
# LVMT: An Efficient Authenticated Storage for Blockchain

Chenxing Li, *Shanghai Tree-Graph Blockchain Research Institute;* Sidi Mohamed Beillahi, *University of Toronto;* Guang Yang and Ming Wu, *Shanghai Tree-Graph Blockchain Research Institute;* Wei Xu, *Tsinghua University;* Fan Long, *University of Toronto*

https://www.usenix.org/conference/osdi23/presentation/li-chenxing

This paper is included in the Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-34-2

Open access to the Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation is sponsored by


# LVMT: An Efficient Authenticated Storage for Blockchain

Chenxing Li Sidi Mohamed Beillahi[†] Guang Yang Ming Wu Wei Xu[‡] Fan Long[†]

*Shanghai Tree-Graph Blockchain Research Institute*

*University of Toronto*[†] *Tsinghua University*[‡]

## Abstract

Authenticated storage access is the performance bottleneck of a blockchain, because each access can be amplified to potentially $O(\log n)$ disk I/O operations in the standard Merkle Patricia Trie (MPT) storage structure. In this paper, we propose a multi-Layer Versioned Multipoint Trie (LVMT), a novel high-performance blockchain storage with significantly reduced I/O amplifications. LVMT uses the authenticated multipoint evaluation tree (AMT) vector commitment protocol to update commitment proofs in constant time. LVMT adopts a multi-layer design to support unlimited key-value pairs and stores version numbers instead of value hashes to avoid costly elliptic curve multiplication operations. In our experiment, LVMT outperforms the MPT in real Ethereum traces, delivering read and write operations six times faster. It also boosts blockchain system execution throughput by up to 2.7 times.

## 1 Introduction

Blockchains that provide decentralized, robust, and programmable ledgers at an internet scale have recently gained increasing popularity across various domains, including financial services, supply chain, and entertainment. For example, smart contracts built on blockchain systems now manage digital assets worth tens of billions of dollars [3].

Early classical blockchain systems like Bitcoin [36] and Ethereum [17] have serious performance bottlenecks in their consensus protocols, which limit the system throughput at under 30 transactions per second. Nevertheless, recent technique evolutions on consensus and peer-to-peer network protocols [8,22,23,26,29,31,33,35,37,44,45,51,52] have driven the achievable blockchain throughput to more than thousands of transactions per second. Consequently, transaction execution, which is dominated the storage access, has emerged as the new system bottleneck. Our investigation (see Sec. 6) shows that 81% of transaction execution time is consumed at the storage layer.

This inefficiency in the blockchain storage layer originates from the requirement for *authentication*. A standard permission-less blockchain system has two types of blockchain nodes: the full nodes and the light nodes. A full node synchronizes and executes all transactions, maintaining the blockchain ledger state. A light node (client) only synchronizes the block headers, excluding transactions and the blockchain ledger state. Blockchain ledger states take the form of key-value pairs. When a light node needs to ascertain the value of a given key, it queries a full node. However, since blockchain nodes are permissionless, light nodes should not trust the responses from full nodes. Therefore, the blockchain protocol requires the block proposer to compute a commitment (termed the *state root*) for the latest ledger state and insert it into the proposed block header. A block header with an incorrect commitment is deemed invalid. When responding to the queries from light nodes, a full node can generate proofs corresponding to the commitments to convince the queriers. This leads to the naming of the ledger state as *authenticated*.

Typically, authenticated storage employs the Merkle Patricia Trie (MPT) [5] structure, a specific variant of the Merkle tree. Each leaf node in an MPT stores a value, and the path from the root to the leaf node corresponds to the key of the stored value. Each inner node in the MPT stores the crypto hash of the concatenated contents of all its children. The MPT's root hash serves as the commitment of the blockchain state for authentication.

Unfortunately, this authentication comes with a heavy performance price. Modifying a key-value pair in the state requires an MPT to update hashes of all nodes along the path from the corresponding leaf node to the root. If not cached, each state update operation could be amplify to $O(\log n)$ disk I/O operations, where $n$ represents the storage size. Note that even a basic payment transaction involves at least two ledger

state updates, – decreasing the sender's balance and increasing the receiver's. As the throughput of recent blockchains approaches thousands of transactions per second, it is not surprise that storage becomes the new bottleneck.

This paper presents LVMT, a novel high-performance authenticated storage framework with significantly reduced disk I/O amplifications. LVMT achieves high efficiency by integrating a multi-level Authenticated Multipoint evaluation Tree (AMT) and a series of append-only Merkle trees. AMT is a cryptographic vector commitment scheme that can update commitment (i.e., the hash root) in constant time [50]. Despite its constant commitment update time, there are several key challenges to address when incorporating AMT into the LVMT design.

The first challenge arises from the expensive elliptic curve multiplication operations employed by the AMT commitment update algorithm. A naive approach would paradoxically result in a slower state update operation on the AMT than the MPT, despite the theoretically reduced amplification. LVMT addresses this challenge with its novel *key-versioned-value* design. It assigns each key a version, incrementing as the value evolves. Rather than storing key-value pairs in the AMT, LVMT employs AMT to keep key-version pairs and uses Merkle Trees to maintain an append-only authenticated list of *key-version-value triples*. Thus, every update in LVMT results in an increment of the stored version within the AMT. Since the AMT algorithm multiplies a precomputed elliptic curve point with the difference between the old value and the new value (i.e., one for a version increment) during a commitment update, LVMT effectively eliminates the expensive multiplication. Also, because the key-version-value triple list is append-only, LVMT only needs to construct these Merkle Trees once during the block commit time, and therefore the process is very efficient.

The second challenge emerges from AMT's limitation in supporting the necessary bit-depth for blockchain state keys. An AMT with $k$-bit key-space requires public parameters with $2^k$ elliptic curve points. To enable efficient update, the AMT also requires pre-computation and caching of elliptic curve points proportional to the public parameters' size. Even for a modest 32-bit key-space, the precomputed metadata size would exceed 256 GB, which is untenable, given that blockchain ledger keys typically comprise 256 bits. To address this challenge, LVMT operates with a novel *multi-level multi-slot structure*, integrating multiple AMTs. Each AMT in this structure has a 16-bit key-space, and the structure can automatically generate a sub-AMT on the next level to accommodate keys-version pair with collided prefix. Since collisions are rare after the first level and creating sub-AMT will make subsequent access more expensive, LVMT also makes each entry in AMTs contain five slots. Therefore expansion to the next level only occur when more than five collisions arise.

The third challenge lies in the costly maintenance of proof generation metadata. While updating the root hash for AMT incurs constant time, maintaining the proof generation metadata still requires $O(\log n)$ time and triggers the same degree of I/O amplifications as MPT. LVMT confronts this issue with a *proof sharding technique*, which distributes the proof generation metadata to multiple nodes. In LVMT, each full node only maintains the proof generation metadata for a shard of the blockchain state (e.g., keys sharing the same 4-bit prefix). Our observation reveals that there are typically thousands of full nodes in a production blockchain, and it's unnecessary for all nodes to maintain proof generation capabilities for all key-value pairs in the total state. Even sharded, for any part of the state, there will still be enough nodes serving proof generation requests from light clients. Within the current Ethereum ecosystem, most light nodes access full nodes from specialized providers, such as Infura, who operate several full nodes to balance query workload. By maintaining proof shards across their nodes, these providers can efficiently generate proof for any key. Note that unlike other sharding designs [18, 29, 34, 51, 53], our proof sharding does not alter the essential obligation of each full node to synchronize and validate blocks, process all transactions, and accurately maintain the state root, thereby preserving security.

We have implemented LVMT [1] and integrated it into Conflux [2, 33], an open-sourced high-performance blockchain production with smart contract support. We evaluated LVMT against OpenEthereum's MPT implementation, RainBlock's MPT structure [40], and LMPTs [20], considering both standalone read/write workload and end-to-end blockchain processing tasks. Our results show that LVMT achieves up to 10x higher throughput on random state read/write operations. When integrated end-to-end with a high-performance blockchain, LVMT achieves up to 2.7x higher throughput for simple payment transactions and up to 2.1x higher throughput for ERC20 [41] token transfer transactions. This boost in performance stems from the considerable reduction in disk I/O amplifications. In terms of amplification, LVMT performs up to 4.1x better than MPT on read operations and up to 8.2x better on write operations.

## 2 Background

In this section, we recall some background on cryptographic concepts that our system builds on. In particular, we introduce the cryptographic building blocks of the Authenticated Multipoint evaluation Tree (AMT) [50], an efficient vector commitment protocol.

**Notations:** We denote $[n]$ as the integers in $\{x \in \mathbb{Z}^+ | 1 \leq x \leq n\}$. $\mathbb{G}$ signifies an elliptic curve group and symbols in upper cases like $G, P$ represent elements in the elliptic curve groups. $\mathbb{Z}_p$ refers to an additive group with order $p$.

## 2.1 Authenticated Storage in Blockchain

In a standard permission-less blockchain system, blockchain nodes can be distinguished into two types: full nodes and light nodes. A full node synchronizes and executes all transactions, maintaining the blockchain ledger state accordingly. A light node (client) synchronizes only the block headers, excluding transactions and blockchain ledger state.

When a full node proposes a new block, it is required to execute transactions in that block and incorporate the commitment of the post-execution ledger state into the block header. The node keeps a write-back cache during transaction execution, committing all modifications to the storage after executing all transactions in a block. The authenticated storage needs to provide two interfaces to the execution engine:

- $\mathsf{Get}(\mathsf{k}) \rightarrow \mathsf{v}$: Retrieves the value $\mathsf{v}$ associated a given key $\mathsf{k}$.
- $\mathsf{Set}(\{(\mathsf{k},\mathsf{v})_i\}, e) \rightarrow \mathsf{comm}$: Flushes a series of key-value pairs $(\mathsf{k},\mathsf{v})$ to the storage with block number $e$, obtaining the commitment $\mathsf{comm}$ of the ledger state after changes.

When a light node wants to know the value of a specific key, it queries a full node, expecting a response of the value along with proof with respect to the ledger commitment. The light client examines whether the commitment exists within the set of verified valid commitments, then checks the validity of the associated proof. So the authenticated storage must provide two algorithms for proof generation and verification:

- $\mathsf{Respond}(\mathsf{k}) \rightarrow (\mathsf{v}, \pi, \mathsf{comm})$: Returns the value $\mathsf{v}$ of key $\mathsf{k}$ with proof $\pi$ with respect to the most recent commitment $\mathsf{comm}$.
- $\mathsf{Verify}(\mathsf{k}, \mathsf{v}, \pi, \mathsf{comm}) \rightarrow \mathsf{true/false}$: Validates the response from the full node.

## 2.2 Elliptic Curve Group

The *elliptic curve group* plays a fundamental role in various cryptographic protocols. This group conducts an additive operation over points on an elliptic curve, such as $\left\{(x,y) \in \mathbb{Z}_q^2 \mid y^2 = x^3 + x + 7\right\}$, where $q$ is a large prime number. An infinite point is included as the identity element. The operation $a \cdot P$ represents $P$ added to itself $a$ times within the group, where $a$ is a positive integer, and $P$ is a point on the curve. An elliptic curve group is characterized by a *starting point* $G$, from which a sequence of points $G, 2 \cdot G, 3 \cdot G, \cdots$ can be generated. If the elliptic curve group is cryptographically secure, this sequence exhibits the following properties:

1. $n \cdot G$ is periodic in $n$, with the period being a large prime integer $p$, i.e., $n \cdot G = (n+p) \cdot G$;

2. For a randomly selected $n$, deriving $n$ from $n \cdot G$ is computationally unfeasible.

## 2.3 KZG Commitment

Kate et al. proposed KZG polynomial commitment protocol [28], enabling someone to commit a polynomial function $f$ to a commitment, and prove the value $f(x)$ of any given position $x$ with respect to that commitment.

The KZG commitment protocol is built on a bilinear map. Consider $G_1$ and $G_2$ as the starting points of two elliptic curve groups $\mathbb{G}_1, \mathbb{G}_2$ respectively, each with the same group order $p$. The bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ is homomorphic such that the equation $e(a \cdot G_1, b \cdot G_2) = ab \cdot e(G_1, G_2)$ holds for any $a, b \in \mathbb{Z}_p$. Here, $\mathbb{G}_T$ denotes another group of the same order $p$. BLS12-381 [14] from BLS families [9] and BN254 [11] from BN families [10] are widely-used deployed systems implementing bilinear maps. The groups $\mathbb{G}_1$ and $\mathbb{G}_2$ are elliptic curve groups of order $p$, and $G_1$ and $G_2$ are their perspective starting points.

For a given polynomial function $f : \mathbb{Z}_p \rightarrow \mathbb{Z}_p$ of degree $n$, the KZG commitment assumes a series of public parameters $\tau \cdot G_1, \tau^2 \cdot G_1, \tau^3 \cdot G_1, \cdots, \tau^n \cdot G_1$ in a trusted setup and commits function $f$ to $C := f(\tau) \cdot G_1$. The public parameters are generated by a trusted party using a random $\tau$, which is forgotten after generation. Secure multi-party computation protocols [15, 16, 25] enable multiple participants to collaboratively generate these public parameters, ensuring that no participant can ascertain the exact value of $\tau$.

For any index $i \in \mathbb{Z}_p$, the expression $x - i$ should divide $f(x) - f(i)$. This suggests that $h_i(x) := \frac{f(x)-f(i)}{x-i}$ is indeed a polynomial. Hence, the proof $\pi$ of $f(i)$ is defined as $h_i(\tau) \cdot G_1$. Given that $h_i(x)$ is a polynomial, the prover can compute the coefficients of $h_i(\tau)$. Thus, $h_i(\tau) \cdot G_1$ forms a linear combination of the public parameters with known coefficients. The prover can compute it in a short time. A verifier, querying $i$ with answer $y = f(i)$ and proof $\pi := h_i(\tau) \cdot G_1$, can verify the proof by checking if

$$e(\pi, (\tau - i) \cdot G_2) = e(C - y \cdot G_1, G_2).$$

If the proof $\pi$ is correctly constructed, the check must pass because

$$\begin{aligned} e(\pi, (\tau - i) \cdot G_2) &= (h(\tau) \cdot (\tau - i)) \cdot e(G_1, G_2) \\ &= e((f(\tau) - f(i)) \cdot G_1, G_2) \\ &= e(C - y \cdot G_1, G_2). \end{aligned}$$

If $f(i) \neq y$, $h(x)$ becomes a fraction, making it difficult to find a proper proof without knowing $\tau$. Kate et al. proved the binding property of this protocol [28].

The KZG commitment also supports the proof of a batch of positions. To prove that $f(x)$ equals to 0 at a set of positions $S$, the proof $\pi$ is constructed by $\frac{f(\tau)}{\prod_{i \in S}(\tau - i)} \cdot G_1$.

A vector commitment scheme can be built with KZG commitment by converting a vector $\vec{a}$ to a polynomial function $f$ by Lagrange interpolation. Formally, for an input vector $\vec{a}$ with $n$ elements, the interpolated function $f$ is defined by $f(x) = \sum_{i=1}^{n} a_i \cdot I_{i,n}(x)$, where $a_i$ is the $i$-th element of $\vec{a}$ and $I_{i,n}(x)$ is a Lagrange function that satisfies $I_{i,n}(i) = 1$ and $I_{i,n}(x) = 0$ for $x \neq i$ and $1 \leq x \leq n$.

When updating the value at position $i$ from $a_i$ to $a_i'$, the corresponding commitment $C$ can be simply updated to

$$C' := C + (a_i' - a_i) \cdot I_{i,n}(\tau) \cdot G_1. \quad (1)$$

If the prover caches results $I_{i,n}(\tau) \cdot G_1$ for all $i$, updating commitment requires only one multiplication and one addition on the elliptic curve $\mathbb{G}_1$, which takes $O(1)$ time.

## 2.4 Authenticated Multipoint Evaluation Tree

Although the KZG commitment enables constant-time updates to the commitment $C$, it requires $O(n)$ time to construct a proof for a given position or to maintain proofs for all positions. In a blockchain system, where the vector being committed to is frequently changing, the KZG commitment cannot generate proofs efficiently for queries with arbitrary indices $i$.

To address this issue, Alin et al. proposed the Authenticated Multipoint evaluation Trees (AMT) commitment protocol [50], which maintains auxiliary information of size $O(n \log n)$ and can generate a proof in $O(\log n)$ time.

Consider an example with $n = 8 = 2^3$. For an input vector $\vec{a}$ with eight elements, AMT computes its Lagrange interpolation $f(x)$ which satisfies $f(i) = a_i$ for $1 \leq i \leq 8$. The function $f(x)$ is then partitioned into two functions $f_0(x)$ and $f_1(x)$. In the subset $x \in [8]$, $f_1(x)$ mirrors $f(x)$ for even $x$ and is zero otherwise, while $f_2(x)$ mirrors $f(x)$ for odd $x$ and is zero otherwise. For values of $x$ outside this subset, $f_1(x)$ and $f_2(x)$ are determined by Lagrange interpolation. Consequently, $f(x)$ can be re-expressed as $f(x) = f_0(x) + f_1(x)$. AMT continues to subdivide $f_0(x)$ recursively into two functions: $f_{0,0}(x)$ and $f_{0,1}(x)$. Here, $f_{0,0}$ mirrors $f(x)$ for $x \in \{4,8\}$, and $f_{0,1}(x)$ mirrors $f(x)$ for $x \in \{2,6\}$. This recursive process of partitioning generates a full binary tree, where each node corresponds to a function. Each inner node's function is the sum of the function at its child nodes, and each leaf node is a multiplication of an identity Lagrange function because it mirrors $f(x)$ at a single point $x$. For example, $f_{0,0,1}(x) = a_4 \cdot I_{4,8}(x)$.

Each inner node of the AMT is associated with two elements: 1) the KZG commitment of its corresponding function and 2) a batch proof for the indices at which the function is zero according to the partitioning process. Detailed definitions of these elements are provided in the appendix. When proving the value of a given entry, e.g., $a_4$, the prover finds the path from the root to the corresponding leaf node: $f(x) \to f_0(x) \to f_{0,0}(x) \to f_{0,0,1}(x)$. It then iteratively decomposes functions along this path to express $f(x)$ into as a sum of four components: $f_1(x) + f_{0,1}(x) + f_{0,0,0}(x) + f_{0,0,1}(x)$. The prover then outputs the associate commitments for $f_1(x)$, $f_{0,1}(x)$, and $f_{0,0,0}(x)$, alongside their batch proofs demonstrating these functions equal to zero at $x = 4$. The verifier checks the correctness of these batch proofs and the consistency among commitments: whether the sum of commitments for $f_1(x)$, $f_{0,1}(x)$, $f_{0,0,0}(x)$, and $f_{0,0,1}(x) = a_4 \cdot I_{4,8}(x)$ equals to the commitment for $f(x)$.

Updating an entry in the AMT involves traversing from the root to the leaf corresponding and updating the associate elements along this path. The remaining are not affected, enabling AMT to maintain the proofs in $O(\log n)$ time.

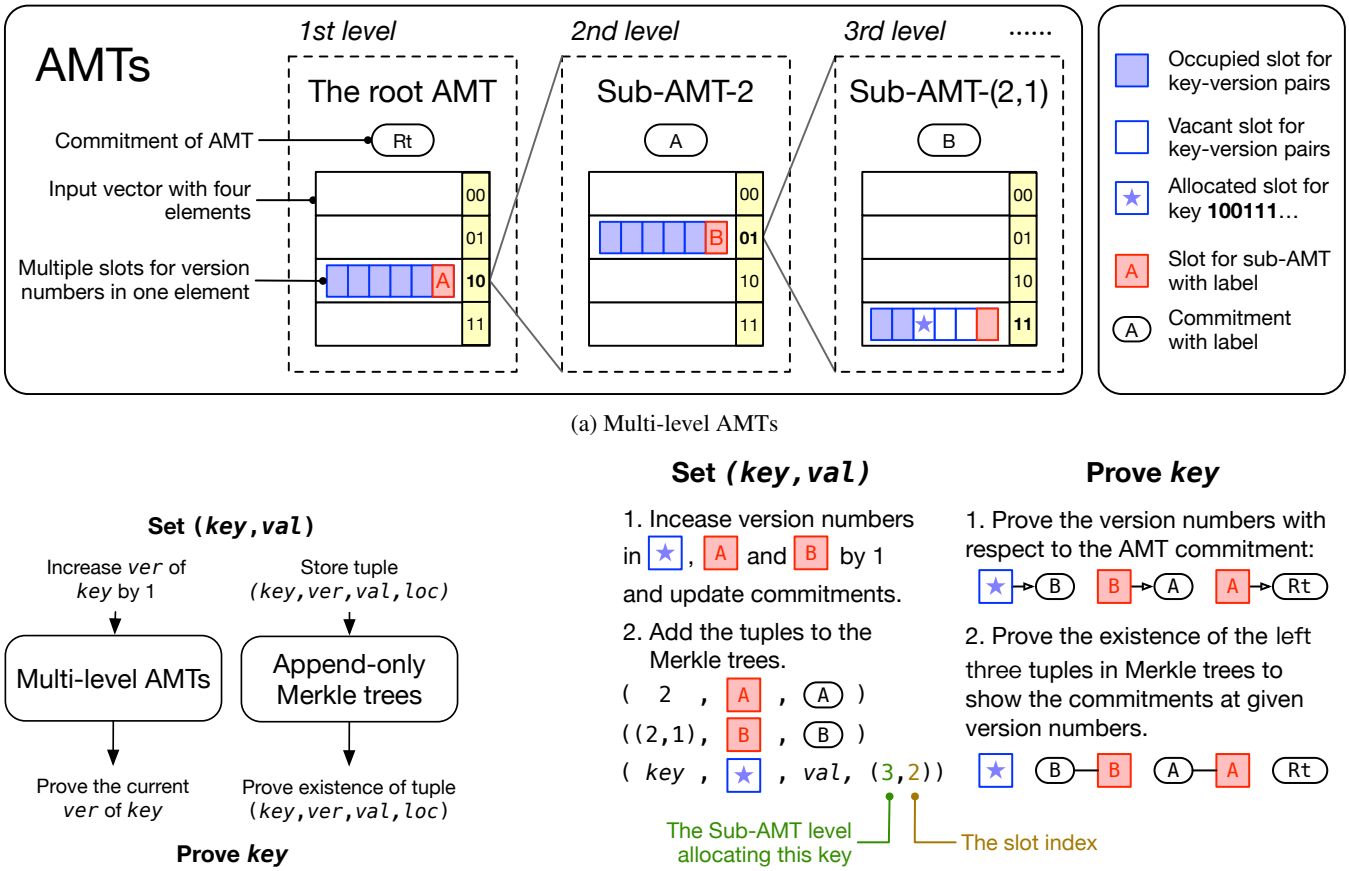
The nodes of the AMT serve as *auxiliary information* for generating proofs only. In a blockchain system, a miner without serving client queries may discard this auxiliary information and only maintain the commitment, which can be updated in constant time.

# 3 Overview

Recent works [32, 42] have shown that the majority of transactions execution time is spent on operations that access the blockchain state. For instance, a profiling experiment [32] shows that read and write operations to the blockchain state account for more than 67% of the execution time for the transaction executing the transfer function of ERC-20 smart contract [4, 41]. In this section, we present an overview of how LVMT tackles this problem. In particular, we propose a new authenticated storage system to reduce the amplification of read and write operations that access the blockchain state.

Our proposed system is based on AMT since it has an ideal time complexity, i.e., constant cost in updating the commitment. In particular, our proposed system solves several challenges to implement an efficient blockchain storage system using AMT:

First, although AMT costs constant time in updating the commitment, the constant ratio is large for a blockchain system. Table 1 shows the result of a micro-benchmark carried on an Intel i9-10900K CPU machine. It shows the time cost for basic cryptographic operations. Note that an elliptic curve multiplication takes about 0.1 ms, which is even much slower




(a) Multi-level AMTs

(b) Versioned key-value database

(c) Maintenance and proving on Multi-level AMTs

Figure 1: LVMT architecture.

| Pairing engines | BLS12-381 | BN254 |
|---|---|---|
| Addition | 0.68 | 0.34 |
| Multiplication | 169 | 92 |

Table 1: Time cost of operations over the primary curve $\mathbb{G}_1$ of pairing functions ($\mu$s).

than an updating operation in MPT.

Second, to support data with $n$ maximum entries, AMT requires precomputed parameters in size of $O(n \log n)$ and maintains auxiliary information in size of $O(n \log^2 n)$. Thus, AMT cannot support key-value pairs for an arbitrary-length bit string. As the size of the blockchain ledger state continues to grow, AMT is not a scalable solution.

Last, a blockchain system must consider the slowest node. Even if most miners do not need to maintain the auxiliary information for proof, the authenticated storage must guarantee the nodes for responding queries can keep up.

We propose the following techniques to resolve the challenges above. First, we design a versioned database that only stores the version number of keys in AMT, thereby avoiding the elliptic curve multiplications. This design also supports arbitrary lengths of values, as they are not stored in AMT. Second, we extend AMT to multiple levels to accommodate version numbers for unlimited keys, making the AMT size relatively small to optimize cache for parameters. To support arbitrary key lengths and minimize deep updates in the multi-level hierarchy, we utilize key hashes to allocate slots for version numbers. Last, we introduce proof sharding to reduce the single node's cost in maintaining auxiliary information for proofs.

## 3.1 Versioned Key-value Database

We designed a versioned authenticated storage to avoid multiplication on the elliptic curve during commitment updates. As shown in Figure 1b, the multi-level AMTs store key-version pairs, which only serve to identify the recent version number of a key. LVMT accumulates the key-version-value tuples in an append-only authenticated data structure consisting of a series of Merkle trees, with each block constructing one Merkle tree from the key-version-value tuples for value changes in

that block.

Imagine a scenario where the blockchain processes a block, setting a key-value pair $(key, val)$. LVMT first locates the corresponding entry of *key* in the multi-level AMTs to increment the stored version number by one. Assume the new version number for *key* is *ver*. LVMT then appends a new tuple $(key, ver, val, loc)$ to the Merkle tree being constructed for the block. Here, *loc* is a tuple $(level, slot)$ that records the level and slot in the multi-level AMTs where the key's version is located. The construction cost of a Merkle tree is linear with the number of version tuples. Once constructed, the Merkle tree for a block remains immutable, except for garbage collection of obsolete nodes. As the blockchain is append-only, the list of these Merkle trees is also append-only.

When generating a proof for a key-value pair $(key, val)$, LVMT first use the multi-level AMTs to prove the most recent version *ver* of the key *key*. It then uses Merkle trees to prove the existence of a tuple $(key, ver, val, loc)$. Since the roots of the Merkle trees are endorsed by the blockchain consensus protocol, light clients can trust that the Merkle trees are generated correctly without duplicate tuples having the same *key* and *ver*. As the location of the version slot is included in the version tuple of the key, the prover can not cheat by providing a version number proof of another slot.

Note that updating one element $a_i$ to $a_i'$ in an AMT requires computing $(a_i' - a_i) \cdot I_{i,n}(\tau) \cdot G_1$ (equation 1), multiplying $a_i' - a_i$ to the elliptic curve point $I_{i,n}(\tau) \cdot G_1$. In the versioned key-value database, $a_i$ is essentially a version number and $a_i' - a_i$ always equal 1. Thus, we eliminate an elliptic curve multiplication in each storage write, saving approximately 100 $\mu$s.

Since the frequency of bumping version number is limited by the block generation rate, we can conserve the bits used for storing version number and store multiple version numbers in a one vector entry. For example, when employing BN254 as the underlying bilinear mapping parameter, each entry is an element in $\mathbb{Z}_p$, where $p$ is a prime integer in $\left(2^{254}, 2^{255}\right)$. This suggests that implies each entry can store at most 254 bits. In a blockchain system generating 10 blocks per second, the version number will not exceed $2^{40}$ in 3000 years. So each entry can be divided into six slots with 40 bits as shown in Figure 1a.

## 3.2 Multi-level AMT

To make AMT scalable and allow it to store the version number for an unlimited number of keys, we introduce multi-level AMTs as shown in Figure 1a. The authenticated storage is initiated by one AMT as *the root AMT*. Each entry in the AMT contains several slots for storing version numbers. One slot in each entry is reserved for storing the version number of the commitment hash of the sub-AMT, with the remaining slots utilized for key-value pairs.

Let $k$ denote the height of the AMT. When allocating a slot for a new key, LVMT accesses the entry in the root AMT whose index aligns with the first $k$ bits of the key hash. If this entry lacks a vacant slot, LVMT accesses the corresponding sub-AMT and locates the entry in the sub-AMT whose index matches the next $k$ bits of the key hash. LVMT recursively visits the sub-AMTs to find a vacant slot for the new key. Figure 1a presents an example with $k = 2$ for allocating a version slot for a key with hash $100111\cdots$. As the first two bits of key hash are 10, LVMT accesses the entry with index 2 and attempts to find a vacant slot. Since all slots in the entry are occupied, LVMT proceeds to the corresponding sub-AMT-2. Picking the next two bits 01, it accesses the entry with index 1, and recursively visits the sub-AMT-(2,1) because there is no vacant slot again. Finally, LVMT finds the third slot at the third level being vacant and allocates this slot.

The commitment of a sub-AMT is treated similarly to a key-value pair, where the key represents the index of the sub-AMT and the value is the commitment. The Merkle trees not only store key-version-value tuples for standard key-value pairs, but they also store the tuples of the sub-AMT index, the version of the sub-AMT commitment, and the commitment hash.

Figure 1c illustrates how LVMT maintains the AMTs and Merkle trees when a block changes the key with hash $100111\cdots$. LVMT first increments the version number for this key by one. This in turn alters the commitment of sub-AMT-(2,1), prompting LVMT to also increase the version number for the commitment (the slot labeled "B") by one. Recursively, the commitment of sub-AMT-2 is changed and the version number labeled "A" is updated. Finally, LVMT gets the updated commitment of the root AMT. LVMT appends the tuples of changed keys and commitments into the Merkle trees along with the normal tuple of the key-value pair.

When generating a proof for this key, LVMT finds the most recent version of tuples for sub-AMT-2, sub-AMT-(2,1) and this key. LVMT proves the existence of these tuples in Merkle trees and confirms the correctness of appeared version numbers with respect to their AMT commitments. When proving the non-existence of a key, LVMT affirms that all the possible slots for this key are vacant or have been allocated to other keys.

## 3.3 Proof Sharding

We recall that the AMT maintains a binary tree, where each node holds a commitment and a batch proof. Each input entry corresponds to a leaf in this tree. When generating a proof, AMT picks commitments and batch proofs from the siblings

of nodes along the path from this leaf to the root. Each node can be updated independently of the other nodes, facilitating the parallelization of tree maintenance. Each blockchain node can maintain a shard of the proof. It picks a subtree of the root AMT and takes responsibility for generating proofs for the leaves in this subtree, and the sub-AMTs extended from these leaves. Multiple blockchain nodes can collaboratively generate proof for any key. Similarly, the storage for the Merkle tree can be distributed to multiple nodes by the block number.

## 4 LVMT Design

Now we formally define LVMT, which utilizes a key-value database as a backend and maintains a tuple of key-value maps $(\mathsf{KM}, \mathsf{AM}, \mathsf{MM}, \mathsf{VM}, \mathsf{LM})$ where KM stores the key-value pairs, AM stores the AMTs data structures, MM stores the Merkle trees, VM stores the version slots metadata for keys, and LM records the position of the most recent tuple for a key or a sub-AMT in the Merkle trees. LVMT decouples the data storage and data authentication: KM stores unauthentication data; AM and MM store the authenticated information; VM and LM store the metadata and indicies for authenticated information. Each AMT in LVMT encompasses the following components:

- comm: the commitment of AMT;
- proof_tree: the proof tree of AMT;
- leaves: a list of leaves; leaves$[i]$ denotes the leaf corresponding to the $i$-th element of the input vector. Each leaf comprises the two lists vers and keys. vers$[0]$ stores the version number for the sub-AMT. vers$[1]$ to vers$[5]$ store the version numbers for the keys keys$[1]$ to keys$[5]$, respectively. Note that only vers contribute to the AMT commitment.

### 4.1 Interfaces to the Transaction Execution

LVMT provides the following two interfaces (instructions) for the blockchain execution layer:

- $\mathsf{Get}(\mathsf{k}) \rightarrow \mathsf{val}$: Reads the value val stored in k;
- $\mathsf{Commit}(\mathbf{W}, e) \rightarrow (\mathsf{aroot}, \mathsf{hroot})$: Flushes the changed key-value pairs in $\mathbf{W}$ with block number $e$ and produces the commitment of LVMT.

These interfaces match the requirements from the blockchain execution engine introduced in Section 2.1. The execution engine uses Get to fetch data from the storage and LVMT simply loads the value correspondingly from KM.

The instruction Commit is invoked after the execution of a block. LVMT commits the key-value pairs $\mathbf{W}$ using the procedure COM defined in Algorithm 1. The returned commitments will be filled in the block header. The commit returned values consist of the roots of both the top-level AMT and MPT.

**Algorithm 1** A procedure to compute a commitment. It takes a list of key-value pairs $\mathbf{W}$ and a block number $e$, and returns the commitments aroot and hroot.

```
1:  procedure COM(W, e)
2:      M ← [ ];  T ← { };
3:      foreach (k, val) in W
4:       (lv, tidx, sidx, ver) ← ComKV(k, val);
5:       M ← (k, ver, val, lv, sidx) :: M;
6:       T ← {(lv, tidx)} ∪ T;
7:      i ←maximum lv in T;
8:      while i ≥ 0
9:       foreach (lv, tidx) in T with lv = i
10:        (C, ver) ← UpdComVer(lv, tidx);
11:        M ← (lv, tidx, ver, comm) :: M;
12:        if lv > 0
13:          T ← {(lv − 1, ⌊tidx/n⌋)} ∪ T;
14:     foreach (k, ver, val, lv, sidx) in M with index i
15:      LM[k] ← (e, i);
16:     foreach (lv, tidx, ver, C) in M with index i
17:      LM[(lv, tidx)] ← (e, i);
18:     Build merkle tree of M and store inner nodes in MM;
19:     mroot ← Merkle root of M;
20:     hroot ←Merkle root of the mroot of all the commits;
21:     aroot ← AM[(0, 0)].comm;
22:     return (aroot, hroot);
```

**Algorithm 2** A procedure to compute the commit of a key-value pair. It returns the level lv, the tree index tidx, the slot index sidx of the changed AMT, and the version ver.

```
1:  procedure COMKV(k,val)
2:      if KM contains k
3:       (lv, sidx) ← VM[k];
4:      else
5:       (lv, sidx) ← ALLOCATESLOT(k);
6:       VM[k] ← (lv, sidx);
7:      (tidx, lf) ← LEAFATLEVEL(lv, k);
8:      ver ← lf.vers[sidx];
9:      lf.vers[sidx] ← lf.vers[sidx] + 1;
10:     Update the corresponding commitments and proofs.;
11:     ver ← ver + 1;
12:     return (lv, tidx, sidx, ver);
```

**Algorithm 3** A procedure to allocate a version slot to a new key. It takes the key k to allocate a slot for, and returns the level and the allocated slot index.

```
1:  procedure ALLOCATESLOT(k)
2:      lv ← 0;
3:      while true
4:       (tidx, leaf) ← LEAFATLEVEL(lv, k);
5:       for j ∈ [5]
6:        if leaf.vers[j] == 0
7:          leaf.keys[j] ← k;
8:          return (lv, j);
9:       lv ← lv + 1;
```

The procedure COM first commits the key-value pairs in $\mathbf{W}$ (Lines 3 to 6) with the sub-procedure COMKV. Then it updates the version numbers of all the affected sub-AMTs from the deepest sub-AMT to the root AMT (Lines 7 to 13) using the procedure UPDCOMVER. This procedure maintains

**Algorithm 4** A procedure to compute the AMT index and the leaf index of a key key at a AMT level lv. It returns the tree index tidx and the leaf leaf corresponding to the key key at level lv.

```
1: procedure LEAFATLEVEL(lv,key)
2:     tidx ← first bit to (k·lv)-th bit of H(key);
3:     lidx ← (k·lv+1)-th bit to (k·(lv+1))-th bit of H(key);
4:     leaf ← AM[(lv,tidx)].leaves[lidx];
5:     return (tidx,leaf);
```

**Algorithm 5** A procedure to update the commitment and version of an AMT at level lv and tree index tidx. It returns the commitment $C$ and the updated version number ver.

```
1: procedure UPDCOMVER(lv, tidx)
2:     C ← AM[(lv,tidx)].comm;
3:     ptidx ← ⌊tidx/n⌋;
4:     plidx ← tidx mod n;
5:     ver ← AM[(lv,ptidx)].leaves[plidx].ver[0];
6:     Increase AM[(lv,ptidx)].leaves[plidx].ver[0] by 1;
7:     Update the corresponding commitments and proofs;
8:     ver ← ver+1;
9:     return (C,ver);
```

the version number for commitments of sub-AMTs similar to COMKV. While maintaining the version numbers, LVMT collects the tuples of keys, versions, values, and other metadata in a list $\mathbb{M}$ (Line 5). The system treats a pair of the sub-AMT index and its commitment similarly to a key-value pair (Line 11). LVMT builds a Merkle tree for $\mathbf{M}$, thereby authenticating the value of a given key and version (Line 19). It also stores the positions of these elements in the Merkle trees (Lines 15 and 17). So when generating a proof, the prover can locate the corresponding Merkle leaves of a key or an AMT commitment.

The sub-procedure COMKV (Algorithm 2) is implemented to maintain and update the version numbers. COMKV(k,val) first finds the allocated version slot for the given key k (Line 3). If the key has not been allocated a version slot, it allocates a slot to it (Line 5). It uses the sub-procedure ALLOCATESLOT (Algorithm 3) to find a vacant slot in the AMT to allocate. In particular, starting from the root AMT, ALLOCATESLOT computes the tree and leaf indices for the given key at each level, checks if the leaf has a vacant slot, and then returns the level and slot indices of the slot; if the leaf doesn't have a free slot, it proceeds to the next level.

Then, COMKV computes the corresponding tree index tidx and the leaf lf for k at level lv (Line 7) using the sub-procedure LEAFATLEVEL (Algorithm 4), which finds the corresponding AMT index and leaf for the key k at the level lv using the hash $H(\mathsf{k})$ of k. Since each AMT has $m$ levels and $2^m$ leaves, the first $m \cdot \mathsf{lv}$ bits of $H(\mathsf{k})$ decides the AMT index and the subsequent $m$ bits locate the leaf in the tree. Finally, COMKV locates the slot for this key and updates its version and other information according to AMT's rule (Line 8 to 10).

**Algorithm 6** A procedure to generate a proof for an existing key k. It returns the proof of the key.

```
1: procedure GENPROOF(k)
2:     keypf ← PROVEKEY(k);
3:     (lv,sidx) ← VM[k];
4:     while lv > 0
5:       tidx ← first bit to (k·lv)-th bit of H(k);
6:       commpfs[lv] ← PROVECOM(lv,tidx);
7:       lv ← lv−1;
8:     return (keypf,commpfs);
```

**Algorithm 7** A procedure to verify the proofs keypf and commpfs with respect to an AMT root aroot and Merkle root mroot.

```
1: procedure VERIFYPROOF(keypf, commpfs, aroot, mroot)
2:     Verify the AMT proofs and the merkle proofs in keypf and
   commpfs;
3:     Verify the commitment in commpfs[1] equals to aroot
4:     if all the verification pass
5:       return true;
6:     else
7:       return false;
```

The sub-procedure UPDCOMVER (Algorithm 5) updates the commitment and its version number given an AMT located by its level and index.

## 4.2 Proving Key-value Pairs

As an authenticated storage, LVMT provides the following two interfaces to allow a user to query a value from an untrusted server and to verify the value with the commitment.

- $\mathsf{GenProof}(\mathsf{k}) \to \pi$: Generates proof $\pi$ for key k;
- $\mathsf{Verify}(\mathsf{k},\mathsf{v},\pi,\mathsf{comm}) \to \mathsf{true/false}$: Verifies the key value pair $(\mathsf{k},\mathsf{v})$ with respect to a ledger state commitment.

When responding to a query k from a light node, a full node will generate proof $\pi$ using the procedure PROVE and responde with the loaded value and the current commitment.

The procedure PROVE (Algorithm 6) consists of two parts: 1) the proof of the value val of the key k with respect to the sub-AMT it belongs to (line 2) using the sub-procedure PROVEKEY; 2) the proof of the commitment for all the sub-AMT along the path from k's sub-AMT to the root AMT (excluding the root AMT) (line 4 to line 7) using the sub-procedure PROVEKEY.

The generated proof consists of a merkle proof for the existence of the tuple of the key (or the AMT index), the value (or the AMT commitment) and the version, an AMT proof for the version number, and other metadatas. We provide the definition for the sub-procedures PROVEKEY (Algorithm 8) and PROVECOM (Algorithm 9) in the supplementary material. In the supplementary material, we also discuss how to generate a non-existing proof.

The light node verifies the proof using the procedure VERIFY (Algorithm 7), which recovers the tuple of Merkle leaves to to be verified from the proof and verifies the AMT proofs and the merkle proofs.

## 5 Implementation

We implemented the AMT using Arkworks [21], a Rust library for elliptic curve operations. AMT is built using the pairing parameters BN254 and supports vector commitment in the length of $2^{16}$. Each entry contains 254 available bits and is divided into six slots with 40 bits. For the public parameters required by the KZG commitment, we utilize the output from the Perpetual-Powers-of-Tau ceremony [27], which conducts an MPC protocol among over 70 participants worldwide in generating secure parameters. Based on the above AMT implementation, we implemented LVMT in Rust [1]. LVMT is compatible with any backend database that provides a key-value interface as defined in rust crate "kvdb" [39].

We ported the implementation of MPT from the OpenEthereum client [48], the most popular high-performance Rust implementation of Ethereum. We also implemented a variant of RainBlock [40], which developed an efficient MPT for distributed in-memory systems, by referencing its implementation [6]. This variant incorporates significant RainBlock features, including caching of top layers in memory, in-memory construction of the Merkle tree using pointers, and the application of lazy hash resolution. Unlike RainBlock, our variant stores the bottom layers on local storage instead of a distributed in-memory system. These implementation are also compatible with the same interface.

For the implementation of LVMT, we applied several optimizations:

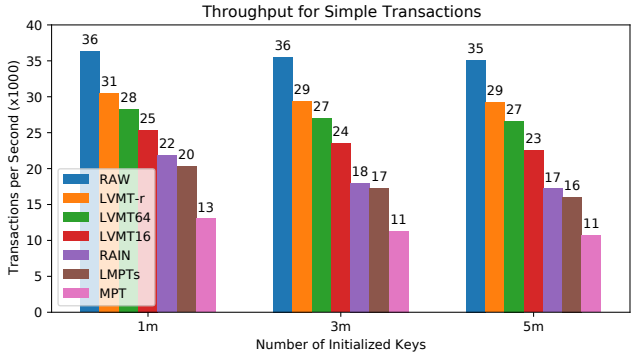
**Combining entries in different maps:** For a given key, we use three maps KM, VM, and LM to store its value, version slot index, and the position of the Merkle tree for the recent change, respectively. In our implementation, we combine these entries into a single key-value pair to save read and write operation for each key.

**Cache the root AMT:** The root AMT is frequently accessed. So its leaves and inner nodes of are always stored in memory. The commitments of the AMTs in the second levels are also cached. Each leaf and inner node of an AMT has two points on the elliptic curve. Given that we set the AMT height as 16, the root AMT and the commitments of AMTs in the second level store about 200,000 elliptic curve points in memory. Each point takes 96 bytes in our parameter, so roughly 20 MB of memory is needed to store them.

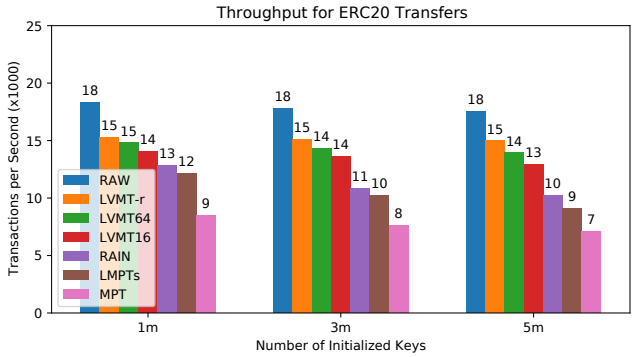
**Cache cryptographic parameters:** We expedited the commitment update procedure by precomputing certain elliptic curve points. For instance, when the input entry at position $i$ increases by $\delta$, the commitment can be updated as $C' = C + P_i$, where $P_i = I_{i,n}(\tau) \cdot \delta \cdot G_1$. Given that each entry is divided into six 40-bit slots, when the version number increases, the difference between the new and the previous version will be one of the following: $1, 2^{40}, 2^{80}, 2^{120}, 2^{160}, 2^{200}$. Thus, LVMT precomputes $P_i^{(j)} = 2^{40j} \cdot P_i$ for all $0 \leq j \leq 5$ and $1 \leq i \leq n$. So LVMT can simplify the commitment update procedure by merely incrementing a precomputed point. In our design, each elliptic curve point requires 96 *bytes* of storage. So a node excluding proof maintenance necessitates around 37 *MB* of memory. However, a node maintaining a shard of proof must cache additional parameters, resulting in a higher memory requirement, approximately 650 *MB*.

**Reduce the coordinates conversion time:** An elliptic curve point is uniquely represented by its affine coordinate $(x, y) \in \mathbb{Z}_q^2$, where $q$ is a large prime number. These points can also be represented through projective coordinates $(x, y, z) \in \mathbb{Z}_q^3$, which accelerate arithmetic operations by eliminating division operations within a large prime field. The conversion of these projective coordinates back to the corresponding affine coordinates is given as $(x/z^2, y/z^3) \in \mathbb{Z}_q^2$. However, a challenge arises from the fact that a single elliptic curve point corresponds to multiple projective coordinates, leading to hashing inconsistencies. To address this issue, LVMT always converts the projective coordinates back to the affine coordinates when computing the hash of a sub-AMT commitment. This conversion process, however, is computationally intensive, taking approximately 60 $\mu$s per conversion and can substantially impact the write speed. To alleviate this, we applied batch conversion of all projective coordinates to affine coordinates at the culmination of each block execution, decreasing the average conversion time to a mere 0.4 $\mu$s.

**Garbage collection of append-only Merkle trees:** As a key's version number increases, the old version tuples within the append-only Merkle trees become unnecessary for future proofs. When a subtree in a Merkle tree only has obsolete children, the entire subtree can be truncated, and only the subtree root is stored. A background thread performs this garbage collection to prevent impacting LVMT's performance under heavy workloads. In a scenario where the append-only Merkle trees have accumulated $m$ version tuples in the past, and only $n$ tuples are currently active, the overhead of storing truncated Merkle trees is about $(\log_2(m/n) + 1) \cdot 2n$. (See appendix for the details.) While this introduces some additional overhead, it remains a practical approach.




(a) Transaction execution for balance transfers.




(b) Transaction execution for ERC-20 transfers.

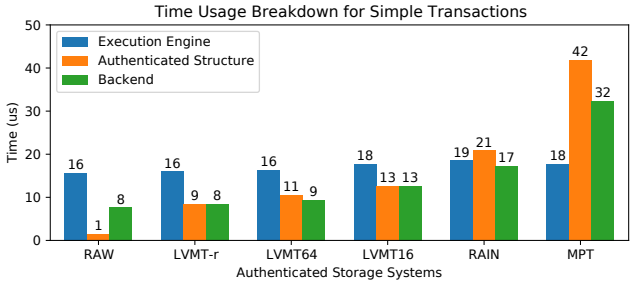
Figure 2: Throughput of transaction execution

## 6 Evaluation

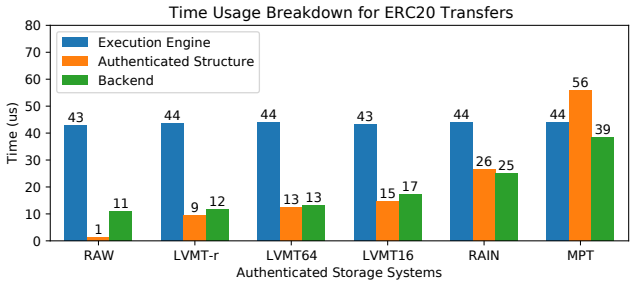
We evaluate LVMT's performance and compare it to other authenticated storage systems using a machine with an Intel i9-10900K CPU, 32 GB DDR4 RAM, and SSD storages. All authenticated storage systems utilize RocksDB [47] as their backend key-value database.

We assess LVMT under different settings: 1) LVMT without associated information (no proof shard), 2) LVMT with 1/64 and 1/16 of the associated information (proof sharding), 3) LVMT with all associated information. In this context, LVMT-r represents LVMT without any associated information, while LVMT64, LVMT16, and LVMT1 signify LVMT with 1/64, 1/16, and complete proof sharding, respectively.

In addition to LVMT, we evaluate various authenticated storage systems for comparison. As previously mentioned, we have ported the MPT in OpenEthereum and have implemented a variant of RainBlock, which we refer to as MPT and RAIN, respectively. We also examine the Layered Merkle Patricia Tries (LMPTs) [20] utilized in Conflux [2, 33], a high-performance blockchain, represented by LMPTs. For reference, we also examine the performance of directly storing data into the backend, bypassing authenticated storage, denoted as RAW.




(a) Time cost breakdown for balance transfers.




(b) Time cost breakdown for ERC20 transfers.

Figure 3: Break down of the time usage in transaction execution on 5 million receivers.

**End-to-end performance:** We assess the end-to-end performance of authenticated storage on Conflux [2, 33], a high-performance blockchain. To gauge peak performance, we set a large block size of 20,000 transactions per block. Thus, all authenticated storage systems can finish executing one block within 0.5 to 5 seconds, aligning with the block generation intervals of major high-performance blockchains. To emulate the prevalent configuration of contemporary blockchains, we employ cgroup to restrict the memory consumption of a blockchain node to 16GB and allocate a 4GB RocksDB cache size. In the experiment, 10,000 senders randomly select addresses from the receiver space and transfer non-zero balances to them, representing simple payment transactions. We evaluate receiver spaces with one million, three million, and five million addresses. Conflux is run for an extended period, ensuring the number of executed transactions is three times larger than the receiver space.

Figure 2a shows that LVMT-r achieves a maximum throughput of 29669 TPS on average and is up to 2.7 times faster than MPT and 1.7 times faster than RAIN. We also evaluate the performance of transactions executing the transfer function of the popular ERC-20 smart contract [41], the most common transactions on the Ethereum blockchain [4]. As shown in Figure 2b, LVMT-r is up to 2.1 times faster than MPT and 1.5 times faster than LMPT in this workload.

To further study the time usage in execution of one transaction, we breakdown the time usage into three parts: 1) Execution Engine, i.e., transactions execution without access to the

authenticated storage, 2) Authenticated Structure, i.e., access to the authenticated storage without accesses the backend database, 3) Backend Database, i.e., accesses to the backend database. Figure 3a shows the breakdown of time usage in executing random balance transfer transactions with 5 million receivers. The execution engine takes the same time 16 us across the different storages. LVMT-r takes a similar time 11 us with RAW in accessing the backend. It implies LVMT-r almost eliminates the overhead of the authenticated storage from backend access. LVMT64 and LVMT16 take a similar time to LVMT. MPT requires 42 us and 33 us to access the authenticated structure and the backend database, respectively, which is more than 4x the time used in LVMT-r. As shown in Table 1, a single elliptic curve multiplication requires 92 us, which is even slower than MPT. Therefore, eliminating the expensive elliptic curve operation is necessary to make LVMT practical. Figure 3b shows the breakdown in executing random ERC20 transfers. The execution engine still takes the same time across the different storages but takes more time than the execution of the balance transfer. This is because the execution of ERC20 transfers requires more I/O accesses (e.g., loading contract bytecode). All the storages take about 20% more time than executing balance transfers.

This experiment shows that LVMT is able to maintain better throughput than MPT for both simple payment transactions and the typical ERC-20 smart contract transfer transactions.

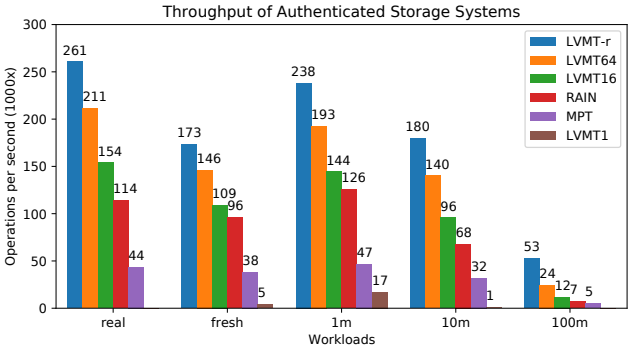
**Stand-alone performance:** We also evaluate the stand-alone performance of authenticated storage systems in micro-benchmarks. We developed an authenticated storage benchmark tools [1] for evaluation. Since most transactions in the real world simply read the accounts of the sender and the receiver and update their balances, we launch a workload of 20 million random "read then write" operations and commit the changes every 100,000 operations, resembling a block being generated every several seconds. The authenticated storage is initiated with random key-value pairs whose size ranges from $10^6$ to $10^8$. Both the key and the value are 256-bit strings. We use "1m", "10m", and "100m" to indicate the initialized size $10^6$, $10^7$ and $10^8$. Since LVMT needs to allocate version number slots for new keys, we also evaluate with a "fresh" setting: the storage has no initialization, and the workload accesses distinct keys.

In addition, to evaluate the performance under the real world access pattern, we extract the I/O trace on Ethereum, the largest smart contract platform. We choose transactions in 2021 winter, when Ethereum is going through its latest boom. We replay the Ethereum transactions from block 13,500,000 to block 13,600,000 to recover the I/O operations. These blocks access 22 million distinct keys, and make 97 million reads and 54 million writes in total. Each block contains an average of 1,500 operations. Considering that high-performance authenticated storage can process over 100,000 operations per second, having only 1,500 operations per block results in an unreasonably short block generation cycle. This considerably impacts RAIN's optimization efforts for lazy hash resolutions. To address this issue, we aggregated operations from every 50 blocks into a single block, making the block size in the real trace workload more closely resemble the size in a random access workload. We use "real" to denote the workload from real world transactions.

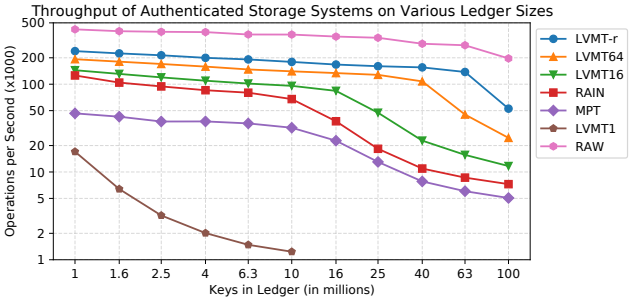
The primary blockchain node like Geth recommands a minimum of 16GB RAM for optimal performance. We assume that half of this memory is allocated for executing smart contracts that access authenticated storage systems, while the remaining half accommodates other functionalities. Consequently, we limited the runtime memory to 8GB using cgroups in our micro-benchmarks. We observed that authenticated storage systems without inherent caching strategies, such as RAW and MPT, perform better when provided with a higher RocksDB memory budget. Conversely, authenticated storage systems incorporating caching strategies, like LVMT and RAIN, show improved performance at a lower memory budget due to the need for an adequate filesystem cache. To optimize performance, we allocated a 4GB RocksDB cache size for RAW and MPT and a 2GB cache size for LVMT and RAIN. As the implementation of LMPTs is highly coupled with the backend database, and the vague boundary separating the authentication structure from the backend database posed a challenge to accurately gauge LMPTs in the micro-benchmarks. We removed LMPTs in micro-benchmarks.

Figure 4a shows the throughput across various workloads. LVMT-r outperforms MPT and RAIN by at least 353% and 80%, respectively. When handling a shard of auxiliary information, LVMT64 and LVMT16 achieve roughly 80% and 60% of LVMT-r's throughput across most workloads. LVMT1 consistently exhibits the weakest performance in all workloads, demonstrating the necessity of proof sharding.Within the Ethereum real trace workload, the ledger size initially comprises 4 million keys and eventually grows to 22 million keys. However, all the authenticated storage systems either outperform or match their performance in the '1m' workload, as the real trace workload provides better access locality than random access.

Figure 4b illustrates the throughput for various ledger sizes. All authenticated storage systems experience a noticeable performance drop when reaching a specific ledger size threshold. This occurs because the ledger size surpasses memory limitations, preventing both RocksDB's cache and the file system cache from effectively storing ledger data. RAIN and MPT performance begins to drop at a ledger size of 16 million,

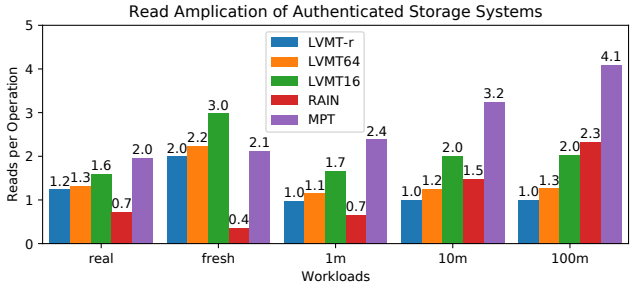



(a) Throughput under different workloads.

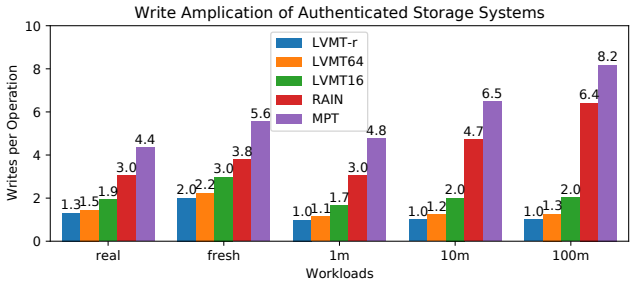



(b) Throughput for various ledger sizes.

Figure 4: Throughput of authenticated storage systems.




(a) Read amplification of authenticated storage systems.




(b) Write amplification of authenticated storage systems.

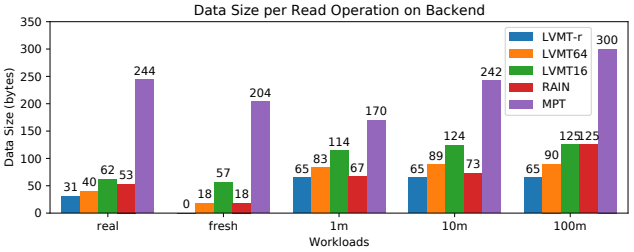
Figure 5: Read and write amplifications of authenticated storage systems.

whereas LVMT declines at a larger size. LVMT16, LVMT64, and LVMT-r demonstrate performance degradation starting from ledger sizes of 25 million, 63 million, and 100 million, respectively. This suggests that LVMT can provide efficient ledger access with a smaller memory usage.

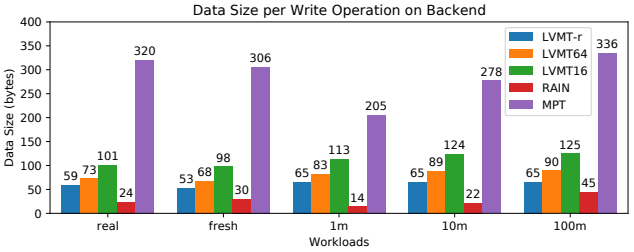
**Read and write amplification:** We further study the read and write amplification at the backend database interface. Here, read amplification represents the ratio of backend read operations to those on authenticated storage systems' interfaces, and write operations are defined similarly. Figure 5a shows the read amplification under the different settings. As the ledger size grows, LVMT-r exhibits consistent read amplifications. The root AMT contains $2^{16}$ entries, and the second level of AMTs $2^{32}$ input entries in total. Since each entry has five slots for key-value pairs, the root AMT can only store 0.3 million keys, and the second level of AMTs accommodate 21 billion keys. So LVMT-r always requires two levels of AMT in all these workloads. The read amplification of a key grows linearly with its level in the AMTs, so it is reasonable for LVMT-r to exhibit similar read amplifications. In contrast, the read amplification of MPT grows from 2.4 to 4.1. RAIN demonstrates a smaller read amplification in the Ethereum real trace, indicating that its cache strategy benefits from better access locality in the real trace. For LVMT with proof shards, the read amplification grows linear with the size of auxiliary information. LVMT16 maintains four times the auxiliary information than LVMT64. So the surplus of LVMT16 compared to LVMT-r is four times larger than the surplus of LVMT64. When accessing the fresh ledger state, allocating slots for the version number increases the read amplification of LVMT-r by 1.

Figure 5b displays the write amplification. The write amplification of LVMT is similar to the read amplification. MPT and RAIN have a larger write amplification than read amplification since MPT nodes are keyed by their hash digests. So each time the storage changes, a write operation and a deletion operation are applied to the backend.

Figure 6a and 6b present the average sizes of read and write operations on backend, while figure 6c and 6d provides a more in-depth analysis of data size percentiles for the "100m" workload. Considering that each MPT node can accommodate up to 16 children, each containing a 32-byte hash, an MPT node may store around 500 bytes. So MPT's performance is negatively impacted by the combination of extensive read amplification and large data size per read operation. By caching the top six layers of MPT in memory, RAIN effectively reduces data sizes for both read and write operations. In RAIN, the first layer on disk represents the seventh layer of MPT, which can house roughly 17 million nodes. Thus, at the largest ledger size in our experiment, which consists of 100 million keys, each node only needs to accommodate six children, leading to a 200-byte node. LVMT-r only accesses elliptic curve points, which are 65 bytes in size. LVMT with proof shards may load 65-byte elliptic curve points and 192-bytes auxiliary information for an AMT node from backend.

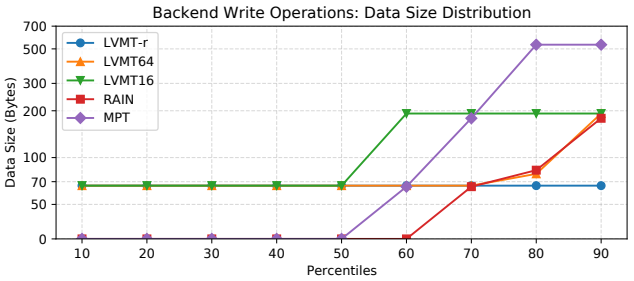



(a) Data size per read operation




(b) Data size per write operation


(c) Data size distribution of backend read operations




(d) Data size distribution of backend write operations

Figure 6: Data size of backend operations

Figure 6 indicates that around 40% of read operations for LVMT16 involve auxiliary information, while about 10% of LVMT64's read operations relate to auxiliary information.

## 7 Related Works

**Improved MPT structures:** mLSM proposes to maintain multiple levels of MPTs [43]. The most recent updates are in the lowest level (level 0). The key-value pairs in a lower level will be merged to higher levels periodically. LMPTs proposes maintaining three MPTs, one large MPT containing old state and two small MPTs containing recent state changes [20]. LMPTs periodically merges small MPTs into large ones. For both mLSM and LMPTs, the concatenation of the Merkle roots of all the MPTs becomes the commitment for the ledger state.

Both LVMT and mLSM employ multi-level structures to minimize write amplification, but their approaches differ. mLSM maintains shallow top-level trees by regularly merging entries from the top-level Merkle trees into lower levels. Since write operations in mLSM only affect the top-level trees, the reduced depth decreases overall costs. LMPTs adopt a similar strategy, keeping a shallow delta MPT and periodically merging it into the snapshot. Conversely, LVMT's multi-level structure is akin to a tree, where each AMT serves as a node. When a write operation modifies an element in a lower-level AMT, all AMTs on the path from the root to the target AMT must be updated. With each AMT capable of accommodating up to 65,536 children, the high degree effectively reduces LVMT's overall depth, thus lowering write amplification.

Their techniques reduce the number of disk I/O operations on the critical path because the recently accessed state will be stored into MPTs with smaller depth, and the merge of MPTs can happen in a background thread. In contrast, LVMT almost eliminate unnecessary read amplification in practice. Our results show that when integrated end-to-end into Conflux, LVMT outperforms LMPTs by up to 2.5x. The mLSM paper only contains its conceptual design without implementation and evaluation [43]. It is unclear how mLSM would perform end-to-end with a blockchain in practice.

**Parallelize storage I/O:** RainBlock [40] introduces three different nodes in a blockchain system to accelerate the transaction execution: the storage prefetchers, the miners executing transactions, and the storage nodes. When executing transactions, the miners obtain needed data from multiple prefetchers and send the updates to multiple storage nodes. Each storage node maintains a shard of MPTs in memory. RainBlock changes the local storage I/O to network distributed storage I/O and benefits from the parallel I/O and in-memory storage. To reduce the read latency of network storage, RainBlock introduces I/O prefetchers and requires the miners to attach all the accessed key-value pairs and the witnesses (MPT nodes) when broadcasting blocks. RainBlock reports the average size of witnesses per transaction is 4 KB and their optimizations reduce the size of witnesses by 95% , so the additional network message per transaction is about 200 bytes, two times of a transaction. However, the inefficient usage of networks brings a bottleneck to a high-performance blockchain system [26]. RainBlock also suffers attacks in data availability. Since in-memory storage is costly, the number of replicas in RainBlock is much less than in Ethereum. As a comparison,

LVMT does not introduce additional network bandwidth consumption and data availability risk. Even if proof of shard in LVMT is lost, the other nodes can recover the auxiliary information of an AMT in minutes.

Both RainBlock and LVMT employ the sharding concept, but with different targets. RainBlock divides the ledger into multiple shards, preventing single nodes from accessing the entire ledger. To address this, RainBlock devised complex protocols between the prefetchers handling ledger reads and the miners executing transactions. Conversely, LVMT utilizes sharding solely to maintain auxiliary information for generating proof, allowing nodes to access the full ledger data during transaction execution. The proof sharding is mainly handled by blockchain node API providers. When users query a key, providers must direct the query to the corresponding node along with the relevant proof.

Another similarity between our RainBlock implementation and LVMT is caching top-level nodes. By default, RainBlock caches six layers of MPT nodes, while LVMT caches a single layer of AMT. As LVMT's nodes having a significantly higher degree than MPT (65,536 vs. 16), LVMT can use less memory to accommodate more entries in the first layer beneath the cached nodes.

**Vector commitment for data sharding:** Several vector commitment protocols [19, 24, 28, 30, 46, 49] have been proposed to reduce the proof size, support revealing elements in batch, or make the commitment efficiently updatable under some requirements. Some research also considers utilizing the vector commitment for data sharding on blockchain. Alin et al. [49] use KZG commitment protocol [28] to replace the underlying Merkle tree for data sharding. Unlike LVMT, the goal of this technique is not to improve the throughput but to reduce the data size of the blockchain storage. It requires the clients to maintain the proofs for their own data, keep updating the proof, and attach the values and proofs for the accessed storage in a transaction. Each client needs to be online and update the proofs of all of its data each time a write operation happens on the blockchain. Note that this protocol takes $O(n)$ time to generate proof or maintain proofs for all data, which costs $O(n)$ time to add a new key-value pair. It is therefore not designed for a high throughput blockchain system. When thousands of transactions are executed on the blockchain per second, a client cannot maintain its proofs efficiently.

Pointproofs [24] proposes an aggregatable and maintainable vector commitment protocol that can maintain the auxiliary information for proofs in $O(\log n)$ time (like AMT) and reveal any $k$-element subset of elements in $O(k)$ time with a batched proof. Pointproofs allows a consensus node to generate a batched proof for all the accessed key value pairs during block execution, so a node without the whole ledger can verify the correctness of execution. However, for every 1024 transactions, Pointproofs takes 5 seconds to maintain the auxiliary information for proofs, which cannot match the requirements in a high throughput blockchain system.

**Accumulators:** Accumulators are cryptographic primitives that commit a set of elements to a short digest (commitment) while supporting operations like addition, deletion, membership proof, and non-membership proof. Merkle trees are one example of accumulators. A recent study [13] designed an RSA accumulator that supports batch operations and stores UTXO sets for a blockchain, with commitments updated in constant time.

In a zk-rollup blockchain [7], it is crucial to convince a light client with a SNARK proof [12] that the ledger root is updated correctly in a given sequence of operations. Ozdemir et al. replaced Merkle trees with RSA accumulators to accelerate SNARK proof generation [38]. Although RSA accumulators require $O(n)$ time to generate a proof or update proofs for all elements, the time savings in SNARK proof generation outweigh the time spent in accumulator proof generation. However, in a high-performance authenticated storage, operations are processed in microseconds, rendering proof updates that require milliseconds per operation as relatively costly.

## 8 Conclusion

LVMT significantly reduces the disk I/O amplifications associated with each blockchain state access. When integrated into a high performance blockchain, LVMT has up to 2.7x higher throughput than the standard MPT structure. The promising results of LVMT demonstrate the potential of eliminating the performance bottleneck at the storage layer with vector commitment schemes.

## Acknowledgements

We express our gratitude to Peilun Li for his detailed guidance on the Conflux test framework, facilitating our end-to-end evaluations. We also appreciate the insightful suggestions from our shepherd, Micheal Wei, and the anonymous reviewers from EuroSys, S&P, and OSDI. Their critique and suggestions considerably improved our evaluation design and enriched our protocol discussion. This research has received support from the Shanghai Committee of Science and Technology, China (Grant No. 21511104600, 20DZ2221800), National Natural Science Foundation of China (Grant No. U2268202), and a gift fund from Nanjing Turing AI Institute.

## References


[1] Authenticated storage benchmarks. https://github.com/ChenxingLi/authenticated-storage-benchmarks.

[2] Conflux rust for authenticated storage benchmarks. https://github.com/Conflux-Chain/conflux-rust/tree/asb-e2e.

[3] DefiLlama - DeFi Dashboard. https://defillama.com.

[4] ERC-20 Top tokens. https://etherscan.io/tokens.

[5] Patricia Tree. https://eth.wiki/en/fundamentals/patricia-tree.

[6] Rainblock protocol. https://github.com/RainBlock/rainblock-protocol.

[7] Zero-knowledge rollups. https://ethereum.org/en/developers/docs/scaling/zk-rollups/.

[8] Vivek Bagaria, Sreeram Kannan, David Tse, Giulia Fanti, and Pramod Viswanath. Prism: Deconstructing the blockchain to approach physical limits. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 585–602, 2019.

[9] Paulo SLM Barreto, Ben Lynn, and Michael Scott. Constructing elliptic curves with prescribed embedding degrees. In *Proceedings of the 2002 International conference on security in communication networks*, pages 257–267. Springer, 2002.

[10] Paulo SLM Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. In *Proceedings of the 2005 International Workshop on Selected Areas in Cryptography*, pages 319–331. Springer, 2005.

[11] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In *Proceedings of the 23rd USENIX Security Symposium*, pages 781–796, 2014.

[12] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pages 326–349, 2012.

[13] Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching techniques for accumulators with applications to iops and stateless blockchains. In *Proceedings of the 2019 Annual International Cryptology Conference*, pages 561–586. Springer, 2019.

[14] Sean Bowe. BLS12-381: New zk-snark elliptic curve construction. https://z.cash/blog/new-snark-curve.

[15] Sean Bowe, Ariel Gabizon, and Matthew D Green. A multi-party protocol for constructing the public parameters of the pinocchio zk-snark. In *Proceedings of the 2018 International Conference on Financial Cryptography and Data Security*, pages 64–77. Springer, 2018.

[16] Sean Bowe, Ariel Gabizon, and Ian Miers. Scalable multi-party computation for zk-snark parameters in the random beacon model. *Cryptology ePrint Archive*, 2017.

[17] Vitalik Buterin. Ethereum whitepaper. https://ethereum.org/en/whitepaper/.

[18] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437*, 2017.

[19] Dario Catalano and Dario Fiore. Vector commitments and their applications. In *Proceedings of the 2013 International Workshop on Public Key Cryptography*, pages 55–72. Springer, 2013.

[20] Jemin Andrew Choi, Sidi Mohamed Beillahi, Peilun Li, Andreas Veneris, and Fan Long. LMPTs: Eliminating storage bottlenecks for processing blockchain transactions. In *Proceedings of the 2022 International Conference on Blockchain and Cryptocurrency*. IEEE, 2022.

[21] Arkworks contributors. `arkworks` zksnark ecosystem. https://arkworks.rs, 2022.

[22] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert Van Renesse. Bitcoin-ng: A scalable blockchain protocol. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation*, pages 45–59, 2016.

[23] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 51–68. ACM, 2017.

[24] Sergey Gorbunov, Leonid Reyzin, Hoeteck Wee, and Zhenfei Zhang. Pointproofs: Aggregating proofs for

multiple vector commitments. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 2007–2023, 2020.

[25] Jens Groth, Markulf Kohlweiss, Mary Maller, Sarah Meiklejohn, and Ian Miers. Updatable and universal common reference strings with applications to zk-snarks. In *Proceedings of the 2018 Annual International Cryptology Conference*, pages 698–728. Springer, 2018.

[26] Yilin Han, Chenxing Li, Peilun Li, Ming Wu, Dong Zhou, and Fan Long. Shrec: Bandwidth-efficient transaction relay in high-throughput blockchain systems. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 238–252, New York, NY, USA, 2020. Association for Computing Machinery.

[27] Koh Wei Jie. Perpetual Powers of Tau. https://github.com/weijiekoh/perpetualpowersoftau.

[28] Aniket Kate, Gregory M Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security*, pages 177–194. Springer, 2010.

[29] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy*, pages 583–598. IEEE, 2018.

[30] Russell WF Lai and Giulio Malavolta. Subvector commitments with application to succinct arguments. In *Annual International Cryptology Conference*, pages 530–560. Springer, 2019.

[31] Yoad Lewenberg, Yonatan Sompolinsky, and Aviv Zohar. Inclusive block chain protocols. In *Proceedings of the 2015 International Conference on Financial Cryptography and Data Security*, pages 528–547. Springer, 2015.

[32] Ao Li, Jemin Andrew Choi, and Fan Long. Securing smart contract with runtime validation. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 438–453. ACM, 2020.

[33] Chenxing Li, Peilun Li, Dong Zhou, Zhe Yang, Ming Wu, Wei Xu, Fan Long, and Andrew Yao. A decentralized blockchain with high throughput and fast confirmation. In *Proceedings of the 2020 USENIX Annul Technical Conference*. USENIX, 2020.

[34] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 17–30, New York, NY, USA, 2016. ACM.

[35] David Mazieres. The stellar consensus protocol: A federated model for internet-level consensus. *Stellar Development Foundation*, 32:1–45, 2015.

[36] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. http://bitcoin.org/bitcoin.pdf.

[37] Gleb Naumenko, Gregory Maxwell, Pieter Wuille, Alexandra Fedorova, and Ivan Beschastnikh. Erlay: Efficient transaction relay for bitcoin. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, page 817–831, 2019.

[38] Alex Ozdemir, Riad Wahby, Barry Whitehat, and Dan Boneh. Scaling verifiable computation using efficient set accumulators. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2075–2092, 2020.

[39] Parity Technologies. Crate kvdb. https://docs.rs/kvdb/0.4.0/kvdb/.

[40] Soujanya Ponnapalli, Aashaka Shah, Souvik Banerjee, Dahlia Malkhi, Amy Tai, Vijay Chidambaram, and Michael Wei. RainBlock: Faster transaction processing in public blockchains. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 333–347, 2021.

[41] Ethereum Improvement Proposals. Eip-20: Token standard. https://eips.ethereum.org/EIPS/eip-20, 2015.

[42] Pandian Raju, Soujanya Ponnapalli, Evan Kaminsky, Gilad Oved, Zachary Keener, Vijay Chidambaram, and Ittai Abraham. mlsm: Making authenticated storage faster in ethereum. In Ashvin Goel and Nisha Talagala, editors, *10th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2018, Boston, MA, USA, July 9-10, 2018*. USENIX Association, 2018.

[43] Pandian Raju, Soujanya Ponnapalli, Evan Kaminsky, Gilad Oved, Zachary Keener, Vijay Chidambaram, and Ittai Abraham. mLSM: Making authenticated storage faster in ethereum. In *Proceedings of the 10th USENIX*

*Workshop on Hot Topics in Storage and File Systems*, page 10, 2018.

[44] Yonatan Sompolinsky, Shai Wyborski, and Aviv Zohar. Phantom and ghostdag: A scalable generalization of nakamoto consensus. *Cryptology ePrint Archive 2018/104*, 2018.

[45] Yonatan Sompolinsky and Aviv Zohar. Secure high-rate transaction processing in bitcoin. In *Proceedings of the 2015 International Conference on Financial Cryptography and Data Security*, pages 507–527. Springer, 2015.

[46] Shravan Srinivasan, Alexander Chepurnoy, Charalampos Papamanthou, Alin Tomescu, and Yupeng Zhang. Hyperproofs: Aggregating and maintaining proofs in vector commitments. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3001–3018, 2022.

[47] Facebook Database Engineering Team. Rocksdb: A persistent key-value store for flash and ram storage. https://rocksdb.org, 2022.

[48] Parity Technologies. Openethereum. https://www.parity.io/ethereum/, 2019.

[49] Alin Tomescu, Ittai Abraham, Vitalik Buterin, Justin Drake, Dankrad Feist, and Dmitry Khovratovich. Aggregatable subvector commitments for stateless cryptocurrencies. In *Proceedings of the 2020 International Conference on Security and Cryptography for Networks*, pages 45–64. Springer, 2020.

[50] Alin Tomescu, Robert Chen, Yiming Zheng, Ittai Abraham, Benny Pinkas, Guy Golan Gueta, and Srinivas Devadas. Towards scalable threshold cryptosystems. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy*, pages 877–893. IEEE, 2020.

[51] Jiaping Wang and Hao Wang. Monoxide: Scale out blockchains with asynchronous consensus zones. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation*, pages 95–112, 2019.

[52] Haifeng Yu, Ivica Nikolić, Ruomu Hou, and Prateek Saxena. OHIE: Blockchain scaling made simple. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy*, pages 90–105. IEEE, 2020.

[53] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 931–948, 2018.

# Appendix

## Formal definition for inner nodes of AMT

We now provide formal definitions for the two elements associated with AMT inner nodes: a polynomial commitment and a batch proof about this polynomial function.

Since the auxiliary information is a binary tree, each node can be located by its depth and index. For a node indexed by $i$ at depth $d$, its left and right children are assigned indices $i$ and $i+2^d$, respectively. The root is indexed by 0.

Let $w$ be an $n$-th root of unity, such that $w^n = 1$, where $n = 2^k$ for some integer $k$. Given an input $\vec{a}$, AMT constructs the vector commitment to $\vec{a}$ to the polynomial commitment to $f(x) : \mathbb{F}_p \to \mathbb{F}_p$ that satisfies $f(w^i) = a_i$, where $\mathbb{F}_p$ is a prime field with order $p$. It is required that $2^k | p-1$.

The interpolation for points is applied on roots in the $\{i \in [n] \mid w^i\}$, instead of $[n]$. This yields a Lagrange function which supports faster algorithms. The Lagrange function is defined as:

$$I_{i,n}(x) = \frac{\prod_{j\in[n]\wedge j\neq i}\left(x-w^j\right)}{\prod_{j\in[n]\wedge j\neq i}\left(w^i-w^j\right)},$$

where the numerator can be simplified to

$$\prod_{j\in[n]\wedge j\neq i}\left(x-w^j\right) = \frac{x^n-1}{x-w^i} = \sum_{j=0}^{n-1}\left(x/w^i\right)^j,$$

and the denominator can be simplified to

$$\prod_{j\in[n]\wedge j\neq i}\left(w^i-w^j\right) = \sum_{j=0}^{n-1}\left(w^i/w^i\right)^j = n.$$

Thus, $f(x)$ is built via Lagrange interpolation as:

$$I_{i,n}(x) = \frac{1}{n}\cdot\frac{x^n-1}{x-w^i} \tag{2}$$

$$= \frac{\sum_{j=0}^{n-1}(x/w^i)^j}{n}. \tag{3}$$

So $f(x)$ can be constructed by Lagrange interpolation as

$$f(x) = \sum_{i=1}^{n} a_i \cdot I_{i,n}(x).$$

Now we consider a node at depth $d$ and index $t$, its associate function $f_{d,t}(x)$ only mirrors $f(x)$ at $x = w^i$ where index $i$ satisfies $i \equiv t \bmod 2^d$, and then covers only the Lagrange interpolation terms of these elements:

$$f_{d,t}(x) := \sum_{i\in T_{d,t}} a_i \cdot I_{i,n}(x), \tag{4}$$

where $T_{d,t} := \{i \in [n] \mid i \equiv t \bmod 2^d\}$. This node is associated with the commitment of function $f_{d,t}(x)$ and the batch proof demonstrating $f_{d,t}(w^i) = 0$ holds for all $i \in [n] \backslash T_{d,t}$. According to the KZG commitment, the commitment for $f_{d,t}(x)$ is $f_{d,t}(\tau) \cdot G_1$, and the batch proof is $h_{d,t}(\tau) \cdot G_1$, where $h_{d,t}(x)$ is defined by

$$h_{d,t}(x) := \frac{f_{d,t}(x)}{\prod_{i \in [n] \backslash T_{d,t}} (x - w^i)}, \tag{5}$$

with the denominator further simplifying to

$$\prod_{i \in [n] \backslash T_{d,t}} (x - w^i) = \frac{\prod_{i \in [n]} (x - w^i)}{\prod_{i \in T_{d,t}} (x - w^i)} = \frac{x^n - 1}{\prod_{i \in T_{d,t}} (x - w^i)}, \tag{6}$$

and where the denominator simplifies to

$$\prod_{i \in T_{d,t}} (x - w^i) = \prod_{i=0}^{2^{k-d}-1} \left( x - w^t \cdot \left( w^{2^d} \right)^i \right) = x^{2^{k-d}} - w^{t \cdot 2^{k-d}}. \tag{7}$$

For a leaf in subtree of this node with index $s$. If $a_s$ increases by 1, $f_{d,t}(x)$ and $h_{d,t}(x)$ will be updated accordingly. By equation 4, $f_{d,t}(x)$ will increases by $I_{s,n}(x)$, denoted as $\bar{f}_s(x)$. By equation 5, $h_{d,t}(x)$ will increase by $\bar{h}_{s,d}(x)$, defined as:

$$\bar{h}_{s,d}(x) := \frac{\bar{f}_s(x)}{\prod_{i \in [n] \backslash T_{d,t}} (x - w^i)},$$

which can be simplified by equation 2, 6 and 7:

$$\begin{aligned} \bar{h}_{s,d}(x) =& I_{s,n}(x) \cdot \frac{x^{2^{k-d}} - w^{s \cdot 2^{k-d}}}{x^n - 1} \\ =& \frac{1}{n} \cdot \frac{x^{2^{k-d}} - w^{s \cdot 2^{k-d}}}{x - w^s} \\ =& \frac{1}{n} \cdot \sum_{j=0}^{2^{k-d}-1} (w^s)^j \cdot x^{2^{k-d}-j}. \end{aligned}$$

In AMT, when increasing $a_s$ by $\delta$, the commitments and proofs of the node along the path from the root to the corresponding leaf will increases by $\delta \cdot \bar{f}_s(\tau) \cdot G_1$ and $\delta \cdot \bar{h}_{d,s}(\tau) \cdot G_1$ respectively. The sequence of $\{\bar{f}_s(\tau) \cdot G_1\}_{s=1}^n$ and $\{\bar{h}_{d,s}(\tau) \cdot G_1\}_{s=1}^n$ for any $d$ can be constructed by FFT. So the AMT can precompute $O(n \log n)$ cached parameters in $O(n \log^2 n)$ time and update the associated elements of each node with two multiplications and two additions on the elliptic curve.

## The overhead of storing the append-only Merkle trees

We provide a rough estimation of the overhead for storing truncated Merkle trees after garbage collection. Considering the roots of Merkle trees are organized in a tree, we can treat them as one large Merkle tree. We assume a full binary Merkle tree has $k$ levels of inner nodes, accumulated $m = 2^k$ version tuples, with $n$ tuples currently active, where $2^l \leq n < 2^{l+1}$ for some integer $l$. A node is not truncated if either itself or its sibling has active descendants, so each active tuple corresponds to at most two nodes per level. The bottom $k - l - 1$ layers have at most $2n \cdot (k - l - 1)$ nodes, less than $2n \cdot \log_2(m/n)$. The first $l + 1$ levels have $2^{l+1} - 1$ nodes, less than $2n$. Therefore, the maximum node count is $(\log_2(m/n) + 1) \cdot 2n$.

**Algorithm 8** A procedure to prove a given key version. It returns the proof of the key version.

```
1:  procedure ProveKey(k)
2:      (tidx, leaf) ← LeafAtLevel(lv, k);
3:      vers ← leaf.vers;
4:      C ← AM[(lv, tidx)].comm;
5:      (e, i) ← LM[k];
6:      val ← KM[k];
7:      (lv, sidx) ← VM[k];
8:      merklepf ←Prove the existence of (k, vers[sidx], val, lv, sidx)
        w.r.t. the current hroot
9:      amtpf ←Prove vers are the version numbers w.r.t. the commitment
        C
10:     return (merklepf, amtpf, vers, sidx, val, C);
```

**Algorithm 9** A procedure to prove the level lv and the tree index tidx of a sub-AMT. It returns the proof of the commitment of the sub-AMT.

```
1:  procedure ProveCom(lv,tidx)
2:      ptidx ← ⌊tidx/n⌋;
3:      plidx ← tidx mod n;
4:      vers ← AM[(lv − 1, ptidx)].leaves[plidx].vers;
5:      C_p ← AM[(lv − 1, ptidx)].comm;
6:      C ← AM[(lv, tidx)].comm;
7:      (e, i) ← LM[(lv − 1, ptidx)];
8:      merklepf ←Prove the existence of (lv, tidx, vers[0], C) w.r.t. the
        current hroot
9:      amtpf ←Prove vers are the version numbers w.r.t. the commitment
        C_p
10:     return (merklepf, amtpf, vers, C_p);
```

## Non-existence proof of LVMT

The process for generating a non-existence proof in LVMT is depicted in Algorithm 10. This procedure proves the non-existence of a key k by demonstrating that all potential version number slots for the key are already allocated to other keys.

It first allocate a version slot for k and followed by an immediate rollback of the allocation (lines 2-3). This process finds the next vacant slot for k.

Then, it proves the version number of this slot is zero, a process similar to Algorithm 6 except that it omits the merkle proof of the key (lines 5-12). This demonstrates that the slot is indeed unoccupied.

Last, it shows that all other potential slots for k are already allocated to different keys. It generates proof for them; the second fields of these proofs can be omitted since they have the same information as commpfs computed in line 11.

Thus, a non-existence proof in LVMT proves the absence of a key by showing that all its potential slots are occupied by other keys.

**Algorithm 10** A procedure to compute the non-existence proof for a given key.

```
 1: procedure NonExistanceProof(k)
 2:     (lv, sidx) ← AllocateSlot(k);
 3:     Roll back the changes in allocating slot for k
 4:     (tidx, leaf) ← LeafAtLevel(lv, k);
 5:     vers ← leaf.vers;
 6:     C ← AM[(lv, tidx)].comm;
 7:     amtpf ←Prove vers are the version numbers w.r.t. the commitment
    C
 8:     zeropf ← (amtpf, vers, sidx, C);
 9:     while lv > 0
10:       tidx ← first bit to (k · lv)-th bit of H(k);
11:       commpfs[lv] ← ProveCom(lv, tidx)
12:       lv ← lv − 1;
13:     L ← [ ];
14:     for i ∈ [sidx − 1]
15:       keypf ← the first component of prove(leaf.keys[i]);
16:       L ← (leaf.keys[i], keypf) ∪ L;
17:     while lv > 0
18:       lv ← lv − 1;
19:       (tidx, leaf) ← LeafAtLevel(lv, k);
20:       for i ∈ [5]
21:         keypf ← the first component of prove(leaf.keys[i]);
22:         L ← (leaf.keys[i], keypf) ∪ L;
23:     keypfs ← L;
24:     return (zeropf, commpfs, keypfs);
```