LVMT: An Efficient Authenticated Storage for Blockchain

Chenxing Li
Shanghai Tree-Graph Blockchain Research Institute

Sidi Mohamed Beillahi
University of Toronto

Guang Yang, Ming Wu, Wei Xu
Shanghai Tree-Graph Blockchain Research Institute

Fan Long
University of Toronto

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. Our experiments show that read and write operations are 7x faster in LVMT than in an MPT. Our experiments also show that LVMT increases the execution throughput of a blockchain system by up to 3.1x.

1 Introduction

Blockchains that provide decentralized, robust, and programmable ledgers at internet scale have recently gained increasing popularity across different domains such as financial services, supply chain, and entertainment. For example, smart contracts built on top of blockchain systems now manage digital assets worth tens of billions of dollars [1].

Although the early classical blockchain systems like Bitcoin [32] and Ethereum [13] have serious performance bottlenecks in their consensus protocols which limit the system throughput under 30 transactions per second, recent technique evolutions on consensus and peer-to-peer network protocols [6,17,18,21,24,26,28,31,33,39,40,46,47] have driven the achievable blockchain throughput to more than thousands of transactions per second. This makes the transaction execution, which is dominated by the storage access, the new bottleneck of the blockchain system. Our investigation (see Sec. 6) shows that 87% of the transaction execution time is consumed at the storage layer.

The inefficiency of the blockchain storage layer originates from the requirement of the *authentication*. In a standard permission-less blockchain system, there exist two types of blockchain nodes: the full nodes and the light nodes. A full node synchronizes and executes all the transactions, and maintains the blockchain ledger state accordingly. A light node (client) only synchronizes the block headers without transactions and blockchain ledger state. The ledger state of a blockchain is in the form of key-value pairs. When a light node wants to know the value of the given key, it must query to a full node. However, because blockchain nodes are permissionless, the light node should not trust the response from the full node. Therefore, the blockchain protocol requires the block proposer to compute a commitment (called the *state root*) for the latest ledger state and fill it in the proposed block header. A block header with an incorrect commitment is invalid. When responding to the queries from light nodes, a full node can generate proofs with respect to the corresponding commitments to convince the queriers. Thus, the ledger state is called authenticated.

The authenticated storage typically employs the structure of Merkle Patricia Trie (MPT) [3], a specific form of Merkle tree. Each leaf node in an MPT stores a value and, as a trie, 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 root hash of the MPT serves as the commitment of the blockchain state for authentication.

Unfortunately, this authentication comes with a heavy performance price. Modifying one key-value pair in the state requires an MPT to update hashes of all nodes in the path from the corresponding leaf node to the root. If not cached,

each state update operation will be amplified into $O(\log n)$ disk I/O operations, where n is the size of storage. Note that even a simple payment transaction contains at least two ledger state updates, i.e., deducting and increasing balances of senders and receivers, respectively. As the throughput of recent blockchains approaches thousands of transactions per second, it is not surprising that the 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 combining 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 [45]. Although it has constant commitment update time, we still face several key challenges when fitting AMT in LVMT design.

The first challenge is that the AMT commitment update algorithm uses expensive elliptic curve multiplication operations. A naive approach would cause a state update operation on AMT slower than MPT in practice in spite of the reduced amplification. LVMT addresses this challenge with its novel *key-versioned-value design*. LVMT assigns the value of each key a version that increments as the value changes. LVMT uses AMT to store key-version pairs instead and uses Merkle Trees to maintain an append-only authenticated list of key-version-value triples. Each update to LVMT will therefore translate into an increment of the stored version in AMT. Because during a commitment update, 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), 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 is that AMT cannot support enough bits for blockchain state keys. An AMT with k-bit key-space requires public parameters with 2^k elliptic curve points. To enable efficient update, AMT also needs to pre-compute and cache elliptic curve points proportional to the same size of public parameters. Even when the number of bits is 32, the pre-computed metadata size would exceed 256 GB, but keys in blockchain ledgers typically have 256 bits. To address this challenge, LVMT operates with multiple AMTs organized in a novel *multi-level multi-slot structure*. Each AMT in the structure has a 16-bit key-space and when a collision happens the structure will automatically create a sub-AMT in the next level to hold collided key-version pairs. Also, because 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. LVMT therefore only expands to the next level if we have more than five collisions.

The third challenge is that maintaining the proof generation metadata is expensive. Unlike just updating the root hash for AMT which takes constant time, maintaining the proof generation metadata still takes $O(\log n)$, and it would trigger the same amount of I/O amplifications as MPT. To address this challenge, LVMT uses a *proof sharding technique* to parallelize the proof generation workload across the blockchain network. In LVMT, each full node only maintains the proof generation metadata for a shard of the blockchain state (e.g., the keys share the same 4-bit prefix). Our observation is that there are typically thousands of full nodes in a production blockchain and that having all nodes to maintain proof generation capabilities for all key-value pairs in the entire state is unnecessary. Even sharded, for any part of the state, there will still be enough nodes serving proof generation requests from light clients. Note that different from other sharding designs [14, 24, 30, 46, 48], our proof sharding does not sacrifice any security. All full nodes in LVMT still maintain the entire blockchain state, execute all transactions, update the root hash of LVMT, and verify all blocks. Only the proof generation workload is sharded.

We implemented LVMT and integrated it into Conflux [28], a high-performance blockchain production with smart contract support. We compared LVMT with MPT on both the stand-alone read/write workload and the end-to-end blockchain processing workload. 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 3.1x higher simple payment transaction throughput and up to 2.7x higher ERC20 [36] token transfer transaction throughput. The performance improvement comes from the significantly reduced disk I/O amplifications. The amplification of LVMT is up to 4x less than MPT on reading operations and is up to 11x less than MPT 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) [45], an efficient vector commitment protocol.

Notations: Let $[n]$ to denote the integers in $\{x \in \mathbb{Z}^+ | 1 \leq x \leq n\}$. $\mathbb{G}$ denotes an elliptic curve group and symbols in upper

cases like G, P denote the elements in the elliptic curve groups. $\mathbb{Z}_p$ denotes 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: the full nodes and the light nodes. A full node will synchronize and executes all the transactions and maintain the blockchain ledger state accordingly. A light node (client) only synchronizes the block headers without transactions and blockchain ledger state.

When a full node proposes a new block, the blockchain protocol requires it to execute transactions in the block and put the commitment of the ledger state after execution in the block header. So a blockchain full node maintains a write-back cache in transaction execution and flushes all the changes to the storage after executing all transactions in a block. The authenticated storage is required to provide two interfaces to the execution engine:

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

When a light node wants to know the value of a given key, it will query with a full node and expect the full node returns the value with proof with respect to the ledger commitment. The light client will check whether the commitment is a know valid commitment and verify the correctness of the proof. So the authenticated storage is required to provide two algorithms for proving and verifying:

- $\mathsf{Respond}(\mathsf{k}) \to (\mathsf{v}, \pi, \mathsf{comm})$: Answeres the value v of key k with proof π with respect to the most recent commitment comm.
- $\mathsf{Verify}(\mathsf{k}, \mathsf{v}, \pi, \mathsf{comm}) \to 0/1$: Verifies the response from the full node.

2.2 KZG Commitment

In [23], Kate et. al. proposed KZG polynomial commitment protocol, which allows someone to commit a polynomial function f to a commitment, and prove the value $f(x)$ of any given position x to verifiers with the commitment.

KZG commitment protocol is built on a bilinear map. Let G_1 and G_2 be the starting points of two elliptic curve groups $\mathbb{G}_1, \mathbb{G}_2$ of the same group order p respectively. A bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is homomorphic such that $e(a \cdot G_1, b \cdot G_2) = ab \cdot e(G_1, G_2)$ holds for any $a, b \in \mathbb{Z}_p$, where $\mathbb{G}_T$ is another group with the same order p. BLS12-381 [10] from BLS families [7] and BN254 [9] from BN families [8] are widely-used deployed systems that implement bilinear maps. $\mathbb{G}_1$ and $\mathbb{G}_2$ are elliptic curve groups in order of p, and G_1 and G_2 are starting points of $\mathbb{G}_1$ and $\mathbb{G}_2$.

For a given polynomial function $f : \mathbb{Z}_p \to \mathbb{Z}_p$ in the degree of n, 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[1] and commits function f to C, $C := f(\tau) \cdot G_1$.

For any index $i \in \mathbb{Z}_p$, $x - i$ divides $f(x) - f(i)$ (i.e., $h_i(x) := \frac{f(x)-f(i)}{x-i}$ is a polynomial). The proof π of $f(i)$ is defined by $h_i(\tau) \cdot G_1$. As long as the prover can compute the coefficient of $h_i(x)$, $h_i(\tau) \cdot G_1$ is a linear combination of public parameters with known coefficients. A verifier querying i with answer $y = f(i)$ and proof $\pi := h_i(\tau) \cdot G_1$ can verify the proof by $e(\pi, (\tau - i) \cdot G_2) = e(C - y \cdot G_1, G_2)$.

If the proof π is constructed correctly, $e(\pi, (\tau - i) \cdot G_2) = (h(\tau) \cdot (\tau - i)) \cdot e(G_1, G_2) = e((f(\tau) - f(i)) \cdot G_1, G_2)$. So it must pass the check. If $f(i) \neq y$, $h(x)$ is a fraction. So it is hard to find a proper proof without knowing τ. In [23], the authors proved the binding property of this protocol.

KZG commitment also supports proving a batch of positions. To prove that $f(x)$ equals to 0 at a set of positions S, the proof π 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 $f(x) = \sum_{i=1}^{n} a_i \cdot I_i(x)$, where a_i is the i-th element of $\vec{a}$ and $I_{i,n}(x)$ is a Lagrange function satisfying $I_{i,n}(i) = 1$ 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

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

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

2.3 Authenticated Multipoint Evaluation Tree

Although KZG commitment can update the commitment C in constant time, it requires $O(n)$ time to construct a proof for a position or maintain proofs for all the position. In a blockchain system, the vector to be committed is changing. KZG commitment cannot generate proofs efficiently for queries with arbitrary index i.

[1] A trusted party generates the public parameters with a random τ and forget τ after generation. Secure multi-party computation protocols for the trusted setup [11, 12, 20] allows multiple participants to generate the public parameters collaboratively while no one can learn the exact τ. More details are in the supplymentary material.

In [45], the authors propose authenticated multipoint evaluation trees (AMT) commitment protocol to solve this problem. AMT maintains auxiliary information in the size of $O(n \log n)$ and can generate a proof in $O(\log n)$ time.

We illustrate the high-level idea of AMT through 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 $1 \leq i \leq 8$. AMT splits $f(x)$ into two functions $f_0(x)$ and $f_1(x)$ in the same degree as $f(x)$ with $f(x) = f_0(x) + f_1(x)$.

$$f_0(x) := \begin{cases} f(x) & x \in \{2,4,6,8\} \\ 0 & x \in \{1,3,5,7\} \end{cases}$$

$$f_1(x) := \begin{cases} f(x) & x \in \{1,3,5,7\} \\ 0 & x \in \{2,4,6,8\} \end{cases}$$

Similarly, AMT also divides $f_0(x)$ into two functions $f_{0,0}(x)$ and $f_{0,1}(x)$ with $f_0(x) = f_{0,0}(x) + f_{0,1}(x)$. $f_{0,0}(x)$ equals $f(x)$ for $x \in \{4,8\}$ and $f_{0,1}(x)$ equals $f(x)$ for $x \in \{2,6\}$. Repeating this process recursively gives a full binary tree. Each node corresponds to a function (e.g., $f_{0,1}(x)$). Each leaf is a multiplication of a Lagrange function (e.g., $f_{0,0,1}(x) = a_4 \cdot I_{4,8}(x)$)

AMT associates each inner node with two elements: 1) the KZG commitment of the corresponding function and 2) a batch proof for the indices that always be zero of this function. The formal definition for associated elements is provided in the appendix. For example, for the node of function $f_{0,0}(x)$, AMT maintains its commitment $f_{0,0}(\tau) \cdot G_1$ and a batch proof $\frac{f_{0,0}(\tau)}{\prod_{1 \leq i \leq 8 \wedge i \notin \{4,8\}} (\tau - i)} \cdot G_1$. When proving the value of a given entry, e.g., a_4, since $f(x) = f_1(x) + f_{0,1}(x) + f_{0,0,0}(x) + a_4 \cdot I_{4,8}(x)$, the prover outputs the associate commitments $(f_1(\tau) \cdot G_1, f_{0,1}(\tau) \cdot G_1, f_{0,0,0}(\tau) \cdot G_1)$ and proves $f_1(4) = f_{0,1}(4) = f_{0,0,0}(4) = 0$ by the associated batch proofs. The verifier checks the correctness of batch proof and if $f(\tau) \cdot G_1 = (f_1(\tau) + f_{0,1}(\tau) + f_{0,0,0}(\tau) + a_4 \cdot I_{4,8}(\tau)) \cdot G_1$.

When updating one entry, AMT finds the path from the root to the leaf corresponding to that entry and update the associate elements on the path. The rest nodes have no change. So AMT can take $O(\log n)$ time to maintain the proofs.

The nodes are *auxiliary information* for generating proofs only. For a blockchain miner which does not serve client queries, it can skim the auxiliary information and only maintain the commitment, which takes $O(1)$ time.

3 Overview

Recent works [27, 37] have shown that the majority of transactions execution time is spent on operations that access the blockchain state. For instance, a profiling experiment done in [27] 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 [2, 36]. 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.

Pairing engines	BLS12-381	BN254
Addition	682	341
Multiplication	168,863	92,042

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

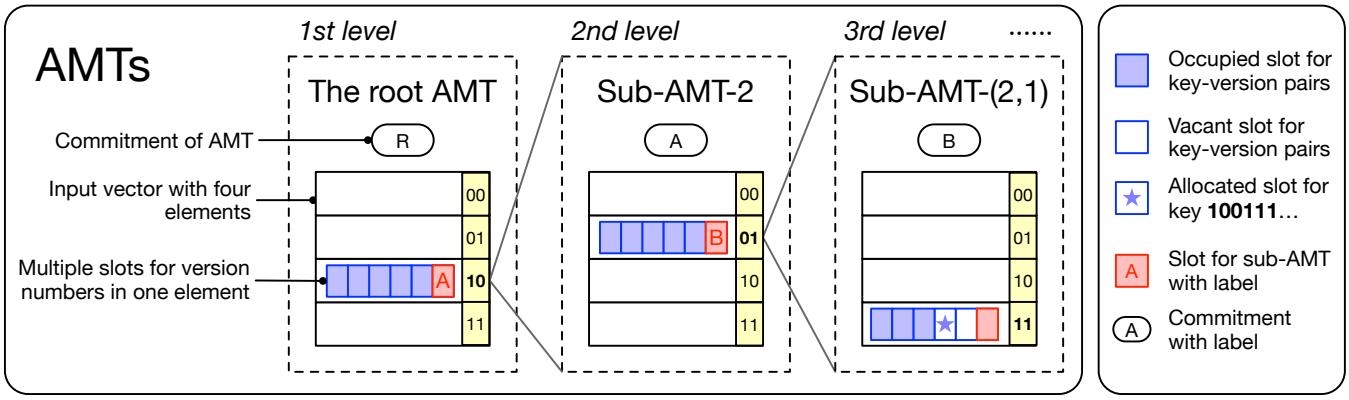
Our proposed system is based on AMT since it has an ideal time complexity, i.e., constant cost in updating the commitment. In particular, our proposed system solves several challenges to implement an efficient blockchain storage system using AMT:

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 than accessing the disk.

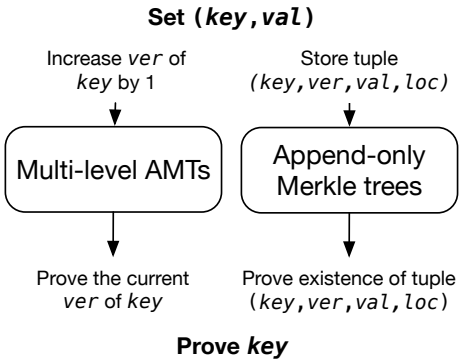
Second, to support data with n maximum entries, AMT requires precomputed parameters in size of $O(n \log n)$ and maintains auxiliary information in size of $O(n \log^2 n)$. Thus, AMT cannot support key-value pairs for an arbitrary-length bit string. As the size of the blockchain ledger state continues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

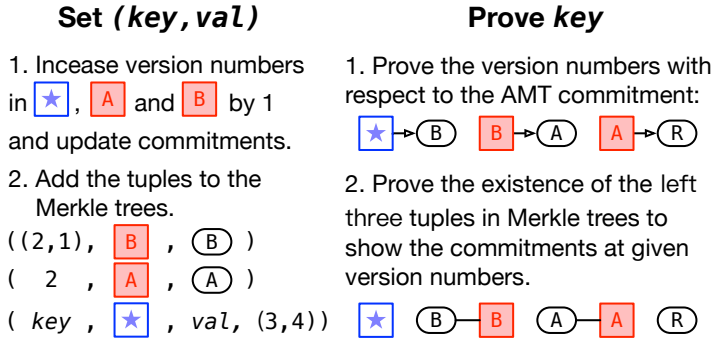
We propose the following techniques to resolve the challenges above. First, we design a versioned database, which only stores the version number of keys in AMT. This can avoid the elliptic curve multiplications and support an arbitrary length of values since the values are not stored in AMT. Second, we extend AMT to multiple levels to store version numbers for unlimited keys. So the size of AMT could be relatively small to save cache for parameters. To support the arbitrary length of keys and avoid deep updates in the multi-level hierarchy, we use key hashes to allocate slots for version numbers. Last, we introduce proof sharding to reduce the cost of a single node in maintaining the auxiliary information for proofs.



(a) Multi-level AMTs



(b) Versioned key-value database



(c) Maintenance and proving on Multi-level AMTs

Figure 1: LVMT architecture.

3.1 Versioned Key-value Database

We designed a versioned authenticated storage to avoid multiplication on elliptic curve in updating commitment. As shown in Figure 1b, the multi-level AMTs stores key-version pairs and are only used to resolve the recent version number of a key. LVMT stores the key-version-value tuples in an append-only authenticated data structure consisting of a series of Merkle trees. Each block constructs one Merkle tree from the key-version-value tuples for value changes in the block.

Suppose when the blockchain processes a block, it sets 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. Suppose the new version number for key is ver. LVMT then appends a new tuple (key, ver, val, loc) to the constructed Merkle tree for the block. Here, loc is a tuple $(level, slot)$ that records at which level and which slot in the multi-level AMTs the version of the key is located. 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). Note that because these Merkle trees will never be modified once they are constructed, their maintenance is very efficient.

Note that updating one element a_i to a'_i requires computing $(a'_i - a_i) \cdot I_{i,n}(\tau) \cdot G_1$ (equation 1), which multiplies $a'_i - a_i$ 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 to 1, so we avoid an elliptic curve multiplication and save about 100 us in each storage write.

Since the frequency of bumping version number is limited by the block generation rate, we can then save the bits used for storing version number and store multiple version numbers in a single vector entry. For example, when using BN254 as the underlying bilinear mapping parameter, each entry is an element in $\mathbb{Z}_p$, where p is a prime integer in $(2^{254}, 2^{255})$. It implies each entry can store at most 254 bits. In a blockchain system generating 100 blocks per second, the version number will not exceed 2^{40} in 300 years. So each entry can be divided into six slots with 40 bits as shown in Figure 1a. Note that the block rate of 100 blocks per second is already unreasonably high according to the practical setup of typical blockchain systems.

3.2 Multi-level AMT

To make AMT scalable and store the version number for unlimited 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, and the other slots are for key-value pairs. Let k be the height of AMT. When allocating a slot for a new key, LVMT accesses the entry in the root AMT whose index is the first k bits of the key hash. If this entry does not have a vacant slot, it accesses the sub-AMT corresponding to this entry and accesses the entry in the sub-AMT whose index is the next k bits of the key hash. LVMT visits the sub-AMTs recursively to find a vacant slot for a new key. Figure 1a presents an example with $k = 2$ for allocating a version slot for a key with hash $100111\cdots$. Since the first two bits of key hash are 10, LVMT accesses the entry with index 2 and tries to find a vacant slot. Since all the slots in the entry are occupied, LVMT visits the corresponding sub-AMT-2. It picks the next two bits 01, 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 fourth slot at the third level being vacant and allocates this slot.

The commitment of a sub-AMT is treated similar with a key-value pair, where the key is the index of the sub-AMT and the value is the commitment. The Merkle trees not only store key-version-value tuples for normal key-value pairs, but also stores the tuples of the sub-AMT index, the version of the sub-AMT commitment, and the commitment hash.

Figure 1c illustrates when a block changes the key with hash $100111\cdots$, how LVMT maintains the AMTs and Merkle trees. LVMT first increases the version number for this key by one. This changes the commitment of sub-AMT-(2,1), so LVMT also increases the version number for the commitment (the slot labeled "B") by one. And 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 proves the correctness of appeared version numbers with respect to their AMT commitments. When proving the non-existence of a key, LVMT proves that all the possible slots for this key are vacant or has been allocated to other keys.

3.3 Proof Sharding

We recall that the AMT maintains a binary tree, in which each node contains a commitment and a batched proof. Each entry in the input corresponds to a leaf in this tree. When generating a proof, AMT picks commitments and batch proofs from the siblings of the nodes on the path from this leaf to the root. Each node can be updated without the information from the other nodes, so the maintenance of the proof tree is easy to parallelize. Each blockchain node can maintain a shard of proof. It picks a subtree of the root AMT and is responsible for generating proofs for the leaves in this subtree and the sub-AMTs extended from this leaf. Multiple blockchain nodes can generate proof for any key collaboratively. 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 keys version slots metadata, and LM stores the position of a key or a sub-AMT in the Merkle trees. LVMT decouples the data storage and data authentication. KM stores data without authentication, AM and MM store the authenticated information, VM and LM store the metadata and indices for authenticated information. Note that each AMT contains the following elements:

- comm: the commitment of AMT;
- $\mathsf{proof\_tree}$: the proof tree of AMT;
- leaves: a list of leaves; $\mathsf{leaves}[i]$ denotes the leaf corresponding to the i-th element of the input vector. A leaf is constituted of the two lists vers and keys. $\mathsf{vers}[0]$ stores the version number for the sub-AMT. $\mathsf{vers}[1]$ to $\mathsf{vers}[5]$ store the version numbers for the keys $\mathsf{keys}[1]$ to $\mathsf{keys}[5]$, 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:

- $\mathsf{Get}(\mathsf{k}) \to \mathsf{val}$: Reads the value val stored in k;
- $\mathsf{Commit}(\mathbf{W}, e) \to (\mathsf{aroot}, \mathsf{hroot})$: 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

Algorithm 1 A procedure to compute a commit changes. It takes the list of key-value pairs $\mathbf{W}$ and an epoch 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);
```

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.

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 that maintains the version number for commitments of sub-AMTs similar to COMKV. While maintaining the version numbers, it collects the tuples of keys, versions, values, and other metadata in a list $\mathbb{M}$ (Line 5). A pair of the sub-AMT index and its commitment

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, tidx);
5:        for j ∈ [5]
6:          if leaf.vers[j] == 0
7:            leaf.keys[j] ← k;
8:            return (lv, j);
9:        lv ← lv + 1;
```

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

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

is treated similarly to a key-value pair (Line 11). LVMT builds a Merkle tree for $\mathbf{M}$, so the value of the given key and version is authenticated (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 used 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 for 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 then returns the level and slot indices of the slot, otherwise, it moves to the next level in case the leaf has no vacant slot. 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

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);
```

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 amtroot
4:     if all the verification pass
5:       return true;
6:     else
7:       return false;
```

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

When responding a query k from a light node, a full node will generate proof π using the procedure Prove and response 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 (except 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 metadata. 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 introduce 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 be verified from the proof and verifies the AMT proofs and the merkle proofs.

5 Implementation

We implemented AMT using Arkworks [4], 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. Based on the above AMT implementation, we implemented LVMT in Rust. LVMT supports any backend database that provides a key-value interface defined in rust crate "kvdb" [34]. We also ported the implementation of MPT from the OpenEthereum client [43], which also supports any backend with the same interface.

Combining entries in different maps: Note that 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. In our implementation, we combine these entries into one key-value pair to save the read and write operation for each key.

Cache the root AMT: Since the root AMT is accessed frequently, the leaves and inner nodes of the first layer are always stored in memory. The commitments of the LVMT in the second layer are also cached. Each leaf and inner node of an AMT has two points on the elliptic curve. Since we set the height of AMT as 16, the root AMT and the commitments of AMTs in the second layer stores about 200000 elliptic curve points in memory. Each point takes 96 bytes in our parameter, so it takes $20MB$ memory to store them.

Cache cryptographic parameters: We recall that the commitment of AMT with polynomial function $f(x)$ is $C = f(\tau) \cdot G_1$. So when increasing the input entry $\vec{a}_i$ by δ, the polynomial function will be updated to $f'(x) = f(x) + \delta \cdot I_{i,n}(x)$, where n is the length of vector input and I is the Lagrange function. So the commitment will be updated to $C' = C + \delta \cdot I_{i,n}(\tau) \cdot G_1$. Thus, LVMT can precompute the elliptic curve point $P_i = I_{i,n}(\tau) \cdot G_1$. Each time the i-th entry is updated from $\vec{a}_i$ to $\vec{a}'_i$, the commitment can be updated by $C' = C + \delta \cdot P_i$. In LVMT, an input entry only changes when the version number increases by 1. As each entry is divided into six slots with 40

bits, the difference between the new value $\vec{a}'_i$ and the old value $\vec{a}_i$ must be one of $1, 2^{40}, 2^{80}, 2^{120}, 2^{160}, 2^{200}$. LVMT precomputes $P_i^{(j)} = 2^{40j} \cdot I_{i,n}(\tau) \cdot G_1$ for $0 \leq j \leq 5$ and $1 \leq i \leq n$, so no matter which version slot changes, LVMT only needs to update the commitment by increasing a precomputed point. Each elliptic curve point takes 96 *bytes*, so a node not maintaining proofs needs only 37 *MB* memory to store them. A node maintaining a shard of proof needs to cache more parameters, which will take about 650 *MB* memory in total.

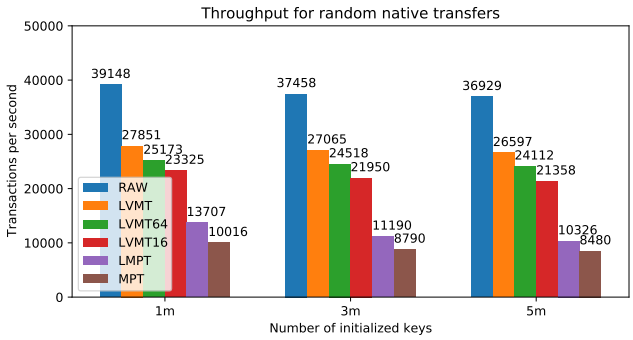
Representation of elliptic curves points: Each point on the elliptic curve has a unique affine coordinate $(x, y) \in \mathbb{Z}_p^2$, where p is a large prime number. A projective coordinate $(x, y, z) \in \mathbb{Z}_p^3$ can also represent an elliptic curve point whose affine coordinate is $(x/z^2, y/z^3) \in \mathbb{Z}_p^2$. Elliptic curve addition on projective coordinates is much faster than it on affine coordinates since it allows to avoid the slow operation, i.e., division on a large prime field, on the projective coordinates. [2] Since elliptic curve points may have multiple projective coordinates, it may cause inconsistency when computing the hash for an elliptic curve point by its projective coordinate. So LVMT must convert the projective coordinates to the affine coordinates when computing the hash of a sub-AMT commitment. However, each conversion takes about 60 us in our experiment, which significantly slows down the write speed. Fortunately, if we convert the projective coordinates in batch, the average time cost can be reduced to 0.4 us. Thus, LVMT postpones computing the hash of elliptic curve points to the end of executing one block and converts all the projective coordinates to the affine coordinates in batch.

Garbage collection of append-only Merkle trees: As the version number of a key increases, the tuples of the old versions of the key in the append-only Merkle trees will no longer be required in future proofs. When the Merkle tree of an old block only contains one or two useful entries, LVMT can garbage collect the remaining entries and instead store only the path from these useful entries to the root. This garbage collection is performed by a background thread to avoid impacting the performance of LVMT under a heavy workload.

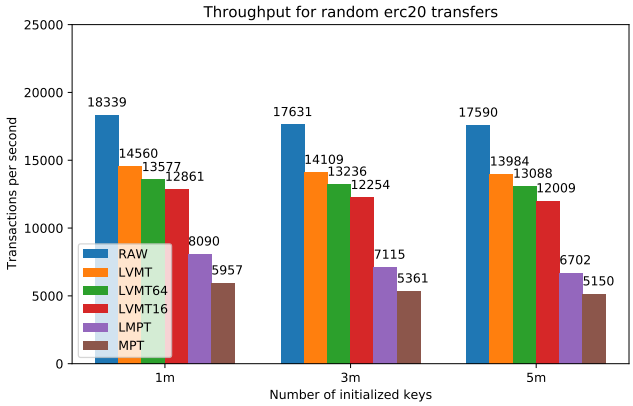
6 Evaluation

We evaluate the performance of LVMT and compare with MPT on a machine equipped with Intel i9-10900K CPU, 32 GB DDR4 RAM, and SSD. Both LVMT and MPT use rocksdb [42] as the backend key-value database with an 1500 MB cache budget. To save the time cost in cryptographic hash function, we replace the keccak hash function with blake2b [5]. We evaluate the end-to-end performance

[2] The benchmarks in Table 1 is computed on projective coordinates.



(a) Transaction execution for balance transfers.



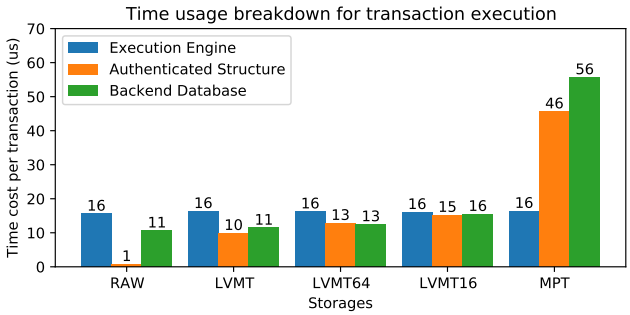
(b) Transaction execution for ERC-20 transfers.

Figure 2: Breakdown of time cost in executing one transaction.

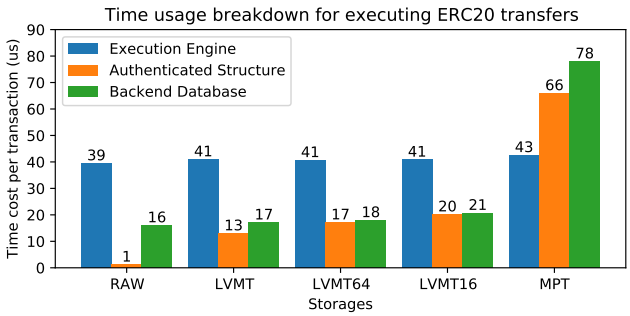
of authenticated storage in Conflux [29], a high-performance blockchain. A single Conflux node executes transactions with different storages:

- LVMT: LVMT without auxiliary information (proof sharding),
- LVMT64 and LVMT16: LVMT with 1/64 and 1/16 of proof sharding,
- MPT: the authenticated storage in OpenEthereum, one of the most popular Ethereum clients,
- RAW (non-authenticated): the read and write operations are applied on the key-value based backend directly.

End-to-end performance: To measure the maximum performance, we set an extremely large block generation rate of 100 blocks per second. In the experiment, 20000 senders randomly pick addresses in the receiver space and transfer the non-zero balance to them, i.e., simple payment transactions. Each sender is initialized with enough balance. We evaluate the receiver space with one million, three million, and five million addresses. We run Conflux for a sufficiently long time, so the number of executed transactions is five times larger than the receiver space. We also compare the end-to-end per-



(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.

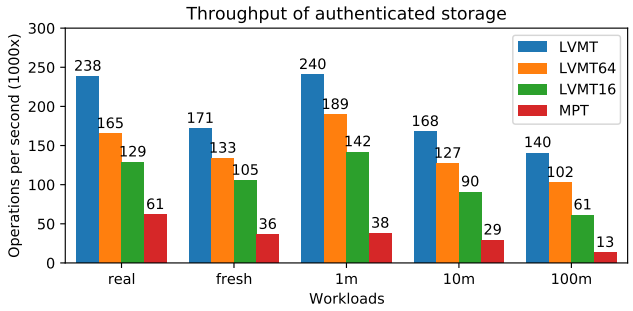


Figure 4: Throughput of authenticated storage on random workload

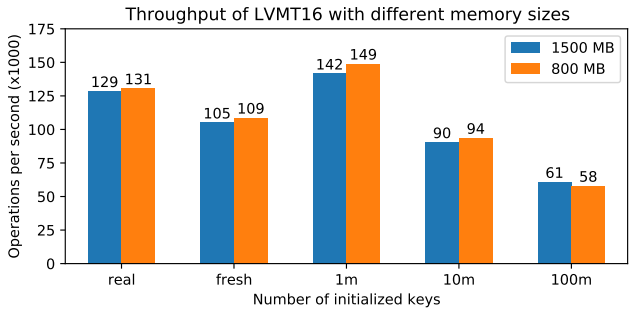


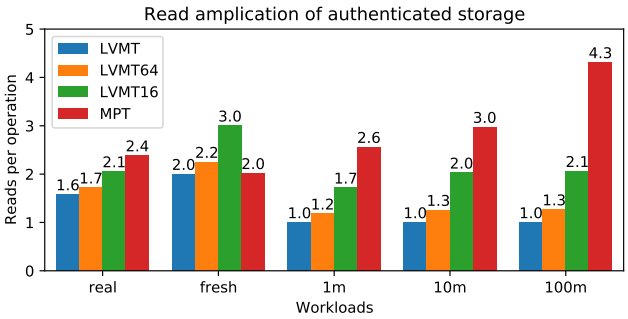
Figure 5: Throughput of authenticated storage on random workload

formance with LMPT [16], a variant of MPT, in the same code base. Figure 2a shows that LVMT achieves a maximum throughput of 27851 TPS on average and is up to 3.1 times faster than MPT and 2.5 times faster than LMPT. We also evaluate the performance of transactions executing the transfer function of the popular ERC-20 smart contract [36], the most common transactions on the Ethereum blockchain [2]. As shown in Figure 2b, LVMT is up to 2.7 times faster than MPT and 2.0 times faster than LMPT in this workload.

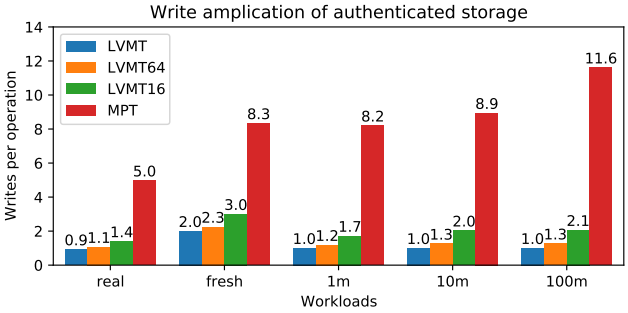
To further study the time usage in execution of one transaction, we breakdown the time usage into three parts: 1) Execution Engine, i.e., transactions execution without access to the authenticated storage, 2) Authenticated Structure, i.e., access to the authenticated storage without accesses the backend 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 takes a similar time 11 us with RAW in accessing the backend. It implies LVMT almost eliminates the overhead of the authenticated storage from backend access. LVMT64 and LVMT16 take a similar time to LVMT. MPT requires 46 us and 56 us to access the authenticated structure and the backend database, respectively, which is more than 4x the time used in LVMT. 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 in micro-benchmarks. Since most transactions simply read the accounts of the sender and the receiver and update their balances, we launch a workload of 10 million random "read then write" operations and commit the changes every 1000 operations. 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. The workload picks the keys from the initialized set randomly. Since LVMT needs to allocate version number slots for new keys, we also evaluate with a "fresh" setting: the storage has no initialization, and

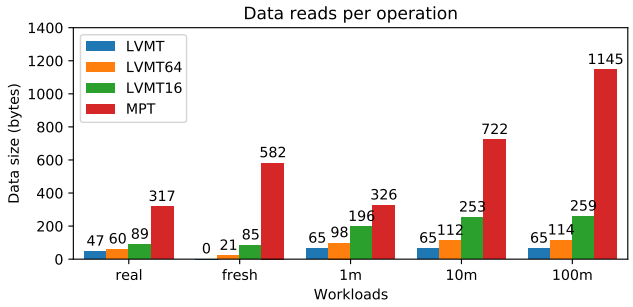


(a) Read amplification of authenticated storage.

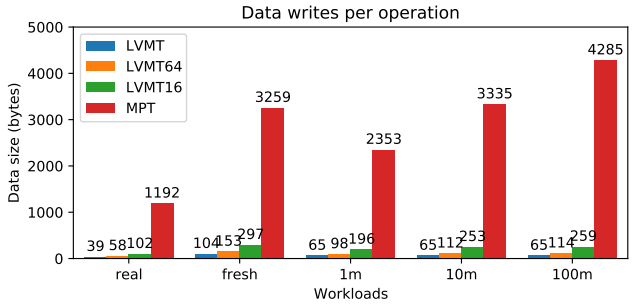


(b) Write amplification of authenticated storage.

Figure 6: Read and write amplifications of authenticated storage.



(a) Size of data read from the backend per operation



(b) Size of data write to the backend per operation

Figure 7: Size of data i/o to the backend per operation

the workload accesses distinct keys. In addition, to evaluate the performance under the real world access pattern, we extract the storage access 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 13,546,700 and recover the I/O operations in these blocks. These transactions access 1.3 million distinct keys, and make 4.2 million reads and 2.4 million writes in total. We use "real" to denote the results from real world transactions.

Figure 4 shows the throughput under the different settings. LVMT is at least five times faster than MPT. When maintaining 1/64 of auxiliary information, LVMT64 achieves 70% to 80% throughput of LVMT. Since the experiment runs 10 million operations in total, in the "fresh" setting, the authenticated storage is initiated with an empty ledger and is ended with a ledger containing 10 million key-value pairs. So the throughput of "fresh" is slightly larger than "10m".

LVMT64 and LVMT16 need to maintain a shard of proof, which takes about 700 MB memory to cache the cryptographic parameters. As a fair comparison, we also evaluate LVMT16 with a lower memory budget 800 MB, instead of 1500 MB, to study how the cache size influences the performance. Figure 5 compares the throughput of LVMT16 between 1500 MB and 800 MB memory budgets and shows that the throughput is not influenced by the memory budget. This is expected since the root AMT is cached in memory and different keys access different parts of the second level of AMTs.

Read and write amplification: We further study the read and write amplification on the interface of the key-value database. Figure 6a shows the read amplification under the different settings. As the size of the initialized set grows, LVMT has similar read amplifications. The root AMT contains 2^{16} input entries, and the second level of AMTs contains 2^{32} input entries in total. Since each entry has five slots for key-value pairs, the first level of AMTs can only store 0.3 million keys, and the second level of AMTs can store 21 billion keys. So LVMT always requires two levels of AMT in the order of millions of keys. Since the read amplification of a key grows linearly with its level in the AMTs, it is reasonable that LVMT has similar read amplifications. In contrast, the read amplification of MPT grows from 2.56 to 4.31. Figure 6a also reveals 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 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 by 1. The read amplification of real-world transactions is approximately equal to the mean value of "1m" and "fresh" settings, because

about half of the write operations creates a new storage entry. MPT has lower read amplification on the "fresh" setting than LVMT since MPT benefits from a smaller tree while LVMT takes one additional read for allocating version slot. Figure 6b shows the write amplification. The write amplification of LVMT is similar with the read amplification. MPT has a much larger write amplification since half of the write operations are deletion. Even excluding this factor, MPT has four to five times larger write amplification than LVMT.

We also study the size of read and write data per operation. Figure 7 shows that MPT reads hundreds of bytes and writes thousands of bytes per operation. Because each inner node in MPT may have at most 16 children, and MPT stores a 32-byte hash for each child. So each node in MPT may contain hundreds of bytes. As comparison, LVMT only loads a small constant size no matter the size of ledger. Because LVMT caches the root AMT and the commitments of AMTs in the second level. For a key whose version is stored in the second level, all the accessing to AMTs can be cached.

7 Related Works

Improved MPT structures: mLSM proposes to maintain multiple levels of MPTs [38]. 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. LMPT proposes maintaining three MPTs, one large MPT containing old state and two small MPTs containing recent state changes [16]. LMPT periodically merges small MPTs into large ones. For both mLSM and LMPT, the concatenation of the Merkle roots of all the MPTs becomes the commitment for the ledger state.

These 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 heights, and the merge of MPTs can happen in a background thread. In contrast, LVMT reduces the disk I/O amplifications directly from $O(\log n)$ to $O(1)$. Our results show that when integrated end-to-end into Conflux, LVMT outperforms LMPT by up to 2.5x. The mLSM paper only contains its conceptual design without implementation and evaluation [38]. It is unclear how mLSM would perform end-to-end with a blockchain in practice.

Parallelize storage I/O: RainBlock [35] 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 [21]. 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.

Vector commitment for data sharding: Several vector commitment protocols [15, 19, 23, 25, 41, 44] 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. In [44], the authors use KZG commitment protocol [23] 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 [19] 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.

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 3.1x 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.

References

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

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

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

[4] arkworks contributors. arkworks zksnark ecosystem, 2022.

[5] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. BLAKE2: simpler, smaller, fast as MD5. In Michael J. Jacobson Jr., Michael E. Locasto, Payman Mohassel, and Reihaneh Safavi-Naini, editors, *Applied Cryptography and Network Security - 11th International Conference, ACNS 2013, Banff, AB, Canada, June 25-28, 2013. Proceedings*, volume 7954 of *Lecture Notes in Computer Science*, pages 119–135. Springer, 2013.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] Sean Bowe. BLS12-381: New zk-snark elliptic curve construction.

[11] 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.

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

[13] Vitalik Buterin. Ethereum whitepaper. Technical report.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] Koh Wei Jie. Perpetual Powers of Tau.

[23] 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.

[24] 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.

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

[26] 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.

[27] 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.

[28] 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.

[29] Chenxing Li, Peilun Li, Dong Zhou, Zhe Yang, Ming Wu, Guang Yang, Wei Xu, Fan Long, and Andrew Chi-Chih Yao. A decentralized blockchain with high throughput and fast confirmation. In Ada Gavrilovska and Erez Zadok, editors, *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 515–528. USENIX Association, 2020.

[30] 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.

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

[32] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report.

[33] 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.

[34] Parity Technologies. Crate kvdb.

[35] 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.

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

[37] 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.

[38] 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.

[39] Yonatan Sompolinsky, Shai Wyborski, and Aviv Zohar. Phantom and ghostdag: A scalable generalization of nakamoto consensus.

[40] 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.

[41] Shravan Srinivasan, Alex Chepurnoy, Charalampos Papamanthou, Alin Tomescu, and Yupeng Zhang. Hyperproofs: Aggregating and maintaining proofs in vector commitments.

[42] Facebook Database Engineering Team. Rocksdb: A persistent key-value store for flash and ram storage, 2022.

[43] Parity Technologies. Openethereum, 2019.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. Rapidchain: A fast blockchain protocol via full sharding.

Supplementary Material

Trusted setup for protocol parameters: For an elliptic curve group $\mathbb{G}$ with a starting point G and the group order p, KZG commitment requires on a series of public parameters $\tau \cdot G, \tau^2 \cdot G, \tau^3 \cdot G, \cdots$, with a random parameter $\tau \in \mathbb{Z}_p$ opaque to anyone. So a deployed system requires a trusted party to generate the public parameters with a random τ and forget τ after generation. This process is called the *trusted setup*. A protocol requiring the trusted setups builds its security on the trustworthy to the third party. To mitigate this problem, several efficient secure Multi-party computation (sMPC) protocols [11,12,20] are proposed to allow unlimited participants to contribute to the generation of public parameters in turn. Anyone cannot recover τ unless collecting the random input of all the participants. A perpetual-powers-of-tau ceremony [22] has started in 2019 to build the public parameters with 2^{28} elements for the groups $\mathbb{G}_1$ and $\mathbb{G}_2$ of BN254.

Formal definition for inner nodes of AMT: Here we formally defines the two elements associated with AMT inner nodes. Since the proof tree 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 child and right child are assigned indices i and $i+2^d$ respectively. The root is indexed by 0.

Let w be a n-th root of unity (notice $n = 2^k$), a.k.a., $w^n = 1$. With an input $\vec{a}$, AMT constructs the vector commitment to $\vec{a}$ to the polynomial commitment $f(x)$ that satisfies $f(w^i) = a_i$. Thus, $f(x)$ can be constructed by Lagrange interpolation as:

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

For a node at depth d and index t, it only covers the input elements whose index i satisfies $i \equiv t \bmod 2^d$. So the corresponding polynomial function $f_{d,t}(x)$ only covers the Lagrange interpolation terms of these elements:

$$f_{d,t}(x) := \sum_{i \in T_{d,t}} a_i \cdot \frac{\sum_{j=0}^{n-1}(x/w^i)^j}{n},$$

where $T_{d,t} := \{1 \le i \le n | i \equiv t \bmod 2^d\}$. So the commitment of this node is $f_{d,t}(\tau) \cdot G_1$ and the corresponding proof is $h_{d,t}(\tau) \cdot G_1$, where $h_{d,t}(x)$ is defined by

$$h_{d,t}(x) = \frac{f_{d,t}(x) \cdot w^{t \cdot 2^{k-d}} \cdot ((x/w^t)^{2^{k-d}} - 1)}{x^n - 1}.$$

For the index s of input, it corresponds to the proof node at depth k with index $R(s)$. Thus, when updating the a_s, for each depth d, the node indexed by $t \equiv s \bmod 2^d$ will be updated. We use $\bar{f}_{d,s}(x)$ and $\bar{h}_{d,s}(x)$ to denote the changes to function $f_{d,t}(x)$ and $h_{d,t}(x)$ when increasing a_s by 1. These two functions can be simplified to:

$$\bar{f}_{d,s}(x) = \frac{\sum_{j=0}^{n-1}(x/w^s)^j}{n},$$
$$\bar{h}_{d,s}(x) = \frac{1}{n} \cdot \sum_{j=1}^{2^{k-d}} w^{j \cdot s} x^{2^{k-d}-j}.$$

Notice that $\bar{f}_{d,s}(x)$ is independent with d, we denote it by $\bar{f}_s(x)$. So in AMT, when increasing a_s by δ, the commitments and proofs of the corresponding nodes will be increased by $\delta \cdot \bar{f}_s(\tau) \cdot G_1$ and $\delta \cdot \bar{h}_{d,s}(\tau) \cdot G_1$.

The sequence of $\bar{f}_s(x) = \{\bar{f}_s(x)\}_{s=1}^n$ and $\{\bar{h}_{d,s}(x)\}_{s=1}^n$ for any d can be constructed by FFT. So 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.

Non-existence proof of LVMT: LVMT proves the non-existence of a key k by proving that all the possible version number slots for the key are allocated to the other keys. In Algorithm 10, we give the procedure for generating the non-existence proof. The procedure first allocate a version slot for k and rolls back the allocation (lines 2-3). Then, it proves

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);
```

the version number of this slot is zero similar to Algorithm 6 except the merkle proof of the key (lines 5-12). Last, it shows the other possible slots of k are allocated to other keys by generating proof for them; the second fields of these proofs can be omitted since they have the same information as commpfs computed in line 11.

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);
```