} = c \cdot \frac{(h - \log(4K)) \cdot 2^h}{4K \cdot \log K}.$$

Now we consider the case involving a tree of height h. Pick any permutation (including duplicate nodes) on the disk and remove all the nodes of height greater than $h - \log(4K)$ and less than h (just "imagine" those nodes are not needed any more). Notably, this removing behavior will not cause any increase of any leaf node's read locality. Next let us analyze how we place the K copies of the root of the tree with 2^h leaves. Since there are exactly $4K$ sub-trees

of $2^h/(4K)$ leaves in this tree, at most $2K$ of those sub-trees have nodes adjacent to one of those copies, which means that at least $2K$ of those sub-trees are not adjacent to any one of those copies. Therefore, in the new permutation, the sum of read locality over all leaves is at least

$$c \cdot \frac{(h - \log(4K)) \cdot 2^h}{4K \cdot \log K} \cdot 4K + 2K \cdot \frac{2^h}{4K}$$

$$= c \cdot \frac{h \cdot 2^h}{\log K} - c \cdot 2^h \cdot \frac{2 + \log K}{\log K} + \frac{2^h}{2} > c \cdot \frac{h \cdot 2^h}{\log K}.$$

Thus the induction step is completed. □

Remark. Theorem 3.9 suggests a way to balance the locality for reads and writes. For example, suppose we are using Merkle trees to design a database system where write operations are rare and read operations are more frequent and suppose our goal is reducing the locality for reads. Moreover, we want to ensure that write operations are not overly expensive, hence we require the write complexity to be at most poly-log time, i.e., modifying at most $\log^c n$ nodes for some constant c. According to Theorem 3.9, if each node can have at most $K = \log^c n$ copies, then the average read locality for all leaves is $\Omega(\log n/\log K) = \Omega(\log n/\log\log n)$, which means the read locality for this Merkle tree (maximum read locality over leaves, see Definition 3.1) will be $\Omega(\log n/\log\log n)$.

3.4 Bounds for sparse Merkle trees

In many blockchain and transparency applications, sparse Merkle trees [2, 13] are used instead of dense Merkle trees. In this subsection, we will extend our discussion in Section 3.2 to sparse Merkle trees and study the lower bounds of their locality.

Our sparse Merkle tree model is similar to [2], where a tree has three types of nodes: root, internal nodes and leaves, and every internal nodes has exactly two children. See Figure 3 in [2] as an example. We consider a sparse Merkle tree of height h with n leaves. Similar to Section 3.2, we are going to show that for any sparse Merkle tree of height h with n leaves, its locality is $\Omega(\log_h n)$.

3.4.1 Definitions and technical outline.
For any fixed sparse tree T, we denote with $\mathrm{num}(T)$ the number of leaves in T. For any permutation σ of T, we denote with $\Lambda_T(\sigma)$ the maximum locality over all its leaves (not sum of locality). Similarly as before, we denote with $\Delta(T)$ the minimum value of $\Lambda_T(\sigma)$ over all σ, and $\delta(T)$ the minimum value of $\Lambda_T(\sigma)$ over all σ such that "T's root r is stored as one end of σ". We call $\Delta(T)$ the minimum locality of T and $\delta(T)$ the partial minimum locality of T.

It is hard to compute $\Delta(T)$ if we do not know the tree structure. However, we can try to compute the minimum value of $\Delta(T)$ over all trees of height h with n leaves. We define

$$\Delta(n, h) := \min\{\Delta(T) : T \text{ is sparse with height } h \text{ and } n \text{ leaves}\}$$

$$\delta(n, h) := \min\{\delta(T) : T \text{ is sparse with height } h \text{ and } n \text{ leaves}\}.$$

We then prove a lower bound for $\Delta(n, h)$. We also need additional definitions for our proof: $\Gamma(h, l)$ is the maximum number of leaves over all sparse trees of height h and minimum locality l, and $\gamma(h, l)$ is defined accordingly wrt the partial minimum locality, i.e.,

$$\Gamma(h, l) := \max\{\mathrm{num}(T) : \mathcal{H}(T) = h \text{ and } \Delta(T) = l\}$$

$$\gamma(h, l) := \max\{\mathrm{num}(T) : \mathcal{H}(T) = h \text{ and } \delta(T) = l\}.$$

Technical outline. We now prove that the locality of any sparse Merkle tree is $\Omega(\log_h n)$.

THEOREM 3.10. $\Delta(n, h) = \Omega(\log_h n)$.

Our proof is based on the computation of $\Gamma(h, l)$. In Section 3.4.2, we will prove some preliminary lemmas, which will be helpful to compute $\Gamma(h, l)$. Next, in Section 3.4.3, we will compute $\Gamma(h, l)$ based on the lemmas in Section 3.4.2. Finally, in Section 3.4.4, we will prove our main result (Theorem 3.10) using $\Gamma(h, l)$.

3.4.2 Preliminary lemmas.

LEMMA 3.11. For any permutation σ of a fixed tree T, there is a cut-and-paste transformation that outputs a new permutation σ' such that

(1) $\Lambda_T(\sigma') \le \Lambda_T(\sigma)$;
(2) The left half part of σ' are all from one subtree and the right half part of σ' are from another subtree (except for the root r);
(3) If in σ, r is on one end, then in σ', r is still on one end with the same neighbor.

PROOF SKETCH. Similar to the proof of Lemma 3.8, we cut-and-paste σ to get σ' (Fig. 2). The only difference here is that now $\Lambda_T(\sigma)$ is the maximum locality of all leaves. However, it is still true that cutting σ at pairs from different subtrees will not cause $\Lambda_T(\sigma)$ to increase, while pasting pairs from the same sub-tree might cause $\Lambda_T(\sigma)$ to decrease, so $\Lambda_T(\sigma') \le \Lambda_T(\sigma)$. The other two claims also follow as in proof of Lemma 3.8. □

LEMMA 3.12. For any fixed sparse tree T, there is one permutation σ such that σ is of the form

$$[a_1, a_2, \ldots, b_1, b_2, \ldots, r_2, r] \text{ or } [r, r_1, a_1, a_2, \ldots, b_1, b_2, \ldots]$$

and such that $\Lambda_T(\sigma) = \delta(T)$.

PROOF. Pick any permutation σ_0 such that $\Lambda_T(\sigma_0) = \delta(h)$ and r is on one end of σ_0. First apply cut-and-paste on σ_0 as Lemma 3.11 described to get σ_1. Now we only need to consider the case that we have neither $(r, r_1) \in \mathcal{N}_{\sigma_1}$ nor $(r, r_2) \in \mathcal{N}_{\sigma_1}$.

Since r is one end of σ_1, assume that the only neighbor of r is $a \in T_1$, and without loss of generality r is on the right end. Then move r_1 from its original place to the middle of (a, r) so that the resulting new permutation σ_2 is of the form we want.

Note that the move of r_1 will not cause the maximum locality of leaves to change, thus we have $\Lambda_T(\sigma_2) = \Lambda_T(\sigma_1) = \Lambda_T(\sigma_0) = \delta(h)$ and σ_2 is the desired permutation. □

LEMMA 3.13. For any fixed sparse tree T, there is one permutation σ such that σ is of the form

$$[a_1, a_2, \ldots, r_1, r, r_2, b_1, b_2, \ldots] \text{ or } [a_1, \ldots, b_1, \ldots, r, r_2, b_i, b_{i+1}, \ldots]$$

and such that $\Lambda_T(\sigma) = \Delta(T)$.

PROOF. Pick any permutation σ_0 such that $\Lambda_T(\sigma_0) = \Delta(h)$. First apply cut-and-paste on σ_0 as Lemma 3.11 described to get σ_1. Now we only need to consider the case that we have neither $(r, r_1) \in \mathcal{N}_{\sigma_1}$ nor $(r, r_2) \in \mathcal{N}_{\sigma_1}$.

Case 1: If σ_1 is of the same type as the first case in Fig. 2, then we just move r_1, r_2 from their original place to be the neighbors of r to get σ_2:

$$[a_1, a_2, \ldots, r_1, r, r_2, b_1, b_2, \ldots].$$

Also, the move of r_1, r_2 will not cause the maximum locality of leaves to change, thus we have $\Lambda_T(\sigma_2) = \Lambda_T(\sigma_1) = \Lambda_T(\sigma_0) = \Delta(h)$ and σ_2 is the desired permutation.

Case 2: If σ_1 is of the same type as the second case in Fig. 2, i.e., σ_1 is of form

$$[a_1, a_2, \ldots, b_1, b_2, \ldots, r, \ldots, b_j]$$

where r is surrounded by nodes in T_2 but r_2 is not its neighbor, then move r_2 from its original place to be a neighbor of r so that the resulting new permutation σ_2 is of the form we want. Again, moving r_2 will not cause the maximum locality of leaves to change, thus $\Lambda_T(\sigma_2) = \Lambda_T(\sigma_1) = \Lambda_T(\sigma_0) = \Delta(h)$ and σ_2 is what we want. □

LEMMA 3.14. $\Delta(2^h, h) = \lfloor h/2 \rfloor + 1, \delta(2^h, h) = \lfloor (h+1)/2 \rfloor + 1.$

PROOF. We prove by induction. If $h = 0$ or 1, then the claim trivially holds. Suppose the conclusion holds for $h < k$. Now we consider $\Delta(2^k, k)$ and $\delta(2^k, k)$. Note that only the full binary tree T is a tree of height k with 2^k leaves. Moreover, since T is full, we have $\Delta(T) = \Delta(2^k, k)$ and $\delta(T) = \delta(2^k, k)$.

According to Lemma 3.12, without loss of generality, there is a permutation σ of form

$$[a_1, a_2, \ldots, b_1, b_2, \ldots, r_2, r]$$

and $\Lambda_T(\sigma) = \delta(T) = \delta(2^k, k)$. When computing $\delta(2^k, k)$, i.e., the maximum locality over all leaves, we need to consider the maximum locality in each sub-tree. In T_1, the maximum locality is $\Delta(2^{k-1}, k-1) + 1$, and in T_2, it is $\delta(2^{k-1}, k-1)$. Thus we have

$$\delta(2^k, k) = \max\{\delta(2^{k-1}, k-1), \Delta(2^{k-1}, k-1) + 1\}$$
$$= \max\{\lfloor k/2 \rfloor + 1, \lfloor (k-1)/2 \rfloor + 2\} = \lfloor (k+1)/2 \rfloor + 1.$$

According to Lemma 3.13, there is also a permutation σ of form

$$[a_1, a_2, \ldots, r_1, r, r_2, b_1, b_2, \ldots] \text{ or } [a_1, \ldots, b_1, \ldots, r, r_2, b_i, b_{i+1}, \ldots]$$

and $\Lambda_T(\sigma) = \Delta(T) = \Delta(2^k, k)$. If σ is of the first form, then

$$\Delta(2^k, k) = \delta(2^{k-1}, k-1) = \lfloor k/2 \rfloor + 1;$$

If σ is of the second form, then

$$\Delta(2^k, k) = \max\{\Delta(2^{k-1}, k-1), \Delta(2^{k-1}, k-1)+1\} = \lfloor (k+1)/2 \rfloor + 1.$$

Now, as $\Delta(2^k, k) = \Delta(T)$ is the minimum locality over all permutations, σ must be of the first form. Thus $\Delta(2^k, k) = \lfloor k/2 \rfloor + 1$. □

This lemma shows that $\Gamma(h, l)$ is only defined over $h \geq 2l - 2$. This is because when $h \leq 2l - 3$, $\Delta(2^h, h) \leq l - 1$, even the full binary tree of height h has no locality-l leaves, so $\Gamma(h, l)$ is only meaningful when $h \geq 2l - 2$. A similar claim shows that $\gamma(h, l)$ is defined over $h \geq 2l - 3$.

3.4.3 Computing $\Gamma(h, l)$. First, based on Lemma 3.14, we have

$$\begin{aligned} \Gamma(h, l) &= 2^h, \text{ if } h = 2l - 2 \text{ or } h = 2l - 1 \\ \gamma(h, l) &= 2^h, \text{ if } h = 2l - 3 \text{ or } h = 2l - 2 \end{aligned} \quad (3)$$

We then discover the recurrence relation on $\gamma(h, l)$ and $\Gamma(h, l)$ so that we can compute $\Gamma(\cdot, \cdot)$ from the initial cases in Eq. (3).

Fix a tree T which has $\mathcal{H}(T) = h$, $\delta(T) = l$ and $\text{num}(T) = \gamma(h, l)$. When $h \geq 2l - 2$, according to Lemma 3.12, without loss of generality, there is a permutation σ of form

$$[a_1, a_2, \ldots, a_i, b_1, b_2, \ldots, r_2, r]$$

and $\Lambda_T(\sigma) = \delta(T)$. Note that $\Lambda_{T_1}([a_1, a_2, \ldots, a_i]) \leq l - 1$, $\mathcal{H}(T_1) = h - 1$ and $\text{num}(T_1) \leq \Gamma(h-1, l-1)$, it must be that $\text{num}(T_1) = \Gamma(h-1, l-1)$, otherwise $\text{num}(T_1) < \Gamma(h-1, l-1)$ then we can find a subtree with $\Gamma(h-1, l-1)$ leaves to substitute T_1 and make $\text{num}(T)$ increase, which is impossible. Similarly, it must be that $\text{num}(T_2) = \gamma(h-1, l)$. Hence, we have

$$\gamma(h, l) = \Gamma(h-1, l-1) + \gamma(h-1, l), \ h \geq 2l - 2. \quad (4)$$

Here we require $h \geq 2l - 2$ because as discussed before, $\gamma(h, l)$ is defined over $h \geq 2l - 3$.

For the recurrence relation on $\Gamma(h, l)$, we prove the following lemma by induction:

LEMMA 3.15. When $h \geq 2l-1$, we have $\Gamma(h, l) = 2\gamma(h-1, l)$, $\Gamma(h-1, l) \leq \Gamma(h, l) \leq 2\Gamma(h-1, l)$ and $\Gamma(h-1, l) \geq \gamma(h-1, l)$.

PROOF. We prove the statement by two-dimensional induction.
Initial step 1: For any $l \geq 1$, when $h = 2l - 1$, according to Eq. (3), $\Gamma(h, l) = 2^{2l-1} = 2 \cdot 2^{h-1} = 2\gamma(h-1, l)$, $\Gamma(h-1, l) = 2^{2l-2} < \Gamma(h, l) = 2^{2l-1} = 2\Gamma(h-1, l)$, $\gamma(h-1, l) = \gamma(2l-2, l) = 2^{2l-2} = \Gamma(h-1, l)$. The statement is correct.
Initial step 2: When $l = 1$, we compute from scratch and we have $\Gamma(h, 1) = 2$, $\gamma(h, 1) = 1$ for any $h \geq 1$. The statement is still correct.
Induction step: For any $s > 1, t \geq 0$, suppose the statement is correct for any pair of (h, l) such that

$$1 \leq l < s, \quad 2l - 1 \leq h \leq 2l + t$$
$$\text{or} \qquad l = s, \quad 2l - 1 \leq h \leq 2l + t - 1,$$

then we are going to prove that the statement is also correct for $l = s$ and $h = 2l + t = 2s + t$.

Now fix a tree T which has $\mathcal{H}(T) = h = 2s + t$, $\Delta(T) = l = s$ and $\text{num}(T) = \Gamma(h, l)$. According to Lemma 3.13, there is a σ of form

$$[a_1, a_2, \ldots, r_1, r, r_2, b_1, b_2, \ldots]$$

or

$$[a_1, a_2, \ldots, b_1, b_2, \ldots, r, r_2, b_i, b_{i+1}, \ldots]$$

and $\Lambda_T(\sigma) = \Delta(T)$. If σ is of the first form, then

$$\Gamma(h, l) = 2\gamma(h-1, l).$$

If it is of the second form, then

$$\Gamma(h, l) = \Gamma(h-1, l) + \Gamma(h-1, l-1).$$

Since $\Gamma(h, l)$ is the maximum number of leaves of trees satisfying that $\Delta(T) = l$ and $\mathcal{H}(T) = h$, we have

$$\Gamma(h, l) = \max\{2\gamma(h-1, l), \Gamma(h-1, l) + \Gamma(h-1, l-1)\}.$$

Now from induction step, we have

$$\begin{aligned} 2\gamma(h-1, l) &= 2(\gamma(h-2, l) + \Gamma(h-2, l-1)) \\ &= 2\gamma(h-2, l) + 2\Gamma(h-2, l-1) \\ &= \Gamma(h-1, l) + 2\Gamma(h-2, l-1) \\ &\geq \Gamma(h-1, l) + \Gamma(h-1, l-1). \end{aligned}$$

Thus $\Gamma(h, l) = 2\gamma(h-1, l) = \Gamma(h-1, l) + 2\Gamma(h-2, l-1)$.

Since $\Gamma(h-1, l) = \Gamma(h-2, l) + 2\Gamma(h-3, l-1)$ and $\Gamma(h-2, l) \leq \Gamma(h-1, l)$, $\Gamma(h-3, l-1) \leq \Gamma(h-2, l-1)$ (from induction step), we have $\Gamma(h-1, l) \leq \Gamma(h, l)$.

From induction step, we have $\Gamma(h-2, l) \geq \gamma(h-2, l)$, so $\Gamma(h-1, l) \geq \Gamma(h-2, l) + \Gamma(h-2, l-1) \geq \gamma(h-2, l) + \Gamma(h-2, l-1) = \gamma(h-1, l)$.

Finally, $\Gamma(h, l) = 2\gamma(h-1, l) \leq 2\Gamma(h-1, l)$.

This completes the induction step. □

Now we can combine Lemma 3.15 with Eq. (4), when $h \geq 2l$, and then we have

$$\begin{aligned}\Gamma(h,l) &= 2\gamma(h-1,l) = 2(\gamma(h-2,l) + \Gamma(h-2,l-1)) \\ &= 2\gamma(h-2,l) + 2\Gamma(h-2,l-1) \\ &= \Gamma(h-1,l) + 2\Gamma(h-2,l-1).\end{aligned} \quad (5)$$

If we solve Eq. (3) and Eq. (5) for $\Gamma(h,l)$ when $h \geq 2l$, we have

$$\Gamma(h,l) = 2^h - 2^l \cdot \sum_{i=0}^{h-2l} 2^{h-2l-i} \binom{l+i-1}{i}. \quad (6)$$

3.4.4 Proof of Theorem 3.10.

PROOF. First of all, we can easily prove by induction that $\Gamma(h,l)$ is non-decreasing when l is increasing: first compute $\Gamma(h,1)$ and $\Gamma(h,2)$ from scratch, and then use Eq. (5) to do the induction step. With this non-decreasing property, if $\Gamma(h,l) < n$, then $\Delta(n,h)$ must be larger than l otherwise it will contradict the definition of $\Gamma(h,l)$. Hence, we have

$$\Delta(n,h) = \min\{l \ : \ \Gamma(h,l) \geq n\}. \quad (7)$$

Note that this formula is a quick way to compute the concrete value of $\Delta(n,h)$ from $\Gamma(h,l)$.

Next, we are going to show that $\Gamma(h, \log_{2h} n) < n$ when $h > 8$. First, when $h > 8$, we have $2 < (2h)^{1/4}, n \leq 2^h < (2h)^{h/4}$, and then

$$h > 4 \log_{2h} n. \quad (8)$$

From generalized binomial theorem [30, 44], we have the following formula

$$\sum_{i=0}^{\infty} \frac{\binom{l+i-1}{i}}{2^i} = 2^l. \quad (9)$$

Then with Eq. (6) and Eq. (9), in order to prove $\Gamma(h, \log_{2h} n) < n$ when $h > 8$, it is equivalent to show that

$$\sum_{i=h-2l+1}^{\infty} \frac{\binom{l+i-1}{i}}{2^i} < \frac{n}{2^{h-l}}, \text{ where } n = (2h)^l, \ h > 8.$$

Now we take a careful look at $\binom{l+i-1}{i}$. When $i > h - 2l + 1$ and $h > 4l$ we have (from Eq. (8))

$$\frac{l+i-1}{i} < \frac{l+(h-2l+1)-1}{h-2l+1} = \frac{h-l}{h-2l+1} < \frac{3}{2}.$$

Then when $i > h - 2l + 1$ we have

$$\begin{aligned}\binom{l+i-1}{i} &= \frac{l \cdot (l+1) \cdots (l+i-1)}{i!} \\ &= \binom{l+(h-2l+1)-1}{h-2l+1} \prod_{j=h-2l+2}^{i} \frac{l+j-1}{j} \quad (10) \\ &< \binom{h-l}{l-1}\left(\frac{3}{2}\right)^{i-(h-2l+1)}.\end{aligned}$$

Therefore, from Eq. (10) we have

$$\begin{aligned}\sum_{i=h-2l+1}^{\infty} \frac{\binom{l+i-1}{i}}{2^i} &< \sum_{i=h-2l+1}^{\infty} \frac{\binom{h-l}{l-1}}{2^{h-2l+1}} \cdot \left(\frac{3}{2 \cdot 2}\right)^{i-(h-2l+1)} \\ &= \frac{\binom{h-l}{l-1}}{2^{h-2l+1}} \sum_{j=0}^{\infty} \left(\frac{3}{4}\right)^j = \frac{2\binom{h-1}{l-1}}{2^{h-2l}} < \frac{2(h-l)^{l-1}}{2^{h-2l}} \\ &= \frac{4(2(h-l))^{l-1}}{2^{h-l}} < \frac{(2h)^l}{2^{h-l}} = \frac{n}{2^{h-l}}.\end{aligned}$$

Now we have proved $\Gamma(h, \log_{2h} n) < n$ when $h > 8$. From Eq. (7), we know that when $h > 8$

$$\Delta(n,h) > \log_{2h} n = \frac{\log n}{1 + \log h} > \frac{1}{2} \log_h n,$$

which means $\Delta(n,h) = \Omega(\log_h n)$. □

4 MERKLE TREE CONSTRUCTIONS WITH IMPROVED LOCALITY

In this section, we present constructions for Merkle trees with improved locality exploiting two ideas: in Section 4.1, we present a family of constructions of full Merkle trees with *duplication* to trade off read/write performance; in Section 4.2, we present a locality-aware construction of 16-ary sparse Merkle trees that are widely used in blockchain systems such as Ethereum [45] as authenticated key-value stores. We will evaluate the performance of these constructions in the next section.

4.1 Full Merkle trees with adjustable read/write performance

Our lower bound in Section 3 precludes the possibility of simultaneously improving read and write locality, so the best one could hope for is to trade write locality (and complexity) for read locality or vice versa based on application needs. Fortunately, such trade-off typically makes sense because many applications have an unbalanced read/write workload. For example, 60% queries to the Merkle Patricia Trie in Ethereum are read in 2021 winter [33], in which case improving read locality could be beneficial overall, even at the cost of increased write complexity. To this end, we introduce a family of Merkle tree constructions that allow adjustable trade-offs between read and write complexity by *duplicating* tree nodes.

A natural way to reduce read locality is to store paths in continuous regions on the persistent storage (which we generically call a disk, while the actual implementation may utilize a different storage medium, such as a database). In an extreme scenario, the read locality can be reduced to 1 at the cost of duplicating height-h nodes 2^h times (i.e., the root is duplicated n times, its children $n/2$ times, and so on). This naive construction has two notable issues. First, it incurs a $\log n$-factor storage overhead which could be significant for big n (e.g., 32× for a typical choice $n = 2^{32}$). Second, write becomes prohibitively expensive since changing a leaf node requires modifying $O(n)$ nodes (the root alone has n copies).

To strike a better balance between read and write overhead, we first show how to avoid the $\log n$-factor storage overhead. Then, we show how to apply the same idea recursively to improve the read locality. Finally, putting the two ideas together, we present DupTree++ that allows adjustable read/write performance.
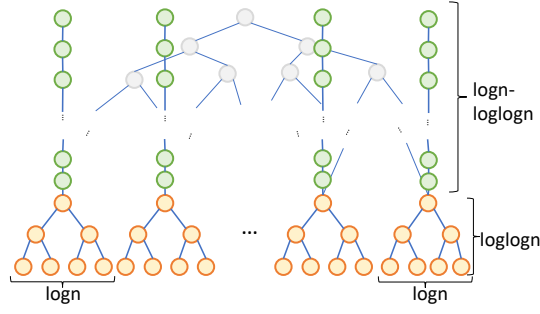
Figure 3: Duplicating nodes with a height above $\log \log n$. This construction has $O(\log \log n)$ read locality and $O(n/\log n)$ write complexity.
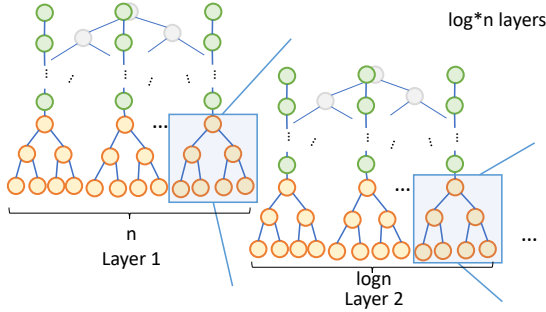

Figure 4: DupTree+ with $O(\log^* n)$ read locality and $O(n/\log n)$ write complexity.

4.1.1 Basic constructions.

To avoid the $\log n$-factor storage overhead, we modify the above naive construction to only store the top part of each path in continuous regions (with duplication), while the lower part is stored without any duplication. We call this construction DupTree. As illustrated in Fig. 3, specifically, DupTree duplicates internal nodes with a height higher than $\log \log n$ so that the top $(\log n - \log \log n)$ nodes of each path are stored in a contiguous region. As a result, an internal node of height $h \geq \log \log n$ has $2^{h - \log \log n}$ copies. The construction of DupTree is formally specified in Fig. 11.

The read locality of DupTree is $O(\log \log n)$, because all of the ancestors with height greater than $\log \log n$ can be read by one jump, and the lower part can be read with at most $O(\log \log n)$ jumps. The space complexity of DupTree is

$$O((\log n - \log \log n) \cdot (n/2^{\log \log n}) + 2^{\log \log n}(n/2^{\log \log n})) = O(n) .$$

To update a leaf node, all copies of its ancestors need to be updated, thus the write complexity is $O(n/2^{\log \log n}) = O(n/\log n)$.

We can further improve the read locality from $O(\log \log n)$ to $O(\log^* n)$ at the cost of a slight ($\log^* n$ factor) increase of space complexity, where $\log^* n$ is iterated logarithm defined as

$$\log^* n := \begin{cases} 0 & \text{if } n \leq 1; \\ 1 + \log^*(\log n) & \text{if } n > 1. \end{cases}$$

We call this construction DupTree+. The key observation is that we can apply the above construction recursively (see Fig. 4 for an
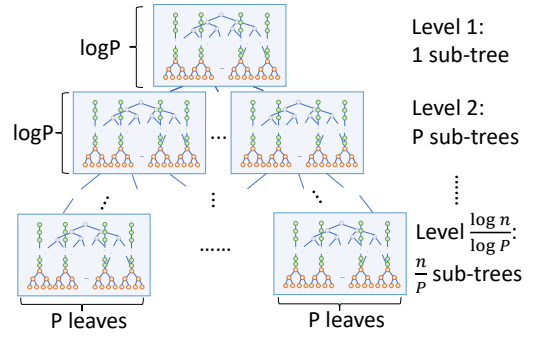

Figure 5: DupTree++ with $O(\log n/\log \log n)$ read locality and $O(\log^2 n/(\log \log n)^2)$ write complexity. We ignore $\log^*$ term which in practice is no larger than 5.

illustration). Consider the read locality for reading a given leaf (and its path to the root) in the previous construction. While the upper part of the path can be read in one jump, the lower part may take up to $O(\log \log n)$ jumps. Now, we recursively apply the previous construction to each subtree. Specifically, in the second layer of recursion, we have $n/\log n$ sub-trees of size $O(\log n)$. We duplicate the first $(\log \log n - \log \log \log n)$ nodes in each path, which requires an additional $O(n)$ disk space[1]. Now, each leaf node in the original tree needs only two jumps to access ancestors of height higher than $\log \log \log n$. Keep iterating the process and the recursion ends after at most $\log^* n$ layers. The resulting construction has $O(\log^* n)$ read locality, $O(n \log^* n)$ disk space, and $O(n/\log n)$ write complexity.

Space reduction. We can reduce the space to linear if we apply DupTree+ to a tree from its root to nodes of height $\log^* n$. Letting $m = n/2^{\log^* n}$, the resulting construction has $O(\log^* m + \log^* n) = O(\log^* n)$ locality for read, $O(m \log^* m + n) = O(n)$ disk space and also $O(m/\log m) = O(n/\log n)$ write complexity.

4.1.2 DupTree++ with tunable read/write performance.

While the read locality of DupTree and DupTree+ is minimal (even when $n = 2^{65536}$, we still have $\log^* n = 5$), its write complexity is unattractively large. In this section, we present DupTree++ that has an adjustable parameter so one can tune the performance to meet the real-world workload.

Consider a tree of $n = 2^h$ leaves and a parameter P ($2 \leq P \leq n$) assumed to be a power of 2. The high-level idea is to divide the tree into subtrees with P leaves and store each of them using DupTree+. Specifically, we partition the tree into $\log n/\log P$ levels (see Fig. 5) and each level consists of subtrees of P leaves. For each of these subtrees, we store it as DupTree+, yielding $O(\log^* P)$ read locality, $O(P/\log P)$ write complexity, and $O(P \log^* P)$ space complexity (c.f. Section 4.1.1). Therefore, to sum up, the entire Merkle tree has $O(\log^* P \cdot \log n/\log P)$ read locality, $O(P \log n/\log^2 P)$ write complexity, and $O(n \log^* P)$ space complexity. If we pick $P = \log n$ and ignore the $\log^*$ term (as we said in Section 4.1.1), the resulting construction has $O(n)$ space complexity, $O(\log n/\log \log n)$ read locality, and $O(\log^2 n/(\log \log n)^2)$ write complexity.

---

[1]to be precise, $n/\log n \cdot (\log \log n - \log \log \log n) \cdot (\log n/2^{\log \log \log n}) = O(n)$

Fig. 12 formally specifies the construction of DupTree++. In order to simplify the description, the specification omits the optimization to remove the $\log^* n$ factor from the disk space so the space complexity is $O(n \log^* n)$.

Interestingly, the above trade-off between read locality and write complexity is very similar to the read-write trade-off of query complexity in [18] (see Section 2.2).

4.2 PrefixTree: Construction for sparse Merkle trees with versions

While DupTree++ is designed for full binary Merkle trees, sparse Merkle trees are widely used in large-scale key-value stores [2, 13, 45] for their scalability. We introduce an approach to constructing locality-aware sparse Merkle trees by arranging tree nodes that are frequently accessed together in close proximity on disk. Our starting point is a simplified version of the Merkle Patricia Trie construction used in Ethereum.
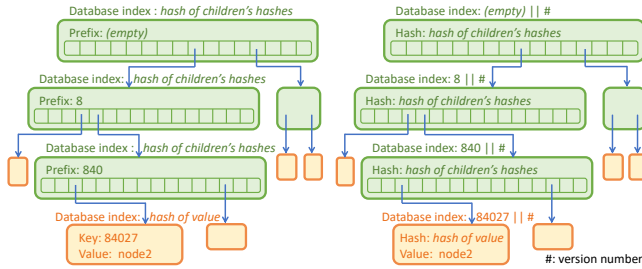

Figure 6: Simplified version of Merkle Patricia Trie (left) and PrefixTree (right).

Merkle Patricia Trie (MPT) In Ethereum [45], smart contract states are stored in an authenticated key-value store implemented by an MPT, a 16-ary prefix tree (or trie) augmented with membership proofs like Merkle trees. To store the $(key, value)$ pair, the key determines the location of the tree node in which $value$ will be stored, allowing for efficient search. Specifically, we consider a simplified version of MPT where each branch node stores 16 hash pointers to its children (possibly null if some children are empty), and the common prefix of the keys of all descendants. Refer to Fig. 6 to see this simplified version of MPT. Searching for a given key in an MPT is the same as searching in a regular prefix tree, starting from the root node and traversing down the tree by following the hash pointers determined by key. This scheme supports versioning so that write operations will create new nodes while keeping existing nodes intact. This allows for efficient rollback needed in blockchain systems. In Ethereum, each block header has a new root hash from which a new version of the MPT can be accessed.

Our construction: PrefixTree. In Ethereum's implementation of MPT, each node is stored in a key-value database (LevelDB, specifically) with its hash as the index (to avoid confusion, we use indices to refer to the keys of the data entries in the database) and the serialized node as the value. As a result, tree nodes are essentially stored in random places in the database, leading to high read and write locality. Based on this observation, we propose PrefixTree

(Fig. 13) that stores nodes with its key prefixes as the index, so that nodes on a Merkle path are stored in adjacent locations.

This change also requires us to change the node data structure. To support versioning, the index of every node is appended with a version number (e.g., the block number). Instead of hash pointers, each branch node stores 16 prefix strings of its children, and the hash of the current node. See Fig. 6 for structure of PrefixTree.

Note that tree nodes in PrefixTree may have a smaller storage footprint since prefixes may be shorter than hash pointers. On the other hand, in order to generate a Merkle proof, in PrefixTree we also need to access the siblings of the nodes on the path to get their hashes, which means the number of nodes to read is concretely higher than MPT. Therefore, whether PrefixTree will outperform MPT is not immediately clear and that is the goal of our experiments in Section 5.2. As a heads-up, when comparing PrefixTree and MPT, we artificially pad the prefix to the same length as hash pointers so that the performance will not benefit from smaller tree node size.

5 EVALUATION

In this section, we report on the implementation details of our constructions and the result of the performance evaluation. We implemented the algorithms of DupTree++ and PrefixTree in about 1600 lines of C++ (available online[2]). For all constructions, we use LevelDB as backend storage (which is how Ethereum, the widely used smart contract blockchain, stores its Merkle Patricia Tries).

All experiments are executed on a server with Intel(R) Xeon(R) Platinum 8380 CPU @ 2.30GHz with 80 cores, 128GB of RAM, and 7TB of SSD.

5.1 DupTree++

5.1.1 Implementation details.

We implemented DupTree++ for binary Merkle trees. For a Merkle tree of height h with 2^h leaves, each node is a $\langle key, value \rangle$ pair where key is the hash of $value$. For internal nodes, $value$ is the concatenation of its two children's keys; for leaf nodes, $value$ stores the data users will query and update.

To store tree nodes in contiguous storage regions, we leverage the fact that key-value pairs in LevelDB are sorted by keys within each level. Therefore when storing a tree node $\langle k, v \rangle$ in LevelDB, we prepend a string id before k (i.e., v is stored with index $id\|k$). By setting appropriate ids, we can control the order of the nodes stored in LevelDB.

Choosing parameters. In our implementation, DupTree++ is used to optimize read efficiency: a Merkle tree is stored on persistent storage according to DupTree++, but when receiving a read operation for one leaf, DupTree++ only reads one copy of all its ancestors. The read locality is therefore $o(\log n))$. To write a node, we need to modify all duplication of its ancestors.

We pick $P = \log n$ so that asymptotically we have around $\log P = \log \log n$ times improvement in read locality while the extra cost in write is at most $O(P/\log^2 P)$ times.

Also, we make some other slight changes on how we organize every $O(P) = O(\log n)$-size sub-tree. In real-world applications, the number of leaves of a Merkle tree roughly lies in the range from 2^{20} to 2^{40} [19]. Thus $\log P \approx \log^* P \approx 5$, and then we naturally

---

[2]https://github.com/wangnick2017/DupTree

12

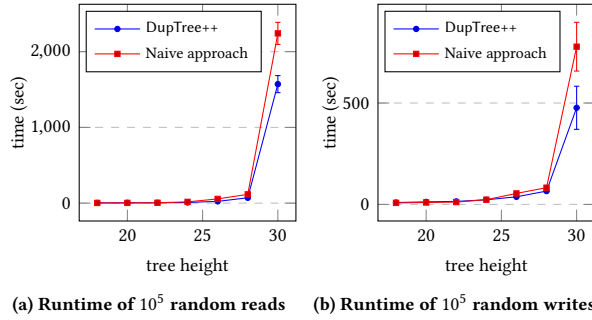(a) Runtime of 10^5 random reads   (b) Runtime of 10^5 random writes

Figure 7: Performance comparison of DupTree++ and a standard Merkle tree.

choose to use the version with $P \log P$ disk space and $O(1)$ read locality for each $O(P)$-size sub-tree. Furthermore, since all the leaf nodes in the sub-trees have read locality only 1, we can put those $\log P$ consecutive nodes in one database entry to improve the read performance. In fact, this method will not introduce too much extra write cost at all since the new (big) database entries are only about $\log P \approx 5$ times as the original (small) ones.

In summary, our implementation is optimized for reads, which has $O(\log n/\log \log n)$ read locality, $O(\log^2 n/\log \log n)$ write complexity, and $O(n \log \log n)$ disk space.

5.1.2 Comparison with naive approach.

We compare the performance of our implementation with a standard implementation of Merkle tree as baseline where nodes are stored in LevelDB with no duplication, in which the locality for both read and write is $O(\log n)$.

Experimental setting. We run experiments for a Merkle tree of height h with 2^h leaves, where h goes from 18 to 30. For each h, we randomly pick 10^5 leaf nodes to conduct read/write experiments for both our approach and naive approach.

To benchmark a read operation of a given leaf node, we read from LevelDB relevant nodes for a given leaf node. To benchmark a write operation, we update relevant nodes in LevelDB, but we should carefully consider the effect of "BatchWrite" in LevelDB. LevelDB allows users to buffer write requests and later make multiple changes in one atomic batch write. According to the official benchmark [25], a single batch of N writes may be significantly faster than N individual writes. Therefore, for both schemes, we have one BatchWrite after 10^4 updates on random leaf nodes so that both schemes enjoy the same speed up.

Results and analysis. We repeat our experiments 20 times and report the average and standard deviation (as error bars) in Fig. 7, for read and write respectively. The coefficient of variation is up to 6.4% on reads, up to 15.4% on writes for baseline, and up to 7.1% on reads, up to 22.2% on writes for DupTree++.

Our approach always has better performance on reads. Since DupTree++ may have more amount of data to read, this means that a smaller read locality can indeed reduce the cost of I/O operations. Although our improvement is not significant (around 1.2× to 1.6× better), the advantage increases with n. As our read locality is $O(\log \log n)$ times smaller than the naive approach, we conjecture

that our construction will exhibit a notable advantage for larger trees. One possible reason why our construction is not $\log \log n > 4$ times faster is that the caching technique in LevelDB partially mitigates the overhead of the naive construction.

For write performance, we observe that our construction is slightly ($\sim 1.3×$) slower than the naive approach for small trees but has equal or even better (up to 1.6×) performance for large trees. However, we believe that in applications heavily rely on reading data, our construction will have better performance among all (read/write) operations.

The runtime of both schemes grows rapidly when $n > 2^{28}$. We conjecture that this rapid growth comes when LevelDB needs to jump multiple times over an extremely large database on the disk.

5.2 PrefixTree

PrefixTree is designed to support large key-value stores such as the world state in the smart contract platform Ethereum. To evaluate its performance, we replay real-world Ethereum workload (read/write queries extracted from Ethereum transactions) and compare the runtime of PrefixTree and standard implementation of Merkle Patricia Trie (thereafter called the baseline). We note that Go Ethereum (geth) has a similar implementation called PathScheme that uses prefix strings as database indices [20]. However, this locality-friendly version was not put into use in practice and its performance improvement is not clear as it also changes the tree node data structure.

Padding of prefixes. As explained in Section 4.2, the concrete read/write complexity of PrefixTree differs from the baseline. To make the comparison completely unbiased, we artificially pad every prefix string to (at least) 64-nibble to match the hash pointer length. We confirmed that PrefixTree with padding always reads and writes more bits than the baseline in our experiments, as shown in Fig. 8. This means that the performance advantage of PrefixTree is due to improved locality.

Real-world workload. The workload we used to benchmark PrefixTree is the real-world Merkle Patricia Trie queries extracted from transactions in the first 4 million blocks (from July 2015 to July 2017). We stopped at 4 million blocks as the trend is clear. After processing 4 million blocks, on average, there are over 100K read/write queries in each block, and the sparse Merkle tree is 60GB and has roughly 100 million nodes.

Experimental settings. We replay the read/write queries in every 100k blocks to both implementation and measure the time for processing these queries. Same as Section 5.1, we set the write batch size of LevelDB to be 10^4 for both schemes to reduce the running time of experiments. Since both schemes have the exact same tree structure and thus the numbers of writes to LevelDB, they enjoy equal speed up from batching, so the comparison remains fair.

Results and analysis. We repeat our experiments 5 times and report the average (we stop at 5 because the standard deviation is low) result in Fig. 9. Each data point (x, y) shows the runtime for processing 100k blocks between block number $(x - 1) \cdot 10^5 + 1$ to block number $x \cdot 10^5$.

Fig. 9 shows that PrefixTree performs much better after about 2.4 million blocks, and the advantage is increasing as the tree grows. At 4 million blocks, PrefixTree performs around 3× faster than the baseline. Note that we artificially added padding to ensure the

(a) Number of bits read from disk during the experiments

(b) Number of bits written to disk during the experiments

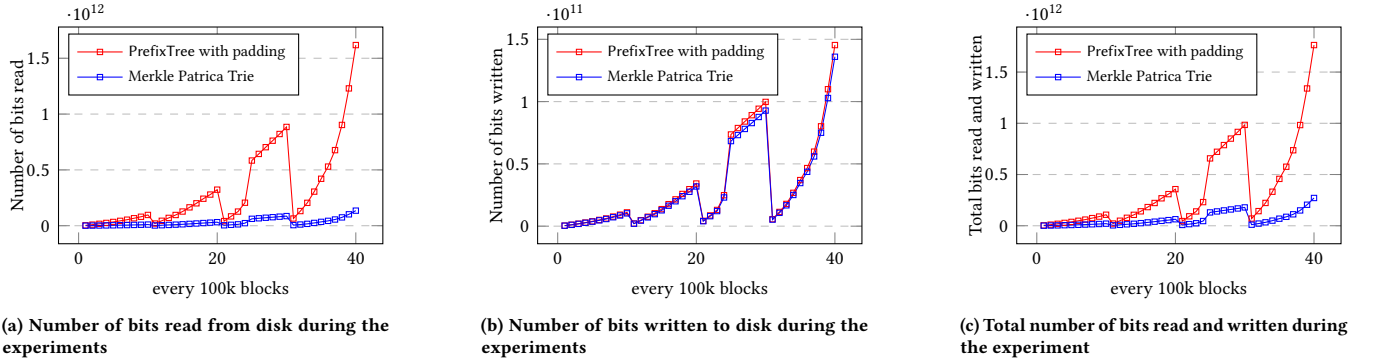(c) Total number of bits read and written during the experiment

Figure 8: By padding the prefix strings, PrefixTree always incurs more I/O overhead (measured as the number of bits read/written) than the baseline, confirming that the performance advantage of PrefixTree is only due to improved locality. For each data point, the y value is for processing 100k blocks between block number $(x - 1) \cdot 10^5 + 1$ to block number $x \cdot 10^5$.
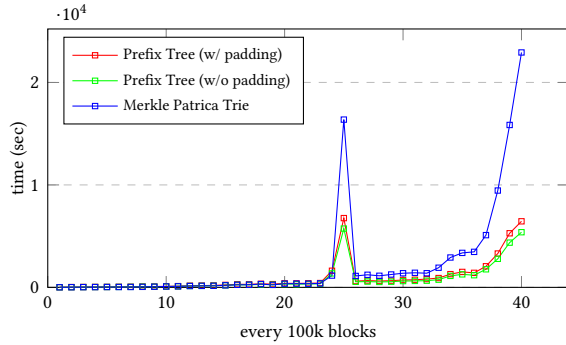

Figure 9: Time to process real-world Ethereum workload in every 100k blocks.

comparison is far, but the padding is not necessary in practice. Without padding, PrefixTree is around 4× faster than the baseline at 4 million blocks. We believe this advantage comes from the fact that PrefixTree has low locality and benefits from the caching of LevelDB. More specifically, according to the official document of LevelDB [27], the unit of disk transfer and caching is a "block" (approximately 4096 uncompressed bytes by default) and adjacent indices will usually be placed in the same block (or adjacent blocks). Therefore, in Merkle Patricia Trie, each query involves loading several separate blocks while in PrefixTree each query might only touch one or more adjacent blocks, which has a significant impact on the resulting performance. We note that the abnormal peak between block number 2.3 million and 2.6 million is caused by DOS attacks to Ethereum[3] that happened in 2016.

In conclusion, this experiment shows that the locality of sparse Merkle trees significantly impacts its performance on real-world workloads. Moreover, the improvement from locality even outweighs the slightly increased read and write complexity.

6 CONCLUSION

In this paper, we introduced and studied the concept of *locality* in memory checkers. We first proved the lower bound of locality

on general memory checking and then we showed some stronger bounds of locality on Merkle trees. Next, we gave some constructions to meet those lower bounds and conducted several experiments to show that our constructions can indeed improve performance.

Future work. Our future work can be partitioned into two categories: theoretical exploration and experimental validation.

Better lower bound of locality on general online memory checkers. Since we have shown that proving lower bound for locality is almost as hard as proving the bound for query complexity, we could prove stronger bounds for query complexity of general online memory checkers. We have the following conjecture on query complexity.

CONJECTURE 1. *For a (Σ, n, q, s) deterministic and non-adaptive online memory checker, with secret space $s < n^{1-\epsilon}$ for some $\epsilon > 0$, it must be that the query complexity $q = \Omega(\log n / \log \log |\Sigma|)$.*

Macrobenchmarks and comparison with other efficient authenticated storage systems. It would be very interesting to run and compare our constructions on other key-value storage systems such as RocksDB [41] or even directly on SSD. In addition, we can compare our construction with other other efficient authenticated storage systems [33, 37, 38] for maintaining a huge set of key-value pairs, or apply our construction to some other tree-structure applications.

ACKNOWLEDGMENTS

REFERENCES

[1] 2023. Bitcoin Developer Network. https://bitcoindev.network/.
[2] Arvind Arasu, Badrish Chandramouli, Johannes Gehrke, Esha Ghosh, Donald Kossmann, Jonathan Protzenko, Ravi Ramamurthy, Tahina Ramananandro, Aseem Rastogi, Srinath Setty, Nikhil Swamy, Alexander van Renen, and Min Xu. 2021. FastVer: Making Data Integrity a Commodity. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 89–101. https://doi.org/10.1145/3448016.3457312
[3] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A survey of attacks on ethereum smart contracts (sok). In *Principles of Security and Trust: 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on*

Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings 6. Springer, 164–186.

[4] Mihir Bellare, Alexandra Boldyreva, and Adam O'Neill. 2007. Deterministic and efficiently searchable encryption. In *Advances in Cryptology-CRYPTO 2007: 27th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2007. Proceedings 27.* Springer, 535–552.

[5] Josh Benaloh and Michael de Mare. 1994. One-Way Accumulators: A Decentralized Alternative to Digital Signatures. In *EUROCRYPT '93*, Tor Helleseth (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 274–285.

[6] Manuel Blum, Will Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. 1994. Checking the correctness of memories. *Algorithmica* 12 (1994), 225–244.

[7] Manuel Blum, William S. Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. 1991. Checking the Correctness of Memories. In *32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1-4 October 1991.* IEEE Computer Society, 90–99.

[8] Dan Boneh, Benedikt Bünz, and Ben Fisch. 2019. Batching Techniques for Accumulators with Applications to IOPs and Stateless Blockchains. In *CRYPTO'19*.

[9] Matteo Campanelli, Dario Fiore, Nicola Greco, Dimitris Kolonelos, and Luca Nizzardo. 2020. Incrementally Aggregatable Vector Commitments and Applications to Verifiable Decentralized Storage. In *Advances in Cryptology − ASIACRYPT 2020*, Shiho Moriai and Huaxiong Wang (Eds.). Springer International Publishing, Cham, 3–35.

[10] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Cătălin Roşu, and Michael Steiner. 2014. Dynamic searchable encryption in very-large databases: Data structures and implementation. *Cryptology ePrint Archive* (2014).

[11] David Cash and Stefano Tessaro. 2014. The Locality of Searchable Symmetric Encryption. In *EUROCRYPT*. Springer, 351–368. https://doi.org/10.1007/978-3-642-55220-5_20

[12] Dario Catalano and Dario Fiore. 2013. Vector Commitments and Their Applications. In *Public-Key Cryptography - PKC 2013 - 16th International Conference on Practice and Theory in Public-Key Cryptography, Nara, Japan, February 26 - March 1, 2013. Proceedings.* 55–72.

[13] Melissa Chase, Apoorvaa Deshpande, Esha Ghosh, and Harjasleen Malvai. 2019. SEEMless: Secure End-to-End Encrypted Messaging with Less Trust. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 1639–1656. https://doi.org/10.1145/3319535.3363202

[14] Alexander Chepurnoy, Charalampos Papamanthou, and Yupeng Zhang. 2018. Edrax: A Cryptocurrency with Stateless Transaction Validation. https://eprint.iacr.org/2018/968.

[15] Ioannis Demertzis, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2018. Searchable encryption with optimal locality: Achieving sublogarithmic read efficiency. In *Advances in Cryptology–CRYPTO 2018: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part I 38.* Springer, 371–406.

[16] Peter J. Denning. 2005. The Locality Principle. *Commun. ACM* 48, 7 (jul 2005), 19–24. https://doi.org/10.1145/1070838.1070856

[17] Thaddeus Dryja. 2019. Utreexo: A dynamic hash-based accumulator optimized for the Bitcoin UTXO set. https://eprint.iacr.org/2019/611.

[18] Cynthia Dwork, Moni Naor, Guy N. Rothblum, and Vinod Vaikuntanathan. 2009. How Efficient Can Memory Checking Be?. In *Theory of Cryptography*, Omer Reingold (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 503–520.

[19] Ethereum. 2023. Etherscan. https://etherscan.io/.

[20] Ethereum. 2023. go-ethereum. https://github.com/ethereum/go-ethereum.

[21] Ethereum. 2023. Optimistic Rollups. https://ethereum.org/en/developers/docs/scaling/optimistic-rollups. Accessed: 2023-05.

[22] Ethereum. 2023. Zero-Knowledge Rollups. https://ethereum.org/en/developers/docs/scaling/zk-rollups/. Accessed: 2023-05.

[23] Zhenhuan Gao, Yuxuan Hu, and Qinfan Wu. 2021. Jellyfish Merkle Tree. (2021).

[24] Eran Gilad, Edward Bortnikov, Anastasia Braginsky, Yonatan Gottesman, Eshcar Hillel, Idit Keidar, Nurit Moscovici, and Rana Shahout. 2020. EvenDB: optimizing key-value storage for spatial locality. In *Proceedings of the Fifteenth European Conference on Computer Systems.* 1–16.

[25] Google. 2011. LevelDB Benchmarks. http://www.lmdb.tech/bench/microbench/benchmark.html.

[26] Google. 2018. LevelDB. https://github.com/google/leveldb. Accessed: 2023-04.

[27] Google. 2018. LevelDB - Key Layout. https://github.com/google/leveldb/blob/main/doc/index.md#key-layout. Accessed: 2023-08.

[28] Google. 2023. Chromium README. https://chromium.googlesource.com/chromium/src/+/HEAD/third_party/leveldatabase/README.chromium/.

[29] Sergey Gorbunov, Leonid Reyzin, Hoeteck Wee, and Zhenfei Zhang. 2020. Pointproofs: Aggregating Proofs for Multiple Vector Commitments. In *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM, 2007–2023.

[30] David Guichard. 2017. *An Introduction to Combinatorics and Graph Theory.* Whitman College.

[31] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. 2010. Constant-Size Commitments to Polynomials and Their Applications. In *ASIACRYPT'10*.

[32] Polygon Labs. 2022. Introducing Plonky2. https://polygon.technology/blog/introducing-plonky2.

[33] Chenxing Li, Sidi Mohamed Beillahi, Guang Yang, Ming Wu, Wei Xu, and Fan Long. 2023. LVMT: An Efficient Authenticated Storage for Blockchain. (2023).

[34] Jiangtao Li, Ninghui Li, and Rui Xue. 2007. Universal Accumulators with Efficient Nonmembership Proofs. In *Applied Cryptography and Network Security*, Jonathan Katz and Moti Yung (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 253–269.

[35] Ralph C. Merkle. 1988. A Digital Signature Based on a Conventional Encryption Function. In *CRYPTO '87*, Carl Pomerance (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 369–378.

[36] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33 (1996), 351–385.

[37] Soujanya Ponnapalli, Aashaka Shah, Souvik Banerjee, Dahlia Malkhi, Amy Tai, Vijay Chidambaram, and Michael Wei. 2021. RainBlock: Faster Transaction Processing in Public Blockchains.. In *USENIX Annual Technical Conference.* 333–347.

[38] Pandian Raju, Soujanya Ponnapalli, Evan Kaminsky, Gilad Oved, Zachary Keener, Vijay Chidambaram, and Ittai Abraham. 2018. mLSM: Making Authenticated Storage Faster in Ethereum.. In *HotStorage.*

[39] Srinath Setty. 2020. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In *Advances in Cryptology–CRYPTO 2020: 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17–21, 2020, Proceedings, Part III.* Springer, 704–737.

[40] Dawn Xiaoding Song, David Wagner, and Adrian Perrig. 2000. Practical techniques for searches on encrypted data. In *Proceeding 2000 IEEE symposium on security and privacy. S&P 2000.* IEEE, 44–55.

[41] Meta Open Source. 2023. RocksDB. https://rocksdb.org/.

[42] Shravan Srinivasan, Alexander Chepurnoy, Charalampos Papamanthou, Alin Tomescu, and Yupeng Zhang. 2022. Hyperproofs: Aggregating and Maintaining Proofs in Vector Commitments. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/usenixsecurity22/presentation/srinivasan

[43] Alin Tomescu. 2020. *How to Keep a Secret and Share a Public Key (Using Polynomial Commitments).* Ph. D. Dissertation. Massachusetts Institute of Technology, Cambridge, MA, USA.

[44] Wikipedia contributors. 2023. Binomial theorem - Wikipedia, the free encyclopedia. 2023. https://en.wikipedia.org/wiki/Binomial_theorem. Accessed: 2023-07.

[45] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

A   CONSTRUCTION DETAILS

- MerkleTreeInit($n, [v_1, \ldots, v_n]$) $\rightarrow T_M$:

  Given an array of n values, this subroutine returns a Merkle tree with n values as leaves, represented as an array of $2n - 1$ nodes. All nodes have the format $<k,v>$ where k is the hash of v; for internal nodes, v is the concatenation of its children's ks, and for leaf nodes, v is some v_i.

- SubTree(i, T_M) $\rightarrow T'$:

  This subroutine returns the sub-tree in T_M with root i, i.e., a subarray of T_M, including node i and all the descendants of i in the tree.

- SubTreeDistance(i, d, T_M) $\rightarrow T'$:

  This subroutine returns the sub-tree with root i and 2^d leaves, i.e., a subarray of T_M, including node i and all the descendants of i that has distance at most d to i in the tree.

Figure 10: Subroutines used in other constructions.

- BasicInit(n, δ, T_M) $\rightarrow T$:

  // T_M is a standard Merkle tree with n leaves built with MerkleTreeInit. δ is a constant such that if the number of leaves is smaller than δ, we use the naive approach to store the tree.

  if $n \leq \delta$ **then**
      return T_M
  else
      $T \leftarrow [\,]$
      for each node i in T_M of height $\log \log n$ **do**
              ▷ There are exactly $n/\log n$ such nodes
        $T \leftarrow T :: $ BasicInit($\log n, \delta,$ SubTree(i, T_M))
        Append i's $(\log n - \log \log n)$ ancestors to T, nodes with smaller height first
      end for
      return T
  end if

- BasicWrite(i, v, T):

  if $n \leq \delta$ **then**   ▷ T is a normal Merkle tree
      Write v into the *value* of node i and update the hashes in all its ancestors;
  else
      $k \leftarrow \frac{|T|}{n/\log n}; j \leftarrow \lfloor \frac{i-1}{\log n} \rfloor$
      BasicWrite($i - j \cdot \log n, v, T[jk+1 : (j+1)k - (\log n - \log \log n)]$)
      Update the hashes in $T[(j+1)k - (\log n - \log \log n) + 1 : (j+1)k]$
      for $h \leftarrow \log \log n + 1$ to $\log n$ **do**
        There are $2^{h - \log \log n}$ copies of the height-h ancestor of i; update the hashes in all of them.
      end for
  end if

- BasicRead(i, T) $\rightarrow$ [$\log n$ nodes to form a Merkle proof]:

  if $n \leq \delta$ **then**   ▷ T is a normal Merkle tree
      Read and return node i and all its ancestors;
  else
      $k \leftarrow \frac{|T|}{n/\log n}; j \leftarrow \lfloor \frac{i-1}{\log n} \rfloor$
      return BasicRead($i - j \cdot \log n, T[jk + 1 : (j+1)k - (\log n - \log \log n)]$) :: $T[(j+1)k - (\log n - \log \log n) + 1 : (j+1)k]$
  end if

Figure 11: The construction of DupTree. Subroutines are defined in Fig. 10.

- GeneralInit$(n, P, \delta, T_M) \rightarrow T$:
  - **if** $n \leq \delta$ **then**
    - **return** T_M
  - **else**
    - $T \leftarrow []$
    - **for** $i \leftarrow 1$ to $\log n / \log P$ **do**
      - **for each** $j \in T_M$ of height $\log n - (i-1) \log P$ **do**
        - $T \leftarrow T :: \text{BasicInit}(P, \delta, \text{SubTreeDistance}(j, \log P, T_M))$
      - **end for**
    - **end for**
    - **return** T
  - **end if**
- GeneralWrite(i, v, T):
  - **if** $n \leq \delta$ **then**　　　　　▷ T is a normal Merkle tree
    - Write v into the *value* of node i and update the hashes in all its ancestors;
  - **else**
    - $k \leftarrow \frac{|T|}{(n-1)/(P-1)}$
    - $j \leftarrow \frac{n/p-1}{p-1} + \lfloor \frac{i-1}{p} \rfloor + 1$
    - $v' \leftarrow v$
    - **for** $j \geq 1$ **do**
      - BasicWrite$(P, v', T[(j-1)k+1 : jk])$
      - $v' \leftarrow T[jk]$'s *key*
      - $j \leftarrow \lfloor \frac{j-2}{P} \rfloor + 1; i \leftarrow \lfloor \frac{i-1}{P} \rfloor + 1$
    - **end for**
  - **end if**
- GeneralRead$(i, T) \rightarrow [\log n$ nodes to form a Merkle proof$]$:
  - **if** $n \leq \delta$ **then**　　　　　▷ T is a normal Merkle tree
    - Read and return node i and all its ancestors;
  - **else**
    - $k \leftarrow \frac{|T|}{(n-1)/(P-1)}$
    - $R \leftarrow []$
    - $j \leftarrow \frac{n/p-1}{p-1} + \lfloor \frac{i-1}{p} \rfloor + 1$
    - **for** $j \geq 1$ **do**
      - $R \leftarrow R :: \text{BasicRead}((i-1)\%P+1, T[(j-1)k+1 : jk])$
      - $j \leftarrow \lfloor \frac{j-2}{P} \rfloor + 1; i \leftarrow \lfloor \frac{i-1}{P} \rfloor + 1$
    - **end for**
    - **return** R
  - **end if**

Figure 12: The construction of DupTree++. Subroutines are defined in Fig. 10 and Fig. 11

Node structure in PrefixTree:
- $key : (prefix; version)$
- $value : (hash; \{key_i\}_{i \in [1,16]}; text)$

- Init$(l) \rightarrow T$:
  - $list \leftarrow [], version \leftarrow 0, T \leftarrow []$
  - Set T's maximum key length to be l
  - **return** T
- Update$(k, t, list, T)$:
  - $list.\text{append}(k, t)$
- Commit$(list, version, T)$:
  - $version \leftarrow version + 1$
  - $stack \leftarrow []$
  - $stack.\text{push}(T.root$ with new $version)$
  - **for** $(k, t) \in list$ **do**
    - Find $p \in stack$ with longest common prefix with k
    - **while** p has a child p' with longer common prefix with k **do**
      - $stack.\text{push}(p'$ with new $version)$
      - $p \leftarrow p'$
    - **end while**
    - **if** $p.prefix = k$ **then**
      - $p.text \leftarrow t$
    - **else**
      - Determine k should lie in p's i-th child
      - **if** p's i-th child is empty **then**
        - $l \leftarrow$ new leaf node for (k, t)
        - $stack.\text{push}(l)$
      - **else**
        - $k' \leftarrow$ common prefix of k and p's i-th child
        - $l \leftarrow$ new leaf node for (k, t)
        - $n \leftarrow$ new internal node with k' as key, l and p's original i-th child as children
        - $stack.\text{push}(n)$
        - $stack.\text{push}(l)$
      - **end if**
    - **end if**
  - **end for**
  - **while** $stack$ is not empty **do**
    - $n \leftarrow stack.\text{pop}()$
    - compute hash of n and write n to the disk
  - **end while**
- Read$(k, T) \rightarrow [$nodes to form a Merkle proof$]$:
  - $proof \leftarrow []$
  - $p \leftarrow T.root$
  - **while** $p.prefix \neq k$ **do**
    - read $value$ of p from disk
    - append keys of siblings in $value$ to $proof$
    - Determine k should lie in p's i-th child
    - $p \leftarrow p$'s i-th child
  - **end while**
  - **return** $proof$

Figure 13: The construction of PrefixTree. When it comes a new block, we first call Update to record each update from the transactions, and then call Commit to compute new nodes and write changes to the disk.