Scalable and Efficient Data Authentication for Decentralized Systems

Soujanya Ponnappalli\textsuperscript{1} Aashaka Shah\textsuperscript{1} Amy Tai\textsuperscript{2} Souvik Banerjee\textsuperscript{1} Vijay Chidambaram\textsuperscript{1,2} Dahlia Malkhi\textsuperscript{3} Michael Wei\textsuperscript{2}

\textsuperscript{1}University of Texas at Austin \textsuperscript{2}VMware Research \textsuperscript{3}Facebook

Abstract

Decentralized systems such as blockchains promise to fundamentally change how untrusted parties exchange data and assets. A key challenge in the decentralized setting is scalably authenticating data in the system. Merkle Patricia trees (MPT) are widely used in decentralized systems like Ethereum, to verify that the data is correct and up-to-date. Unfortunately, MPTs incur significant overhead due to random IO operations, serialization, and hashing. This paper presents the Distributed Merkle Patricia Tree (DMPT), a novel data structure that reduces the overheads of the MPT. A DMPT vertically shards the MPT across the memory of multiple nodes, eliminating the IO bottleneck. DMPTs reduce network bandwidth utilization using a combination of novel techniques such as witness compaction and node bagging. Compared to the MPT used in Ethereum, put and get operations in the DMPT achieve 80–160× better throughput, even when the Ethereum MPT is stored in memory. We demonstrate the effectiveness of DMPTs by building FVCHAIN, a modified version of Ethereum which uses DMPT instead of MPT. In a four-region geo-distributed deployment, FVCHAIN can verify 20,000 transactions per second (20× higher than Ethereum).

1 Introduction

Authenticated data structures such as Merkle trees [37] are widely used in file systems (ext4 [17], ZFS [1]), mobile storage systems (Android boot partition [23, 24]), and decentralized systems (Ethereum [53]). They are used to efficiently verify that a given data item belongs to a large data set, and to ensure that the data set as a whole has not been tampered with. Since the integrity verification happens in the critical path before the data is provided to the user, the performance of these data structures significantly affects the overall performance of the system; for example, the Ethereum developer community is working to increase Merkle Tree performance since it limits overall throughput [22, 36, 44, 45].

While authenticated data structures are actively researched on [13, 26, 30, 31, 34, 38, 42, 43, 46, 48], Merkle trees and their variants remain the most commonly used authenticated data structure. A Merkle tree constructs a tree of hashes over an ordered list of data items. The data are at the leaves, and each inner node is the hash of its children. Thus, a change in any node is propagated all the way to the root node. The root hash serves a concise representation of the data set.

Unfortunately, Merkle trees suffer from high overhead due to poor IO performance and expensive hashing and serialization operations. Verifying a data item belongs to the set requires reading all the nodes from the leaf to the root, causing a number of random reads. Similarly, updating a data item involves updating a path from leaf to root, leading to random writes. Merkle trees are often serialized using encodings like Recursive Length Prefix (RLP) [12], and are stored in a key-value store such as LevelDB [29], which further exacerbates the problem due to read and write amplification [47, 48]. Apart from the storage bottlenecks, serialization and hashing of Merkle tree nodes also contribute to the overhead. As a result, Merkle tree implementations get poor overall performance. For example, the Ethereum Merkle tree variant (the Merkle Patricia Tree) can only perform 1400 gets per second, even when it is stored in memory.

This paper presents the DMPT, a novel authenticated data structure that aims to reduce the storage and computation overheads of Merkle trees and their variants. The DMPT builds on a type of Merkle tree termed the Merkle Patricia Tree, and turns it into a distributed, sharded data structure that offers transactional capabilities to applications. The DMPT eliminates storage overhead by storing the entire tree in memory. The in-memory representation is optimized by using memory pointers for navigation (instead of hashes). Computational overhead from hashing and serializing of Merkle tree nodes is reduced using a collection of novel techniques such as lazy hash resolution and hash memoization. DMPT employs vertical sharding, where the tree is sharded (each shard is a direct child of the root) and stored in the memory of a different node. Since both reads and writes in Merkle trees operate on a vertical path down the tree, vertical sharding enables parallel reads and writes on different nodes. DMPTs associate a cache with each application which stores the top of the tree and recently accessed items. DMPTs use the cache in conjunction with techniques like witness compaction to reduce the network traffic between the application cache and the storage nodes. As a result of the careful design and optimizations, DMPTs achieve a performance of over 210,000 gets/second on a single node, 150× higher than the Ethereum Merkle Patricia tree even when it is stored in memory.

To demonstrate the utility of DMPTs for real applications, we built FVCHAIN (Fast Verification blockchain), a version of the Ethereum decentralized blockchain platform ported to use DMPTs instead of Merkle Patricia Trees. Ethereum verifies its transactions by checking each value that is read.
against the Merkle Patricia Tree. We modified the architecture of FVCHAIN to take full advantage of the high throughput and parallelism offered by DMPTs. Users submit transactions to a FVCHAIN client. The client then performs speculative pre-execution on the transactions, verifying the reads of each transaction by contacting the DMPT storage nodes. For each verified read, the client obtains a proof called the witness from the storage shard. Several clients can perform the IO-intensive step of speculative pre-execution in parallel, talking to different storage shards. The client then submits the transaction and the witnesses to the miner. The miner can verify the transaction using the witness, without contacting the storage nodes or performing IO; this is key to fast verification. The miner executes the transaction, applies the result of the transaction to its DMPT cache, and then updates the DMPT storage nodes. FVCHAIN has the same trust assumptions as Ethereum: different components such as miners and storage shards do not trust each other.

To evaluate FVCHAIN, we generate synthetic workloads that mirror transactions on the Ethereum mainnet blockchain. We analyzed Ethereum transactions and observed that the accounts involved in transactions have a zipf distribution: 90% of transactions involve the same 10% of accounts. We observed that only 10–15% of Ethereum transactions involved smart contracts. Our workload generator faithfully reproduces these distributions in its transactions. We evaluated FVCHAIN using this workload and found it could verify 30,000 transactions per second in a single node setting, and 20,000 transactions per second in a geo-distributed setting with four regions spread across three continents.

We would like to stress that FVCHAIN is only a proof-of-concept prototype built to evaluate the performance and utility of DMPTs. It is not a mature blockchain framework that can be compared to Ethereum. It is encouraging that FVCHAIN has a high verification throughput; this indicates that the Merkle tree is no longer the bottleneck for overall throughput. DMPTs could enable Ethereum to target different designs, such as packing more transactions into large blocks; this was previously prevented by the slow verification speed of the Ethereum Merkle Patricia Tree. However, more work is required to identify all the implications of such a design; our focus in this work is to merely demonstrate that such designs are possible using DMPTs.

The DMPT data structure are not without limitations. DMPTs depend upon the application for availability and fault tolerance. As DMPT creates multiple versions of data items for concurrency control, effective garbage collection is crucial for reclaiming space; if the abort rate increases, garbage collection may stall the system. Despite these limitations, DMPTs represent an interesting new point in the design space of authenticated data structures.

This paper makes the following contributions:

- An empirical analysis of Ethereum performance showing that authentication using Merkle Patricia Trees is a significant bottleneck (§2).
- The design and implementation of the novel DMPT, an authenticated data structure (§3).
- The architecture and implementation of the FVCHAIN decentralized blockchain framework which demonstrates how to effectively utilize DMPTs (§4).
- A workload generator for Ethereum transactions based on analysis of Ethereum transactions, and an empirical evaluation of DMPTs and FVCHAIN (§5).

### 2 Background

In this section, we provide some background on authenticated data structures and their overheads. We first describe the Merkle tree variant used by Ethereum, the Merkle Patricia Tree (MPT) (§2.1). We then experimentally show that the poor IO performance of MPTs is a significant bottleneck for Ethereum (§2.2).

![Merkle Patricia Tree](image)

**Figure 1: Merkle Patricia Tree.** Nodes A and C are branch nodes, B and D are extension nodes, and the rest are leaf nodes. These nodes are stored in a key-value store keyed by their hash, which randomizes their location on disk. Reading the account 626 for example, would require reading node A, looking up C using \( h(C) \), then looking up F using \( h(F) \).

#### 2.1 Merkle Patricia Trees

A Merkle Patricia Tree (MPT) is a combination of the Merkle tree [37] and Patricia tree data structures. The MPT constructs a tree of hashes on an ordered list of key-value pairs. The MPT stores one character of the key at each level. Each node can have multiple children. Keys with common prefix share paths in the tree. The non-leaf nodes in a MPT store the hashes of their children. The root hash reflects changes to any of the values as it hashes the entire list. The root hash thus represents the entire list with a unique constant-sized root hash.

MPTs can prove that a value exists in the list by providing a witness: for each node in the path from the leaf node to
the root, the witness contains its sibling hashes required to recompute its parent hash. A witness is verified against a root-hash: if recalculating the hashes from the leaf to the root results in the given root hash, the value belongs in the list represented in the root hash.

Modified MPT. The classical MPT data structure is inefficient since it stores a single character of the key at each level. For example, if the key is 64 hex characters long, a Patricia tree is 64 nodes deep, requiring a kilobyte of extra space to store one node per character at each level, and takes full 64 steps for a lookup or a delete operation. The modified MPT solves this problem by introducing three types of nodes: the branch node, the extension node, and the leaf node. A branch node has 16 branches (one branch per hex character, keys are hex strings), and a value. An extension node stores a string of characters encoding a path without branches in the tree, and a pointer to the next node. A leaf node stores the remaining characters in a key and the value. These different nodes compress the unique paths, resulting in an efficient modified Patricia tree, as shown in the Figure 1. Ethereum uses the modified MPT, that employs Recursive Length Prefix [12] encoding for serializing its nodes and secure KECCACK-256 for hashing the serialized nodes.

2.2 Overheads of Ethereum MPT

We analyze the costs of the Ethereum MPT in terms of storage IO and network IO. We also analyze the data structure’s impact on Ethereum verification time. We employ a widely-used Ethereum blockchain client, Parity 2.2.11 [10], in our experiments. We use Parity to sync a Ethereum full node and measure the various costs.

Storage IO cost. We find that verifying a single block with around 100 transactions results in over 10K random I/Os from reading nodes in the MPT. Figure 2 (a) shows the number of IO operations incurred during the sync of various blocks in the Ethereum blockchain; most of these IO operations are incurred due to the MPT. Ethereum’s MPT is stored using the LevelDB [29] key-value store, which amplifies the read and write cost further [47, 48].

Network IO cost. Figure 2 (b) shows the witness size required for verifying each block in the Ethereum blockchain. These witnesses would need to be transmitted over the network. In Ethereum where secure, 256-bit hashes are used, the witness for a single read of a 100 byte account can be around 4KB, an overhead of 40×. The witness size also increases over time as the total data in Ethereum increases.

The spikes in the Figure 2 (a) and (b) are the result of a DDOS attack [52] on the Ethereum’s world state, which created accounts with no balances, increasing the values in the MPT and thereby reducing the transaction execution and verification rate on the Ethereum state.

Verification time. Verifying a block involves checking that each transaction in the block is correct, and that executing the transactions lead to the world state summarized by the root hash embedded in the block. As verification involves reading witnesses from storage, verification is affected by the storage overhead mentioned above. Processing an Ethereum block with about 100 transactions takes hundreds of milliseconds due to the I/O operations required, even on a datacenter-grade NVMe SSD. Figure 2 (c) shows the time taken to sync Ethereum blocks along with number of IO operations required.

2.3 Summary

Though we analyze the Ethereum implementation of the MPT, our analysis is generally applicable to any implementation of the MPT or the Merkle tree. The MPT incurs significant overhead in terms of storage IO and network IO. This overhead will increase as the total amount of data served by the data structure increases. Our analysis also shows that the performance of these data structures significantly affects overall application throughput, since verification happens in the critical path. Thus, the challenge is to design an authenticated data structure that avoids the overheads experienced by the Ethereum MPT and offers high read and write throughput.
3 DMPTs

This section presents the DMPT, a novel distributed, authenticated data structure that seeks to eliminate the overheads of MPTs and Merkle trees.

3.1 Goals

**Fast authentication.** Creating and verifying witnesses should be fast and efficient, avoiding storage IO, serialization, and hashing overheads.

**Scalability.** As the amount of data increases, we should be able to add nodes to DMPTs to maintain or increase overall throughput.

**High concurrency.** DMPTs should achieve a high degree of concurrency, allowing multiple simultaneous readers and writers. DMPTs should provide consistent snapshots to applications.

**Low network overhead.** DMPTs should minimize both coordination among components and utilization of network bandwidth.

3.2 Overview

The DMPT is a distributed authenticated data structure built on top of the modified Merkle Patricia Tree used by Ethereum. DMPTs are designed to avoid the storage and network overheads caused by MPTs. DMPTs distribute the authenticated data without compromising safety.

DMPTs use three main techniques to achieve the design goals. First, DMPTs store the entire authenticated data structure in memory, rather than on solid state drives or magnetic hard drives. DMPTs use an optimized representation for the in-memory data: using memory pointers instead of hashes, and reducing serialization and hashing overhead using techniques like hash memoization and lazy hash resolution.

Second, DMPTs shard the tree vertically across multiple nodes. Vertical sharding is particularly suitable for DMPTs because witness creation and verification require vertical slices of the tree; these operations do not require nodes across the width of the tree. Operations can happen in parallel across vertical shards as long as conflicts are resolved at the root. Sharding also allows DMPTs to scale beyond the memory capacity of a single machine.

Finally, DMPTs employ caching and witness compaction to reduce the size of the witness transmitted over the network. DMPTs provide a cache for each application node that caches the top part of the DMPT, and storage nodes only send the un-cached part of the witness to the application. The size of the cache provides a configurable knob for trading memory consumption for network bandwidth utilization.

DMPTs provide support for ACID transactions with serializability. Transactions build on the snapshot they read at the start of the transaction. DMPTs are able to provide high concurrency for writers as they employ a form of multi-version concurrency control. The commit of a transaction results in a new snapshot of the tree; concurrent writers produce different snapshots which can be resolved by the application. DMPTs do not require the use of locks across its distributed nodes, yet provide strong, meaningful isolation guarantees to applications.

The design of the DMPT argues that a large random-access data structure can get higher throughput by serving it from memory over a network rather than creating it from local storage. While RAMCloud [41] proposed a similar design for low latency, DMPTs employ it to obtain high throughput. The design takes advantage of the nature of a Merkle Patricia tree: the cache-friendliness of the top of the tree, and the fact that witness creation only requires a vertical slice of the tree. Finally, the design also argues that it is better to create witnesses once and serve them over the network rather than create them on demand from local storage. Overall, DMPTs present a new point in the design space of authenticated data structures.

3.3 API

Table 1 presents the API of DMPTs. DMPTs provide the standard authenticated operations of get, put, and verify. Additionally, DMPTs provide ACID transactions with serializability to support applications that require transactional guarantees.

### Table 1: API. The table lists DMPT operations.

| API | Function |
|-----|----------|
| `beginTx()` | Begins the transactional context for further operations. |
| `endTx(tx-root)` | Commits or aborts a transaction. |
| `get(key, [root-hash]) = status` | Returns the value associated with the given key in the transactional snapshot or optional root-hash snapshot. |
| `put(key, value, [root-hash]) = status` | Inserts or updates the key-value pair in the context of the transactional snapshot or optional root-hash snapshot. |
| `verify(key, value, witness, root-hash) = status` | Verifies the given key-value pair in the root-hash snapshot using the given witness. |

3.4 Optimized In-Memory Representation

We observed that when the Merkle Patricia Tree is flattened and stored in a key-value store, there is significant overhead. This overhead comes both from the random read and write IO operations, and the serialization and hashing that must be performed when reading from or writing to storage.

To reduce these overheads, DMPTs store the tree in memory. We observed that merely storing the tree in an in-memory key-value store was not enough; the serialization and hashing overheads were as significant as the cost of doing random IO.
An in-memory DMPT (subtrees starting with a direct child of the root), and place the accessed witnesses (vertical paths down the tree). The cache scales with the number of shards. As witnesses are vertical paths down the tree, and the cache understands the DMPT structure and access patterns, unlike parallel. As witnesses are vertical paths down the tree, and the cache understands the DMPT structure and access patterns, unlike

3.5.2 Application Caches
DMPTs associate an in-memory cache with the application that caches the top part of the tree. The cache stores the top part of the tree (that changes more often) and the recently accessed witnesses (vertical paths down the tree). The cache understands the DMPT structure and access patterns, unlike

Pruning. The DMPT caches can be pruned to reclaim memory. They provide a configurable parameter called the retention level $r$. A prune operation discards all the nodes beyond $r$ by replacing the nodes in the level $(r + 1)$ with hash nodes. Hash nodes store a single hash of the node that is pruned; the hash is used to fetch the node from storage. As the DMPT cache uses memory pointers, hash nodes provide a mechanism to identify nodes in a machine-independent fashion. Pruning allows DMPTs to efficiently cache witnesses by constructing a partial tree. Reducing Witness Size. As data gets added to the DMPT, the size of the witnesses increases. To reduce the size of the witnesses transmitted over the network, DMPTs use a technique termed witness compaction. Witness compaction works in conjunction with the cache to reduce the size of witnesses. Since the application cache has the most up-to-date version of cached nodes, the shards only need to send the un-cached part of the witness, termed a compact witness, to the application. Witness for multiple items are collected together and sent in a node bag, which eliminates node duplicates across witnesses.

3.6 Concurrency Control
DMPTs provide ACID transactions with serializability. Concurrency control in DMPT does not require distributed locks though concurrent transactions read from and write multiple shards. DMPTs take advantage of the fact that every transaction creates new versions of the key-value pairs; there are no in-place updates.

Every transaction creates a unique snapshot when it commits, with new nodes in the DMPT. As a result, concurrent

5
transactions do not interfere with each other.

3.7 DMPT Operations

Transaction Begin. \( \text{txBegin}() = \text{tx-root} \). When the transaction begins, the latest committed transaction root hash at the application cache is used as the transaction snapshot, and returned as \( \text{tx-root} \). Gets in the transaction return values from the transactional snapshot.

Get. \( \text{get(key, [root-hash]) = value, [witness]} \). Gets return key-value pairs from the transactional snapshot or if provided, from \( \text{root-hash} \). Gets may optionally return a witness to the application. If the values and witnesses are not present in the application cache, they are retrieved from the storage shards and added to the cache.

Put. \( \text{put(key, value, [root-hash]) = status.} \) A put adds a new node to the application cache. The path from the leaf to the root is not immediately recalculated, due to lazy hash resolution. This results in fast put operations.

Verify. \( \text{verify(key, value, witness, root-hash) = status.} \) Verify uses the supplied key-value pair and witness to verify whether the key-pair belongs to the \( \text{root-hash} \). In the common case, verify does not cause any reads at the storage shards.

Transaction End. \( \text{endTx(tx-root) = status, new-tx-root}. \) When the transaction ends, the new hash root \( \text{new-tx-root} \) is calculated based on all puts performed in the transaction. The updates are first logged and applied to the application cache. The \( \text{new-tx-root} \) is provided to the application, and the application’s replication protocol is triggered to send the updates to other replicas; in parallel, the updates are sent to the shards. When the replication protocol completes, the transaction is durably committed. If replication among application caches fail, the transaction is aborted, and \( \text{new-tx-root} \) is discarded and later garbage-collected.

3.8 Availability and Fault Tolerance

DMPTs depend on the distributed application for availability and fault tolerance. The application ensures that at least one replica will always be available, and that the replicas agree on the root hash of the last committed transaction. The application is free to use any replication and consensus protocol as long as this guarantee is provided. Since the DMPT caches are embedded in the application replicas, they inherit the availability and fault tolerance guarantees.

DMPTs have novel failure-recovery mechanisms: if the storage shards lose their data due to a crash, they recover from the caches (instead of caches recovering from storage shards). This is because caches are the authoritative source of data, receiving updates before the storage shards. Each cache has a write-ahead-log where logical updates are persisted; this is used to reconstruct a shard that has lost data. The caches reconstruct data from other replicas if they lose data.

The application’s availability guarantee ensures at least one replica will be accessible.

3.9 Tuning

The performance of DMPTs can be tuned based on a single parameter \( r \): the retention level of the DMPT cache at the application. This parameter affects the size of the compact witnesses transmitted over the network. A higher \( r \) results in better performance and lower network utilization, but will require more memory for the cache.

4 FVCHAIN

To demonstrate the utility of DMPTs, we build FVCHAIN, a version of Ethereum modified to use DMPTs.

Overview. Ethereum suffers from two major problems: miners maintain the large authenticated state on-disk locally, and perform expensive IO in the critical path of transaction execution. FVCHAIN tackles both the problems by replacing the local on-disk MPT in Ethereum with DMPTs, and introducing speculatively pre-executing clients which perform IO prior to transaction execution. FVCHAIN achieves high transaction throughput by parallelizing the IO on DMPTs using multiple clients, and with novel techniques like witness revision, that maximize its overall transaction throughput.

Our goal in building FVCHAIN was to demonstrate that DMPTs could potentially be used to increase the throughput of blockchain platforms. The public nature of Ethereum along with its support for complex smart contracts make it an ideal use case for DMPTs. FVCHAIN is a not a mature system that is a replacement for Ethereum; the implications of our design choices needs to be examined further.

We first provide background on Ethereum, then describe the architecture of FVCHAIN, discuss its security and trust assumptions, and finally discuss possible bottlenecks arising from the design.

4.1 Ethereum Overview

Ethereum [5, 53] is as a decentralized service which provides its users access to a database known as the world state. The world state consist of a mapping between an address and accounts or smart contracts.

Transactions. To transfer funds to another account, the account owner sends a message with the payment to a destination address. If the destination is a smart contract, it is executed. The message is signed with the owner’s private key, and sent to the Ethereum transaction pool \( (\text{txpool}) \) via a gossip protocol. Participants in Ethereum known as miners select transactions from the txpool and assemble them into blocks, chained together using Proof-of-Work consensus. These blocks become part of the blockchain that can be replayed to reproduce the world state.

Ethereum stores the world state in a modified Merkle Patricia Tree. Miners include a Merkle tree root hash for the world state in additions to transactions in each block. The
Merkle root hash is broadcast to all participants in the system, and allows miners to prove that an account balance exists in the world state. The keys in the MPT are address hashes. KECCAK-256 hashing is used to transform keys into uniformly distributed 32-byte values. Witnesses can be large in the MPT: for a tree of height 10, a witness for a 100 byte account is typically around 4KB, resulting in a $40 \times$ size overhead for authentication. Updating a leaf node requires rehashing it and updating each node to the root, resulting in significant amount of random I/O [22, 36, 44, 45, 54].

**The quest for higher throughput.** While the Proof of Work [25, 40] consensus used by Ethereum regulates the block creation rate to approximately one block every 10 seconds, consensus does not limit the number of transactions per block, or the *block size*. The block size is limited mainly by the block verification time.

Larger blocks will lead to higher verification times due to the additional IO required. Verification is carried out by each Ethereum node in the critical path; verification must be completed before the block can be accepted and sent to other Ethereum nodes. Higher block verification times causes increased forking, compromising the safety and live-ness of the blockchain. Ethereum strives to have each block be received and verified by a majority of the nodes in 10–12 seconds [49]. The block size is kept small lest the verification time increase beyond acceptable levels [54].

The stateless client proposal [19] from Ethereum replaces local storage IO with network IO from servers that maintain the world state. This proposal would require that witnesses be sent over the network. As shown in Figure 2 (b), witness sizes are increasing over time, so this proposal would lead to unacceptable network bandwidth utilization.

If Ethereum could be modified to have large blocks without increasing verification time or placing a large strain on the network, Ethereum would have high transaction throughput (even though the block creation rate is fixed by the Proof-of-Work consensus). DMPTs enable this point in the design spectrum; we build FVCHAIN to demonstrate the idea.

4.2 Architecture

Figure 3 illustrates the novel architecture of FVCHAIN. FVCHAIN has three components: the storage nodes, the clients, and the miners. The local storage in Ethereum’s miners is replaced by the caches and storage shards of DMPT. The clients read data from DMPT shards, and submit transactions (along with witnesses for transaction data) to the miners. Each miner has an associated DMPT cache. The miners verify and execute transactions, and update their DMPT cache and the DMPT storage shards.

The FVCHAIN architecture maximizes parallelism. Several clients can read from different shards at the same time. Clients submit both transactions and witnesses to the miners; in the common case, the miners do not have to perform reads at the storage nodes. As a result, the miners are not disk IO-bound, verifying and executing transactions efficiently. Once a transaction is executed, miners update their application cache first, and then update the storage nodes. Only clients block for IO when collecting data and witnesses for transactions. However, as clients are performing IO in parallel, the overall throughput of the system is high.

4.2.1 Storage nodes

Storage nodes in FVCHAIN each contain a DMPT shard. FVCHAIN shards the total state in Ethereum based on the first four bits of the key, and contains 16 shards.

**Updates.** Storage nodes receive updates from miners, corresponding to the transactions in one block of the blockchain. They apply the updates to their in-memory DMPT shard, re-calculate the root hash, and confirm it matches the root hash in the block supplied by the miner. They also verify the block header and reject updates from invalid blocks.

**Reads.** Clients read account values from the storage nodes. Storage nodes provide a *witness node bag*, a collection of nodes comprising the witness for the read and the updated accounts. Witness for multiple accounts are combined, removing the duplicate nodes. The storage nodes provide compact witnesses based on the miner’s cache size.

**Forks.** Multiple miners can write different competing versions of the same block to the storage nodes at the same time. This is termed a *fork* in the blockchain. The storage nodes store both competing versions of the block. As the blockchain grows, older forks which are not built on are discarded.

4.2.2 Speculative Pre-Execution by Clients

One of the novel aspects of FVCHAIN is how the clients operate. In Ethereum, miners perform IO for witness creation and verification. In contrast, only clients perform IO in FVCHAIN (in the common case). Clients contact storage nodes, collect
witnesses that are pre-fetched. For example, Figure 4 shows
will re-execute the transaction again to maintain correctness.

witnesses from the storage nodes. The execution is speculative
witnesses in the form of node bags, and submit transactions
and node bags to the miners. Miners can then verify transac-
tions without performing any IO, drastically increasing their
performance.

The client receives a transaction from the client. The client
speculatively pre-executes the transaction, reading values and
witnesses from the storage nodes. The execution is speculative
for two reasons. First, the miner does not trust the client.
Second, the transaction might use the timestamp or block
number of the block it appears in during execution. The miner
will re-execute the transaction again to maintain correctness.

The speculative pre-execution produces the witness node
bags for accounts and keys read in the transaction. The client
then sends the node bags along with the transactions to the
miner. The miner tolerates incorrect or incomplete node bags,
fetching node bags from the storage nodes as required.

Speculative Values. When the transaction depends on an ex-
ternal value such as the block number or the timestamp, the
client speculatively provides a value. We find that this is effec-
tive in obtaining the correct node bags from the storage nodes;
in other words, the inaccuracy introduced by guessing such
values does not significantly alter execution. For example, the
CryptoKitties mixGenes function repeatedly references the
current block number and its hash. Because these numbers
are only used to generate randomness of written values in the
function, substituting inaccurate values does not affect the
witnesses that are pre-fetched. For example, Figure 4 shows
an example of a function from the popular CryptoKitties con-
tract that repeatedly references the current block number and
its hash, where substituting inaccurate values does not affect
the witnesses that are pre-fetched.

Stale data. We make a similar observation that clients can
pre-execute with stale data and still prefetch the correct node
bags. For example, the CryptoKitties giveBirth function is a
fixed-address contract, where the addresses read (loads from
the kitties array) only depend on the inputs from the
message call. To deal with rare variable-address contracts, the
miner may asynchronously read from a storage node after the
transaction has been submitted. Even in these cases, the client
will have retrieved some of the correct nodes required for the
transaction (e.g., the to and from accounts).

Aborting transactions at the client. Another advantage of
the speculative pre-execution is that it can filter out transac-
tions that miners would abort. Contrast this with the Ethereum
blockchain, in which aborted transactions are no-ops, but still
take up valuable space in the public ledger. In FVCHAIN,
clients can prevent miners from spending valuable cycles
executing transactions that will be eventually aborted. The
tradeoff is that staleness might cause clients to abort transac-
tions conservatively. For example, in Figure 5, if the client
has a stale view of the matron kitty, it might abort on Line 5
if it believes the matron is not ready to give birth. If account
holders believe that a client has incorrectly aborted their trans-
action, they can send their transaction to other clients.

4.3 Miners

A miner’s task is to verify the transactions sent by the client,
execute the transactions, and update the storage nodes. There
are multiple miners in FVCHAIN. Each miner has its own
DMPT cache, which provides a compact, consistent, and au-
thenticated view of the FVCHAIN world state.

When a miner needs to read or update an account, it con-
sults both its DMPT cache and the node bag submitted by the
client. Miners exploit the Merkle tree for both authentication
and consistency. Verifying that an account value hashes to the
root of a block proves two things: that the value belongs to
that block, and that the value has not been tampered with.

The miner groups together transactions and mines a new
block. The miner informs other miners of the new block along
with the node-bag nodes required to process the block.

The miner then sends the block with a list of changes that
were made to storage nodes, which verify and process the
updates. The miner can then compact its in-memory tree by
pruning it to a configurable height in the tree.

Witness Revision. Miners while processing transactions may
end up with stale nodes in the node bags submitted by the

---

**Figure 4:** Indeterminate contract values do not affect wit-
ness prefetching. Psuedocode of the CryptoKitties mixGenes
function. The function makes repeated calls to curBlock, the
block of the executed transaction. Because this is not known
at pre-execution time, the client substitutes an approximate
value, which doesn’t affect witness prefetching because these
numbers only affect written values in the function.

```
1: function mixGenes(mGenes, sGenes, curBlock)
2:     uint256 randomN ← curBlock.blockHash
3:     randomN ← KeccakHash(randomN, curBlock)
4:     MemoryAry babyGenes ← mix(mGenes, sGenes, randomN)
5:     return babyGenes
```

**Figure 5:** Fixed-address contract example. Psuedocode of
the CryptoKitties giveBirth function, which indicates that
storage reads (accesses to the kitties array) only depend on
function arguments.

```
1: function giveBirth(matronId)
2:     Kitty matron ← kitties[matronId]
3:     assert(matron.isValid)
4:     assert(matron.isReadyToGiveBirth)
5:     xCoord ← matron.xCoord
6:     yCoord ← matron.yCoord
7:     newKitty ← createKitty(matron, sire, childGenes)
8:     assert(newKitty.isValid)
9:     if newKitty.isReadyToGiveBirth.
10:     return newKitty
```
clients. However, miners replace the stale nodes with more up-to-date versions using the DMPT cache, as shown in Figure 6. After processing tx1, node C is modified to C' but, the node bag still has stale node C. The stale node prevents the miner from processing tx2, which accesses node G. Using witness revision, the miner revises the node bag to reflect the recent changes, making it possible to reach Node G from the modified node C'. With witness revision, miners process transactions without relying on clients for the latest witnesses.

Witness revision can be explained with an analogy from version control: it is similar to doing git push (trying to upload your changes), finding out something else in the repository has changed, doing a git pull (obtaining the change in the repository), merging and then doing a git push. Witness revision is important for maximizing the benefit of the node bags provided by the client and allows the DMPT to benefit from provided nodes when a simple witness would otherwise be rejected.

Incomplete Node Bags. The node bags from the clients might have insufficient nodes to process a transaction if the DMPT cache is pruned aggressively. Consider that a miner prunes its cache after processing tx1, as shown in Figure 7. A hash node with the hash of C' is present in the DMPT cache, and the miner has insufficient nodes to reconstruct the witness for the value in node G. In such cases, the miner can tolerate the incomplete node bags by fetching the required nodes from the shards. Alternatively, the miner can reject the transaction, forcing the client to perform IO.

Figure 7: Incomplete Transactions. In this example, the tree is pruned after applying tx1, resulting in only a hash node for the new node C'. When tx2 arrives, there are insufficient nodes in the node bag to reach node G, since tx2 only has the stale node C. As a result, tx2 is incomplete, so the miner must reject the transaction or fetch C from the bottom layer.

4.4 Examining the impact of the design

We now describe how the design affects trust, security, and incentives. We discuss new attacks and bottlenecks result from the design of FVCHAIN.

Trust Assumptions. FVCHAIN has the same cryptographic assumptions as Ethereum and assumes no additional trust between its components: the clients, miners, and the storage nodes. Miners don’t trust clients: they re-execute the transactions, verify the node bags, and can perform network I/O to handle incomplete transactions. Miners don’t trust the storage shards: miners verify their reads from the storage nodes and compare the reads against their cache, operating without trusting the client and the storage shards.

Storage shards do not trust miners and clients. Storage shards verify updates from the miners, and create a new snapshot for the update, and thereby do not affect existing state.

Clients read data from the storage shards and can verify their reads. Clients can resubmit transactions to other miners if their transactions were unfairly handled by a miner.

Security. Our system assumes the same threat model as Ethereum. Like Ethereum, FVCHAIN components are incentivized to behave correctly. Some examples of incorrect behavior include: clients submitting transactions with empty node bags, storage shards sending stale witnesses which cannot be revised, and miners rejecting transactions.

Incentives. Clients pay storage nodes for access to the authenticated data. Storage nodes are incentivized to send correct data (as clients can verify their reads) and to send recent data (as clients can detect staleness by observing the recent state of the system). Miners are paid through block rewards and are
incentivized to broadcast the correct updates to storage nodes to aid the acceptance of their block. Users can run clients (or pay someone to run clients for them) that pre-execute and submit transactions to the miners. Users can detect malicious clients from the results of pre-execution and execution, which incentivizes clients to pre-execute correctly. While we have developed the basic ideas of this incentives solution, we have not analyzed it rigorously: doing so would involve game theoretical and economic models beyond the scope of this paper.

**Attacks.** Storage nodes add another threat vector where miners could land Denial of Service attacks on storage nodes by sending invalid blocks and reducing their availability. However, correctness is not compromised as storage nodes identify and reject updates with invalid blocks.

Clients can serve read only transactions without having to rely on miners to mine those transactions. Clients can be bombarded with empty transactions, just like miners in Ethereum.

**Bottlenecks.** DMPTs provide better performance for witness creation and verification, enabling large blocks in FVCHAIN. Consensus protocols like Proof of Work only limit the block creation rate, not the size of the blocks.

Large blocks, if they take higher block propagation + verification time, will cause forking, compromising the system. DMPTs enable larger blocks by reducing the overheads involved in verification.

Block advertisement time in FVCHAIN could go up significantly. However, FVCHAIN blocks contain only transactions (no witnesses), so advertising the block is efficient. There is the potential for the network to become a bottleneck to advertise the nodes required to verify blocks. However, miners can control the block sizes, and thus limit the network bandwidth used for advertising node bags.

### 4.5 Implementation

We implement DMPT and FVCHAIN in Typescript, targeting node.js. The miners and storage nodes use the DMPT as a library. The performance-critical portions of the code, such as secp256k1 key functions for signing transactions and generating keccak hashes, are written as C++ node.js bindings. To execute smart contracts, we implement bindings for the Ethereum Virtual Machine Connector interface (EVMC) and use Hera (v0.2.2), which can run contracts implemented using Ethereum flavored WebAssembly (ewasm) or EVMC1 bytecode through transcompilation. Our speculative pre-executing client is implemented in C++. The DMPT and FVCHAIN implementations (15K lines of code), will be made open source.

### 5 Evaluation

In this section, we evaluate the performance of DMPTs and FVCHAIN. We seek to answer the following questions:

1. What is the performance of the DMPT for various operations on a single node? (§5.2)

2. How does the size of the DMPT cache impact the witness sizes, memory overhead, and rate of transaction aborts? (§5.3)

3. What is the performance of FVCHAIN on various end-to-end workloads that characterize the Ethereum public blockchain? (§5.4)

#### 5.1 Experimental Setup

We run the experiments in a cloud environment on instances which are similar to the m4.2xlarge instance available on Amazon EC2 with 32GB of RAM and 48 threads per node. We use Ubuntu 18.04.02 LTS, and node.js v11.14.0. For the end-to-end benchmarks, each storage node, miner, and client is deployed on its own instance.

#### 5.2 Evaluating DMPT on a single node

First, we evaluate the DMPT running on a single node. This tests the performance of the optimized in-memory representation of DMPT. We measure the throughput of point put and get operations for a variety of tree sizes against the state-of-the-art Ethereum MPT.

To make a fair comparison, we compare DMPT with the in-memory implementation of Ethereum MPT [8] that uses

![Figure 8: Performance of DMPT on a single node. (a) The figure shows the absolute put and get throughput of DMPT. Throughput relative to the Ethereum MPT is shown on the bars. As the number of accounts increase, DMPT throughput increases relative to Ethereum MPT. (b) This figure shows the memory used by DMPT and Ethereum MPT across varying number of accounts. The trend line captures the height of the MPT. DMPTs are orders of magnitude more memory-efficient than Ethereum MPT. Note the log scale on the axes.](image-url)
We dump the Ethereum world state every 100K blocks until 4M blocks and use it to micro-benchmark DMPTs; every key in these benchmarks is a 160-bit Ethereum address and values are RLP-encoded Ethereum accounts [12].

**Gets.** DMPTs with 1.19M accounts, obtain a get throughput of \( \approx 216K \text{ ops/s} \), that is \( 150 \times \) the throughput of Ethereum MPT. The main reason for the DMPT’s better performance is the use of in-memory pointers. To fetch a node, the DMPT simply needs to follow a path of in-memory pointers to the leaf node. On the other hand, walking down a tree path means looking up the value (node) at a particular hash for each node in the Ethereum MPT. Even though this database is in-memory, looking up values in an in-memory key-value map is still more expensive than a few pointer lookups. Furthermore, the larger the world state, the better DMPT’s in-memory Merkle tree performs over the Ethereum MPT. This is simply because the larger the state, the taller the tree, so the more nodes on the path to a leaf, see Figure 8 (a).

**Puts.** DMPTs with 1.19M accounts obtain a put throughput of \( \approx 245K \text{ ops/s} \), that is \( 160x \) the throughput of Ethereum MPT. Due to lazy hash resolution, a put does not need to adjust any values in the path from the leaf to the root; in contrast, every node in the path has to be updated in the Ethereum MPT. put throughput in the DMPT is more than two orders of magnitude higher than in the Ethereum MPT.

**Tree Size.** Figure 8 (b) shows that DMPTs are significantly smaller than Ethereum MPTs when the same number of accounts are stored. With 1.19M accounts, Ethereum MPT consumes \( \approx 26021MB \) and DMPTs consume \( \approx 775MB \), using \( 34 \times \) lesser memory. The primary reason for this is the efficient in-memory representation of DMPTs. Ethereum MPT is not-memory efficient as it uses 32-byte hashes as pointers and relies on memdown [9] to store the flattened MPT as key-value pairs. The significantly reduced size of the DMPT, along with sharding, enables DMPTs to be stored entirely in memory, eliminating the IO bottleneck.

**Lazy hash resolution.** We run an experiment where we trigger a root hash calculation after every \( N \) write (put or delete) operations. As \( N \) increases, the performance of DMPT write operations also increases. At \( N = 1000 \) (the root hash is read every 1000 writes), DMPT is \( 4-5 \times \) faster than Ethereum MPT. Since the root hash calculation is expensive (requiring RLP serialization of nodes), performing it even once every 1000 writes reduces DMPT performance from \( 150 \times \) Ethereum MPT performance to \( 5 \times \).

**5.3 Impact of cache size**

Next, we evaluate the performance of the distributed version of the DMPT when the cache size is changed. Pruning the cache reduces memory consumption but results in larger witnesses being transmitted, and more transactions being aborted due to insufficient witness caching. We evaluate these effects.

**Memory consumption.** We evaluate the reduction in the application memory utilized, from pruning the DMPT cache, across varying cache sizes \( r \). Figure 9 (a) shows that lower \( r \) will result in higher memory savings, with a tree of depth
five consuming only 40% of the memory consumed by the full tree. However, this means that either 1) DMPT shards will have to provide larger witnesses or 2) the application will experience a higher abort rate due to insufficient witness caching.

**Witness Compaction.** DMPTs transmit compact witnesses which include only the un-cached parts of the witness. DMPTs employ node-bagging where they combine multiple witnesses and eliminate duplicate nodes. Figure 9 (b) shows the reduction in witness size due to node bagging and witness compaction, based on the height of the cached tree \( r \). Witness compaction and node bagging together reduce witness sizes by up-to 95% of their original size.

**Transactions.** Pruning the cache discards cached witnesses. Since transactions abort if the witnesses are not cached, this increases the abort rate. To study the effect of varying the cache size on transaction abort rate, we use FVCHAIN with 16 storage nodes, 1 miner, and enough clients to saturate the miner. Transactions are generated by selecting two random accounts from a set of \( N \) accounts. Figure 9 (c) shows that the transaction abort rate is dependent on two factors: the DMPT cache retention level, and the number of accounts. In particular, increasing \( r \) reduces the transaction abort rate. More importantly, with large number of accounts \( N \), the contention on Merkle tree nodes reduces, reducing the abort rate for fixed \( r \), making DMPTs practical for application with low available memory.

### 5.4 End-to-End Blockchain Workloads

Finally, we evaluate the end-to-end performance of FVCHAIN against synthetically generated workloads that mirror transactions on the Ethereum public mainnet blockchain.

**Challenges.** Since Ethereum transactions are signed, the public transactions are not conducive to experiments: we cannot change transaction data or the sources accounts, because we do not have the \( \text{secp256k1} \) private key. Since FVCHAIN runs transactions at a much higher rate than Ethereum, we quickly run into state mismatch errors, and eventually, exhaust the available transactions.

To tackle this challenge, we analyze the public blockchain to extract salient features, and develop a synthetic workload generator which generates accounts with private keys we control so our clients can run and submit signed transactions.

**Synthetic Workload Generator.** We analyze the transactions in the Ethereum mainnet blockchain to build a synthetic workload generator. We analyzed 100K recent (since block 7M) and 100K older blocks (between blocks 4M and 5M) in the Ethereum blockchain to determine: 1) the distribution of accounts involved in transactions, 2) what fraction of all transactions are smart contract calls. We observe that 10-15% of Ethereum transactions are contract calls and the rest are simple transactions. This is true of both recent blocks and older blocks. It is also the case that a small percentage of accounts are involved in most of the transactions. Figure 11 shows a CDF of the number of times accounts are called, where the accounts are sourced from the blocks we inspected. Based the analyzed data, we generate workloads where 90% of accounts are called 10% of the time, and 10% of the accounts are called 90% of the time. Smart contracts are invoked 15% of the time.

**Throughput.** Figure 10 (a) shows the transaction throughput results. First, this figure shows that the FVCHAIN can achieve an end-to-end verification throughput of 30,000 transactions per sec. It also demonstrates the scalability of the DMPT and FVCHAIN, which scales as more clients are added. By varying the DMPT retention level at the miners from 0 to 6, the DMPT shard throughput increases by 7x, from 1.3K ops/s to 9.4K ops/s, increasing the scalable creation and transmission of witnesses.

![Figure 10: End-to-End Blockchain Workloads. (a) The figure shows the scalability of DMPTs in FVCHAIN with increasing number of clients and varying cache retention levels (r). The workload used in the experiment is representative of the account distributions in Ethereum transactions. Miners in FVCHAIN can verify about about 30K ops/s with 4 clients each, when configured at r = 7. (b) This figure shows the overall throughput of FVCHAIN in a geodistributed deployment. Miners at r = 8 can verify about 20K ops using 4 clients each, when communicating with the DMPT across WAN.](image)

![Figure 11: CDF of # account calls in a transaction. This figure shows that, consistently throughout the blockchain, few accounts participate in the majority of transactions. Note the log-scale on x-axis. We include data for accounts encountered in the blocks that we analyzed.](image)
Geo-distributed Experiment. We also ran a geo-distributed experiment, with varying numbers of regions across 3 continents. Each region has 4 clients, 1 miner, and 16 storage nodes, caching eight levels of the DMPT tree ($r = 8$). Figure 10 (b) reports the throughput experienced by the FVCHAIN. FVCHAIN in a single region achieves a throughput of $\approx 25K$ transactions/sec; when we scale to four regions, the throughput drops to $\approx 20K$ transactions/sec, thus retaining 80% of the performance in a geo-distributed setting.

Contract Calls. We also ran a workload where accounts repeatedly call the OmiseGO Token, which is an ERC-20 token contract [4]. Four clients repeatedly called the token contract against a single FVCHAIN miner with DMPT cache configured at $r = 8$, achieving a throughput of $17.9K \pm 796$ contract calls per second. This demonstrates that even for pure contract contract calls, FVCHAIN can provide orders of magnitude higher transaction throughput than other blockchains.

6 Related Work

In this section, we place FVCHAIN in the context of related work. Table 2 shows the major differences between FVCHAIN and related work. We compare FVCHAIN to the stateless clients proposal, HyperLedger fabric, and prior work on sharding. Finally, we discuss the recent work on efficient authenticated data structures in the context of DMPTs.

6.1 Ethereum’s Stateless Client

Ethereum’s stateless client proposal [19] tackles the IO bottleneck at miners. It proposes inserting witnesses into the blocks, enabling miners to verify a block without performing IO. Despite active discussion [3, 6, 11], stateless clients have not been implemented due to practical concerns of witness sizes: witnesses for a single, simple ethereum transaction can be $\approx 4-6$KB resulting in $40-60 \times$ the network overhead ($\S 2$). FVCHAIN handles this by using witness compaction and node bagging to reduce the size of witnesses transmitted over the work (by as much as 95%). FVCHAIN uses clients which prefetch the required witnesses, removing the I/O burden on the miners.

6.2 Pre-execution

The Hyperledger Fabric [14] is a permissioned blockchain that optimistically executes transactions and uses signatures, instead of witnesses, to verify the execution. However, Fabric has special nodes that maintain the centralized state, which can quickly become a bottleneck, as clients by design, must contact multiple nodes to submit transactions. In contrast, FVCHAIN is a permission-less blockchain that uses clients to pre-fetch witnesses in parallel, eliminating the storage bottleneck in the speculative pre-execution of transactions.

6.3 Sharding and Consensus

We discuss sharding and novel consensus proposals in the context of FVCHAIN. Blockchains using sharding [32, 33, 35, 50, 56] run independent parallel chains over subsets of the state, speeding up root calculation and shrinking witnesses. However, sharding requires syncing the chains for consistency, requiring expensive locking or communication protocols, potentially eliminating the benefits of sharding. FVCHAIN uses DMPTs which shard the results, instead of executing over the sharded state [51, 57].

In the context of consensus, FVCHAIN is orthogonal to the underlying consensus and increases the overall throughput by increasing the number of transactions that can be included in a block without increasing the verification time. Work that proposes alternatives to the proof-of-work consensus [7, 15, 20, 27, 28, 39, 55] are orthogonal to our work.

6.4 Dynamic Accumulators

Merkle trees are widely used in blockchains [2, 5] for witness verification. However, the size of Merkle witnesses grow with the increasing amount of data. Constant-sized dynamic accumulators [16] provide fixed-sized witnesses; however, the verification throughput is low. Improving the verification rate of constant-sized accumulators is an ongoing effort [18, 21]. In contrast, DMPTs provide both small witnesses through witness compaction and fast verification.

7 Conclusion

Providing authenticated data scalably is a challenge in the decentralized setting, as structures like Merkle trees, that authenticate data, suffer from poor IO performance and low scalability. As decentralized services become more popular, their data will increase commensurately, increasing the overheads of data authentication. DMPTs address this limitation by leveraging the fundamental properties of a Merkle tree in a distributed manner: the verification and authentication properties of the root hash allow the DMPT cache to achieve strong consistency and the in-memory shards enable DMPTs to achieve high scalability. FVCHAIN uses DMPTs with speculatively pre-executing clients and novel techniques like witness revision to achieve a transaction rate of over 20K tx/s in geo distributed settings, demonstrating how DMPTs scalably authenticate large amounts of data. We expect future work on cryptographic accumulators and Byzantine consensus to improve the performance of DMPT and FVCHAIN further, enabling practical, high performance decentralized systems to be realized.

| Proposals          | Stateless Clients | Hyperledger | FVCHAIN |
|--------------------|------------------|-------------|---------|
| Permissionless     | ✓                | ×           | ✓       |
| PoW Consensus      | ✓                | ×           | ✓       |
| Pre-execution      | ×                | ✓           | ✓       |
| Sharded Storage    | ×                | ×           | ✓       |

Table 2: Related Work. We highlight the core differences in the Stateless Clients proposal, Hyperledger, and FVCHAIN.
References

[1] 2005. ZFS End-to-End Data Integrity. https://blogs.oracle.com/bonwick/zfs-end-to-end-data-integrity.

[2] 2019. Bitcoin. https://bitcoin.org/en/.

[3] 2019. Detailed analysis of stateless client witness size, and gains from batching and multi-state roots. https://ethresear.ch/t/detailed-analysis-of-stateless-client-witness-size-and-gains-from-batching-and-multi-state-roots/862.

[4] 2019. ERC20 Token Standard. https://theethereum.wiki/w/index.php/ERC20_Token_Standard.

[5] 2019. Ethereum. https://github.com/ethereum/.

[6] 2019. Ethereum Improvement Proposals Repository. https://github.com/ethereum/EIPs.

[7] 2019. Hybrid Casper FFG. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1011.md.

[8] 2019. Implementation of the modified merkle patricia tree as specified in the Ethereum’s yellow paper. https://github.com/ethereumjs/merkle-patricia-tree.

[9] 2019. In-memory abstract-leveldown store for Node.js and browsers. https://github.com/Level/memdown.

[10] 2019. Parity Ethereum 2.2.11-stable. https://github.com/paritytech/parity-ethereum/releases/tag/v2.2.11.

[11] 2019. A possible solution to stateless clients. https://ethresear.ch/t/a-possible-solution-to-stateless-clients/4094.

[12] 2019. Recursive Length Prefix Encoding. https://github.com/ethereum/wiki/wiki/RLP.

[13] Aris Anagnostopoulos, Michael T. Goodrich, and Roberto Tamassia. 2001. Persistent Authenticated Dictionaries and Their Applications. In Information Security, George I. Davida and Yair Frankel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 379–393.

[14] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. 2018. Hyperledger fabric: a distributed operating system for permissioned blockchains. In Proceedings of the Thirteenth EuroSys Conference. ACM, 30.

[15] Shehar Bano, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn, and George Danezis. 2017. Consensus in the Age of Blockchains. CoRR abs/1711.03936 (2017). arXiv:1711.03936 http://arxiv.org/abs/1711.03936

[16] Josh Cohen Benaloh and Michael de Mare. May 1994. One-way accumulators: A decentralized alternative to digital signatures (extended abstract). In EUROCRYPT’93, volume 765 of LNCS. 274–285.

[17] Eric Biggers. 2018. fs-verity: filesystem-level integrity protection. https://lwn.net/Articles/763441/.

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

[19] Vitalik Buterin. 2017. The Stateless Clients Concept. https://ethresear.ch/t/the-stateless-client-concept/172.

[20] Christian Cachin and Marko Vukolic. 2017. Blockchain Consensus Protocols in the Wild. CoRR abs/1707.01873 (2017). arXiv:1707.01873 http://arxiv.org/abs/1707.01873

[21] Jan Camenisch and Anna Lysyanskaya. August 2002. Dynamic accumulators and application to efficient revocation of anonymous credentials. In CRYPTO 2002, volume 2442 of LNCS. 61–76.

[22] cdetrio. 2018. Research alternatives to leveldb (like badger). https://github.com/ethereum/go-ethereum/issues/15717.

[23] Anrin Chakraborti, Bhushan Jain, Jan Kasiak, Tao Zhang, Donald E. Porter, and Radu Sion. 2017. dm-x: Protecting Volume-level Integrity for Cloud Volumes and Local Block Devices. In Proceedings of the 8th Asia-Paciﬁc Workshop on Systems, Mumbai, India, September 2, 2017. ACM, 16:1–16:7. https://doi.org/10.1145/3124680.3124732

[24] Jonathan Corbet. 2011. dm-verity. https://lwn.net/Articles/459420/.

[25] Cynthia Dwork and Moni Naor. 1993. Pricing via Processing or Combating Junk Mail. In Proceedings of the 12th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO ’92). Springer-Verlag, London, UK, UK, 139–147. http://dl.acm.org/citation.cfm?id=646757.705669
[26] C. Chris Erway, Alptekin Küpçü, Charalampos Papamanthou, and Roberto Tamassia. 2015. Dynamic Prov- able Data Possession. *ACM Trans. Inf. Syst. Secur.* 17, 4, Article 15 (April 2015), 29 pages. https: //doi.org/10.1145/2699909

[27] Ittay Eyal, Adem Efe Gencer, Emin Gun Sirer, and Robbert Van Renesse. 2016. Bitcoin-NG: A Scalable Blockchain Protocol. In 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16). USENIX Association, Santa Clara, CA, 45–59. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/eyal

[28] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP ’17)*. ACM, New York, NY, USA, 51–68. https://doi.org/10.1145/3132747.3132757

[29] Google. 2019. LevelDB. https://github.com/google/leveldb

[30] Alexander Heitzmann, Bernardo Palazzi, Charalampos Papamanthou, and Roberto Tamassia. 2008. Efficient integrity checking of untrusted network storage. In *Proceedings of the 2008 ACM Workshop On Storage Security And Survivability, StorageSS 2008, Alexandria, VA, USA, October 31, 2008*, Yongdae Kim and William Yurcik (Eds.). ACM, 43–54. https://doi.org/10.1145/1456469.1456479

[31] Fangyong Hou, Dawu Gu, Nong Xiao, Fang Liu, and Hongjun He. 2009. Performance and Consistency Improvements of Hash Tree Based Disk Storage Protection. In *International Conference on Networking, Architecture, and Storage, NAS 2009, 9-11 July 2009*, Zhang Jia Jie, Hunan, China. IEEE Computer Society, 51–56. https://doi.org/10.1109/NAS.2009.15

[32] Vitalik Buterin Joseph Poon. 2019. Plasma: Scalable Autonomous Smart Contracts. https://plasma.io/plasma.pdf

[33] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford. 2018. OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*. 583–598. https://doi.org/10.1109/SP.2018.000-5

[34] Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid Reyzin. 2006. Dynamic Authenticated Index Structures for Outsourced Databases. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD ’06)*. ACM, New York, NY, USA, 121–132. https://doi.org/10.1145/1142473.1142488

[35] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. 2016. A Secure Sharding Protocol For Open Blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS ’16)*. ACM, New York, NY, USA, 17–30. https://doi.org/10.1145/2976749.2978389

[36] Max Gillett. 2018. Block processing time slows down following trie persistence. https://github.com/ethereum/go-ethereum/issues/16674.

[37] Ralph C Merkle. 1987. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*. Springer, 369–378.

[38] Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi. 2014. Authenticated Data Structures, Generically. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL ’14)*. ACM, New York, NY, USA, 411–423. https://doi.org/10.1145/2535838.2535851

[39] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. The Honey Badger of BFT Protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS ’16)*. ACM, New York, NY, USA, 31–42. https://doi.org/10.1145/2976749.2978399

[40] Satoshi Nakamoto et al. 2008. Bitcoin: A peer-to-peer electronic cash system. (2008).

[41] John K. Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen M. Rumble, Ryan Stutsman, and Stephen Yang. 2015. The RAMCloud Storage System. *ACM Trans. Comput. Syst.* 33, 3 (2015), 7:1–7:55. https://doi.org/10.1145/2806887

[42] Charalampos Papamanthou, Elaine Shi, Roberto Tamassia, and Ke Yi. 2013. Streaming Authenticated Data Structures. In *Advances in Cryptology – EUROCRYPT 2013*, Thomas Johansson and Phong Q. Nguyen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 353–370.

[43] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. 2010. Optimal Authenticated Data Structures with Multilinear Forms. In *Pairing-Based Cryptography - Pairing 2010*, Marc Joye, Atsuko Miyaji, and Akira Otsuka (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 246–264.
[44] Péter Szilágyi. 2018. Currently the primary bottleneck on go-ethereum is disk IO. https://github.com/ethereum/go-ethereum/pull/15857.

[45] Péter Szilágyi. 2018. Efficient Garbage Collection to reduce state on disk and increase the block processing rate. https://github.com/ethereum/go-ethereum/pull/16810.

[46] Xiaoyao Qian. 2018. Improved authenticated data structures for blockchain synchronization. Ph.D. Dissertation.

[47] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP ’17). Shanghai, China.

[48] Pandian Raju, Soujanya Ponnapalli, Evan Kaminsky, Gilad Oved, Zachary Keener, Vijay Chidambaram, and Ittai Abraham. 2018. mLSM: MakingAuthenticated Storage Faster in Ethereum. In 10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18). USENIX Association, Boston, MA.

[49] Vitalik Buterin. 2014. Toward a 12-second Block Time. https://blog.ethereum.org/2014/07/11/toward-a-12-second-block-time/.

[50] Marko Vukolić. 2015. The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication. In International workshop on open problems in network security. Springer, 112–125.

[51] Michael Wei, Amy Tai, Christopher J Rossbach, Ittai Abraham, Maithem Munshed, Medhavi Dhawan, Jim Stabile, Udi Wieder, Scott Fritchie, Steven Swanson, et al. 2017. vcorfu: A cloud-scale object store on a shared log. In 14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17). 35–49.

[52] Jeffrey Wilcke. 2016. The Ethereum network is currently undergoing a DoS attack. https://ethereum.github.io/blog/2016/09/22/ethereum-network-currently-undergoing-dos-attack/.

[53] Gavin Wood et al. 2019. Ethereum: A secure decentralised generalised transaction ledger. (2019).

[54] Renlord Yang, Toby Murray, Paul Rimba, and Udaya Parampalli. 2019. Empirically Analyzing Ethereum’s Gas Mechanism. 310–319. https://doi.org/10.1109/EuroSPW.2019.00041

[55] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. 2018. HotStuff: BFT Consensus in the Lens of Blockchain. arXiv preprint arXiv:1803.05069 (2018).

[56] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. 2018. RAPIDCHAIN: Scaling Blockchain via Full Sharding. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS ’18). ACM, New York, NY, USA, 931–948. https://doi.org/10.1145/3243734.3243853

[57] Tao Zhu, Zhuoyue Zhao, Feifei Li, Weinig Qian, Aoying Zhou, Dong Xie, Ryan Stutsman, Haining Li, and Huiqi Hu. 2018. Solar: towards a shared-everything database on distributed log-structured storage. In 2018 {USENIX} Annual Technical Conference ({USENIX} {ATC} ’18). 795–807.