# mLSM: Making Authenticated Storage Faster in Ethereum

Pandian Raju[1] Soujanya Ponnapalli[1] Evan Kaminsky[1]
Gilad Oved[1] Zachary Keener[1] Vijay Chidambaram[1,2] Ittai Abraham[2]
[1]*University of Texas at Austin* [2]*VMware Research*

## Abstract

Ethereum provides authenticated storage: each read returns a value and a proof that allows the client to verify the value returned is correct. We experimentally show that such authentication leads to high read and write amplification (64$\times$ in the worst case). We present a novel data structure, Merkelized LSM (mLSM), that significantly reduces the read and write amplification while still allowing client verification of reads. mLSM significantly increases the performance of the storage subsystem in Ethereum, thereby increasing the performance of a wide range of Ethereum applications.

## 1 Introduction

Modern crypto-currencies such as Bitcoin [21] and Ethereum [26] seek to provide a decentralized, cryptographically-secure currency built upon the blockchain [25]. Ethereum provides a distributed platform on which decentralized applications can be built. While ethereum's cryptocurrency *ether* is widely used and adopted, having the notion of a platform opens the opportunities for a wide range of decentralized applications apart from just crypto-currencies. Ethereum has a market cap of 64 billion dollars at the time of writing this paper, with over 1.5 billion dollars being traded over the last 24 hours. Thus, improving the performance of Ethereum will have significant impact.

In this work, we view Ethereum as a distributed storage system, and seek to improve the performance of its reads and writes. Ethereum is especially interesting because it provides *authenticated storage*. Reads in Ethereum return both the value and a *Merkle proof* that the value is indeed the correct value. Each key being read is part of a Merkle tree [19]; the inner nodes of the tree are simply hashes of their children. The proof consists of the hashes of each node along the path from the root to the leaf containing the key. The root is globally published, and thus any client receiving a value for a key can independently determine that the value it received is correct. Each write changes the hashes of all nodes along the path from the root to the leaf. Thus, reads and writes are expensive operations in Ethereum.

We examine the overhead of Ethereum reads using carefully designed experiments based on real-world public Ethereum blockchain data. Ethereum stores its data in the LevelDB [15] key-value store. We show that reading a single key (*e.g.,* the amount of ether in a given account) can result in 64 LevelDB reads, while writing a single key can lead to a similar number of LevelDB writes. Internally, LevelDB induces extra write amplification [23], further increasing overall amplification. Such write and read amplification reduces throughput (storage bandwidth is wasted by the amplification), and write amplification in particular significantly reduces the lifetime of devices such as Solid State Drives (SSDs) which wear out after a limited number of write cycles [1, 16, 20]. Thus, reducing the read and write amplification can both increase Ethereum throughput and reduce hardware replacement costs.

We trace the read and write amplification in Ethereum to the fact that it provides authenticated storage. The challenge then becomes: *is it possible to design an authenticated storage system which minimizes read and write amplification?* To overcome this challenge, we design a novel data structure called *Merkelized LSM* (mLSM) which combines merkle trees and log-structured merge trees [22], resulting in a write-optimized data structure for authenticated reads and writes. The key insights behind mLSM are maintaining multiple independent tries and decoupling the two current uses of tries in Ethereum: lookup and authentication. Similar to a log-structured merge tree, mLSM has multiple levels. Each level contains a number of immutable merkle trees, with merkle trees becoming bigger with higher-numbered levels. Updates to mLSM are batched in memory, and written to Level 0 as a new merkle tree. When each level has a threshold number of merkle trees, they are compacted into a single merkle tree in the next level. mLSM introduces several challenges unique to authenticated storage, such as the need for deterministic compaction.

Log-structured merge trees have been widely used in key-value stores [2, 3, 7, 13–15, 18, 23, 24, 27]; mLSM demonstrates a novel application of the ideas behind this data structure in a new domain. By adopting such ideas into Ethereum, we hope to increase the performance of reads and writes, and therefore increase the performance of multiple Ethereum applications. The ideas behind mLSM are not limited to Ethereum, and are broadly applicable to any authenticated storage.

## 2 Background

This section provides some background on blockchain and Ethereum. It first provides a high level overview of the blockchain technology on which Ethereum is built (§2.1). Then, it describes the Merkle patricia tree (§2.2) and how it is used as an authenticated data structure in blockchain. Finally, it provides an overview of how Ethereum works (§2.3).

### 2.1 Blockchain

A blockchain is a continuously growing list of blocks which are chained together. Each block consists of a set of transactions which happened during some particular time interval. A blockchain is also cryptographically secured using authenticated data structures such that the data cannot be silently altered. Each block is linked to the previous block by a cryptographic hash. We arrive at the current state of the system by processing transactions in order from the first block to the latest block.

The blockchain data (list of blocks and transactions) is distributed across thousands of nodes across the world. This is a powerful property because the data is no longer controlled by any single authority. There are consensus protocols for different nodes to agree upon the blocks that should be added to the chain. To keep the data cryptographically secure, hashes are extensively used in the blockchain to store values. For example, every block, every transaction, every account *etc.* is identified by a unique hash.

The process of adding a new block to the blockchain is called mining. Mining requires solving a cryptographically complex puzzle, commonly called the proof of work algorithm [21]. Other nodes on the network will validate the block that gets added. Typically multiple miners compete to mine a block at any point in time and the miner who succeeds is awarded some amount of cryptocurrency (as a reward for adding data to the blockchain). Some crypto-currencies also use a proof-of-stake algorithm [17] to verify the blocks.

### 2.2 Merkle tree

The data in a blockchain is securely stored using a Merkle tree [19]. A merkle tree is a tree where every parent stores the combined hash of all of its children; the values are stored in the leaves. Each node contains a list of the hashes of its children. This also means that if the value of any node *N* in the tree changes, the hash of every node in the path from that node *N* to the root of the tree will change. This allows us to detect if any piece of data in the tree has been modified in which case the root hash will not match. The root of the tree is publicly available to clients, and clients can use this to verify reads.

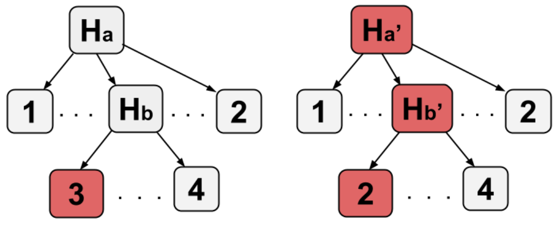

Figure 1: **Update in Merkle tree**. *This figure shows an update in a merkle tree where leaf nodes are represented by numbers. The non-leaf nodes $H_a$, $H_b$, $H_a$' and $H_b$', store the hash of their children. When a value 3 in the merkle tree is updated to 2, the hash of its parent node changes from $H_b$ to $H_b$' and so the hash of the root node from $H_a$ to $H_a$'. Any update to a node changes the hashes of all nodes in the path from that node to the root.*

### 2.3 Ethereum protocol

The Ethereum nodes connect to each other through remote procedure calls (RPC) to exchange block information. Each block header consists of a monotonously increasing unique block number and a cryptographic hash of the previous block in the chain (this chains the blocks together). Ethereum uses Keccak-256 [4] for generating cryptographic hashes. It uses a Merkle Patricia tree (§3.1) to store the data. There are two local tries (data from each block) and a global state trie. The global state trie tracks the state of each account, and stores details such as account id and balance.

**ETH and LES protocols**. There are different types of nodes in an ethereum network - `fullNode`, `fastNode` and `lightNode`. A `fullNode` downloads the entire history of blockchain from the beginning while a `lightNode` downloads only the block headers and gets the block bodies on demand from a `fullNode` through RPC calls. Ethereum runs ETH protocol between two `fullNode` in the cluster, and LES protocol [9] to manage the interaction between a `lightNode` and a `fullNode` in the cluster. A `fullNode` returns a merkle proof along with the requested data to a `lightNode`. A merkle proof is simply the list of hash values in the tree on the path from the requested value node to the root node. A `lightNode` can verify if the data is cryptographically correct or not using the merkle proof.

## 3 Storage in Ethereum

LSM-based key-value stores like LevelDB [15], RocksDB [13], and PebblesDB [23] provide high write throughput and reasonable read throughput. They excel at random reads and writes. Ethereum uses cryptographic hashes as identifiers for its data; hence it uses LevelDB as its data store, storing the hash as the key, and the data as the value. Ethereum uses Recursive Length Prefix encoding [12] to encode its data.

### 3.1 Merkle Patricia Trie

Ethereum uses a Merkle Patricia trie [11] to store the data, where the keys are stored in hexadecimal format. A merkle patricia trie in Ethereum consists of four types of nodes; `fullNode`, which branches out into 17 child nodes (16 children one each for each character of hex and 1 child if there's a value terminating at that node); shortNode, an optimization which uses a prefix as the key if there's no branching at that prefix; hashNode, which is used to convert a hash to a node doing a LevelDB lookup; valueNode, which is the leaf node which contains the value stored for that key. Ethereum trie maintains both in-memory trie and on-disk trie. On doing a commit, the in-memory trie is committed to the disk by writing key-value pairs to LevelDB, by introducing hash nodes.

### 3.2 LevelDB

Ethereum uses LevelDB extensively to store all of its data. For example, a key would be the hash of a block header, and the value would be the encoded value of the block header. Transaction data is also stored as key-value pairs with the transaction ID as the key. Ethereum uses a number of merkle patricia tries to store state; each node of the trie is also stored in LevelDB. As a result, Ethereum does a significant number of reads and writes to LevelDB during normal operation.

### 3.3 State trie

The State trie is a global trie which contains the state of all the accounts in the public network. Each account id in Ethereum is a randomly generated 20 bytes address. Ethereum uses the 256-bit hash of the account ID to store its data: the hash becomes the path through the patricia trie, with the data at the end of the path. Each node in the patricia trie is 4 bits; thus, resolving 256 bits requires traversing 64 nodes (each of which requires reading a LevelDB key) to get to the data. To improve performance, Ethereum uses a `shortNode` in the patricia trie to compact many nodes along a path into a single node. Even with this optimization, reading a single value in Ethereum requires tens of LevelDB reads.

## 4 The Ethereum Storage Bottleneck

Ethereum's authenticated storage has an IO amplification problem: every Ethereum read requires a large number of LevelDB reads, and every Ethereum write requires multiple LevelDB writes. Attackers have capitalized on this storage bottleneck to mount a DOS attack on the public network [8].

The root cause of the storage bottleneck is the design of the authenticated storage. Just building a patricia trie on top of LevelDB leads to significant amplification. In the worst case, an Ethereum read can lead to 64 LevelDB `get()` requests, which internally can each lead to multiple storage reads.

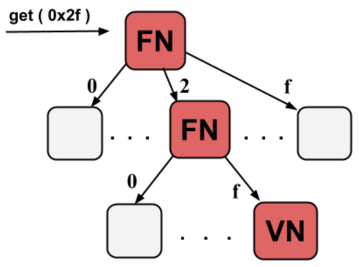

Figure 2: **Extensive use of LevelDB**. *This figure shows a single account value lookup. The account is encoded as a byte (here, 0x2f), the lookup on the account translates to 2 LevelDB lookups, one at each FullNode (FN) before reaching the ValueNode (VN). Thus, for accounts encoded using keccak 256 hash (32 bytes), a lookup on the value of the account translates to 64 LevelDB lookups.*

**Experimental setup**. We started a private ethereum network in which we imported the initial 1.6M blocks of the real-world public Ethereum blockchain [6]. We used the go Ethereum client (`geth`) [10] to start a `fullNode` that imports the blockchain data. We ran this setup on a machine with 16 GB of RAM and the block chain data was stored on a 2TB Intel 750 series SSD (using a software raid0). The following section provides some preliminary results on the number of storage reads.

**Results**. Our experiment processes all 1.6M blocks, parsing the transactions in each block, extracting the unique account IDs and obtaining the balance for each account. We augmented internal metrics provided by `geth` with our own to obtain the following results:

**getBlock:** `getBlock(blockNum)` returns the block body of the block with the number `blockNum`. We found that it resulted in roughly 8M LevelDB get calls for 1.6M blocks (around 5 LevelDB gets per block).

**getTransaction:** `getTx(txHash)` returns the details of the transaction with the hash `txHash`. Retrieving 5.2M transactions resulted in 10.4M LevelDB gets (2 LevelDB gets per transaction). The number of LevelDB gets per transaction is low because the transactions are retrieved from local transaction tries rather than global state trie.

**getBalance:** `getBalance(addr)` returns the amount of ether balance present in the account `addr`. The experiment resulted in around 1.4M LevelDB gets for 0.22M account addresses. The observed amplification is 7$\times$ instead of the worst-case 64$\times$ as the total amount of data is small, leading to a depth of only seven in the patricia trie. As the amount of data in the blockchain increases, the depth of the patricia trie will also increase, leading to higher amplification.

**Interaction with light nodes**. An Ethereum `lightNode` sends data request to a `fullNode` and the `fullNode` per-

| Metric | Value | Metric | Value |
|---|---|---|---|
| **# blocks** | 1.6M | **# account lookups** | 0.22M |
| **# transactions** | 5.2M | **# leveldb gets** | 1.4M (7×) |
| **# accounts** | 0.22M | **depth of state trie** | 7 |
| **Total time** | 562 s | **Time in storage** | 527 s |

Table 1: **Preliminary metrics**. *The table shows some metrics from the Ethereum public main network for the first 1.6M blocks. Looking up ether balance for 0.22M accounts results in 1.4M leveldb gets, with a read amplification of 7×. 93.8% of the overall time taken is spent in the storage layer.*

forms the IO to retrieve the data, and sends it back to the `lightNode` along with the merkle proof. A `fullNode` that is responding to many `lightNode` requests will be under significant IO pressure.

**Latency metrics.** To gauge the influence of storage layer in the overall latency of normal operations, we measure the overall time taken to read 1.6M block details, and count the number of transactions and accounts within those blocks, and the corresponding amount of time spent in the storage layer. Table 1 shows the numbers. For this experiment, we by-passed the javascript console provided by ethereum in order to eliminate the overhead of RPC calls, and we directly interact with the geth client (with the golang layer). We see that around 96.2% of the time is spent in the storage layer (to read from LevelDB).

Note that we have analyzed the worst case where sustained reads are being serviced by a single node, and networking and RPC overhead are eliminated. In a real-world deployment, a node could experience similar stress on its storage if it is serving hundreds or thousands of light nodes which are each requesting blocks. This is possible for workloads such as analytics or auditing.

Thus from Table 1, we can see that Ethereum's authenticated storage design significantly amplifies (by 7×) the number of high-level reads and writes to LevelDB, reducing throughput and increasing latency. The problem will only get worse over time, as more and more data is added to the Ethereum blockchain.

## 5 Proposed solution: Merkelized LSM

The challenge is to design an authenticated storage system that reduces the IO amplification, yet allows reads to be authenticated. To address this challenge, we present a novel data structure called *Merkelized LSM* (mLSM).

mLSM combines techniques from merkle trees and log-structured merge trees. We will first describe a trivial solution to optimize the reads and then explain the shortcomings of the trivial approach. We will then explain the intuition behind mLSM and then discuss its design. We will then analyze mLSM and show that it improves storage performance.

### 5.1 Caching merkle proofs

The main problem is that each Ethereum read results in multiple LevelDB reads. A straightforward solution is to just cache the value and the merkle proof for each Ethereum read in LevelDB, and just use the cache to serve Ethereum reads. For a read-only workload, this works great; the problem arises with writes. Each write updates several nodes in the Patricia Trie, including the root. As a result, all the cached merkle proofs are invalidated. The entire cache being invalidated on every write makes this an impractical approach, and demonstrates the unique challenges of authenticated storage.

### 5.2 Merkelized LSM

**Insight**. The key insight behind mLSM follows directly from the shortcoming of the trivial solution. The trivial solution did not work because there was tight coupling between any two nodes in the tree (since all the nodes formed a single tree under a single root). So, mLSM strikes at that invariant to maintain *multiple independent tries* such that a change of value in one tree doesn't affect other trees. When a write happens, only the cached values of the affected tree have to be invalidated. Another key insight derived from the trivial solution is to *decouple lookup* of value from the trie itself. The main purpose of the trie is to provide authentication, but it is not necessarily the only source for lookups. Decoupling lookup and authentication can reduce the number of LevelDB gets during a lookup.

**Design**. We combine the Merkle Tree and Log-Structured Merge Trees data structures in Merkelized LSM (mLSM). The shortest way to describe mLSM is to take LSM and replace immutable sstables with immutable patricia trees. In mLSM, we envision multiple levels similar to log-structured merge trees. Each level will have multiple immutable patricia tries. mLSM also contains a lookup cache per level; looking up a key would return its value and its merkle proof. To connect all the patricia tries together (so that reads can be verified), mLSM introduces a master root node. The merkle proofs in each level do not contain the master root hash; this way, we make sure that the tries are independent of each other. mLSM replaces merkle patricia tries with static binary merkle trees (fan out of 2 instead of 16) since the binary trees balance data better.

**mLSM Writes**. Similar to log-structured merge trees, writes and updates are buffered in memory and then written to storage as a large batch. mLSM does not update the binary merkle trees in place; instead, it writes new binary merkle trees to Level 0. When the number of binary merkle trees in Level 0 reaches a certain threshold, they

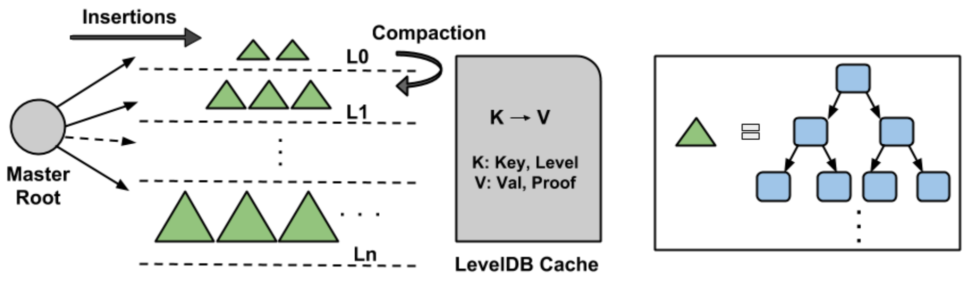

Figure 3: **Merklized LSM Design**. *mLSM has many binary merkle trees in every level (L0, L1, ..., Ln) and a LevelDB Cache, that maps a key in each level (key, level) to its value and the associated MerkleProof (value, proof). Insertions are performed to Level0 (L0) and the data in every level is periodically compacted to the higher indexed levels.*

are compacted (merged to form a larger binary merkle tree), and written as large batched I/O to the next level. Note that a write only affects the new merkle binary trees and the master root node; nothing else in the storage system is affected, avoiding one of the big problems in the trivial solution. The key to our solution is having multiple patricia trees that are independent of each other, except for the root node.

**mLSM Reads**. Similar to an LSM read, an mLSM read might require inspecting data in multiple levels. In Level 0, there might be multiple tries containing the key we are searching for; in all other levels, mLSM maintains the invariant that there is a single trie containing the required key. Once the right merkle binary tree is identified, it is queried to get the value and the merkle proof. The merkle proof is then extended to include the master node, and the value and the merkle proof are returned to the user.

**Caching**. mLSM caches both the values and authentication for each key. mLSM uses LevelDB for caching, with the key being the original key, and the value being a combination of the data and the merkle proof. mLSM reads inspect the different levels one by one starting with the lowest level and returning once the key is found in a level. There is a cache for each level in mLSM and reads which hit the cache do not need to traverse through the merkle tree to read the value and merkle proof. The Level 0 cache needs to handle multiple copies of the same key using something like a version number.

**Challenges**. Note that mLSM introduces read and write amplification of its own, due to the multiple levels. We propose to handle the read amplification with Bloom Filters [5]. Bloom filters can be used to optimize the lookups efficiently such that we only read the trie which can possibly contain the key. With such optimizations, on an average, we will read only one trie for a Get request and the value and merkle proof can be retrieved with a single LevelDB get from the corresponding cache.

The write amplification introduced by the mLSM structure is another challenge. We could potentially borrow ideas from our previous work, Fragmented Log-Structured Merge Trees [23], to reduce the write amplification while maintaining read performance.

Another subtle challenge is that since the mLSM structure is used to verify reads, different nodes should reflect the same mLSM state at any point in time. This means that the compaction of tries should be deterministic and should have the same behavior in all the nodes – this is not the case with LSM-based key-value stores today, where compaction is triggered in the background in a non-deterministic way. Deterministic compaction could possibly be enforced by using an opcode for compaction which forces compaction of the state rather than triggers from the thresholds. Other challenges include efficiently storing bloom filters in memory and limiting the number of level-0 tries.

**Cost Analysis**. When bloom filters are added to each trie in mLSM, each mLSM read will only need to read one trie with high probability. Thus, on a cache miss, the cost of the mLSM read will be $O(D)$ where $D$ is the depth of the merkle binary tree where the key is located. We believe this will be significantly smaller than the depth of the merkle patricia trie used in Ethereum today. For all consecutive reads of a key (during a cache hit), the cost of the read is $O(1)$ since it can be served with just one LevelDB lookup.

Each mLSM write will be buffered in memory, and written as an immutable merkle binary tree to Level 0. Each key will be written only once to each level, so the write cost will be $O(H)$ where $H$ is the height of mLSM. We believe this will also be significantly lower than the write cost in Ethereum today, which has to update all nodes on the path from the leaf to the root. Additionally, since the writes are buffered in memory and written as a batch, the write cost is amortized among all the keys.

## 6 Conclusion

We show that Ethereum suffers from read and write amplification due to its authenticated storage system. We present a novel data structure, Merkelized LSM (mLSM), that lowers read and write amplification while still providing authenticated reads. mLSM is applicable to any authenticated storage. We plan to modify Ethereum to introduce the mLSM structure and evaluate the potential increase in storage performance.

## References

[1] Agrawal, N., Prabhakaran, V., Wobber, T., Davis, J. D., Manasse, M. S., and Panigrahy, R. Design Tradeoffs for SSD Performance. In *Proceedings of the 2008 USENIX Annual Technical Conference* (2008), pp. 57–70.

[2] Ahn, J.-S., Seo, C., Mayuram, R., Yaseen, R., Kim, J.-S., and Maeng, S. ForestDB: A Fast Key-Value Storage System for Variable-Length String Keys. *IEEE Transactions on Computers 65*, 3 (2016), 902–915.

[3] Balmau, O., Didona, D., Guerraoui, R., Zwaenepoel, W., Yuan, H., Arora, A., Gupta, K., and Konka, P. TRIAD: Creating synergies between memory, disk and log in log structured key-value stores. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC 17)* (Santa Clara, CA, 2017), pp. 363–375.

[4] Bertoni, G., Daemen, J., Peeters, M., and Assche, G. V. The Road from Panama to Keccak via RadioGatún. In *Symmetric Cryptography, 11.01. - 16.01.2009* (2009), H. Handschuh, S. Lucks, B. Preneel, and P. Rogaway, Eds., vol. 09031 of *Dagstuhl Seminar Proceedings*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany.

[5] Bloom, B. H. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM 13*, 7 (1970), 422–426.

[6] Bloopish. Ethereum blockchain download. `http://bloopish.com/tools/ethereum/`, 2016.

[7] Dayan, N., Athanassoulis, M., and Idreos, S. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017* (2017), S. Salihoglu, W. Zhou, R. Chirkova, J. Yang, and D. Suciu, Eds., ACM, pp. 79–94.

[8] Ethereum. The Ethereum network is currently undergoing a DoS attack. `https://blog.ethereum.org/2016/09/22/ethereum-network-currently-undergoing-dos-attack/`, 2016.

[9] Ethereum. Ethereum Light Client Protocol. `https://github.com/ethereum/wiki/wiki/Light-client-protocol`, 2017.

[10] Ethereum. Ethereum Golang client. `https://github.com/ethereum/go-ethereum`, 2018.

[11] Ethereum. Merkle Patricia Trie Specification. `https://github.com/ethereum/wiki/wiki/Patricia-Tree`, 2018.

[12] Ethereum. Recursive Length Prefix in Ethereum. `https://github.com/ethereum/wiki/wiki/RLP`, 2018.

[13] Facebook. RocksDB — A persistent key-value store. `http://rocksdb.org`, 2017.

[14] Golan-Gueta, G., Bortnikov, E., Hillel, E., and Keidar, I. Scaling concurrent log-structured data stores. In *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015* (2015), L. Réveillère, T. Harris, and M. Herlihy, Eds., ACM, pp. 32:1–32:14.

[15] Google. LevelDB. `https://github.com/google/leveldb`, 2018.

[16] Grupp, L. M., Caulfield, A. M., Coburn, J., Swanson, S., Yaakobi, E., Siegel, P. H., and Wolf, J. K. Characterizing Flash Memory: Anomalies, Observations, and Applications. In *Proceedings of 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42)* (2009), IEEE, pp. 24–33.

[17] King, S., and Nadal, S. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. *self-published paper, August 19* (2012).

[18] Lu, L., Pillai, T. S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST 16)* (2016), pp. 133–148.

[19] Merkle, R. C. A Digital Signature Based on a Conventional Encryption Function. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology* (London, UK, UK, 1988), CRYPTO '87, Springer-Verlag, pp. 369–378.

[20] Mielke, N., Marquart, T., Wu, N., Kessenich, J., Belgal, H., Schares, E., Trivedi, F., Goodness, E., and Nevill, L. R. Bit Error Rate in NAND Flash Memories. In *Proceedings of the IEEE International Reliability Physics Symposium, (IRPS 08)* (2008), IEEE, pp. 9–19.

[21] Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system.

[22] O'Neil, P., Cheng, E., Gawlick, D., and O'Neil, E. The log-structured merge-tree (LSM-tree). *Acta Informatica 33*, 4 (1996), 351–385.

[23] Raju, P., Kadekodi, R., Chidambaram, V., and Abraham, I. PebblesDB: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 497–514.

[24] Sears, R., and Ramakrishnan, R. bLSM: a General Purpose Log Structured Merge Tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (2012), ACM, pp. 217–228.

[25] Wikipedia. Blockchain. `https://en.wikipedia.org/wiki/Blockchain`, 2018.

[26] Wood, G. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper 151* (2014), 1–32.

[27] Wu, X., Xu, Y., Shao, Z., and Jiang, S. LSM-trie: An LSM-Tree-Based Ultra-Large Key-Value Store for Small Data Items. In *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC 15)* (2015), pp. 71–82.