LMPT: A Novel Authenticated Data Structure to Eliminate Storage Bottlenecks for High Performance Blockchains

Jemin Andrew Choi*, Sidi Mohamed Beillahi*, Srisht Fateh Singh*, Panagiotis Michalopoulos*,
Peilun Li†, Andreas Veneris*, Fan Long*
*University of Toronto †Shanghai Tree-Graph Blockchain Research Institute
choi@cs.toronto.edu, sm.beillahi@utoronto.ca, srishtfateh.singh@mail.utoronto.ca,
p.michalopoulos@mail.utoronto.ca, peilun.li@confluxnetwork.org,
veneris@eecg.toronto.edu, fanl@cs.toronto.edu

***Abstract*—We present the Layered Merkle Patricia Trie (LMPT), a performant storage data structure for processing transactions in high-throughput systems when compared to traditional Merkle Patricia Tries used in Ethereum clients. LMPTs keep smaller intermediary tries in memory to alleviate read and write amplification from high-latency disk storage. As an additional feat, they also allow for the I/O and transaction verifier threads to be scheduled in parallel and independently. LMPTs can ultimately reduce significant I/O traffic that happens on the critical path of transaction processing. Empirical results show that LMPTs can process up to $\times 6$ more transactions per second on real-life ERC20 smart contract workloads when compared to existing Ethereum clients.**

***Index Terms*—Blockchain, data storage, transaction execution, Merkle Patricia trie**

I. Introduction

Popularized by cryptocurrencies [3], [30], blockchain platforms have become increasingly prevalent today. Among others, they enable decentralized ledgers at Internet-of-scale that fuel innovation in diverse industries such as finance [14], supply chain [42], and healthcare [27]. One issue that continues to challenge their wider adoption is their limited transaction throughput. To ensure safety, the consensus protocols used by blockchain platforms like Bitcoin and Ethereum conservatively apply slow block generation rates and restrict block sizes. Consequently, this allows them to process only 7 to 30 transactions per second (TPS) [23], [37], when compared to traditional centralized systems that can parse thousands of transactions per second.

To address this bottleneck, new consensus protocols have been proposed in recent years. For example, Algorand [18], Conflux [25], Prism [12], and OHIE [47] can process thousands of transactions per second. In doing so, innovations in high throughput ledgers also revealed an important but overlooked challenge in the blockchain community: *transaction execution* performance. Particularly, transactions that frequently access the blockchain state tend to become the new performance bottleneck that limits the overall throughput of a blockchain platform. For instance, when importing previously downloaded transactions, popular Ethereum clients like GoEthereum [4] and OpenEthereum [5] can only process 700 TPS on a laptop with a SSD, which is significantly lower than the capability of many new consensus protocols [24].

In the Ethereum blockchain, we can distinguished two types of nodes: full and light nodes. A full node synchronizes and executes all transactions and maintains the blockchain state. A light node only synchronizes blocks headers without transactions and the blockchain state. The blockchain state in Ethereum is a key-value structure that maps account addresses and persistent state to the corresponding account metadata and values. If a light node requires the value of a given key, it queries a full node. However, since Ethereum follows permissionless protocol there exist a mechanism to ensure that the light node does not need to trust the full node in order to use its response (known as the authenticated ledger state). In particular, the Ethereum protocol requires miners to compute a commitment (known as the *state root*) of the blockchain state and add the latest computed commitment in the generated blocks headers. Thus, when responding to a query from a light node, a full node generates a proof based on the latest commitment of the blockchain state that can be verified against the blocks headers the light node keeps.

Previous studies have shown that the bottleneck of executing transactions in Ethereum clients is of processing read/write operations on the underlying blockchain state [24], [35]. In particular, to ensure the authenticated ledger state mechanism Ethereum stores its state as a Merkle Patricia Trie (MPT) [44]. Each node in the trie has up to sixteen children where each path from the root to a leaf node corresponds to a hexadecimal-encoded key and the leaf node holds the corresponding value of the key. Furthermore, each inner node in the MPT contains the hash result of all of its children. As such, a Merkle proof of a key-value pair consists of hashes of all nodes along the path to the leaf node of the key. The root hash value is published globally in the header of each Ethereum block so that anyone can verify the key-value pair with the proof. Thus, a blockchain full node maintaining the entire state can generate authenticated proofs of key-value pairs in the state, and a light node can verify the proofs without the need to trust the full node.

The trust resulting from the authenticated ledger state mechanism comes with costly performance drawbacks since read/write operations in MPT are slow: 1) a read/write to a key-value pair is amplified to multiple I/O operations of all nodes along the corresponding path of the key in the MPT, 2) a write operation recomputes the hashes of all inner nodes along the path in the MPT, and most importantly, 3) the transaction execution thread has to wait for costly read/write operations to complete before it continues to the next instruction or transaction. Notably to those observations is the fact that to ensure deterministic execution outcomes, blockchain clients execute transactions sequentially in a single thread.

This paper presents the Layered Merkle Patricia Trie (LMPT), a novel authenticated storage structure for high performance blockchains. LMPTs can directly operate with transaction execution engines that implement the Ethereum Virtual Machine (EVM) [43], [44]. The empirical results presented show that LMPTs speed up the transaction execution throughput by up to 6 times. The net-effect is that, in conjunction with existing innovations on consensus algorithms, LMPTs can significantly improve the transaction throughput of blockchain platforms.

The LMPT consists of a snapshot MPT and a flat key value store that holds the blockchain state at a recent block height, an intermediate MPT that contains updates to the blockchain state on top of the snapshot MPT, and a delta MPT that contains updates on top of the intermediate MPT. LMPT records new updates to the blockchain state first into the smallest delta MPT. For a predetermined number of blocks (*e.g.*, 1000 blocks), LMPT merges the updates from the intermediate MPT into the snapshot MPT to form a new one. Then, the old delta MPT becomes the new intermediate MPT and the new delta MPT is emptied.

One advantage of the LMPT design is reduced intensity and amplification of read and write operations. Because the intermediate MPT and the delta MPT only hold recent updates to the blockchain state, they are small enough to be stored entirely in memory. Evidently, the small depths of the two tries reduce the I/O amplification of reads and writes. In addition, as more decentralized applications (DApps) move into the blockchain ecosystem, popular smart contracts are expected to have greater localized access patterns on blockchain state [7]. Consequently, most reads and writes in the transaction execution thread will only access the intermediate and/or delta MPTs, which are cached in memory.

Another advantage of LMPT is to decouple the expensive disk I/O operations from the critical transaction execution thread as much as possible. Furthermore, the blockchain clients can parallelize the construction of snapshot MPTs with the transaction execution thread. Reads that do not require authentication can be executed from an internal flat key value store, instead of querying the full trie.

We evaluated LMPT with real-world workloads and benchmarks for simple payments and ERC20 smart contracts [17], [1]. We sampled $500,000$ transactions from Ethereum, and packed blocks to simulate blocks on the real network based on gas limits. Our results show that LMPT is able to considerably outperform the Ethereum MPT for larger genesis states under the same hardware constraints and system usage. LMPTs are able to sustain up to 3000 TPS for simple payments and 2000 TPS for ERC20 smart contracts for 10 million senders in the genesis state, which is roughly $\times 6$ faster than the existing MPT structure in Ethereum clients. We also evaluated LMPT with $500,000$ transactions workload of the more complex Uniswap smart contract that is widely used in decentralized finance (DeFi) [8], [45]. We show that LMPT is roughly $\times 2$ faster than the existing MPT structure in Ethereum clients for Uniswap smart contract for 10 million senders in the genesis state. These results show that LMPT is increasingly suitable for blockchain systems as the state trie grows exponentially bigger.

In summary, the paper makes the following contributions:

- **LMPT:** We present a novel authenticated storage structure called LMPT that significantly reduces the amplification effect of read/write operations and decouples expensive disk I/O operations from the critical transaction execution thread.
- **EVM Transaction Execution Engine with LMPTs:** We present the design, implementation, and evaluation of an EVM transaction execution engine integrating LMPTs that empirically enables the transaction execution engine to process up to 3000 TPS (*i.e.,* $\times 6$ times compared to traditional MPTs).

The remainder of this paper is organized as follows. Section II presents the background and the overview of LMPT. Section III presents the design of LMPT. We evaluate the implementation of the LMPT on real world benchmarks in Section IV. In Section V we discuss related work. We finally conclude the paper in Section VI.

II. Background and Overview

In this section we first describe how Ethereum uses the MPT to store the ledger state and why read/write operations on MPTs are a performance bottleneck during transaction execution. We then present an overview of LMPTs and how they tackle this bottleneck.

A. Background

Ethereum is a state machine constituted of a genesis state and transactions that modify the state [43], [44]. Ethereum supports two types of accounts: user accounts and smart contract accounts. Smart contract accounts are software objects that manage transactions. Each account is associated with a unique address. The state includes account information, which consists of the nonce, account Ether balance, the storage root hash of the account's storage trie, and the Ethereum Virtual Machine (EVM) code hash if the account is a smart contract. The state is kept in a top level state trie, where there is a mapping between the `Keccak256` hash of the account address and the state. Users can interact with the blockchain by issuing transactions using their user accounts. Ethereum then executes state transition functions using the EVM. Transactions are packed into blocks that are hashed with previous blocks. To check whether a block is valid in a chain, the block header stores the cryptographic hash of the MPT root. Hence, any tampering of the block state can easily be detected by verifying the root hash of the MPT.

To efficiently store authenticated state, Ethereum uses a modified MPT structure to compress key-value pair hashes. The key is a 256-bit hash of the account address, which maps to the stored account data as the value. Since light clients in Ethereum do not have full access to all the data in the blockchain, it is crucial to have authenticated data reads and writes so light clients can verify the state with partial proofs with the help of a full client that has access to all the data in the blockchain.

In a MPT, we distinguish three types of nodes: branch, extension, and leaf nodes. A branch node stores up to 16 pointers, one per hexadecimal, that point to either a leaf node, extension node, or another branch node. An extension node compresses a byte sequence that can be used to compress nodes with a shared hash sequence and contains the pointer to the next node in the tree. A leaf node stores the encoded path and the value itself. Finally, the root of the tree is used to create a hash that is dependent on all the leaf values, which can be used by light clients to verify data originated from a full client with access to the entire blockchain state.

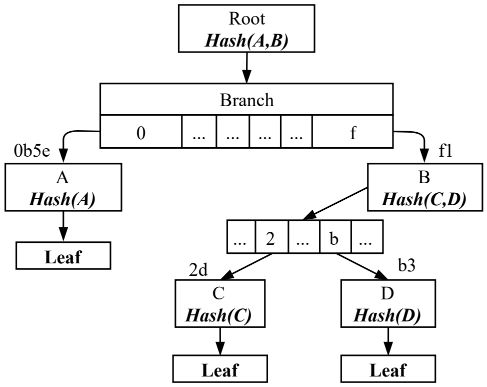


Fig. 1: MPT used in Ethereum. A node contains the hash of its children nodes, preventing data tampering. A path of the trie can be travelled by one hexadecimal at a time, shown by the branch node, until a leaf node is reached.

Fig. 1 illustrates the MPT structure. It shows how a path can be constructed from the root to extension and branch nodes, down to the leaf nodes. The Ethereum block header contains the `Keccak256` hash of the root to allow both efficient storage and verification of block data. From the root, there are branch nodes for each hexadecimal that contain pointers to the next node in the path of the trie. In addition, each node contains a hash of its children nodes, which allows to efficiently compare whether two trees have the same data by checking the hash of their roots. By traversing the path down the tree to its leaf node, we can verify the existence of a particular account key-value in a blockchain state.

In addition, when an authenticated read query is executed on MPT, it requires a proof that shows a valid path between the root node and the leaf node. This path is then used to recompute the signature independently, and verify that the read value exists in the trie. This is imperative for data access in light clients that do not store the entire state trie. Authenticated reads in a standard MPT are costly due to high read amplification as clients track down nodes in the MPT. Since each node access requires an additional database read, each authenticated read in Ethereum can have a read amplification of 64, or one per hexadecimal in the 256-bit hash of the address.

B. Observations and Motivation

We now describe an experiment using OpenEthereum, a popular and fast open-sourced Ethereum client [5]. We use OpenEthereum to import blocks containing transactions that regularly access the blockchain state and use `perf` [6] to profile transaction execution. We observed that as transactions access the blockchain state more frequently, the transaction processing throughput became lower. Our findings are consistent with prior work [24]: the majority of transactions execution time is spent on operations that access the blockchain state, *e.g.*, EVM opcodes such as SLOAD and SSTORE.

Observation 1: The blockchain storage is the primary performance bottleneck for transaction execution.

In particular, we observe that transaction execution threads frequently become blocked waiting for the disk

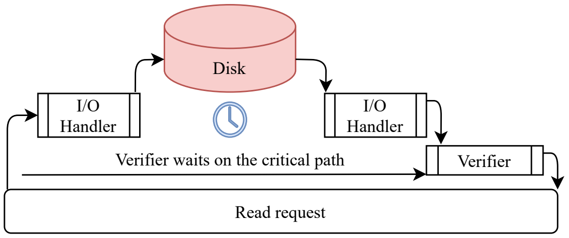
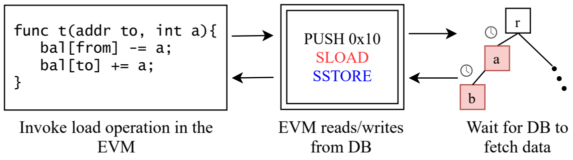


Fig. 2: I/O Blocking for Transaction Execution



```
func t(addr to, int a){
    bal[from] -= a;
    bal[to] += a;
}
```

Fig. 3: A sequential thread execution for database (DB) reads in the EVM

I/O operations to finish. Fig. 2 illustrates our profiling findings. While the verifiers wait for disk I/O operations to finish, resources like CPU and memory are underutilized and idle during transaction execution.

Fig. 3 presents an example to illustrate the root cause of the latency-bound issue. The left part of Fig. 3 presents a Solidity code snippet which reads an array stored in the MPT. In Ethereum, read/write operations will be translated into SLOAD/SSTORE EVM instructions, as shown in the middle of Fig. 3. Because the EVM is designed to execute transactions and EVM instructions sequentially, the transaction execution thread has to wait for the results of SLOAD before it can execute the next instruction. The SLOAD execution reads the data from the MPT and is eventually amplified into multiple key-value read operations, shown on the right of Fig. 3.

Similar latency-bound issues exist for MPT write operations and SSTORE instructions. In particular, each Ethereum block contains the MPT state root hash that existing clients have to compute and verify. Thus, the transaction execution thread will wait for all MPT write operations associated with one block to finish before it continues to the next block. Although the latency of the write operations only happens at the block level, it is on the critical path for the performance because it cannot be mitigated by memory cache in the Ethereum client.

> **Observation 2:** The primary performance bottleneck of MPT: transaction execution thread waits for expensive disk I/O operations and becomes latency bound.

Table I presents our experimental findings on the `OpenEthereum` client with different memory cache sizes for the MPT database. We import blocks containing random simple payment transactions and report the transaction throughput under different cache sizes. In Table I, we observe that increasing the memory cache size in OpenEthereum had an immediate effect on the cache hit rate, *i.e.* around 25%. However, the cache sizes had no significant impact on overall performance, and throughput increased by no more than 5%. These results show that simply enlarging the memory cache of MPT or naively allocating more memory to the process may *not* improve the transaction execution sufficiently.

TABLE I: Cache hit rates in OpenEthereum

Cache size (MB)	Hit Rate	TPS
50	0.635	1238
100	0.758	1256
500	0.862	1278
1000	0.879	1292

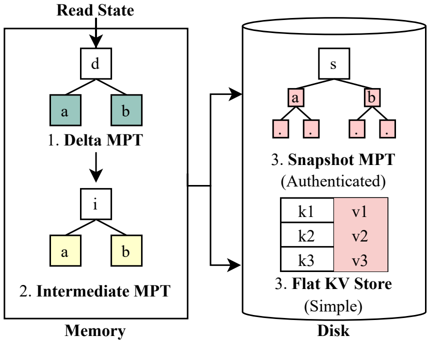


Fig. 4: Authenticated and simple state reads in LMPT

C. LMPT Overview

To reduce I/O amplification and separate the critical path of blocking threads, we propose a new data structure, namely Layered Merkle Patricia Trie (LMPT), to store authenticated Ethereum state. The LMPT consists of three distinct MPTs that act as "caches" for any authenticated access: delta, intermediate, and snapshot MPTs. For every read access to the state tree, the request first searches the delta MPT. If the requested data is not found, then the intermediate MPT is searched, then finally, the snapshot MPT is checked. This hierarchical cache structure reduces read amplification on the key path (especially for hot data) and reduces very costly accesses to disk. Fig. 4 shows how authenticated and simple reads access data in the LMPT. The smaller delta and intermediate tries are stored in memory to allow for faster access to hot data, while the larger snapshot trie and the flat key-value store are stored in disk. For authenticated reads that require Merkle proofs, the snapshot trie provides information about the account. Simple reads requested from the full node itself can be read from a flat key-value store, which reduces any read amplification for accessing a value.

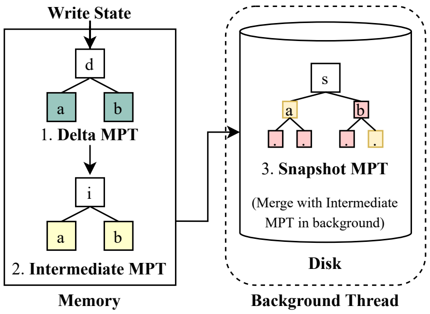


Fig. 5: Writing state in LMPT. Writes are periodically flushed from intermediate trie to snapshot trie using background threads.

```
struct Trie {
  root:       uint256,
  kv:         Map
}
struct LMPT {
  delta, interm:  Trie, // In memory
  snapshot:       Trie, // In disk
  flat:           Map   // In disk
}
```

Fig. 6: LMPT Data Structure

For a write, instead of immediately flushing changes to disk, the delta MPT is updated. Caching writes allow to have a consistent view of the entire system at the small cost of storing and updating the delta MPT in memory. To keep the MPTs small, the changes are flushed at periodic checkpoints, where the delta MPT changes are merged to the intermediate MPT and the intermediate MPT is merged into the snapshot MPT. Fig. 5 shows how the delta MPT is stored inside the memory while a background process merges any larger changes between the intermediate and snapshot MPT. As a result, the writes are periodically batched and completed independently of the critical path of transaction verification.

III. LMPT Design

In this section, we outline the design of LMPTs and describe the fundamental improvements they bring to transaction throughput in the storage layer for blockchain clients.

A. *Definitions and Data Structures*

In Fig. 6, we define the data structures used to architect the LMPT. We first define the data type *trie*, which consists of a `uint256` root hash and a key-value map as an abstraction for storing authenticated data. The LMPT data structure is comprised of four components: the delta, intermediate, and snapshot tries, and the flat key-value store map. The delta and intermediate trees are stored in memory and contain frequently accessed state. The snapshot tree and flat key-value map store the entire blockchain state on disk, and return the values for an authenticated and non-authenticated access, respectively.

```
T := LMPT()
fn write_LMPT(k,v) {
  root := T.delta.put(T.delta.root,k,v)
  T.delta.root := root
}
fn read_LMPT(k) -> <v,p> {
  <v,$p_1$> := T.delta.get(delta.root,k)
  if v is present
    return <v,$p_1$>
  <v,$p_2$> := T.interm.get(T.interm.root,k)
  if v is present
    return <v,$p_1 + p_2$>
  if auth_proof
    <v,$p_3$> := T.snapshot.get(T.snapshot.root,k)
    return <v,$p_1 + p_2 + p_3$>
  else
    v := flat.get(k)
    return <v,$\epsilon$>
}
```

Fig. 7: LMPT read and write operations

B. *Read and Write Operations*

In Fig. 7, we present the pseudocode for LMPT read and write operations. For a write, the value is always updated on the delta trie, which is kept in memory so that hot data can be queried quickly. For a read, we first query the delta trie, and if a value does exist in the delta trie, we can verify existence for that value and simply return the value and path. If the value does not exist, then we need to return proof by showing that the two adjacent paths, *i.e.*, a path that is immediately greater and immediately less than the value, exist in the tree instead. Using this returned proof of adjacent paths, we query the intermediate trie to check for the value. If the intermediate trie contains the corresponding value for the key, we return the resulting data and the combined proof from the delta and intermediate tries. Finally, if the key is not present in the delta or intermediate trie, then it is queried from disk. If the client requires an authenticated read, then it must query the snapshot trie on disk for the value. If the client can trust the authenticity of the data, e.g., reading state from its own database, then the client can query the flat store map to eliminate any read amplification. By querying the disk last, we can delay costly reads from disk and reduce incurring large read amplification on bigger tries by having smaller, intermediary authenticated data structures in memory.

C. *Trie Merge Operations*

In Fig. 8, we give the two step merge process of the different trie structures behind the LMPT. The `merge_compute` function updates the snapshot trie and flat store map on disk. At predefined intervals, `merge_compute` is called to update and append all the changes from the intermediate trie to its snapshot

```
fn merge_compute(T) -> (root′, flat′) {
   flat′ := T.flat
   root′ := T.snapshot.root
   for <k, v> in T.interm.kv(T.interm.root)
     root′ := T.snapshot.append(root′, k, v)
     flat′ := flat′.set(k, v)
   return (root′, flat′)
}
fn merge_update(T, root′, flat′) {
   T.flat := flat′
   T.snapshot.root := root′
   T.interm := T.delta
   T.interm.root := T.delta.root
   T.delta := Trie()
   T.delta.root := None
}
```

Fig. 8: LMPT merge operations

```
block_cnt := 0
T := LMPT(genesis_state)
while Block is processing
  for transaction in Block
    T.update_trie(transaction)
  block_cnt += 1
  if block_cnt % merge_interval == 0
    Wait for last spawned thread to end
    merge_update(T, root′, flat′)
    spawn_thread(root′, flat′=merge_compute(T))
```

Fig. 9: Flushing updates to disk on a background thread

trie and flat store map, and returns the new snapshot root and key-value map. This function allows to batch writes to disk at once and allows the snapshot MPT on disk to be updated efficiently without having to update every single interior node on the MPT, which greatly reduces write amplification. In addition, the merging can be parallelized and distributed to multiple threads, which prevents blocking the main execution thread on the critical path for I/O accesses.

The `merge_update` function defines how the tries in the LMPT are updated. `merge_update` accepts the new snapshot root and flat store map that were returned by the function `merge_compute`. Then, the intermediate trie is set to the smaller delta trie, and the delta trie is flushed and initialized by a new empty trie.

Finally, Fig. 9 shows a procedure that merges new data using a background thread so it does not block the critical path for the client. While a new block is being processed by the node, the incoming transactions in the block are written into the delta and intermediate tries of the LMPT. After each block is processed, a block counter is incremented as the tries in memory are filled with new incoming data. When the counter reaches a particular threshold, defined as the merge period interval, the process waits until all the remaining transactions are processed and threads that are merging tries finish. Then, the process calls the `merge_update` function to update the tries and flat store map computed by `merge_compute`. This two parts process allows data to be batched and flushed from memory to disk by a background thread so the main execution thread continues verification normally and only accesses the disk for the merge period intervals. After the tries are merged, a new background thread is spawned so that the incoming data can be integrated into the snapshot trie and flat store and flushed to disk in the next merge period.

D. Integration with Blockchain Clients

The LMPT can replace the standard MPT in Ethereum-like systems with the following modifications:

1) **State encoding:** Ethereum uses a 32-byte root hash of the MPT representing the resulting state of each executed block. LMPT consists of three tries, but we can use one-way cryptographic hash functions like `Keccak256` to combine the root hashes of the tries to generate a single 32-byte root hash representing the state.
2) **Proof verification process:** The authentication proof contains the proof combination of multiple tries, if the value is not found in the delta trie. Thus, we need to update the proof verification process accordingly so that the proof combination from the delta, intermediate, and snapshot tries is accepted by the verifier.

IV. Empirical Evaluation

In this section, we evaluate the transaction throughput on Ethereum clients with and without LMPT using different workloads based on simple payment transfers, ERC20 smart contracts, and Uniswap smart contracts.

A. Implementation

To compare LMPT's storage performance with existing Ethereum MPT implementations, we modify the `OpenEthereum` client to implement LMPT instead of the standard Ethereum MPT. The OpenEthereum client is implemented in Rust programming language and is one of the fastest Ethereum clients available [2]. In particular, we modify the existing storage engine of OpenEthereum to integrate the delta trie, intermediate trie, snapshot trie, and flat store instead of the single MPT structure. In addition, we alter the verification engine of OpenEthereum so that the LMPT merging process from Section III is integrated into the client.

B. Experimental Setup

The first two experiments are run on an *AWS EC2 i3.xlarge* instance with 4 vCPU, 30GB memory, and 1TB SSD storage and the last experiment is run on an *AWS EC2 c6a.8xlarge* instance with 32 vCPU and 64GB memory, and 1TB SDD storage. We run our LMPT implementation and compare it with the standard OpenEthereum v3.1.0, available on Github [5]. In order

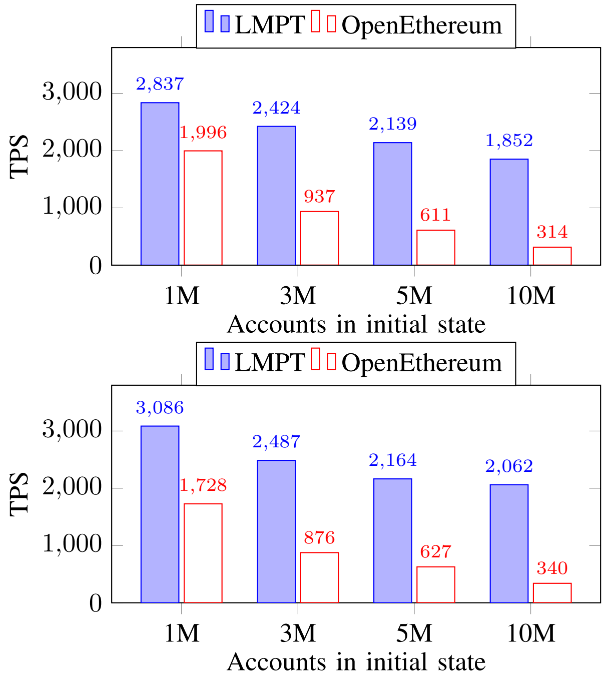


Fig. 10: TPS for LMPT-based OpenEthereum and the standard OpenEthereum for simple payment transactions. The top graph corresponds to the Ethereum traces benchmark and the bottom graph corresponds to the random senders traces benchmark.

to purely compare storage performance, we turn off the consensus engine and run the experiments in a private network containing a single node, so the blocks can be instantly mined and network effects will be negligible. We further collect a sample trace of $500,000$ real transactions from the Ethereum network for the first two experiments of simple payments and ERC20 transfers. We replicate the transaction behavior and pack blocks to mimic real world conditions. The blocks are created to reflect real gas limits, which is 150 transactions per block for simple payments and 20 transactions per block for ERC20. For ERC20 workloads, we sample transfer transactions for the *Tether token*, which is one of the most popular ERC20 tokens on Ethereum [1]. We monitor memory usage for both the LMPT implementation and standard OpenEthereum to ensure that the average memory usage for both experiments are relatively equal.

C. Simple Payments

Ethereum traces benchmark: We re-create blocks with the transaction traces collected from real Ethereum simple payment transactions. This allows us to import the blocks and measure the true performance of the authenticated storage structures. Since real Ethereum simple payments require their respective private keys of the senders, we create a one-to-one mapping between each public address and a generated public-private key pair. This enables one to send and sign transactions using the generated private keys to keep the integrity of the real-life workloads on the main network.

Random senders traces benchmark: In addition, we create another benchmark where we send simple payment transactions from a set of random senders addresses. We define each random sender with a high initial ETH balance in the genesis block, and send transactions with an evenly distributed load. Although the number of accounts in the initial states differs, every unique sender is guaranteed to send at least one transaction to a random receiver. Similar to the Ethereum traces benchmark, we send a total of $500,000$ transactions from the random senders pool.

Initial State: In the experiments, we prepare different *initial states* with an increasing number of accounts in the genesis block and measure the throughput in transactions per second for importing blocks on the client. This is because in our initial tests, the number of accounts in the genesis state does have a significant impact on performance. Contrarily, the number of transactions has little effect on the overall TPS, aside from storage warm up times (cache loading) for the initial transactions. Even as transactions increase, we do not observe significant difference in transaction import times. We track workloads with large numbers of senders and receivers, which would not fit entirely in the program memory and require I/O accesses from storage.

Fig. 10 shows the size of initial state versus performance for LMPT-based OpenEthereum and the standard OpenEthereum for simple payment transactions for the two benchmarks. The X-axis corresponds to the number of accounts in the genesis state (in millions) and the Y-axis corresponds to the throughput (in TPS) when the blocks are imported from disk. Our results show that the standard OpenEthereum's MPT model handles a relatively small initial state fairly well, and can reach up to 2000 TPS for 1 million accounts for both the Ethereum and random sender traces benchmarks. However, it drastically slows down to about 1000 TPS in importing blocks when the initial state is 3 million accounts. At 10 million accounts in the initial state, the standard OpenEthereum starts to significantly slow down on our 30GB memory machine, and for more than 10 million accounts, it fails to make much progress on the machine.

On the other hand, the LMPT-based OpenEthereum can achieve around 3000 TPS for 1 million accounts, a 50% improvement over the standard OpenEthereum. It is also able to sustain much higher performance for a large number of senders, and gets up to 2000 TPS for 10 million senders. For both benchmarks, the

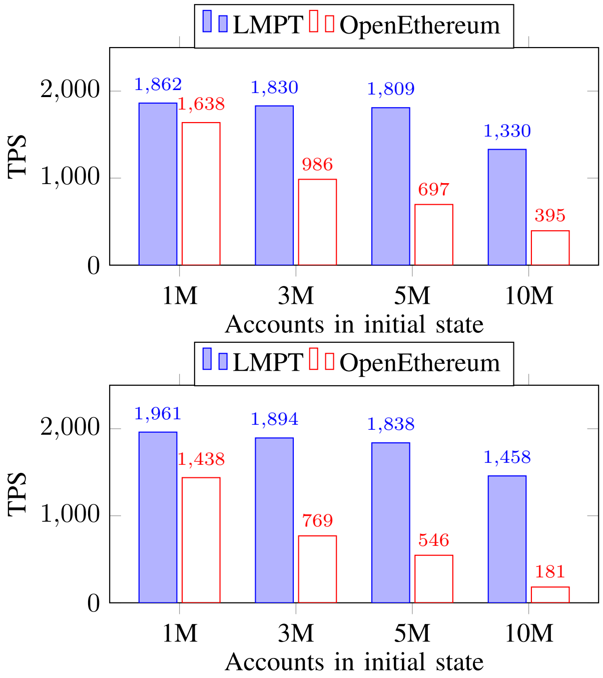


Fig. 11: TPS for LMPT-based OpenEthereum and the standard OpenEthereum for ERC20 transfer transactions. The top graph corresponds to the Ethereum traces benchmark and the bottom graph corresponds to the random senders traces benchmark.

LMPT outperforms the standard client by a factor of 6 for a large initial state. After the initial state reaches around 20 million accounts, we finally see a noticeable drop and saturation in performance for the LMPT-based OpenEthereum, which is twice the threshold reached by the standard OpenEthereum.

The LMPT structure allows for higher sustained performance because as the state trie gets bigger, LMPT can still cache hot data and account information into its delta and intermediate tries. In addition, since merging the snapshot trie on disk is done in parallel to the main execution thread, there is minimal blocking when state is imported. Contrarily, the standard OpenEthereum needs to execute increasingly more state reads from disk as the state grows larger, which slows it down drastically.

D. ERC20 Transfers

Similar to the simple payment traces, we sample transactions for the `Tether token` to generate real life workloads for the ERC20 contract. We deploy the ERC20 contract on a private network, initialize the contract address, synthesize a set of accounts, and fund them with some initial tokens. Then, we use our generated senders to call the transfer function and send the tokens according to our sampled transactions trace. For the random senders benchmark, we initialize senders addresses with enough tokens and call the transfer function with an even distribution.

Fig. 11 illustrates the size of initial state versus performance for LMPT-based OpenEthereum and the standard OpenEthereum for ERC20 tokens transfers transactions for the two benchmarks. The performance on ERC20 contracts are noticeably lower because they require more computation and gas. However, the results are similar to the simple payments as the standard OpenEthereum reaches saturation much more quickly as the state grows in size. For 1 million accounts, LMPT-based OpenEthereum had around 2000 TPS and could sustain that performance for $3-5$ millions accounts. On the other hand, the standard OpenEthereum had around 1600 TPS for 1 million accounts and performance quickly dropped as the size of the initial state increases. For 10 million accounts, LMPT-based OpenEthereum outperforms standard OpenEthereum by a factor 4. This shows that LMPT is able to maintain better throughput as the initial state grows. These results also suggest that LMPT is an effective solution for blockchains that support smart contracts and require more complex state reads and writes.

E. Uniswap Exchange

In this experiment, we compare LMPT against OpenEthereum MPT using a widely used smart contract that requires more state reads and writes than simple payment and ERC20 transfer. In particular, we use the `Uniswap` smart contract, a very popular decentralized finance (DeFi) protocol [8], [45]. Uniswap is an exchange protocol that also offers flash loan services. Uniswap exchange protocol consists of liquidity providers and traders. A liquidity provider supplies a pool of two ERC20 tokens that can be exchanged, i.e., creating an exchange market between the two ERC20 tokens. A trader exchanges one type of ERC20 token to the pool and receive the other ERC20 token out of the pool. The exchange rate between the ERC20 tokens in the pool is determined using an automated liquidity protocol by computing the relative number of the two tokens the pool has taking to account a small percent as reward for the liquidity pool provider. For instance, in a given liquidity pool with an amount X of a token A and an amount Y of a token B, the output amount o of token B a user receives for selling an input amount i of token A is given as follows:

$$o = \frac{Y \cdot i \cdot (1 - f)}{X + i \cdot (1 - f)} \tag{1}$$

The constant f represents the reward earned by the liquidity provider for the exchange. In the Uniswap smart contract implementation, we can distinguish two main exchange functions `swapExactTokensForTokens` and `swapTokensForExactTokens`. swapExactTokensForTokens sells a specific amount of tokens fixed by the caller for another (the outputted amount is determined using the Uniswap exchange formula). On the other hand, swapTokensForExactTokens buys a specific amount of tokens fixed by the caller.

In this experiment, we send a total of $500,000$ transactions from the random senders addresses pool that uniformly call one of the two functions swapExactTokensForTokens and swapTokensForExactTokens. We initialize each random sender with a sufficient ETH and ERC20 tokens balances. Similar to before, we prepare different *initial states* with an increasing number of accounts in the genesis block.

Fig. 12 shows the size of initial state versus performance for LMPT-based OpenEthereum and the standard OpenEthereum for Uniswap exchange transactions initiated by random senders addresses. The performance for Uniswap transactions are lower versus ERC20 transactions because they require more computation and storage accesses. The results of the experiment are similar to the results of the simple payment and ERC20 transfer as the standard OpenEthereum reaches saturation much more quickly as the state grows in size. In the Uniswap experiment, we use a machine with superior computing and memory capabilities. Thus, both LMPT-based OpenEthereum and standard OpenEthereum reach saturation much slower in Fig. 12 compared to Fig. 11 even though Uniswap requires more computation and storage accesses than ERC20. For instance, for 3 millions accounts in standard OpenEthereum the TPS for ERC20 is smaller than the TPS for Uniswap, this is because standard OpenEthereum reaches saturation in the inferior machine. On the other hand, for 3 millions accounts in LMPT-based OpenEthereum the TPS for ERC20 is bigger than the TPS for Uniswap while the Uniswap is run on a more superior machine. Eventually, for 10 millions accounts, LMPT-based OpenEthereum reaches saturation in the inferior machine, and the TPS for ERC20 is smaller than the TPS for Uniswap that is run on a superior machine. Notice that for 10 million accounts, LMPT-based OpenEthereum outperforms standard OpenEthereum by almost a factor 2 for Uniswap transactions. The results of this experiment also show that LMPT is an effective solution for blockchains that support the advanced DeFi smart contracts that does more complex computation and storage state reads and writes.

V. Related Work

Layered storage hierarchy has been studied extensively in the context of storage systems, distributed systems, and databases to ensure efficient data accesses for data intensive applications [19], [31], [36], [28], [29], [20], [21], [22], [46], [16]. However, layered storage design based on MPTs was not studied in the context of global state storage in blockchain platforms. In particular, the problem LMPTs are solving are for the main blockchain networks, *i.e.*, `Layer-1` solutions. As a result, we discuss other recent Layer-1 solutions that improve blockchain throughput here.

Distributed MPTs: A number of works [33], [35] study distributed MPTs to improve storage performance. In [35], the authors introduce mLSM, which splits the storage layer into multiple MPTs. This allows to reduce the authenticated read and write amplification. By decoupling the verifier with the lookup, mLSM reduced the I/O workload between reads and writes. However, increasing the number of levels in the MPT structure introduces a separate write amplification between layers and performance considerations need to be made when doing compaction between different tries.

In [33], the authors introduce `Rainblock`, which uses distributed sharding for the MPTs to improve storage performance in Ethereum based clients [33]. The underlying architecture proposes to decouple nodes into clients, miners, and storage nodes. This allows the storage nodes to use a distributed and sharded MPT in order to provide witness proofs to verify blocks based on the Merkle root. However, `Rainblock` requires major changes to existing Ethereum clients, as there is no such distinction between clients, miners and storage nodes. On the other hand, our LMPT design does not require major architectural changes and can be applied directly to existing nodes in Ethereum with few modifications, as we discussed in Section III.

Consensus protocols: There are many works on improving transaction throughput in blockchain systems by improving the underlying consensus protocols with different tradeoffs, *e.g.*, [11], [13], [18], [25]. Although

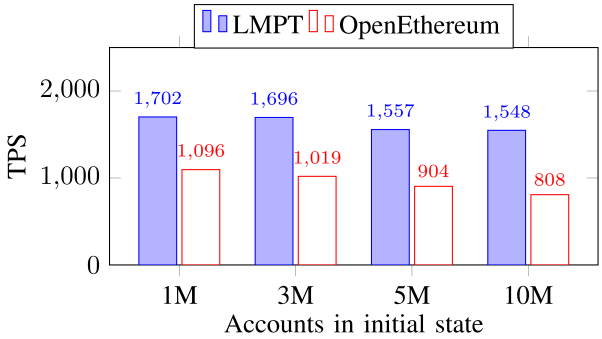


Fig. 12: TPS for LMPT-based OpenEthereum and the standard OpenEthereum for Uniswap swapExactTokensForTokens and swapTokensForExactTokens transactions from a set of random senders.

improving consensus protocols is an important concern, the transaction execution will still be a bottleneck by blocking I/O calls made by clients. As the blockchain state increasingly grows, the storage bottleneck will be the main problem faced by blockchain clients to overcome for scaling transaction throughput. Our proposed LMPT can be implemented with any consensus mechanism, and it allows further improvements in performance.

Sharding in Blockchains: There are a number of works on improving throughput in blockchain platforms through sharding transaction execution and sharding the blockchain state [10], [15], [26], [32]. The Ethereum community has also been receptive to sharding consensus solutions as a part of the ETH2 protocol [9]. Sharding proposes validator nodes to split up into smaller committees and validate a portion of the entire blockchain state. By separating groups of validator nodes, the nodes can also validate blocks with fewer resources, as it only needs to keep track of a small portion of the state and can allow more validators to participate on a limited set of computing power. However, sharding also introduces the problem of malicious nodes gaining easier access to attack the blockchain. This is because the state is more vulnerable to fragmentation, and sharding requires stricter network guarantees and fewer overall validators in each shard committee [34], [38]. In addition, sharding often requires heavy cross-shard communication and more networking overhead as nodes need to coordinate transactions with other nodes that have different portions of the state [40]. All in all, sharding is orthogonal to the problem LMPTs are solving by enabling a more performant storage structure.

VI. Conclusion

The LMPT is a novel storage structure that can significantly improve transaction processing in the storage layer of blockchain systems. This paper shows that it is able to be easily integrated to existing blockchain clients and can be used to improve throughput, in addition to novel consensus mechanisms. Ultimately, our results show that the LMPT is able to parallelize execution in the critical path and is effective for improving import performance in block catchup, especially for large states.

LMPT is integrated in production in the Conflux [25] protocol, an EVM-based high performance blockchain, replacing the traditional MPT.

There are several interesting avenues for future work for improving the efficiency of blockchain platforms. In particular, new vector commitment schemes such as authenticated multipoint evaluation Trees (AMT) [41] and Hyperproofs [39] permit to achieve faster authenticated storage reads and writes than MPT. Furthermore, Hyperproofs provides an efficient proof aggregation mechanism that enables to build a blockchain protocol with states sharding. Thus, an interesting direct future work is to investigate how to extend the layered LMPT design to AMT and Hyperproofs.

References

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

[2] Ethereum Node and Clients. https://ethereum.org/en/developers/docs/nodes-and-clients/.

[3] Etherum White Paper. https://github.com/ethereum/wiki/wiki/White-Paper.

[4] Go Ethereum. https://geth.ethereum.org/.

[5] OpenEthereum. https://github.com/openethereum/openethereum.

[6] Perf tools. https://github.com/torvalds/linux/tree/master/tools/perf.

[7] Top 20 Gas Consuming Smart Contracts. https://www.theblockcrypto.com/data/on-chain-metrics/ethereum.

[8] Uniswap Protocol. https://uniswap.org/.

[9] Validated, staking on eth2: Sharding Consensus. https://blog.ethereum.org/2020/03/27/sharding-consensus/.

[10] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. Chainspace: A sharded smart contracts platform. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018.

[11] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[12] Vivek Bagaria, Sreeram Kannan, David Tse, Giulia Fanti, and Pramod Viswanath. Prism: Deconstructing the blockchain to approach physical limits. In *Proceedings of the2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 585–602, New York, NY, USA, 2019. Association for Computing Machinery.

[13] Vitalik Buterin, Daniël Reijsbergen, Stefanos Leonardos, and Georgios Piliouras. Incentives in ethereum's hybrid casper protocol. *Int. J. Netw. Manag.*, 30(5), Sep 2020.

[14] Luisanna Cocco, Andrea Pinna, and Michele Marchesi. Banking on blockchain: Costs savings thanks to the blockchain technology. *Future Internet*, 9:25, 06 2017.

[15] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. Towards scaling blockchain systems via sharding. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 123–140, New York, NY, USA, 2019. Association for Computing Machinery.

[16] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In Thomas C. Bressoud and M. Frans Kaashoek, editors, *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, pages 205–220. ACM, 2007.

[17] Vitalik Buterin Fabian Vogelsteller. "eip-20: Erc-20 token standard," ethereum improvement proposals, 2015.

[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] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach, 5th Edition*. Morgan Kaufmann, 2012.

[20] Song Jiang, Xiaoning Ding, Feng Chen, Enhua Tan, and Xiaodong Zhang. DULO: an effective buffer cache management scheme to exploit both temporal and spatial localities. In Garth Gibson, editor, *Proceedings of the FAST '05 Conference on File and Storage Technologies, December 13-16, 2005, San Francisco, California, USA*. USENIX, 2005.

[21] Hyojun Kim, Sangeetha Seshadri, Clement L. Dickey, and Lawrence Chiu. Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches. *ACM Trans. Storage*, 10(4):15:1–15:21, 2014.

[22] Jaehyung Kim, Hongchan Roh, and Sanghyun Park. Selective I/O bypass and load balancing method for write-through SSD caching in big data analytics. *IEEE Trans. Computers*, 67(4):589–595, 2018.

[23] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In *Proceedings of the 25th USENIX Conference on Security Symposium*, SEC'16, page 279–296, USA, 2016. USENIX Association.

[24] Ao Li, Jemin Andrew Choi, and Fan Long. Securing smart contract with runtime validation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 438–453, New York, NY, USA, 2020. Association for Computing Machinery.

[25] 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*. USENIX Association, USA, 2020.

[26] 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.

[27] Thomas McGhin, Kim-Kwang Raymond Choo, Charles Zhechao Liu, and Debiao He. Blockchain in healthcare applications: Research challenges and opportunities. *Journal of Network and Computer Applications*, 135:62–75, 2019.

[28] Nimrod Megiddo and Dharmendra S. Modha. ARC: A self-tuning, low overhead replacement cache. In Jeff Chase, editor, *Proceedings of the FAST '03 Conference on File and Storage Technologies, March 31 - April 2, 2003, Cathedral Hill Hotel, San Francisco, California, USA*. USENIX, 2003.

[29] Michael P. Mesnier, Feng Chen, Tian Luo, and Jason B. Akers. Differentiated storage services. In Ted Wobber and Peter Druschel, editors, *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pages 57–70. ACM, 2011.

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

[31] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K page replacement algorithm for database disk buffering. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 26-28, 1993*, pages 297–306. ACM Press, 1993.

[32] George Pîrlea, Amrit Kumar, and Ilya Sergey. Practical smart contract sharding with ownership and commutativity analysis. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211*, pages 1327–1341. ACM, 2021.

[33] 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. USENIX Association, July 2021.

[34] Tayebeh Rajab, Mohammad Hossein Manshaei, Mohammad Dakhilalian, Murtuza Jadliwala, and Mohammad Ashiqur Rahman. On the feasibility of sybil attacks in shard-based permissionless blockchains. *arXiv preprint arXiv:1710.09437*, 2020.

[35] Pandian Raju, Soujanya Ponnapalli, Evan Kaminsky, Gilad Oved, Zachary Keener, Vijay Chidambaram, and Ittai Abraham. mlsm: Making authenticated storage faster in ethereum. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, Boston, MA, July 2018. USENIX Association.

[36] Benjamin Reed and Darrell D. E. Long. Analysis of caching algorithms for distributed file systems. *ACM SIGOPS Oper. Syst. Rev.*, 30(3):12–21, 1996.

[37] Sara Rouhani and Ralph Deters. Performance analysis of ethereum transactions in private blockchain. In *2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, pages 70–74, 2017.

[38] Alberto Sonnino, Shehar Bano, Mustafa Al-Bassam, and George Danezis. Replay attacks and defenses against cross-shard consensus in sharded distributed ledgers. *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 294–308, 2020.

[39] Shravan Srinivasan, Alexander Chepurnoy, Charalampos Papamanthou, Alin Tomescu, and Yupeng Zhang. Hyperproofs: Aggregating and maintaining proofs in vector commitments. *IACR Cryptol. ePrint Arch.*, page 599, 2021.

[40] Yuechen Tao, Bo Li, Jingjie Jiang, Hok Chu Ng, Cong Wang, and Baochun Li. On sharding open blockchains with smart contracts. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1357–1368, 2020.

[41] 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.

[42] Yingli Wang, Jeong Hugh Han, and Paul Beynon-Davies. Understanding blockchain technology for future supply chains: a systematic literature review and research agenda. *Supply Chain Management: An International Journal*, 24, 12 2018.

[43] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2012. https://ethereum.github.io/yellowpaper/paper.pdf.

[44] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2017.

[45] Karl Wüst and Arthur Gervais. Do you need a blockchain? In *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*, pages 45–54. IEEE, 2018.

[46] Xinan Yan, Linguan Yang, Hongbo Zhang, Xiayue Charles Lin, Bernard Wong, Kenneth Salem, and Tim Brecht. Carousel: Low-latency transaction processing for globally-distributed data. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 231–243. ACM, 2018.

[47] Haifeng Yu, Ivica Nikolić, Ruomu Hou, and Prateek Saxena. Ohie: Blockchain scaling made simple. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 90–105, 2020.