Aadit Kapare, 2026, 14:5
ISSN (Online): 2348-4098
ISSN (Print): 2395-4752

International Journal of Science, Engineering and Technology
An Open Access Journal

Owner-Partitioned Merkle Trees: Hash-Cost Analysis and Benchmark Motivated by MerkleChain

Aadit Kapare[1], Rachit Singh[2], Tanish Venkatesh[3], Reshma Sonar[4]
Department of Computer Science and Technology, MIT World Peace University, Pune, India

Abstract – MerkleChain is a recently proposed owner-oriented blockchain architecture in which data owners keep their own separate Merkle trees and participating miners store only the structures and cryptographic commitments relevant to their subscriptions. The design aims to support privacy, controlled redaction, and parallel processing. This study examines a narrower question: does dividing a fixed number of records into owner-specific Merkle trees reduce the cryptographic work needed for construction, proof verification, and updates, if the owner roots must still be fed into a common root? A reproducible Python microbenchmark compares one large balanced global tree with sixteen balanced owner trees and a top-level tree of owner roots, at four data sizes from 256 to 16,384 records. The implementation uses SHA-256 with domain-separated inputs for leaves and internal nodes. In all tested power-of-two workloads, both arrangements require N − 1 internal-node hashes to build the full commitment structure, and the total proof path remains $\log_2(N)$ levels. Measured verification and update times are generally higher for the partitioned model at these small scales, while construction times stay nearly the same. These results do not refute MerkleChain's wider claims about privacy and parallelism, because the experiment includes no chameleon hashes, distributed consensus, or concurrent miners. They show instead that ownership partitioning is a systems-design decision and should not be assumed to reduce hash complexity

Keywords— Blockchain, chameleon hash, data partitioning, integrity proof, Merkle tree, redactable blockchain, reproducible benchmarking.

I. INTRODUCTION

The way that blockchain systems work is to make records verifiable by linking cryptographic commitments together over a distributed ledger; this approach is valuable since any unnoticeable alteration should not appear to be the original record. However, it can be hard to handle permanent data that has been widely replicated when an entry is wrong, includes information that should not be made public, or is owned by someone who needs more control over whether it is disclosed. Research into redactable blockchains deals with this dilemma by permitting carefully authorised changes while at the same time keeping evidence that the ledger adheres to a specified protocol [1]–[3].

MerkleChain, developed by Mukkamala, Olariu and Kalari [1], makes data ownership a central feature of its architecture; its published abstract describes owner-controlled Merkle trees, chameleon hashes, selective miner participation and the parallel processing of separate trees, with healthcare as the example application. The design poses a practical question beyond the cryptographic techniques themselves: what actual effect does splitting the data in a ledger into independent trees specific to each owner have on the amount of work required to verify or update a record?

This paper addresses that question by means of a deliberately limited experiment. It does not attempt to reproduce the MerkleChain implementation, since the complete text of that implementation was not available to the authors when carrying out this study. Rather, it focuses on one feature which can be examined on its own: a fixed set of records is committed either using a single balanced Merkle tree or by means of multiple owner trees whose roots are combined in a top-level Merkle tree. This comparison separates out the cost associated with the tree structure from the other individual issues of redaction policy, trapdoor management, network consensus, and privacy leakage.
The key point is straightforward yet easy to miss: if both the number of records and the number of

© 2026 Aadit Kapare his is an Open Access article distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/4.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly credited.

owner partitions are powers of two, then dividing the entire tree into owner trees and adding a root tree does not decrease the total number of internal-node hashes needed to construct the commitment. Similarly, the combined depth of the local proof path and the owner-root path is equal to that of a global proof path. Although partitioning can still enable access separation, allow parallel processing, and offer more flexible participation, these advantages are not the same as having a lower asymptotic hash count. The experiment therefore views partitioning as an architectural compromise whose value will vary depending on the workload and the system-level properties.

The three aspects of the contribution are a transparent model of global and owner-partitioned Merkle trees, a reproducible benchmark that includes the construction phase, proof verification, and updates, and an analytical explanation of why identical tree depth does not ensure equivalent measured runtime. The remainder of the paper covers the cryptographic background, defines the comparison, presents the measurements, and examines what the results can and cannot establish.

II. Technical Background

1. Merkle Trees as Commitments

A Merkle tree calculates a hash for each leaf node and combines pairs of the child hashes to generate the parent hashes; the hash at the top of the tree is known as the Merkle root. If a trusted Merkle root is available, a verifier can determine whether a particular record is part of the committed set by reworking the path of sibling hashes rather than having to download all the records. This kind of logarithmic proof mechanism is one of the reasons why Merkle trees are useful for authenticated data structures and for including transactions in a blockchain [4], [5].

Let x_i be the ith record and let H be a cryptographic hash function; in this experiment the inputs to the leaves and to the internal nodes are separated by a one-byte prefix. The leaf hash and parent hash are

$$L_i = H(0x4C \,||\, x_i) \quad (1)$$

$$P = H(0x4E \,||\, L \,||\, R) \quad (2)$$

where L and R are the hash values of the left and right children and || denotes concatenation of byte sequences. The prefixes differentiate leaf inputs from internal-node inputs. SHA-256 is used via Python's standard hashlib library; SHA-256 is part of the Secure Hash Standard issued by the National Institute of Standards and Technology (NIST) [4].

For a balanced binary tree with N leaves, where N is a power of two, the number of internal nodes and the proof depth are

$$\text{Internal-node count} = N - 1 \quad (3)$$

$$d = \log_2(N) \quad (4)$$

where $\log_2(N)$ is the base-two logarithm of N; for instance, a tree with 256 leaves has a proof depth of 8 since 2^8 is 256. When a single record is updated, the leaf hash is altered and the hashes along the path to the root are recomputed; the amount of work involved at the internal nodes is therefore proportional to $\log_2(N)$. This is the model employed here; actual implementations may vary with respect to memory layout, batching, serialization, network transfer, and hash-library performance.

2. Chameleon Hashes and Controlled Redaction

A conventional cryptographic hash is designed such that it is infeasible to find a different input that has the same digest. A chameleon hash is a similar construction, but allows collision-finding for an entity that possesses trapdoor information. Redactable blockchain systems utilise variations of this idea to allow certain content to be replaced without invalidating every downstream commitment in the same way that a conventional hash link would [2], [3].

The ability to edit, however, results in a governance problem: a system must specify who can authorise such a change, how misuse is detected, and how participants determine which version is legitimate.

Recent work demonstrates that redaction is not a single operation. Huang et al. describe a dynamically redactable blockchain using decentralised chameleon hashing and a consistency-check mechanism [3]. Other approaches incorporate attribute-based permissions, committees, or traceability to constrain editing authority [6]. These designs differ in their assumptions and protocols, so

a tree-structure benchmark cannot substitute for their security analysis.

3. MerkleChain's Architectural Motivation

The published MerkleChain abstract describes a system where miners may hold Merkle trees and chameleon hashes for data to which they subscribe, rather than every miner storing all owner data. It also identifies parallel processing of owner-controlled trees as a route to higher throughput [1].

This suggests two distinct dimensions: a data-organisation dimension, where commitments are partitioned by owner, and a cryptographic-authorisation dimension, where controlled changes are permitted. The experiment below examines only the first dimension.

The distinction is important because a smaller local tree is not the whole proof if its root must be authenticated against a common commitment. A verifier who receives a proof for a record in one owner's tree must also establish that the owner root is included in the top-level commitment. The local path is shorter, but the top-level path adds the missing levels. Counting only the local path would overstate the savings.

Related designs help to clarify why the distinction matters. Conventional blockchain commitments generally prioritise stable, globally verifiable history; redactable systems introduce an authorised mechanism for revising selected content. Chameleon-hash approaches differ in how they distribute the trapdoor, define editing permissions, record revision history, and allow other nodes to validate the resulting state. Huang et al. emphasise decentralised redaction and consistency checking, while PRBC introduces attribute-based control and identity tracing [3], [6]. MerkleChain's owner-oriented tree organisation addresses a different but complementary question: which data structures and commitments a participant needs to maintain.

These dimensions should not be collapsed into one performance claim. A system may improve selective participation without reducing the number of hashes in a complete proof, and it may provide controlled editing without guaranteeing that every participant sees only non-sensitive metadata.

III. Research Question and Analytical Model

For a fixed set of N records belonging to K owners, each having N/K records, does the two-level owner-partitioned Merkle structure reduce the work or depth of proof required to construct, verify, and perform a single-record update, assuming all the owner roots are authenticated by a single top-level tree?

The baseline is a single balanced Merkle tree containing all N records. The partitioned arrangement is K balanced owner trees, each with N/K records, and a top-level Merkle tree whose leaves commit to the K owner roots. Both arrangements use the same leaf and internal-node functions. N is tested at 256, 1,024, 4,096, and 16,384; K is 16. All values are powers of two so that both arrangements have completely balanced trees and there is no ambiguity about partitioning odd-numbered sets.

1. Complexity of Construction

For the global tree, the count of internal-node hashes is

$$H_{\text{global}} = N - 1 \tag{5}$$

For the partitioned arrangement, the owner trees require

$$H_{\text{owners}} = K \times (N/K - 1) = N - K \tag{6}$$

and the top-level tree requires

$$H_{\text{top}} = K - 1 \tag{7}$$

The sum of the two is

$$H_{\text{partitioned}} = (N - K) + (K - 1) = N - 1 \tag{8}$$

So the two arrangements have the same count of internal-node construction hashes under the given assumptions. The partitioned arrangement hashes the K owner roots as leaves of the top-level tree, so it performs K more leaf-hash operations than the single-tree arrangement.

2. Proof Depth

The global proof depth is

$$d_{\text{global}} = \log_2(N) \tag{9}$$

The partitioned proof includes $\log_2(N/K)$ sibling hashes inside the owner tree and $\log_2(K)$ sibling hashes in the owner-root tree, resulting in

$$d_{\text{partitioned}} = \log_2(N/K) + \log_2(K) = \log_2(N) \quad (10)$$

The combined number of internal-node proof steps is equal. The partitioned verifier also hashes the owner root as a leaf of the top-level tree. Similarly, one record's update recomputes $\log_2(N/K)$ local-level hashes and $\log_2(K)$ top-level hashes, for $\log_2(N)$ internal-node recomputations, plus an additional owner-root leaf hash.

These equations give the deliberately limited hypothesis that partitioning alone will not reduce asymptotic tree-hash work when all owner roots are committed in a balanced top-level tree. The runtime may still vary due to the additional object processing, function call overhead, cache effects, or implementation detail. Parallel execution could change the result, but the current benchmark is sequential and does not test parallel throughput.

3. Trust Assumptions

The global-tree baseline assumes a verifier has an authenticated root and a record with a valid proof. The partitioned model assumes the verifier also has an authenticated top-level root and knows the owner tree in which the record lies.

The experiment does not say how either root becomes trusted; in a deployed blockchain this would involve block validation, consensus, signatures, and other protocol rules. Nor does a matching root prove that a record is a factually correct value. It only provides evidence that the record matches the committed values under the hash assumptions. The model thus evaluates the integrity-path cost, not the truth, confidentiality, or legitimacy of a redaction.

IV. Methodology

1. Implementation

All of the Merkle trees stored a list of values for each level, with the leaves at level zero and the root at the final level. If there was an odd number of nodes at a level, the final node was paired with itself. The reported power-of-two experiments do not require this fallback, but it ensured the implementation was defined for other inputs.The global case built one tree from the entire record list. The partitioned case divided the same ordered list into 16 equal chunks, built one tree for each chunk, and built a top-level tree from the owner roots.

A proof was verified by hashing the selected record, applying each sibling hash in the correct left/right order, hashing the resulting owner root as a top-level leaf, and applying the top-level siblings. An update replaced a leaf's value and recalculated only the ancestors along the affected path. In the partitioned case, the corresponding owner root was then updated in the top-level tree.

2. Workloads and Timing

For each record count, construction was measured 7 times and the median duration was reported. Verification latency was estimated by repeating the same valid proof 2,000 times for each measurement and taking the median of 5 measurement batches. Update latency was estimated from 200 deterministic, pseudo-random record updates per run, with 7 runs and the median reported. The same record count, owner count, and SHA-256 implementation were used in both arrangements.

The reported experimental environment was Python 3.13.5 under Linux on an Intel Xeon Platinum 8573C host. These timings are microbenchmark observations from one environment and should not be interpreted as universal performance claims. The machine's broader workload, interpreter state, operating-system scheduling, and measurement noise could affect small differences. Consequently, the third decimal place in milliseconds should not be interpreted as a stable system characteristic.

To reduce avoidable variation, the data-generation procedure and update indices were deterministic, the same workload was used for both structures, and medians rather than single timings were reported. Verification used a fixed valid proof so that the comparison measured verification work rather than proof-generation work. The benchmark did not intentionally interleave unrelated tasks, but the host

was not a dedicated real-time test machine. The code and CSV results are intended as companion materials so that the experiment can be rerun and extended.

3. Measures and Scope

- Tree-construction time in milliseconds.
- Proof-verification time in microseconds.
- Update time in microseconds.
- Analytical counts of internal-node hashes and proof levels.

The benchmark did not consider network traffic, persistent storage, miner participation, cryptographic trapdoor operations, authorisation checks, redaction auditing, or concurrent execution. Specifically, it does not test the privacy properties or the full MerkleChain protocol. It isolates the computational impact of the Merkle-tree partitioning model.

V. Results

1. Construction Cost

Table I shows the median construction times. The global and partitioned implementations were close at the tested sizes: the partitioned implementation was slightly slower at 256, 4,096, and 16,384 records and slightly faster at 1,024 records. The differences were small enough that the measurements do not demonstrate a strong or general construction-time advantage for either arrangement.

TABLE I
MEDIAN TREE-CONSTRUCTION TIME

Records (*N*)	Owners (*K*)	Global tree (ms)	Partitioned trees (ms)
256	16	0.230	0.256
1,024	16	1.000	0.985
4,096	16	4.015	4.074
16,384	16	16.801	17.368

The analytical hash count explains the closeness of construction times: each arrangement performs $N - 1$ internal-node hashes. The partitioned design redistributes that work across K owner trees and a top-level tree rather than eliminating it, and it adds K leaf hashes for the owner roots in the top-level tree. As N increases, the extra K leaf hashes are a smaller proportion of the overall leaf processing, but the timing was still subject to Python object creation and list management.

2. Verification and Update Cost

Table II shows the measured verification and update latencies. The partitioned implementation had higher verification and update latencies across all four workloads, but the differences were small in absolute terms, and they do not suggest a sequential latency advantage for partitioning in this implementation. The additional owner-root hashing and the separate tree structures are plausible contributors, but the experiment does not separate the impact of these factors.

TABLE II
MEDIAN VERIFICATION AND UPDATE LATENCY

Records (*N*)	Global verif. (µs)	Partitioned verif. (µs)	Global update (µs)	Partitioned update (µs)
256	5.844	6.444	6.601	7.714
1,024	7.895	8.413	9.572	10.959
4,096	9.224	10.218	11.203	13.144
16,384	10.275	11.796	14.135	15.597

The proof depth did not decrease: 256 records have a global proof with eight internal-node steps, while the partitioned proof has four steps inside the owner tree and four steps in the owner-root tree. At 16,384 records, the split is ten local levels plus four top-level levels, again totalling fourteen. The same applies to updates: local path recomputation plus top-level recomputation equals the global path depth.

C. Analytical Hash Counts

The analytical counts in Table III are shown separately from the measured timing values. They are consequences of the balanced-tree model, not estimates based on the timing results.

TABLE III
INTERNAL-NODE CONSTRUCTION HASHES AND PROOF-PATH STEPS

Records (N)	Global internal hashes	Partitioned internal hashes	Global proof steps	Partitioned proof steps
256	255	255	8	8
1,024	1,023	1,023	10	10
4,096	4,095	4,095	12	12
16,384	16,383	16,383	14	14

The equality of counts does not imply that the two implementations will take the same time: the partitioned implementation must additionally process the owner root as a leaf of the top-level tree, and it uses separate tree structures. These overheads can affect measured runtime even when the number of internal-node steps is the same.

4. Reading the Results Correctly

The measurements do not show that owner partitioning is a bad architecture. They show that a claim of lower hash complexity cannot be supported by the size of the owner tree alone. The local proof is shorter, but a verifier that must authenticate the owner root must also process the upper path. The value of partitioning may lie in selective storage, access control, independent scheduling, or parallel processing of updates for different owners. Those benefits would require measurements and security analysis at the full-system level.

VI. DISCUSSION

When evaluating data structures it is common to examine the cost of an operation within a single component, and for owner-partitioned Merkle trees this can give the impression that the local tree is cheaper than the global tree since it contains N/K leaves instead of N. However, a record proof on a shared commitment is not complete until the owner root has been authenticated, and the full cost is then the local path plus the aggregation path. For balanced binary trees these path lengths add back up to the original global depth.

The resulting principle is familiar from systems design: shifting a cost across a boundary is not the same as eliminating it. A two-level design could improve organisational separation while maintaining the same number of cryptographic path steps. It could still be beneficial if different owners' data is processed independently, if miners subscribe to only selected data, or if privacy policies forbid unnecessary data distribution. These properties are architectural and operational, and cannot be derived from a single-threaded timing table.

The results also bear on the throughput-latency distinction. If owner trees can be processed concurrently, total completed operations per second could increase even if a single proof or update requires slightly more work. A proper follow-up experiment would vary the number of workers, the distribution of updates among owners, and the contention at the top-level root. If every update requires synchronous rewriting of a shared root, this shared commitment could become a coordination point; if root updates can be batched or otherwise coordinated safely, the trade-off would be different. This study does not evaluate those protocol designs. It is important to separate privacy from the size of the data structure. Storing fewer owner trees could reduce the amount of underlying data accessible to a miner, but hashes and metadata can still reveal relationships, update frequency, and membership information depending on the surrounding protocol. Likewise, chameleon-hash permissioning requires governance and auditability. The literature on decentralised and attribute-controlled redactable blockchains treats these as distinct security concerns [3], [6]. The speed of a Merkle proof does not establish confidentiality, and a controlled redaction mechanism does not establish that the remaining data is private.

For a healthcare deployment the distinction would affect the system's design decisions. Partitioning into owner trees could allow easier restriction of which hospital's records each participating node maintains, but confidentiality at the record level would depend on encryption, access control, key management, and the metadata which the network exposes. Redaction would require a policy which defined valid reasons

for modification, which actors were permitted to approve it, and what evidence is retained for auditing. If multiple owners update records at the same time, the implementation would also need to define how their root changes are ordered or batched. Without these mechanisms, simple separation of the local tree is not sufficient for a privacy or governance model.

VII. Limitations and Future Work

This study has several limitations. First, it considers only balanced Merkle trees and record counts that are powers of two; the behaviour in the case of skewed owner sizes, empty owners, changing owners, or dynamically growing or shrinking trees could be different. Second, the implementation is written in Python and stores all levels in memory; it is not a production-grade blockchain implementation. Third, the benchmark runs on a single machine using sequential execution and cannot observe throughput benefits from parallelism.

Fourth, no chameleon-hash algorithm, trapdoor, authorisation policy, consensus protocol, miner subscription mechanism, or healthcare workflow was implemented; the results therefore must not be presented as a replication or validation of MerkleChain's full security or performance claims. Fifth, the timing values are short microbenchmarks; even with repeated measurements and median calculation, they can be affected by host load and interpreter overhead. The analytical hash-count result is more general than the timing results, but it depends on the stated balanced-tree and root-aggregation assumptions.

An additional limitation is the equal-partition assumption. Real owners rarely contribute identical numbers of records. If one owner has an extremely large tree and the others have small trees, the maximum local proof depth will be driven by the largest partition, while the top-level path still depends on K. An uneven workload can therefore change latency distributions even if the total internal-node count is roughly the same order. A useful extension would report per-owner percentiles and examine skewed distributions rather than only aggregate medians.

Future work would benefit from implementing and measuring a production-grade concurrent owner-tree construction and update-processing system, including realistic record sizes and measuring throughput, memory consumption, proof bytes, and network communication; comparing different root-commitment strategies; and evaluating the possibility of batching root updates without weakening verification. A separate security study would model the redaction authority, trapdoor compromise, authorised and unauthorised edits, audit trails, and the information exposed by miners' subscriptions. Finally, replication in a second implementation language and on a second hardware platform would help determine which timing differences are specific to Python.

VIII. Conclusion

This paper examined one aspect of MerkleChain's owner-oriented data organisation: whether dividing a fixed set of records into separate Merkle trees reduces the hash work needed to verify and update a record. Given sixteen equal owner trees and a top-level tree that commits to their roots, the answer is no: although the owner path is shorter, the root path restores the total proof depth to $\log_2(N)$. Internal-node construction work remains $N - 1$, and the internal-node work for a single update is $\log_2(N)$.

The Python measurements showed comparable construction times and consistently higher, though small in absolute terms, sequential verification and update times for the partitioned implementation in the described environment. These results are not an indictment of MerkleChain as a whole-architecture design. They suggest that its potential benefits should be evaluated in the dimensions where they are expected to arise: selective data storage, access boundaries, redaction governance, and concurrent processing. A careful analysis of blockchain design must account for the work along the entire verification path, not only the innermost components.

DATA AVAILABILITY

The benchmark code (Python 3, standard library only) and the CSV results reported in this paper are provided as supplementary material with this article.

REFERENCES

1. P. Vas, Parameter Estimation, Condition Monitoring, and Diagnosis of Electrical Machines. Oxford, U.K.: Clarendon Press, 1993.
2. W. T. Thomson and M. Fenger, "Current signature analysis to detect induction motor faults," IEEE Industry Applications Magazine, vol. 7, no. 4, pp. 26–34, 2001.
3. A. Bellini, F. Filippetti, C. Tassoni, and G. A. Capolino, "Advances in diagnostic techniques for induction machines," IEEE Transactions on Industrial Electronics, vol. 55, no. 12, pp. 4109–4126, 2008.
4. S. Nandi, H. A. Toliyat, and X. Li, "Condition monitoring and fault diagnosis of electrical motors—A review," IEEE Transactions on Energy Conversion, vol. 20, no. 4, pp. 719–729, 2005.
5. J. Travis and J. Kring, LabVIEW for Everyone: Graphical Programming Made Easy and Fun, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall, 2006.
6. National Instruments, NI-DAQmx User Manual. Austin, TX, USA: National Instruments.
7. National Instruments, LabVIEW User Manual. Austin, TX, USA: Na-tional Instruments.