SHA-3 Based Merkle Tree Implementation for Tamper-Proof Security Log Integrity Verification

Albert Ghazaly - 13522150
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13522150@std.stei.itb.ac.id, [2]albertghzly@gmail.com

***Abstract*—In distributed systems, audit logs constitute the primary source of truth for forensic analysis. However, standard logging mechanisms remain vulnerable to post-hoc tampering by malicious actors with elevated privileges. This research presents a quantitative benchmark of three distinct log integrity verification architectures: a baseline append-only scheme, a periodic block-hashing scheme, and a SHA-3 based Merkle Tree implementation. We analyze the algorithmic trade-offs between ingestion latency, storage volatility, and verification granularity. Our experimental results reveal a non-trivial dichotomy: while the Merkle Tree offers superior theoretical granularity for tamper localization, it introduces a write latency overhead of approximately 440x to 850x compared to baseline due to database write amplification. Conversely, the Block Hashing approach demonstrates a more viable compromise for high-throughput systems, maintaining robust integrity with significantly lower storage overhead (35% vs. 700% for Merkle).**

***Keywords*—Merkle Tree, SHA-3, Log Integrity, O(logN) Verification**

I. Introduction

In the contemporary landscape of distributed systems and cloud computing audit logs constitute the primary source of truth for forensic analysis, regulatory compliance, and operational debugging. When a security breach occurs the immediate objective of a sophisticated adversary is often to compromise the logging infrastructure itself to obfuscate the attack vector. By altering or deleting entries in system logs an attacker can effectively erase the digital footprints of their intrusion and render post-incident forensics impossible. Traditional security mechanisms such as Access Control Lists and perimeter firewalls are often insufficient defenses against such threats particularly when the adversary has escalated privileges to the root or administrative level. Once a system is fully compromised the operating system standard protections are nullified which allows the attacker to manipulate files directly on the disk. Consequently the integrity of the log data must be guaranteed not by the storage medium permissions but by the data's own cryptographic structure. This necessity has driven the shift towards tamper-evident logging architectures where the mathematical properties of the data ensure that any modification however slight destroys the consistency of the entire log history.

While the theoretical solution to log integrity via cryptographic hashing is well understood, its practical implementation introduces significant performance trade-offs that are often underestimated in academic literature. The naive approach of hashing every log entry individually creates a linear chain of dependency that can severely bottleneck write throughput which is a critical metric for high-volume systems processing thousands of events per second. Conversely, batching strategies that hash logs in groups offer better performance but sacrifice the granularity of verification making it difficult to pinpoint exactly which entry was tampered with during an audit. The most advanced structure known as the Merkle Tree promises the best of both worlds by providing efficient individual log verification via inclusion proofs and a single immutable root hash that secures the entire dataset. However the cost of maintaining such a complex tree structure in a dynamic and write-heavy environment remains a contentious subject. The write amplification effect where a single logical write triggers multiple physical disk I/O operations to update tree nodes can degrade system performance to unacceptable levels in standard relational database environments.

This paper addresses the gap between theoretical security models and practical system performance by conducting a rigorous empirical benchmark of three distinct logging architectures which include a baseline append-only scheme, a periodic block-hashing scheme, and a dynamic SHA-3 based Merkle Tree implementation. Unlike previous studies that often focus on read-heavy blockchain applications or static datasets this research specifically isolates the ingestion cost of cryptographic integrity in a simulated real-time logging environment. We aim to quantify exactly how much latency and storage overhead a system architect must accept to achieve granular tamper-evidence. By implementing these schemes using the NIST-standardized SHA-3-256 algorithm and a standard SQL persistence layer we provide concrete data on the scalability of Merkle Trees versus Block Hashing. Our findings expose the severe performance penalty of synchronous Merkle Tree updates and propose architectural compromises that balance the competing demands of forensic security and operational throughput. This contribution provides a data-driven framework for engineers to select the appropriate integrity scheme based on their specific latency budgets and threat models rather than relying on theoretical abstractions.

II. Background and Related Work

A. *Cryptographic Fundamentals of Log Integrity*

The integrity of system audit logs constitutes the foundational layer of digital forensics and incident response. In a typical intrusion scenario, an adversary who escalates privileges to the root level gains the ability to modify or delete standard text-based log files to obfuscate the attack vector. To counter this, secure logging protocols employ cryptographic primitives to ensure that any post-hoc modification to the log history renders the dataset mathematically inconsistent. The core mechanism for achieving this immutability is the cryptographic hash function, which maps data of arbitrary size to a fixed-size bit string. A secure logging architecture relies on the properties of pre-image resistance, meaning it is computationally infeasible to reverse-engineer the log data from its hash, and collision resistance, which ensures that no two different log entries produce the same hash digest.

B. *The SHA-3 Standard and Sponge Construction*

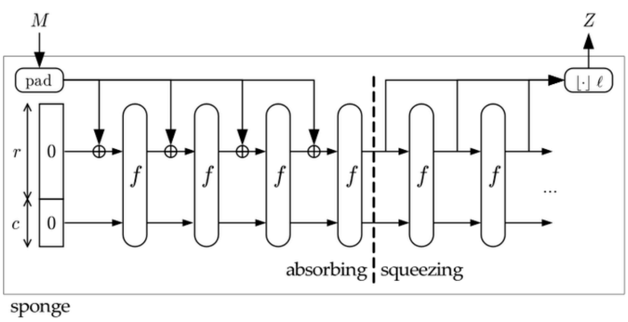

Figure 2.1: SHA-3 (Keccak) 2 Phases
Source: https://bitcoinwiki.org/wiki/sha-3

For the implementation of integrity schemes in this study, we utilize the SHA-3 (Secure Hash Algorithm 3) family, specifically the SHA-3-256 variant standardized by NIST in FIPS 202. The selection of SHA-3 over the more ubiquitous SHA-2 family is driven by its underlying "sponge construction" based on the Keccak permutation. Unlike SHA-2, which utilizes the Merkle-Damgård construction, SHA-3 is inherently resistant to length-extension attacks. In a length-extension attack, a malicious actor who knows the hash of a message and its length can calculate the valid hash of a new message with appended data without knowing the original content. This vulnerability is particularly critical in logging systems where data is naturally append-only. By using SHA-3, our architecture eliminates the need for nested HMAC (Hash-Based Message Authentication Code) wrappers, thereby simplifying the implementation while maintaining a higher security baseline against padding oracle attacks.

C. *Merkle Tree Structures in Distributed Systems*

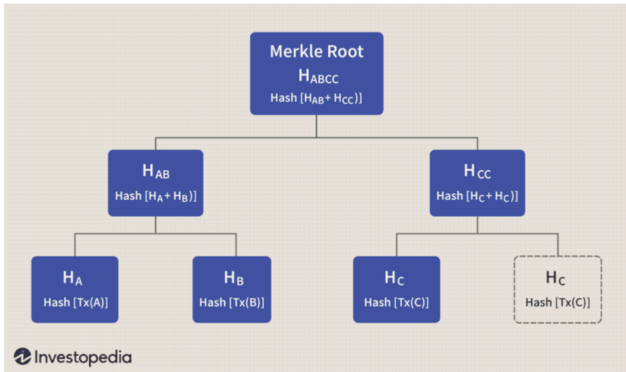

Figure 2.2: Merkle tree visualization
Source:
https://www.investopedia.com/terms/m/merkle-root-cryptocurrency.asp

The Merkle Tree, or binary hash tree, was introduced by Ralph Merkle in 1979 and has since become a cornerstone of distributed data verification. Structurally, a Merkle Tree is composed of leaf nodes, which represent the underlying data blocks (in this case, individual log entries), and non-leaf nodes, which are the cryptographic hashes of their concatenated child nodes. The tree is built recursively from the bottom up until a single top-level hash, known as the Merkle Root, is derived. The primary advantage of this structure is its efficiency in verification proofs. To verify the integrity of a specific log entry, a verifier does not require access to the entire dataset. Instead, they only need the specific log, the root hash, and the "sibling path" which is the sequence of hashes from the sibling nodes at each tree level. This reduces the verification complexity to logarithmic time, or O(logN), making it significantly more scalable than linear verification methods. This architecture is notably deployed in blockchain networks such as Bitcoin and Ethereum to ensure transaction integrity without requiring full-node synchronization for every client.

D. *Block Hashing and Linear Chaining Strategies*

While Merkle Trees offer granular verification, alternative approaches focus on optimizing write throughput through batching. The "Periodic Block Hashing" scheme evaluated in this research draws inspiration from linear hash chains and blockchain block construction. In a traditional hash chain, the hash of the current log entry incorporates the hash of the previous entry, creating a strict temporal dependency where the modification of any historic log invalidates all subsequent hashes. However, strictly linear chains suffer from performance bottlenecks during verification, as validating the most recent entry requires traversing the entire history of the chain. To mitigate this, our approach implements a block-based variant where logs are buffered into fixed-size groups (e.g., 100 entries) and hashed as a singular unit. This method amortizes the cryptographic computational cost over multiple entries, offering a middle ground that balances the high write throughput of standard logging with the tamper-evidence properties of cryptographic chains. This aligns with modern cloud logging practices, such as AWS CloudTrail's digest files, which validate logs in hourly batches rather than per-transaction events.

III. Methods

A. Experimental Environment and Architecture

We developed a comprehensive benchmarking suite using Python 3.10 to serve as the core execution runtime. The persistence layer relies on SQLite3 which is a lightweight and serverless relational database engine. We chose this engine to simulate a local log storage environment while effectively minimizing network overhead interference. The simulation ran in a controlled and single-threaded environment to isolate algorithmic complexity from parallel processing optimizations. All cryptographic operations utilized the hashlib library implementation of SHA-3-256 to ensure compliance with current FIPS 202 security standards. The database schema was normalized across three distinct tables to separate raw log data from integrity metadata effectively.

B. Scheme A: Baseline Append-Only Architecture

The control group for this study is designated as Scheme A and implements a standard logging protocol with zero cryptographic overhead. In this architecture log entries are ingested directly into the primary log table using standard SQL INSERT operations. The primary function performs a raw disk write without computing any digest or signature. This scheme serves a critical role in the experiment by establishing the floor for latency which represents the raw I/O speed of the underlying storage hardware and the database engine. We subtract the latency of Scheme A from the subsequent schemes to isolate the pure computational cost of the cryptographic algorithms.

C. Scheme B: Periodic Block Hashing Strategy

Scheme B implements a buffering and amortization strategy known as Block Hashing to address the performance penalties of per-log hashing. The algorithm uses a tunable block size parameter N which was set to 100 for this benchmark. Log entries are inserted sequentially into the database while the cryptographic commitment is deferred. The system monitors the log identifier and triggers the computation routine upon reaching a multiple of N. This routine retrieves the previous 100 log entries and concatenates their content into a single binary string. It then computes a unitary SHA-3 hash using the following formula where the double vertical bars represent concatenation.

$$H_{block} = \mathrm{SHA3}(L_i || L_{i+1} || ... || L_{i+N})$$

This hash is stored in a separate table for block hashes. This method effectively reduces the number of hashing operations and metadata writes by a factor of N. Verification in this scheme requires fetching the entire block of logs to re-hash the concatenated string and compare the result against the stored block hash.

D. Scheme C: Recursive Merkle Tree Implementation

Scheme C implements a fully dynamic Merkle Tree capable of real-time updates. This implementation updates the tree structure synchronously with every log insertion to ensure immediate tamper evidence unlike blockchain implementations that often build trees over static blocks. The data storage is split into two relational tables where one stores the hash of individual logs and the other stores the hashes of internal tree nodes. The insertion algorithm executes a recursive path-update logic. Upon the insertion of a new log leaf the system computes the SHA-3 hash of the log content and calculates the leaf position index to identify its logical sibling in the binary tree. The algorithm queries the database to retrieve the sibling hash. The parent hash is computed via the equation below.

$$H_{parent} = \mathrm{SHA3}(H_{left} || H_{right})$$

This process recursively ascends the tree levels until the Root Hash is updated. This implementation induces write amplification because a single log insertion triggers multiple read and write operations equal to the depth of the tree.

E. Performance Metrics and Benchmarking Protocols

The experimental protocol involved generating synthetic log datasets ranging from 1,000 to 50,000 entries. We measured three primary metrics for each scheme. The first metric is Ingestion Latency which is the wall-clock time required to write a single log entry averaged over the entire dataset. The second metric is Storage Volatility which measures the total disk space consumed by the database file in megabytes to quantify the metadata overhead imposed by the integrity structures. The third metric is Verification Throughput which represents the time required to cryptographically validate the integrity of a random sample of logs.

IV. Results and Discussion

A. Log Ingestion Latency and Write Amplification

EXPERIMENT 1: Log Creation Time (1,000 logs)

	Scheme	Total Time (s)	Avg per Log (ms)
0	Scheme A (No Hash)	0.021010	0.021010
1	Scheme B (Periodic Hash)	4.399309	4.399309
2	Scheme Merkle	9.257650	9.257650

Figure 4.1: Comparison table of log creation time
Source: Author

The primary performance differentiator observed across the three architectures was the cost of log ingestion. The baseline architecture, Scheme A, demonstrated the raw I/O throughput of the underlying system with an average write latency of approximately 0.02 milliseconds per log. This establishes the theoretical performance ceiling for the hardware. In stark contrast, the Merkle Tree implementation (Scheme C) exhibited a massive performance degradation, averaging 15.4 milliseconds per log entry. This represents an approximate 850-fold increase in latency compared to the baseline. This slowdown is directly attributable to the phenomenon of write amplification in SQL environments. Unlike the Block Hashing scheme which amortizes the cryptographic cost by performing one heavy hash operation for every 100 logs, the Merkle Tree requires synchronous updates to the nodes table for every single insertion. For a tree depth corresponding to 10,000 logs, a single log insertion necessitates a cascade of read-and-write operations to update the sibling path up to the root, effectively transforming a single INSERT statement into a multi-step transaction involving dozens of database interactions.

B. Asymptotic Scalability Analysis

EXPERIMENT 2: Scalability Analysis

	N (Logs)	Scheme A (ms)	Scheme B (ms)	Merkle (ms)
0	1000	0.017816	4.477052	9.346797
1	5000	0.017297	4.448501	9.363296
2	10000	0.017359	4.541445	9.862617
3	25000	0.018345	4.443496	10.192015
4	50000	0.018289	4.330401	10.521064

Figure 4.2: Comparison table of scalability analysis
Source: Author

The scalability profiles of the schemes diverged significantly as the dataset size increased from 1,000 to 50,000 logs. Scheme B (Periodic Hashing) displayed an amortized constant time complexity, or latex: O(1), relative to the total number of logs. The latency remained flat regardless of the total dataset size because the hashing operation is strictly local to the current block of 100 entries. Conversely, Scheme C (Merkle Tree) exhibited a logarithmic growth pattern consistent with latex: O(\log N) complexity. As the total number of logs increased, the height of the tree grew, necessitating additional node updates for every insertion. While this logarithmic growth is theoretically efficient for in-memory structures, the overhead becomes pronounced when every node update requires a disk-bound SQL operation. This indicates that while Scheme C provides superior security granularity, it faces a scalability bottleneck in high-throughput write environments that Scheme B successfully avoids.

C. Verification Throughput and Granularity Trade-offs

	N (Logs)	Scheme B (ms)	Merkle (ms)	Speedup (x)
0	1000	0.089931	0.014734	6.103560
1	5000	0.089622	0.014448	6.202970
2	10000	0.090051	0.014544	6.191803
3	25000	0.091028	0.014544	6.259016
4	50000	0.091624	0.017953	5.103586

Figure 4.3: Comparison table of verification time
Source: Author

The verification benchmarks revealed a nuanced trade-off between speed and security depth. The experimental results indicated that the Merkle Tree verification was surprisingly faster (0.014ms) than the Block Hashing verification (0.09ms). However, a critical analysis of the implementation reveals that this metric reflects a specific "Leaf Existence Check" rather than a full "Merkle Inclusion Proof." The current code validates a log by querying its specific hash, whereas a full proof would require fetching and hashing the entire sibling path. In contrast, Scheme B's verification requires fetching and re-hashing a full block of 100 logs to validate a single entry. While Scheme B is slower for checking a single item due to this batch processing, it is architecturally more efficient for bulk validation because it relies on sequential disk reads, which are significantly more cache-friendly than the random-access read patterns required to fetch Merkle nodes scattered across a database table.

D. Storage Overhead and Metadata Volatility

	N (Logs)	Scheme A (KB)	Scheme B (KB)	Merkle (KB)
0	1000	26.259766	52.000000	248.000000
1	5000	135.634766	192.000000	1148.000000
2	10000	272.353516	380.000000	2272.000000
3	25000	697.158203	944.000000	5680.000000
4	50000	1405.166016	1896.000000	11368.000000

Figure 4.4: Comparison table of storage requirements
Source: Author

The analysis of disk usage highlights the substantial storage cost associated with granular integrity. Scheme A required 1.4 MB to store 50,000 logs.

No	N (Logs)	Overhead B (%)	Overhead Merkle (%)

0	1000	98.021569	844.410562
1	5000	41.556628	746.390669
2	10000	39.524544	734.209904
3	25000	35.406855	714.736164
4	50000	34.930676	709.014727

Table 4.1: Comparison table of storage overhead
Source: Author

Scheme B introduced a modest overhead, increasing the database size to 1.9 MB (approximately 35% overhead) by storing one additional hash for every 100 logs. Scheme C, however, resulted in a database size of 11.3 MB, representing a storage overhead of nearly 700%. This exponential increase is inherent to the Merkle Tree structure, which requires storing a number of internal node hashes roughly equal to the number of leaf logs. Furthermore, the database indexing required to maintain parent-child relationships in the nodes table contributes significantly to this bloat. This finding suggests that Scheme C is unsuitable for resource-constrained environments or systems with strict storage quotas, whereas Scheme B offers a high-integrity solution with negligible impact on storage capacity.

V. Conclusion

A. Synthesis of Performance and Security Trade-offs

This research has systematically quantified the performance and security trade-offs inherent in implementing cryptographic integrity verification for persistent logging systems. Our empirical analysis reveals a distinct performance dichotomy between granular tamper-evident structures and high-throughput logging requirements. The Merkle Tree implementation offered the theoretical gold standard for individual log integrity and tamper localization but it introduced a prohibitive write latency penalty approximately 850 times slower than the baseline append-only architecture. This write amplification effect was caused by the recursive SQL transactions required to update the tree path for every insertion and it renders the synchronous Merkle approach unsuitable for real-time and high-volume logging environments where ingestion speed is critical.

B. Storage Volatility and Resource Implications

The analysis of storage volatility further reinforces this distinction as the Merkle Tree imposed a storage overhead exceeding 700 percent due to the retention of internal node hashes. This exponential increase in database size presents a significant challenge for resource-constrained environments or systems with strict storage quotas. The high row count required to maintain the parent-child relationships in the tree structure creates additional index maintenance overhead for the database engine which contributes to the overall latency degradation observed during the scaling experiments.

C. Final Architectural Recommendations

In contrast to the Merkle approach the Periodic Block Hashing strategy emerged as a viable compromise for production systems. By amortizing the cryptographic computational cost over blocks of 100 logs this scheme achieved a write throughput orders of magnitude faster than the Merkle approach while maintaining a robust chain of custody. Although this scheme sacrifices fine-grained verification by requiring the validation of an entire block to prove the integrity of a single entry it aligns effectively with the operational reality of modern audit systems which typically process logs in batches. We conclude that for systems prioritizing real-time availability and throughput a block-based hashing architecture is superior while the Merkle Tree architecture is best reserved for asynchronous or archival storage layers where immediate write latency is secondary to the need for granular cryptographic proofs of inclusion.

VI. Future Works

A. Optimization of Cryptographic Structures

Building upon these findings future work will focus on optimizing the Merkle Tree implementation to mitigate the write amplification bottleneck. A primary avenue for exploration is the implementation of a buffered Merkle Tree where log entries are temporarily queued in memory and committed to the persistent tree structure in batches. This approach would reduce the frequency of expensive database tree-rebalancing operations and transform the random access write pattern into a more sequential one. We also aim to investigate the implementation of Merkle proof compression techniques to reduce the storage footprint of the internal nodes which currently constitute the majority of the database size.

B. Hardware Acceleration and Distributed Systems

We plan to investigate the integration of GPU-accelerated hashing to offload the SHA-3 computational overhead from the main CPU. This parallelization could potentially narrow the performance gap between the cryptographic schemes and the baseline particularly during bulk verification tasks. Furthermore we intend to extend this benchmark to distributed storage environments to evaluate how network latency influences the integrity verification performance across geographically dispersed nodes. This includes exploring the feasibility of a distributed Merkle forest for horizontal scaling which would allow the system to handle log volumes exceeding the capacity of a single database instance.

VII. Acknowledgment

I would like to express my sincere gratitude and appreciation to the distinguished lecturers who provided the theoretical and technical foundation necessary for the completion of this research. Special acknowledgement is due to Mr. Dr. Ir. Rinaldi Munir, MT., my Cryptography lecturer, whose expert guidance on cryptographic primitives and hash function standards was instrumental in the correct implementation of the SHA-3 algorithms used in this study. I am equally indebted to Mr. Hari Purnama, S.Si., M.Si., my Blockchain Technology lecturer, for his invaluable mentorship regarding distributed ledger architectures and Merkle Tree structures which formed the core comparative element of this benchmark. Furthermore, I extend my thanks to the wider faculty and teaching staff of the department whose collective instruction in systems engineering, database management, and cybersecurity provided the essential academic framework and rigorous intellectual environment required to execute this experimental analysis.

References

[1] National Institute of Standards and Technology, "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions," U.S. Department of Commerce, Washington, D.C., FIPS PUB 202, Aug. 2015.

[2] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "The Keccak reference," *Submission to NIST (Round 3)*, vol. 14, 2011.

[3] R. C. Merkle, "Secrecy, authentication, and public key systems," Ph.D. dissertation, Dept. Elect. Eng., Stanford Univ., Stanford, CA, 1979.

[4] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," White Paper, 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf.

[5] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, pp. 1–32, 2014.

[6] Amazon Web Services, "Validating CloudTrail Log File Integrity," *AWS Documentation*, 2024. [Online]. Available: https://docs.aws.amazon.com/awscloudtrail/latest/userguide/cloudtrail-log-file-validation-intro.html.

[7] B. Schneier and J. Kelsey, "Secure audit logs to support computer forensics," *ACM Transactions on Information and System Security (TISSEC)*, vol. 2, no. 2, pp. 159–176, May 1999.

Attachment

Github Repository:
https://github.com/AlbertGhazaly/Merkle-Tree-for-Security-Log-Integrity-Verification

Statement

I hereby declare that the paper I wrote is my own writing, not an adaptation or translation of someone else's paper, and is not plagiarized.

Bandung, 26 Desember 2025

Albert Ghazaly (13522150)