MerkleAudit: A Merkle Tree-Based Blockchain Framework for Tamper-Evident Audit Logging in Digital Record Systems

Rashi Vakhare[1], Sakshi Chavan[2]
[1,2]*Department of Science and Mathematics, Deccan Education Society Pune University*
doi.org/10.64643/ijirt.208359-459

***Abstract*—In digital record-keeping systems, ensuring the integrity and authenticity of audit logs is critical for regulatory compliance, security assurance, and post-incident forensic analysis. Conventional centralized logging architectures remain vulnerable to tampering by insiders with administrative access and constitute single points of failure: an adversary who compromises the log store, or an operator who abuses legitimate access, can retroactively rewrite the historical record without leaving evidence of the change. Blockchain technology offers a compelling countermeasure, since data committed to a well-secured blockchain is computationally infeasible to alter retroactively. However, anchoring every individual log entry on-chain introduces prohibitive latency, storage overhead, and transaction cost, limiting its practicality for high-throughput logging environments such as enterprise transaction systems, hospital record systems, or academic record-keeping platforms.**

This paper presents MerkleAudit, a hybrid logging framework that combines Merkle tree-based authenticated data structures with blockchain anchoring to address these limitations. MerkleAudit batches log entries off-chain into Merkle trees and commits only the resulting root hash to the blockchain, decoupling log-ingestion throughput from on-chain transaction frequency and substantially improving scalability while preserving tamper-evidence. The framework further supports efficient, privacy-preserving inclusion proofs that allow verification of a single log entry without disclosing the full log set or requiring direct database access, and restricts log submission to an authorized, role-based allow-list, combining the auditability of public verification with the access control of a permissioned system. We implemented and evaluated a prototype of MerkleAudit on the Ethereum Sepolia test network, demonstrating that it can process high volumes of log entries with minimal on-chain storage footprint, rapid proof verification, and reliable detection of tampering, deletion, and forgery attempts. A gas-cost benchmark across batch sizes from 10 to 5,000 entries shows savings of up to 99.97% relative to a naive per-entry logging baseline, while a 25-test adversarial suite and an independent static-analysis pass confirm that no exploitable vulnerabilities were introduced by the batching design. MerkleAudit thus offers a practical, cost-efficient, and privacy-preserving approach to verifiable audit logging for security-critical digital record systems.

Index Terms*—blockchain, Merkle trees, consortium ledger, audit logging, tamper-evident storage, O (log N) inclusion proofs, data integrity, privacy-preserving verification, smart contract security

I. INTRODUCTION

Reliable audit logs are foundational to trust and accountability in digital systems. Financial institutions, healthcare providers, and educational institutions all depend on the assumption that a recorded log entry a financial transaction, a modification to a patient record, or a grade change faithfully represents what actually occurred. Regulatory regimes across sectors, from financial reporting to healthcare data handling, explicitly require that such records be traceable, attributable, and resistant to undetected alteration. In practice, this assumption does not always hold: when log storage is controlled by a centralized authority, that authority, or any attacker who compromises it, can silently alter or delete historical records without detection, undermining both regulatory compliance and post-incident forensic analysis.

Blockchain technology is frequently proposed as a solution, since data committed to a blockchain is computationally infeasible to alter retroactively without controlling a majority of the network's consensus power. However, naively writing every

individual log entry to a blockchain does not scale: transaction cost and storage overhead grow linearly with the number of log entries, making this approach prohibitively expensive for systems that generate large event volumes, such as enterprise systems that may produce thousands of loggable events per day. This tension between tamper-evidence and cost efficiency motivates the design of logging frameworks that preserve the security guarantees of blockchain anchoring while minimizing the volume of data written on-chain.

This paper introduces MerkleAudit, a framework that decouples the rate of log ingestion from the frequency of blockchain writes. Log entries are first collected off-chain and grouped into fixed-size batches. Each batch is organized into a Merkle tree, and only the resulting 32-byte root hash regardless of how many entries the batch contains is committed on-chain. Given a log entry and its corresponding Merkle proof, any party can verify, in time that scales logarithmically with batch size, that the entry was included in a specific batch and has not been altered since. Because only authorized submitters may anchor new batches, MerkleAudit additionally provides access control over who can extend the audit trail, while leaving verification open to any party a separation of write authority from read/verify authority that mirrors how audit systems are typically governed in regulated organizations.

This work makes three contributions: (1) the design and implementation of a Merkle-batch anchoring system that combines role-based, permissioned log submission with publicly verifiable blockchain anchoring; (2) an empirical evaluation that measures the on-chain cost of the system under realistic batch sizes and compares it against a naive per-entry logging baseline; and (3) a security evaluation combining adversarial testing covering entry tampering, non-membership, and cross-batch replay attacks with independent static analysis of the smart contract code.

The remainder of this paper is organized as follows. Section 2 reviews related work in authenticated data structures and blockchain-backed logging. Section 3 defines the threat model. Section 4 states the problem formally. Section 5 describes the MerkleAudit architecture. Section 6 details the implementation. Section 7 presents the security evaluation. Section 8 reports experimental results. Section 9 discusses limitations. Section 10 outlines future work, and Section 11 concludes.

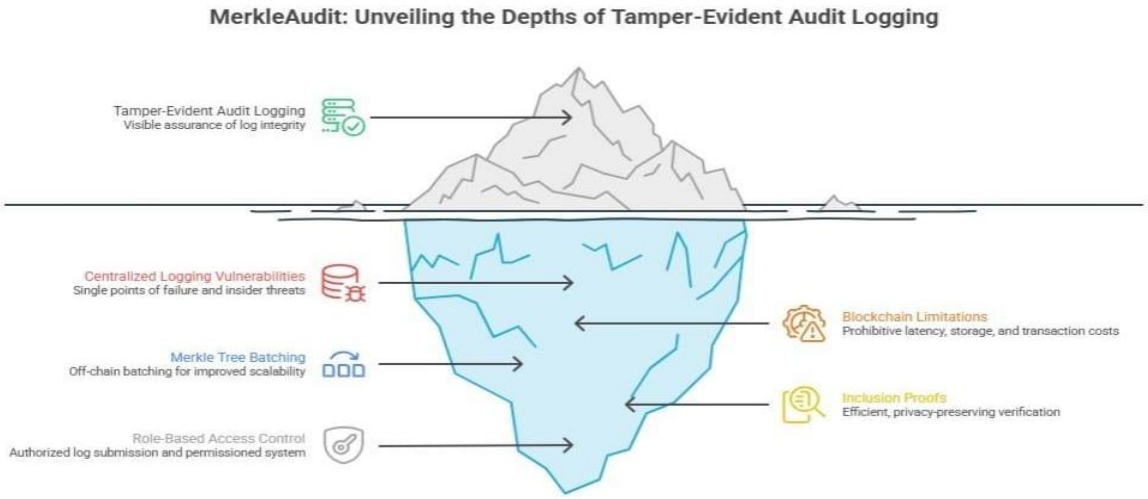

Fig. 1. Conceptual overview of MerkleAudit: the visible, tamper-evident logging guarantee rests on underlying mechanisms Merkle-tree batching, role-based access control, and privacy-preserving inclusion proofs that address the limitations of centralized logging and naive blockchain anchoring

II. RELATED WORK

2.1 Authenticated Data Structures

Crosby and Wallach proposed a tamper-evident logging scheme built on Merkle history trees that enables inclusion and consistency checks in time logarithmic in log size [1]. Hartung extended this line of work to support verification of log excerpts without revealing the remainder of the log, which is useful when only part of a log needs to be disclosed to a third

party [2]. Pulls and Peeters introduced Balloon, a forward-secure, append-only authenticated data structure that preserves integrity guarantees even if the storage system is later compromised, addressing a threat model closer to a fully malicious server [3]. At larger scale, the IETF Certificate Transparency standard applies Merkle trees to verify the authenticity of issued digital certificates across a federation of public log servers, demonstrating that Merkle-tree-based auditing can operate at internet scale [4]. Collectively, this body of work establishes Merkle trees as an effective and well-studied foundation for tamper-evident logging; however, none of these approaches incorporate blockchain anchoring or the permissioned batch-submission model that MerkleAudit introduces, and several assume a single trusted log operator rather than a decentralized, publicly verifiable anchor.

2.2 Blockchain-Backed Audit Logging

Putz et al. built an audit-logging layer on top of Hyperledger Fabric, using a permissioned blockchain to guarantee log immutability through Fabric's native ordering and endorsement mechanisms [5]. Ahmad et al. proposed BlockAudit, which relies on Practical Byzantine Fault Tolerance (PBFT) consensus to ensure log transparency and verifiability across a set of validating nodes [6]. Both systems provide strong non-repudiation guarantees, since every log entry is itself committed to a replicated, consensus-ordered ledger, but this comes at the cost of deploying and operating a full blockchain network provisioning validator nodes, managing consensus configuration, and maintaining ledger state which entails significant setup and operational overhead that may be disproportionate for organizations that need tamper-evidence but do not otherwise require a full distributed ledger deployment.

2.3 Lightweight Tamper-Evident Approaches

More recent work has explored tamper-evident logging without relying on a blockchain at all. A 2026 scheme designed for IoT edge devices uses Merkle trees together with adaptive data-chunking to enable log verification without a full blockchain backend, targeting resource-constrained environments where blockchain participation is impractical [7]. This approach shares MerkleAudit's use of Merkle trees but forgoes blockchain anchoring, sacrificing the independent, publicly verifiable tamper-evidence that a blockchain provides a guarantee MerkleAudit preserves despite its additional cost. MerkleAudit's batching strategy keeps on-chain cost approximately constant regardless of batch size, positioning it between full on-chain logging systems, which are secure but expensive, and blockchain-free approaches, which are inexpensive but lack independent public verifiability.

2.4 Positioning of This Work

Table 1 summarizes the systems discussed above along four dimensions relevant to MerkleAudit's design: the underlying authenticated data structure, whether the system anchors data to a blockchain, its permissioning model, and whether it supports privacy-preserving inclusion proofs (i.e., proving a single entry's membership without revealing the rest of the log). MerkleAudit is distinguished by combining root-only blockchain anchoring, which keeps on-chain cost independent of batch size, with role-based permissioned submission and privacy-preserving proofs, a combination not present as a whole in any single system reviewed here.

Table 1. Comparison of MerkleAudit with related tamper-evident and blockchain-backed logging systems.

System	Data Structure	Blockchain Anchoring	Permissioning	Privacy-Preserving Proofs
Crosby & Wallach [1]	Merkle tree (history tree)	No	Not addressed	Partial
Hartung [2]	Merkle tree	No	Not addressed	Yes (excerpts)
Balloon [3]	Authenticated skip list	No	Not addressed	Partial

System	Data Structure	Blockchain Anchoring	Permissioning	Privacy-Preserving Proofs
Certificate Transparency [4]	Merkle tree	No (public log servers)	Log-operator based	No
Putz et al. [5]	Ledger-native	Yes (Hyperledger Fabric)	Consortium (Fabric)	No
BlockAudit [6]	Ledger-native	Yes (PBFT chain)	Consortium (PBFT)	No
Yagiz et al. [7]	Merkle tree, adaptive chunking	No	Not addressed	Partial
MerkleAudit (this work)	Batched Merkle tree	Yes (root-only anchoring)	Role-based allow-list	Yes (inclusion proofs)

III. THREAT MODEL

MerkleAudit is designed to operate under the following threat model. The adversary is assumed to have access to the off-chain application layer that stores raw log entries and constructs Merkle trees, and may attempt to modify, delete, insert, or reorder log entries after they have been anchored.

The adversary does not control a majority of the underlying blockchain's consensus power and cannot rewrite blocks that have already been finalized; this is a standard assumption inherited from the base blockchain platform and is not a property MerkleAudit itself provides. The adversary may also attempt to submit unauthorized batches to the MerkleAuditAnchor contract, or attempt to reuse a valid (entry, proof) pair from one batch to falsely claim membership in a different batch.

Under this model, MerkleAudit aims to guarantee that (i) any modification, deletion, or insertion of a previously anchored log entry is detectable by any verifier holding the anchored root; (ii) only addresses on the contract's allow-list can extend the audit trail with new batches; and (iii) a valid inclusion proof cannot be transplanted across batches. MerkleAudit does not, by itself, guarantee the confidentiality of off-chain log contents against an adversary who gains read access to the off-chain store, nor does it protect against a compromise of the private keys controlling allow-listed submitter addresses; both are discussed further in Section 9.

IV. PROBLEM STATEMENT

Given a digital system that generates a high volume of log entries requiring verification, the problem this work addresses is: how can log integrity be assured such that (a) any verifier can detect post-hoc modification, deletion, or insertion of entries; (b) verification does not require trusting the system operator; (c) the cost of maintaining integrity guarantees does not grow linearly with the number of log entries; and (d) only authorized parties are permitted to submit new log entries.

Formally, for a sequence of log entries $E = \{e1, e2, ..., en\}$ partitioned into batches $B1, B2, ..., Bk$, the system must produce, for each batch Bi, a compact commitment ci (the Merkle root) such that (1) $|ci|$ is constant in the size of Bi, (2) any entry $ej \in Bi$ can be proven to be a member of Bi in time $O(\log |Bi|)$ given ej, ci, and a proof path, and (3) no computationally bounded adversary can produce a valid proof for an entry not actually contained in Bi, except with negligible probability, under the collision resistance of the underlying hash function.

V. PROPOSED SYSTEM: MERKLEAUDIT ARCHITECTURE

5.1 System Overview

MerkleAudit consists of two layers. An off-chain layer handles log ingestion and batch construction. An on-chain layer, MerkleAudit Anchor, stores the root hash of each batch and exposes a public verification

interface. Because only the root hash is stored on-chain, the cost of anchoring a batch is independent of whether the batch contains 10 entries or 1,000, which is the central design property that allows MerkleAudit's on-chain cost to remain flat as logging volume grows (Section 8.1).

5.2 Off-Chain Layer

Each log entry is hashed using keccak256 and inserted as a leaf into a Merkle tree. All entries within a batch are combined into a single tree, yielding one root hash per batch. The full entry data and tree structure are retained off-chain, allowing inclusion proofs to be generated for any entry at a later time.

Batch size is a configurable parameter: smaller batches anchor more frequently, reducing the window during which unanchored entries are not yet tamper-evident, but at higher aggregate on-chain cost; larger batches amortize the fixed anchoring cost over more entries but increase this exposure window. This trade-off is discussed further in Section 9.

5.3 On-Chain Layer: MerkleAudit Anchor

The MerkleAuditAnchor smart contract restricts batch submission to an allow-listed set of authorized submitters. Its anchor Batch function records a batch's root hash, entry count, timestamp, and submitting address in a single transaction. It verifies Entry function is publicly callable and checks whether a given log entry and Merkle proof are consistent with a previously anchored root hash; this check runs in time logarithmic in batch size. Administrative functions allow the contract owner to add or remove addresses from the submitter allow-list, providing a mechanism for onboarding or revoking logging permissions without redeploying the contract.

5.4 Baseline Comparison System

To quantify the benefit of batching, we implemented BaselineAuditLog, a comparison system that commits each log entry to the blockchain individually. This baseline is included solely to illustrate the cost of unbatched, per-entry logging and is not proposed as a practical solution; it represents the naive approach that MerkleAudit's batching design is intended to improve upon.

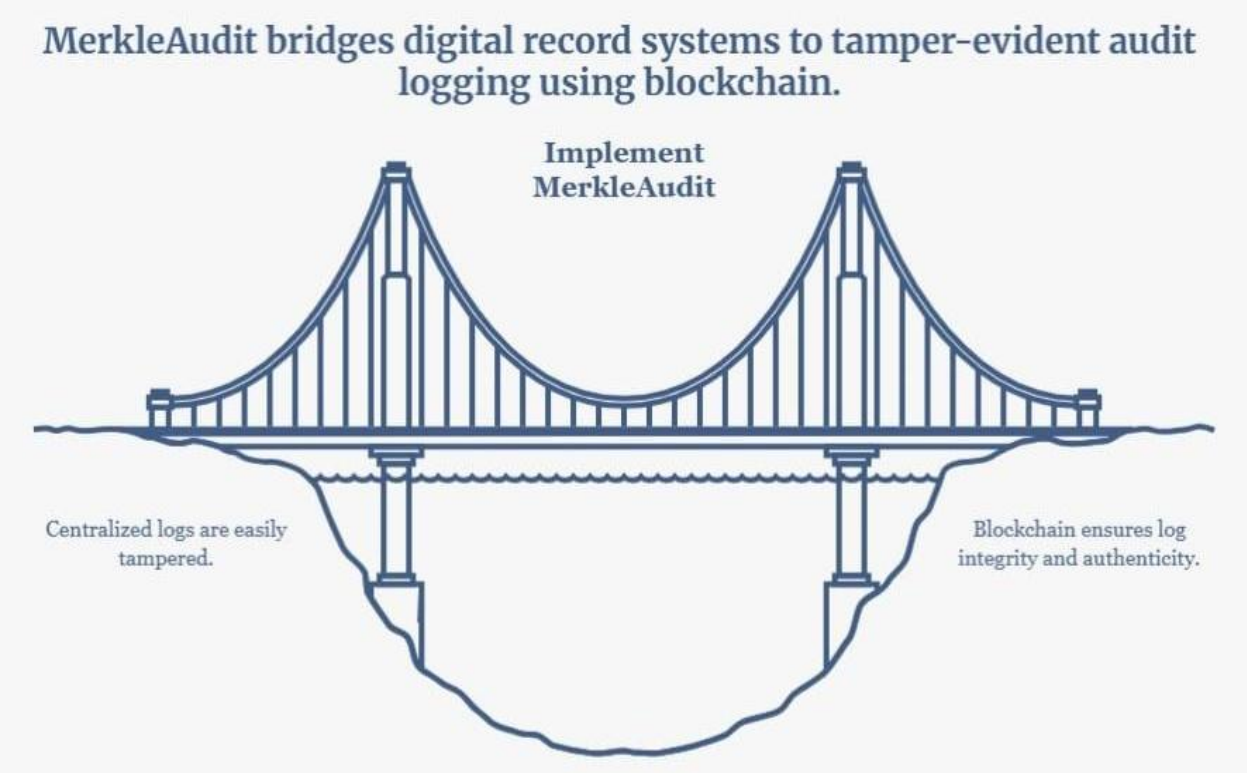

Fig. 2. MerkleAudit as a bridge between vulnerable, centralized digital record systems and blockchain-anchored guarantees of log integrity and authenticity.

VI. IMPLEMENTATION

Both the MerkleAuditAnchor and BaselineAuditLog contracts were implemented in Solidity 0.8.24 and built using the Hardhat development framework, which was used for compilation, local test-network deployment, and automated testing via its integrated Mocha/Chai test runner. The off-chain component was implemented in JavaScript, using a keccak256-based Merkle tree library to construct trees and generate inclusion proofs. Gas measurements were collected using Hardhat's built-in gas reporting against a local Hardhat network instance configured to match Ethereum mainnet opcode gas costs, and cross-checked against the deployed Sepolia contracts. The final system was deployed to the Ethereum Sepolia test network and validated end-to-end, from off-chain batch construction through on-chain anchoring and proof verification.

VII. SECURITY ANALYSIS

7.1 Adversarial Test-Based Evaluation

We developed a suite of 25 automated tests targeting five categories aligned with the threat model in Section 3: tampering detection, non-membership rejection, cross-batch replay resistance, access-control enforcement, and allow-list management. Table 2 summarizes the objective, size, and outcome of each category. All 25 tests passed (100% pass rate), including tests confirming that only allow-listed addresses can submit batches and that contract ownership can correctly manage the allow-list.

Table 2. Breakdown of the 25-test adversarial evaluation suite by category and outcome.

Test Category	Objective	Number of Tests	Result
Tampering detection	Verify that a modified log entry fails verify Entry () against its original anchored root	8	8/8 passed
Non-membership rejection	Verify that a log entry never included in any batch fails verification	6	6/6 passed
Cross-batch replay resistance	Verify that a valid (entry, proof) pair from one batch is rejected when checked against a different batch's root	5	5/5 passed
Access-control enforcement	Verify that only allow-listed addresses can successfully call anchorBatch()	4	4/4 passed
Ownership / allow-list management	Verify that the contract owner can correctly add and remove submitters	2	2/2 passed
Total	—	25	25/25 passed (100%)

It is important to characterize what a 100% pass rate on this suite does, and does not, establish. A passing adversarial test confirms that the specific attack scenario it encodes is correctly rejected by the current implementation; it does not constitute a formal proof that no other attack strategy exists, nor does it bound the probability of an undiscovered vulnerability. The suite was designed to cover the primary attack surfaces identified in the threat model entry-level tampering, non-membership claims, cross-batch proof reuse, and access-control bypass — but, consistent with the limitations of test-based verification generally, does not exhaustively enumerate the input space. We treat the 25-test suite as a necessary but not sufficient form of assurance, complemented by the independent static analysis reported in Section 7.2, and note formal verification of the verifyEntry logic as a natural direction for strengthening this assurance further (Section 10).

7.2 Independent Static Analysis (Slither)

The smart contracts were additionally analyzed using Slither, a static analysis tool for Solidity. Three findings were reported, none of which were security-critical. The tool flagged the equality comparison used within verify Entry (); on inspection, this exact-match

comparison is required and correct behavior for Merkle-proof verification, since successful verification depends on an exact match between the reconstructed and anchored root hashes, and weakening it to an approximate comparison would itself be a vulnerability. Slither also flagged the use of block.timestamp in anchor Batch (); this value is recorded solely as audit metadata and has no bearing on authorization or proof validity, so the limited manipulability of block.timestamp by miners/validators does not create an exploitable condition here. The remaining finding that certain state variables could be declared immutable was a gas-optimization recommendation rather than a security concern, and was addressed by declaring the owner variable immutable in the final implementation. No exploitable vulnerabilities were confirmed. Combining automated static analysis with targeted adversarial testing provided stronger assurance than either method would have provided in isolation, since static analysis is effective at surfacing classes of common implementation bugs while adversarial testing validates protocol-level properties specific to MerkleAudit's design that a generic analyzer is not aware of.

VIII. RESULTS AND EVALUATION

8.1 Gas Cost Benchmark

We measured the gas consumption of logging a batch of N entries via BaselineAuditLog (N individual transactions) against a single anchorBatch call in MerkleAuditAnchor, across batch sizes ranging from N = 10 to N = 5,000, to evaluate the framework under synthetic high-throughput logging conditions. Results are summarized in Table 3.

Table 3. Gas cost comparison: MerkleAudit batch anchoring vs. per-entry baseline logging (N = 10 to 5,000).

Batch Size (entries)	Baseline Gas (total)	MerkleAudit Gas (1 anchor call)	MerkleAudit Gas / Entry	Gas Savings
10	979,170	138,569	13,856.90	85.85 %
50	4,827,426	138,569	2,771.38	97.13 %
100	9,637,740	138,569	1,385.69	98.56 %
500	48,119,976	138,581	277.16	99.71 %
1,000	96,222,756	138,581	138.58	99.86 %
5,000	481,044,288	138,581	27.72	99.97 %

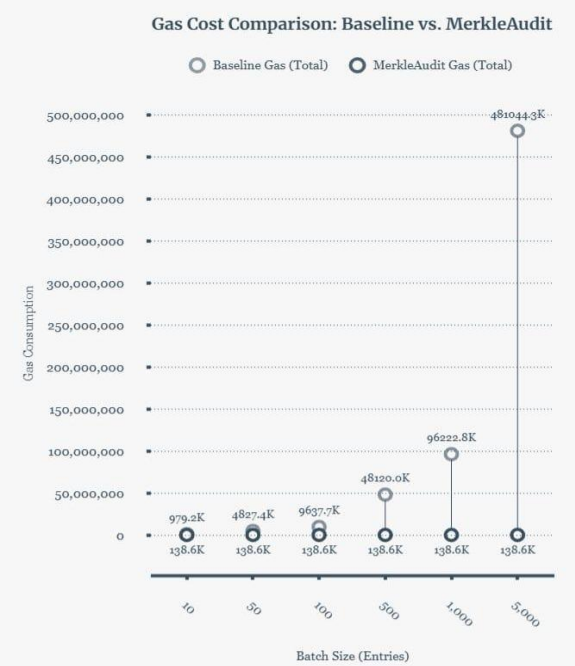

Fig. 3. Gas consumption of MerkleAudit versus the per-entry baseline across batch sizes, plotted on the same linear scale, illustrating the widening gap in total gas cost as batch size increases.

MerkleAudit's on-chain anchoring cost remains effectively constant (approximately 138,569–138,581 gas) regardless of batch size, while the baseline's cost scales linearly with the number of entries, confirming the intended decoupling of log-ingestion volume from on-chain anchoring cost. Gas savings increase with batch size, from 85.85% at N = 10 to 99.97% at N = 5,000, where MerkleAudit's per-entry cost is approximately 27.72 gas compared to roughly 96,209 gas per entry under the naive baseline. This result supports the framework's design objective of supporting high-throughput logging scenarios with minimal on-chain storage overhead. The marginal increase in MerkleAudit's absolute anchoring cost between the $N \leq 100$ and $N \geq 500$ rows (138,569 to 138,581 gas) is attributable to Solidity's storage-slot packing behavior for the entry-count field at larger values, and is negligible relative to the linear growth exhibited by the baseline.

8.2 Proof Size Scaling

For a 16-entry batch, Merkle proof length remained at or below 4 elements, consistent with the expected logarithmic relationship between batch size and proof length (⌈log2 16⌉ = 4), in contrast to the linear growth that a non-batched verification approach would exhibit. This behavior is expected to generalize to larger batches: a batch of 1,000 entries requires at most ⌈log2 1000⌉ = 10 proof elements, meaning that even substantially larger batches remain cheap to verify.

8.3 Deployment Verification

Both contracts were deployed to the Sepolia test network and verified on Etherscan, with the published source code matching the deployed bytecode. This provides independently verifiable evidence that the system described in this paper corresponds to a real, working deployment rather than a local simulation, and allows third parties to inspect the exact contract logic being anchored against.

IX. DISCUSSION AND LIMITATIONS

The consortium-style permissioning implemented in this work is enforced through role-based access control (an allow-list checked in the anchor Batch function) on a single Ethereum test network, rather than through a true multi-node consortium consensus protocol such as Hyperledger Fabric or Hyperledger Besu. This distinction is significant: in Fabric- or Besu-style consortium deployments, multiple independently operated nodes participate in an ordering or consensus process (e.g., Raft, IBFT, or PBFT-style protocols), so that no single organization can unilaterally determine which transactions are recorded, and the system tolerates a bounded number of faulty or malicious participants. MerkleAudit's current allow-list mechanism, by contrast, determines who may submit a batch, but does not itself distribute the decision of whether a submitted batch is accepted across multiple independent parties that decision inherits its trust properties from the underlying Ethereum network's consensus, not from a dedicated multi-organization consortium layer. This is a deliberate scoping decision for the current implementation rather than an oversight, and is stated explicitly here so that the paper's claims are not overstated: MerkleAudit provides permissioned write access and publicly verifiable read access, but not, in its present form, Byzantine-fault-tolerant multi-party governance over which batches are anchored. Extending the design with a genuine multi-node consortium layer is discussed as future work in Section 10.

A second limitation concerns confidentiality. Off-chain entry data is currently assumed to be held by a trusted application layer and is not encrypted; consequently, an attacker who gains access to the off-chain store while unable to alter historical entries without detection, since any alteration would no longer match the anchored root could still read their contents. This means MerkleAudit's guarantees are integrity- and availability-of-evidence guarantees rather than confidentiality guarantees, and deployments handling sensitive data (e.g., healthcare or financial records) would need to layer encryption or access control over the off-chain store independently of the mechanisms described here.

A third limitation is that the trade-off between batching latency and anchoring cost that is, how frequently batches should be anchored was not analyzed in this work. More frequent anchoring reduces the time window during which recently generated log entries are not yet covered by an on-chain commitment, at the cost of higher aggregate gas expenditure; less frequent anchoring amortizes cost more effectively but widens this exposure window. Selecting an appropriate anchoring interval is application-dependent and is left as a parameter for future empirical study.

Finally, submitter identity in the current design is tied to possession of an Ethereum private key corresponding to an allow-listed address. Compromise of such a key would allow an attacker to submit fraudulent batches until the compromise is detected and the address is removed from the allow-list by the contract owner; MerkleAudit does not currently include key-compromise detection or automated revocation mechanisms, and relies on standard key-management practice outside the scope of the smart contract itself.

X. FUTURE WORK

Several directions could extend MerkleAudit beyond the scope of the present prototype. Incorporating zero-knowledge succinct proofs (zk-SNARKs) is a natural next step, potentially allowing an auditor to confirm

that a log entry is valid and included in an anchored batch without observing the entry's contents a property particularly relevant to privacy-sensitive domains such as healthcare records. This direction was not pursued in the current work due to the circuit-design effort required within the available project timeline, and is noted here as a direction for future research rather than something attempted.

A second direction is to deploy the consortium layer on a genuine multi-node permissioned network such as Hyperledger Besu operating under IBFT 2.0 consensus, in order to evaluate the permissioning model under real distributed consensus rather than a single-node simulation, and to directly measure the latency and throughput trade-offs this introduces relative to the current Ethereum Sepolia deployment.

A third direction is a systematic empirical study of the batch-size and anchoring-interval trade-off identified in Section 9, measuring end-to-end detection latency (the time between an entry being generated and its inclusion being provably anchored on-chain) as a function of batch size and anchoring frequency, to provide deployment guidance for organizations with different latency and cost sensitivities.

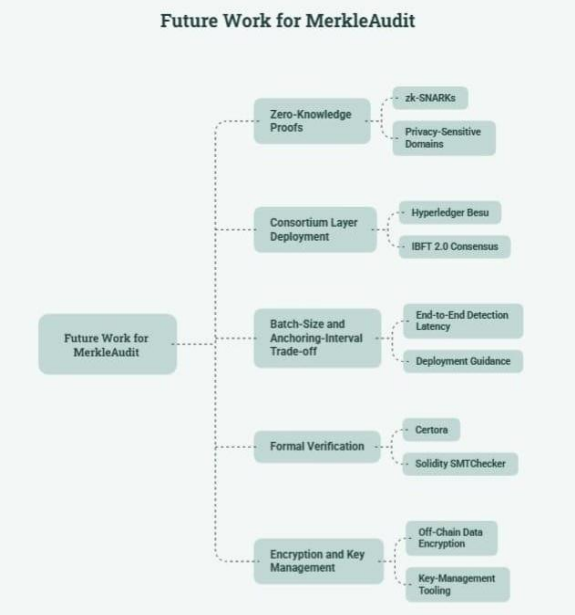

Fig. 4. Summary of proposed future-work directions for MerkleAudit, spanning privacy (zero-knowledge proofs), decentralized governance (consortium-layer deployment), deployment tuning (batch-size and anchoring-interval trade-off), assurance (formal verification), and confidentiality (encryption and key management)

A fourth direction is formal verification of the verifyEntry and anchorBatch functions using a tool such as Certora or the Solidity SMTChecker, to complement the adversarial testing and static analysis reported in Section 7 with machine-checked correctness guarantees for the core proof-verification logic. Finally, integrating optional encryption of off-chain entry data, together with key-management and submitter-key-revocation tooling, would address the confidentiality and key-compromise limitations discussed in Section 9.

XI. CONCLUSION

This paper presented MerkleAudit, a blockchain-based framework for tamper-evident audit logging that anchors Merkle roots of batched log entries rather than individual entries combined with role-based permissioned batch submission. Deployed and verified on the Ethereum Sepolia testnet, the system was validated through a 25-test functional and adversarial suite spanning five attack categories, independent Slither static analysis, and an empirical gas-cost comparison against a naive on-chain logging baseline across batch sizes from 10 to 5,000 entries, achieving gas savings of up to 99.97% at a batch size of 5,000 while preserving complete tamper-evidence guarantees against modification, deletion, and replay attacks. At the same time, the evaluation clarified the boundaries of these guarantees: the current implementation provides permissioned, publicly verifiable write access rather than multi-party Byzantine-fault-tolerant governance, and integrity rather than confidentiality of off-chain data. Taken together, these results demonstrate a validated, cost-efficient path toward scalable, publicly verifiable audit trails for digital record systems, while identifying concrete directions multi-node consortium deployment, zero-knowledge proofs, and formal verification for closing the remaining gap between this prototype and a production-grade audit-logging system.

REFERENCES

[1] S. A. Crosby and D. S. Wallach, “Efficient data structures for tamper-evident logging,” in *Proc. 18th USENIX Security Symp.*, Montreal, Canada, 2009, pp. 317–334.

[2] G. Hartung, “Secure audit logs with verifiable excerpts,” in Topics in Cryptology – CT-RSA 2016, Lecture Notes in Computer Science, vol. 9610, Springer, 2016, pp. 183–199.

[3] T. Pulls and R. Peeters, “Balloon: A forward-secure append-only persistent authenticated data structure,” in Computer Security – ESORICS 2015, Lecture Notes in Computer Science, vol. 9327, Springer, 2015, pp. 622–641.

[4] B. Laurie, A. Langley, and E. Kasper, “Certificate Transparency,” Internet Engineering Task Force (IETF), RFC 9162, Dec. 2021.

[5] B. Putz, M. Menges, and G. Pernul, “A secure and auditable logging infrastructure based on a permissioned blockchain,” Computers & Security, vol. 87, Art. no. 101602, 2019.

[6] A. Ahmad, S. U. Malik, M. F. Zaffar, et al., “BlockAudit: Distributed blockchain-based audit logging for smart contract systems,” Cluster Computing, vol. 22, 2019.

[7] M. A. Yagiz, et al., “Lightweight tamper-evident log integrity verification for IoT edge environments: A Merkle tree pipeline with adaptive chunking,” *arXiv preprint arXiv:2605.00065*, 2026.