TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Nikolai Ovtšinnikov 242803IVCM

Performance Comparison of AVL Hash Trees and Sparse Merkle Trees

Master's Thesis

Supervisor: Ahto Truu
PhD

Tallinn 2026

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Nikolai Ovtšinnikov 242803IVCM

AVL-räsipuude ja hõredate Merkle'i puude jõudluse võrdlus

Magistritöö

Juhendaja: Ahto Truu
PhD

Tallinn 2026

Author's declaration of originality

I hereby certify that I am the sole author of this thesis and that this thesis has not been presented for examination or submitted for defense anywhere else. All used materials, references to the literature, and work of others have been cited.

Author: Nikolai Ovtšinnikov

20.05.2026

Acknowledgements

Artificial intelligence tools supported selected parts of the work behind this thesis.

Codex [1] was used as an AI coding assistant during implementation work and while preparing technical documentation related to the benchmark artifacts.

ChatGPT [2] was used with task-specific prompts to check the clarity of written sections, grammar and punctuation. Additionally, ChatGPT was used to explain relevant concepts.

The use of these tools was limited to drafting support, review, implementation assistance, and technical clarification. They were not treated as independent academic sources for the thesis findings.

The author remains responsible for all conclusions presented in the thesis.

Abstract

Authenticated data structures play a central role in blockchain state management and verifiable storage, but there is still little direct empirical evidence to help choose between AVL Hash Trees and Sparse Merkle Trees under comparable conditions. Most existing work studies variants within the same structural family rather than comparing these two data structures directly. This thesis addresses that gap through a controlled performance comparison aimed at informing real implementation choices.

To address this goal, both structures were implemented in Go programming language and benchmarked in the same single-threaded in-memory environment using the same SHA-256 hash function, fixed 32-byte keys and values, hardware, and workload families. The benchmark covered full builds, ordered builds, post build insertion of new keys, existing element updates, and inclusion and exclusion proof generation and verification. The results show that AVL had lower update latency and lighter measured memory behavior, while SMT produced noticeably smaller proofs. Inclusion proof verification remained comparatively close, while AVL was faster for exclusion proof verification.

The thesis shows that AVL is the better choice when update throughput, prover work, or measured memory footprint is the dominant constraint, whereas SMT is stronger when small proof size matters most. It also provides a practical baseline for future work on concurrent execution, persistent storage, and broader proof workloads.

The thesis is in English and contains 58 pages of text, 6 chapters, 10 figures, 8 tables.

Annotatsioon

AVL-räsipuude ja hõredate Merkle'i puude jõudluse võrdlus

Autentivatel andmestruktuuridel on oluline roll plokiahela olekuhalduses ja kontrollitavas andmesalvestuses, kuid otseseid empiirilisi tõendeid AVL-räsipuude ja hõredate Merkle'i puude (SMT) vahel valimiseks võrreldavates tingimustes on vähe. Suurem osa olemasolevaid töid uurib erinevaid variante sama autentivate puude perekonna sees, kuid ei võrdle neid kahte struktuuri omavahel. Käesolev lõputöö täidab seda lünka kontrollitud jõudlusanalüüsiga, mille eesmärk on toetada reaalseid rakendusotsuseid.

Selle eesmärgi saavutamiseks realiseeriti mõlemad struktuurid Go programmeerimiskeeles ning mõõdeti nende jõudlust samas ühelõimelises muutmälusiseses keskkonnas, kasutades samu krüptograafilisi primitiive nagu SHA-256 räsifunktsiooni, lisaks fikseeritud 32-baidiseid võtmeid ja väärtuseid, sama riistvara ning samu töövooge. Analüüs hõlmab puu ehitust juhuslikus järjekorras ja järjestatud andmetest, uute võtmete lisamist peale puu eelnevat valmisehitust, olemasolevate võtmete väärtuste muutmist ning kaasamis- ja välistamistõendite genereerimist ja kontrollimist. Tulemused näitavad, et AVL-räsipuul oli madalam latents ja väiksem mälukasutus, samas kui SMT genereeris märgatavalt väiksemaid tõendeid. Kaasamistõendite kontrollimine jäi mõlemal struktuuril sarnaseks, samas kui välistamistõendite kontrollimisel oli AVL-räsipuu kiirem.

Lõputöö näitab, et AVL-räsipuu on parem valik, kui olulisem on andmete lisamine ja uuendamise jõudlus, tõestaja töökoormus või mälukasutus, samas kui SMT eelis on tõendite suurus. Samuti pakub töö praktilist alust tulevasteks uuringuteks paralleeltöötluse, püsiva salvestuse ja laiema tõestuskoormuse alal.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 58 leheküljel, 6 peatükki, 10 joonist, 8 tabelit.

List of abbreviations and terms

API	Application Programming Interface
AVL Tree	Adelson-Velsky and Landis Tree
CPU	Central Processing Unit
CSV	Comma-Separated Values
DMT	Dynamic Merkle Tree
FPGA	Field-Programmable Gate Array
HMT	Hybrid Bonsai Merkle Tree
I/O	Input/Output
JMT	Jellyfish Merkle Tree
MPT	Merkle Patricia Tree
NVMe	Non-Volatile Memory Express
PAD	Persistent Authenticated Dictionary
PoR	Proofs of Retrievability
RAM	Random Access Memory
RQ	Research Question
SHA-256	Secure Hash Algorithm 256-bit
SMT	Sparse Merkle Tree
SNARK	Succinct Non-Interactive Argument of Knowledge
SSD	Solid-State Drive
UTXO	Unspent Transaction Output
WSL2	Windows Subsystem for Linux 2

Table of contents

List of figures

List of tables

1 Introduction

1.1 Motivation

Modern blockchain networks sustain millions of operations per day, making the efficiency of their underlying data structures increasingly important for system performance. Among these structures, authenticated trees provide cryptographic verification of data integrity and membership and are used in transaction verification and state management. Two dominant structures in this domain are the AVL Hash Tree (also called the Merkle AVL Tree or AVL Merkle Tree) [3, 4, 5, 6] and the Sparse Merkle Tree (SMT) [7, 8, 9, 10]. Both AVL Hash Trees and SMTs offer O(log n) time complexity for insertion, update, and delete operations, as well as proof generation. However, their structures and design philosophies differ substantially.

Because of these implementation differences, real-world performance may differ. Despite the widespread deployment of both data structures, there is no direct, systematic, system-aware comparison under real-world workloads. Most current literature compares data structures within the same algorithmic family rather than AVL Hash Trees and SMTs against each other. This thesis addresses that gap by measuring and comparing the performance of both structures.

1.2 Research Problem

The fundamental research problem addressed by this thesis is the absence of empirical performance data comparing AVL Hash Trees and SMTs under equivalent conditions. While both authenticated data structures serve critical roles in blockchain systems and provide cryptographic verification of data integrity, their comparative real-world performance remains unexplored in existing literature. This knowledge gap creates practical challenges for system architects and developers who must select appropriate data structures without access to direct performance comparisons.

The problem is significant because both structures theoretically have O(log n) complexity

for key operations, yet their different design philosophies suggest practical performance differences. AVL Hash Trees maintain balance through rotations and can benefit from spatial locality, while SMTs use fixed-depth sparse structures with deterministic paths and high parallelization potential. Both structures support inclusion (membership) and exclusion (non-membership) proofs, but with different implementation tradeoffs. Knowing which structure performs better under specific workload characteristics is valuable for optimizing blockchain transaction verification, state management, and authenticated data storage systems.

The research problem extends beyond theoretical analysis to practical implementation aspects, including CPU efficiency, memory consumption, and proof-generation overhead, all of which directly impact system throughput and scalability.

1.3 Research Goal

The primary goal of this thesis is to conduct a systematic, empirical performance comparison between AVL Hash Trees and SMTs as authenticated data structures. The performance comparison provides quantitative evidence regarding their real-world performance characteristics under equivalent implementation conditions and varied workloads. The research measures and compares CPU usage during insertion, and update operations, RAM usage and memory-consumption patterns, and proof-generation efficiency. Proof efficiency includes proof size and proof generation and verification time for inclusion and exclusion proofs.

1.4 Research Scope

The scope consists of implementing and evaluating both structures with comparable optimization effort in a controlled environment using the same programming language. The chosen language is Go programming language because of its relevance in decentralized and blockchain-related applications. The environment uses the same cryptographic hash functions and hardware infrastructure, with evaluation across varied workload patterns representative of real-world blockchain applications. Tests include baseline build scenarios, ordered prehashed build variants, insertion of new keys after the build, existing element updates, inclusion proof scenarios over existing keys, and exclusion proof scenarios over

absent keys.

This thesis has limitations. Performance measurements reflect the specific implementations created for this work and may not capture all optimization techniques documented in the literature. The selected workloads are representative but cannot exhaustively cover all real-world scenarios. The focus is on the data structures themselves rather than their integration into a complete blockchain system, therefore, factors such as network latency, consensus mechanisms, and storage-layer optimizations are outside the scope of this work. Additionally, while concurrency is identified in the literature as a significant source of performance improvement, the primary focus of this work is on single-threaded performance characteristics.

1.5 Research Questions

The thesis answers the following research questions:

- **RQ1:** In a direct comparison of AVL Hash Tree and Sparse Merkle Tree under equivalent insertion and update workloads, which tree has lower CPU usage and lower CPU time per operation?
- **RQ2:** How does RAM usage differ between AVL Hash Tree and Sparse Merkle Tree during insertion and update operations?
- **RQ3:** How do AVL Hash Tree and Sparse Merkle Tree compare in inclusion and exclusion proof efficiency, defined as proof size, proof generation time by the prover, and proof verification time by the verifier?

1.6 Novelty

The novelty of this thesis is a direct, systematic performance comparison between AVL Hash Trees and SMTs. The comparison is conducted under equivalent implementation conditions and controlled workloads. Existing literature has compared AVL-based authenticated dictionaries against each other or evaluated SMT implementations with different caching strategies.

Thus, while existing literature extensively optimizes and compares variants within the same structural family, no study has empirically compared these two fundamentally different

authenticated data structures against each other. Both structures are widely deployed in blockchain systems and share a theoretical O(log n) complexity.

This research addresses that gap by implementing both structures with comparable optimization effort and measuring their real-world performance across CPU usage, RAM usage, and proof-generation efficiency. The main contribution is quantitative performance data that enables informed selection between these two dominant authenticated data structures, based on empirical evidence rather than theoretical assumptions or indirect comparisons.

2 Literature Review

2.1 Theoretical Background

Authenticated data structures combine data organization with cryptographic commitments so that clients can verify inclusion and exclusion claims against a trusted anchor. Inclusion proofs show membership, while exclusion proofs show non-membership. In blockchain and other verifiable storage systems, this property enables state verification without trusting the storage provider.

This review focuses on two authenticated tree families: AVL Hash Trees and Sparse Merkle Trees. An AVL Hash Tree augments a self-balancing AVL search tree with node hashing. Updates may trigger rotations to preserve balance, and hash recomputation then propagates from modified nodes to the root.

An SMT maps a large key space to a fixed-depth binary Merkle structure, usually indexed by key-digest bits. Most nodes are represented implicitly through default hashes, and only accessed branches are materialized. This structure provides deterministic path length and efficient support for inclusion and exclusion proofs.

Because AVL Hash Trees and SMTs maintain authenticated state through different structural mechanisms (dynamic balancing versus fixed-depth sparsity), their behavior can diverge under different workloads. The literature in this chapter is therefore analyzed across common dimensions: update cost, proof characteristics, memory behavior, and implementation tradeoffs in blockchain and verifiable storage contexts.

2.2 Search Strategy

For this study, a variety of resources were utilized, including books, conference and research papers, articles, publications and yellow papers. A systematic literature review was performed using SCOPUS, IEEE Xplore, ACM Digital Library, SpringerLink, and ePrint Archive as primary databases. Supplementary grey literature was collected through

Google Scholar, supervisor recommendations, preprint databases such as arXiv, and code repositories such as GitHub. The focus was on works published primarily between 2010 and 2026 with occasional older works if they were highly relevant. The works tackle authenticated data structures, blockchain state verification, and performance optimizations of Merkle-like trees.

The Scopus filter used for finding Sparse Merkle Tree related papers was:

- `TITLE-ABS-KEY("sparse merkle tree*" OR "sparse hash tree*")`

The Scopus filter used for finding AVL Hash Tree related papers was:

- `TITLE-ABS-KEY("AVL tree*" OR "AVL hash tree*" OR "authenticated AVL tree*")`

Additionally, there was another, more general, Scopus filter utilized:

- `TITLE-ABS-KEY((("AVL" OR "binary search tree" OR "balanced tree") AND (authenticated OR merkle OR cryptographic)) OR "sparse merkle tree*")`

All provided Scopus queries were refined iteratively to remove unrelated works and target performance-oriented studies.

2.3 Inclusion and Exclusion Criteria

To ensure the relevance and quality of the sources and studies analyzed in this literature review, specific inclusion and exclusion criteria were established. The inclusion criteria include peer-reviewed or otherwise verifiable technical documents published between 2010 and 2026. Primary focus was on the performance or scalability aspects of AVL Hash Tree and Sparse Merkle Tree structures. Studies were further required to include implementations or evaluations related to blockchain systems or other verifiable storage systems and to be written in English. On the contrary, studies were excluded if full-text access was unavailable, if they dealt exclusively with unrelated cryptographic primitives, or if they consisted of non-peer-reviewed blog posts or opinion articles. Additionally, theoretical works lacking empirical performance evaluations were omitted to maintain a

practical and comparative perspective on system performance.

2.4 Selection Process

The selection process began with an initial screening, during which search results were filtered based on the relevance of their titles and abstracts. As a result of the initial screening 37 papers were selected. Studies that met the initial criteria were then subjected to a full-text review to assess their depth and overall relevance to the research topic. Following this stage, key information such as research findings, proposed systems, cryptographic schemes, noted limitations and performance metric were systematically extracted. In addition to the primary database searches, backward snowballing was employed to identify further relevant studies through reference lists of selected papers. Grey literature sources were also explored to ensure comprehensive coverage of the topic. The collected studies were subsequently synthesized by grouping and comparing them to identify prevailing trends, existing research gaps, and potential opportunities for further contribution. Finally, a manual verification was conducted to ensure the inclusion of only the most suitable and relevant studies in the final selection. Manual verification of the studies resulted in 22 studies being included in this literature review.

2.5 Comparative Analysis of the Selected Studies

Across systems, performance decomposes into several dimensions: update cost (e.g., rotations and re-hash span), proof size and shape with verification time for inclusion and exclusion proofs, and system-level amortization from caching strategies, batch updates, concurrency, and I/O layout. The selected papers can be grouped into two broad clusters: balanced AVL Hash Tree-based structures and SMT or trie-based structures.

2.5.1 AVL-Based Designs and Update-Heavy Authenticated Sets

Experimental evidence for AVL hashing comes from workloads with frequent updates to authenticated sets. Zhang et al. study TICK for Bitcoin UTXO maintenance [11]. TICK replaces a lexicographically ordered Merkle tree with an AVL hash tree. In their evaluation, updating after a new block takes about 0.3 seconds with the AVL hash tree. The ordered Merkle tree baseline takes about 84 seconds. TICK also keeps proofs small. Inclusion

proofs are about 1.6 KB, while exclusion proofs are at most about 3.2 KB. More than 99% of proofs are generated within 1 ms. These results show that AVL hashing can support frequent insertions and deletions with compact proofs and fast proof generation.

In dynamic cloud-storage verification, RB-MAT (Rank-Based Merkle AVL Tree) introduces a rank-based Merkle-AVL tree to make queries explicit, reduce rebalancing overhead, and enable batch-update verification [12]. In RB-MAT, each node stores a rank value indicating the number of leaves in its subtree, allowing efficient block lookup and verification of block indices. Qi, Tang, and Huang applied this structure to verify integrity and authenticity of stored data at block level. The authors later extended this work with RBMV-MAT (Rank-Based Multi-Version Merkle AVL Tree), which adds multi-version support via a special lock and relaxed balancing to reduce rebalancing frequency, improving throughput for concurrent verifiable updates across versions [13].

In 2023, Wang, Zhang, Qin, Ma, and Huang proposed a verifiable symmetric searchable encryption scheme based on an AVL tree (VSSE-AVL) [14]. The scheme indexes complete keyword labels rather than segmented trie paths, which avoids degeneration when many keywords share long prefixes and reduces storage overhead relative to trie- based designs. The paper also adds verification for empty search results and protection against substitution attacks, and experimentally reports lower storage, search, and verification costs than verifiable SSE-2. SSE-2 is Curtmola et al.'s [15] static index-based searchable symmetric encryption construction, which allows a server to search encrypted files using client-generated trapdoors without learning the plaintext contents.

Reyzin et al. report that their AVL plus authenticated dictionary gives shorter proofs than two earlier authenticated dictionary designs [16]. The proofs are about 1.4 times shorter than authenticated skip lists, and about 2.5 times shorter than red black trees. The same comparison also applies to prover and verifier time in their implementation. When many operations are verified together, their proof compression reduces total proof size by about another factor of two. This reduction is relative to sending separate proofs for each operation. In a simulated blockchain verification workload, their verifier ran about 20 times faster than a full verifier using an on disk data structure.

Overall, these studies indicate that AVL-based balancing can improve verifier efficiency and

general system performance, especially when workload locality is high and multi-version or batch verification is required.

2.5.2 Sparse Merkle Trees, Non-Membership, and Batch Updates

SMTs are relevant when applications need predictable proof shape, compact exclusion proofs, and adjustable memory use. Dahlberg, Pulls, and Peeters define SMTs recursively and give security models for them [17]. They also present three cache strategies, called B, B- (B minus), and B+ (B plus). These caches support inclusion and exclusion proofs in practically constant time. The cache choice lets implementations trade memory use against proof generation time. In their evaluation, proofs take less than four milliseconds with SHA 512/256. This holds even when the cache is smaller than the authenticated dataset. They also analyze security in the multi instance setting.

In his 2016 dissertation, Östersjö proposed a general SMT representation with explicit space-time tradeoff models tied to cache policy and tree layout [18]. The work also shows how SMTs integrate with other authenticated data structures, such as Balloon. Because full SMT representation is intractably large, the dissertation proposes using a non-authenticated auxiliary structure (e.g., a map) to maintain inserted key-value pairs.

At system scale, concurrency and batching are decisive. Kalidhindi, Kazorian, Khera, and Pari (2018) proposed Angela, a sparse, distributed, and highly concurrent Merkle tree [19], reporting roughly 2x speedup over a naive global-lock approach used in systems such as Trillian. The architecture supports distribution across Amazon EC2 instances via Ray and Amazon Aurora. In 2021, Chen, Qi, Du, Zhang, and Jin proposed PEEP (Parallel Execution Engine for Permissioned Blockchain Systems) [20]. PEEP offers deterministic scheduling and parallel state-trie updates on radix and Merkle tries such as MPT, reducing trie-depth costs that otherwise dominate sequential order-execute pipelines.

Taken together, these studies show that SMTs are particularly well suited to settings where fast exclusion proofs, predictable proof structure, and parallelizable updates are important. At the same time, the reviewed literature does not support a general claim that SMTs are inherently better or worse than AVL-based hash trees, especially with respect to memory usage under comparable conditions. Reported space consumption varies with representation, auxiliary indexing, cache policy, and workload, indicating that the more

appropriate choice depends on system requirements rather than a universal advantage of either structure.

2.5.3 I/O and Workload as Performance Drivers

The literature also shows that I/O and proof-layer architecture often dominate micro-level AVL-versus-SMT tradeoffs. At the storage layer, Burke et al. (2025) quantified compute and metadata-I/O costs on the critical path for storage-level hash trees [21]. They show that optimal tree shape is workload dependent, reduce the design problem to optimal prefix codes, and propose Dynamic Merkle Trees (DMTs) that self-adjust during operation. The authors report up to 2.2x throughput and latency improvements over state-of-the-art implementations in realistic AWS EC2 NVMe SSD settings.

The QMDB study by Zhang et al. (2025) bridges the customary separation between key-value storage and Merkle proof generation [22]. The study proposes an append-only twig-based authenticated data structure, where a twig is a fixed-size immutable subtree of grouped updates. The design performs each read with at most one SSD access, each update with constant-time I/O complexity, and all merkleization in memory. The authors report 6x throughput over RocksDB and 8x over NOMT, plus up to 2.28M state updates per second. These findings indicate that optimizing SSD IOPS and write amplification can outweigh asymptotic data-structure differences.

Shadab et al. (2023) researched hardware implementations of Merkle trees [23] and proposed HMT, a hardware-centric hybrid Bonsai Merkle tree on FPGAs. HMT uses level-partitioned write-back caches, speculative buffers, and parallel write-backs to achieve up to 7x bandwidth improvement. The study also reports 4.5x subsystem latency reduction and 14% end-to-end speedup on a Xilinx U200 secure-memory prototype. Ponnapalli's 2023 Ph.D. dissertation similarly explores reducing I/O bottlenecks through storage-layout customization [24].

Taken together, these studies, along with [24], show that I/O elimination, batching, and pipeline design are major drivers of end-to-end performance.

2.5.4 Proof Generation as a Basis of Authenticated Data Structures

Proof ecosystems and architectural placement determine which structure's strengths matter most. Verdict, proposed by Tzialla et al. in 2021, composes indexed Merkle trees with Phalanx, a SNARK tailored for epoch-level amortization [25]. A SNARK (Succinct Non-Interactive Argument of Knowledge) is a fixed-size, stateless proof that can be verified without a third party. Verdict delivers publicly verifiable proofs of correct operation. Clients retain O(log n) inclusion and exclusion checks while epoch proofs remain constant-size and verify in milliseconds, shifting heavier work to the prover [26].

The Alphabill Yellowpaper by Buldas, Saarepera, Laanoja, and Truu demonstrates how production-grade ledgers layer multiple tree-derived certificates [27]. In this context, a certificate is a compact proof of inclusion or exclusion in an authenticated data structure. These include Unit, State, Shard, and Unicity certificates, each with explicit creation and verification routines. This implies that the best tree choice depends on certificate placement and traffic patterns. In addition, Buldas, Truu, Laanoja, and Rogojin (2025) presented the Unicity Execution Layer, which moves almost all work off-chain and leaves only double-spend prevention on-chain around an append-only, globally consistent spent-state dictionary [28].

Complementary results from Iris (Stefanov et al., 2012) show that with enterprise-side caching and a Proofs of Retrievability (PoR) protocol, an authenticated file system can sustain high throughput [29], approximately 260 MB/s end-to-end for 100 simultaneous clients while preserving freshness and retrievability. The paper suggests that careful placement and amortization of verification are often more decisive than selecting between balanced AVL-like trees and SMTs. For historical authenticated queries, Persistent Authenticated Dictionaries (PADs) formalize the question "was e in S at time t," as studied by Anagnostopoulos et al. (2001) [30]. Although that work focuses on Red-Black trees and skip-list-based PADs, it notes that other balanced trees, such as AVL or 2-3 trees, are also suitable.

Finally, recent optimization work by Pham et al. (2025) surveys SMT, Merkle Sum, and Patricia variants and proposes a hybrid Merkle Sparse_Sum Tree [31]. The proposed structure reports promising performance for finance and banking workloads, pointing to

a broader design space where sparsity and aggregation are combined. Production SMT systems also continue to evolve. Gao et al. (2021) presented Jellyfish Merkle Tree (JMT), a space- and computation-efficient SMT optimized for Diem-like blockchains [8], and a practical example of SMT deployment in production contexts.

2.5.5 Conclusion

This literature review examined current research on AVL Hash Trees and Sparse Merkle Trees as authenticated data structures. It analyzed 22 selected studies published mainly between 2010 and 2026. The review identifies a research gap in direct comparison between the two structures. The gap exists because prior work usually studies each structure within its own design family. AVL research often compares authenticated AVL dictionaries with other balanced tree designs. SMT research usually evaluates sparse tree variants, cache strategies, or concurrent update methods. These studies show useful improvements, but they do not test AVL Hash Trees and SMTs under the same conditions. Therefore, both data structures should be evaluated in a controlled performance environment.

The reviewed literature reveals a clear pattern where research contributions focus on optimizing and comparing variants within the same algorithmic or structural family. Studies on AVL Hash Trees, such as TICK client [11], RB-MAT and its derivatives [13, 12] and authenticated dynamic dictionaries [16] demonstrate improvements over other balanced-tree approaches or ordered Merkle trees. These studies report performance gains such as sub-second batch updates and 1.4-2.5x reductions in proof sizes.

Similarly, research on Sparse Merkle Trees, including work on efficient cache strategies [17], distributed implementations like Angela [19], and optimized variants like JMT [8], focuses on comparing SMT derivatives with each other. Additionally, these variants and implementations are compared against naïve implementations, achieving constant-time proof generation in under four milliseconds and 2x speed-ups through concurrency optimizations.

Generic frameworks can build authenticated data structures from ordinary data structure code. Miller et al. proposed the $\lambda\bullet$ (lambda-auth) programming language for this purpose [32]. The language helps developers define authenticated operations and generate the corresponding proof logic. This makes it easier to implement authenticated versions of

different structures. However, such frameworks still do not provide a direct systematic comparison between AVL Hash Trees and SMTs.

The literature does provide insights into theoretical strengths, such as that AVL hash trees excel in update-heavy workloads with special locality and batch operations. Conversely Sparse Merkle Trees demonstrated advantages when uniform proof sizes, efficient exclusion proofs, and parallelizability are important. However, when theory and separate optimization studies do not show how the two structures perform under the same conditions, direct empirical evaluation is more appropriate.

No study in the reviewed literature directly compares the performance characteristics of AVL Hash Trees against Sparse Merkle Trees under equivalent conditions and workloads. Both structures have the same asymptotic O(log n) complexity for key operations, but this does not determine actual runtime or memory behavior. Therefore, it is difficult to draw practical conclusions from asymptotic complexity alone. The two structures organize data differently, so their measured performance may still differ under the same workloads. Real-world performance depends on constant factors, memory access patterns, cache efficiency, CPU usage efficiency, and proof generation/verification overhead. Provided factors can only be accurately assessed through direct benchmarking under real-world conditions.

This literature review therefore establishes the foundation and justification for the present research that is a systematic performance comparison of AVL Hash Trees and Sparse Merkle Trees. By implementing both structures with equivalent optimization efforts and evaluating them across varied workloads, this thesis aims to provide empirical evidence that directly addresses the identified research gap.

3 Methodology

This chapter describes how the study was designed and executed. It explains the benchmark design, the compared implementations, the input generation process, the analysis protocol, and the main methodological limits. It also discusses validity, reliability, and ethical considerations.

3.1 Research Design

The study used a quantitative benchmark design. It compared two authenticated in-memory data structures under controlled laboratory conditions. The compared structures were an AVL Hash Tree and a Sparse Merkle Tree with path compression. Here, path compression means that chains of single child SMT nodes are stored as one path segment. This reduces the number of explicit nodes while preserving the logical key path. The design was comparative because both structures were subjected to the same workloads and measurement rules. It was also explanatory because the benchmark was intended to show not only which tree was faster or more space efficient, but also why differences appeared under particular workload patterns.

This design matches the research questions introduced in Chapter 1. RQ1 asks which structure has lower computational cost during build and update workloads. It is therefore mapped to total measured phase time, average time per operation, insert timing buckets, and representative diagnostic CPU and heap profiles. RQ2 asks how RAM usage differs between the structures. It is mapped to the change in allocated memory during the measured phase, total allocated memory during the phase, and heap object count recorded after the phase. RQ3 asks how proof efficiency differs for inclusion and exclusion proofs. It is mapped to average serialized proof size, proof generation time, and proof verification time. Here, a phase means one measured benchmark segment, such as a build run, an update run, proof generation, or proof verification. This mapping links each research question to directly observed variables instead of to high level intuition alone.

A controlled benchmark is appropriate for this thesis because the aim is not to evaluate a full blockchain node. It is to compare the cost profile of two authenticated dictionaries as implemented in the project. End to end blockchain integration would have coupled tree behavior to storage layout, I/O policy, networking, transaction execution, and consensus choices [21, 22]. Those factors are important in production systems, but they would have obscured the structural comparison that the thesis set out to make. By holding the surrounding environment constant and changing the tree implementation, the study increased internal validity.

The design also focused on comparison at the structure level because both trees can be used inside larger systems in many different ways. An AVL Hash Tree can appear inside update intensive authenticated indexes. An SMT can appear inside sparse state commitment systems or storage services with different caching policies. A direct comparison at the implementation level provided a narrower but more defensible methodological target than trying to emulate every larger architecture cited in the literature [19, 13, 8]. The benchmark therefore asks a limited question. Given equal language, equal hash function, equal measurement setup, and equal hardware, how do these two concrete authenticated dictionaries behave?

The design targeted a specific set of properties at the structure level and deliberately excluded others from the benchmark scope. It measured in-memory build cost, update cost on a prebuilt structure, proof generation cost, proof verification cost, and memory behavior under controlled workloads. It did not measure persistence overhead, disk amplification, distributed execution, concurrent mutation, consensus interaction, transport compression, or long running service behavior. These omissions were deliberate, as they reduced realism in exchange for a cleaner comparison of authenticated structure behavior.

3.1.1 Benchmark Object of Study

The benchmark object of study was not the abstract AVL family and the abstract SMT family in every possible form. It was two concrete Go implementations developed for this thesis (see Appendix 2). The first implementation was an AVL Hash Tree that maintained authenticated balanced search tree state through node hashes, subtree hashes, and AVL rotations. The second implementation was a Sparse Merkle Tree with path compression

that maintained authenticated sparse state through compressed bit paths, branch nodes, and leaf nodes.

This distinction matters for interpretation. The study does not claim that every authenticated dictionary based on AVL behaves like the measured AVL prototype. It also does not claim that every SMT variant behaves like the measured SMT prototype. The benchmark instead compares two implemented authenticated dictionaries under equal measurement conditions. That is the level at which evidence can be collected and defended, because the benchmark measures those concrete implementations directly.

Comparability in the benchmark came from matched workloads, shared measurement conditions, and comparable optimization effort rather than from identical internal design. The workloads were chosen to compare both structures at the same tree scales. This made tree size the main controlled variable and allowed scaling differences to be observed. Both implementations were written in the same language. Both used SHA-256. Both were measured through the same benchmark framework on the same machine. Both used the same key width, the same value width, and the same benchmark workloads. Both produced a root commitment and supported proof generation and proof verification. These equalities did not erase structural differences, but they reduced sources of unfairness that would otherwise weaken the comparison. Examples include using different programming languages, different SHA-256 implementations, or different cryptographic libraries.

3.2 Data Collection Methods

The empirical data were produced by executing benchmark workloads on the two software prototypes. Both prototypes were in-memory authenticated key value stores. This implementation style kept the comparison focused on behavior at the structure level. It also excluded storage engine effects that often dominate Merkleized workloads at system scale [21, 22].

Table 1. Core implementation decisions.

Aspect	**Implementation Choice**	**Reason**
Programming language	Go for both implementations	Same compiler and runtime context with strong tooling support
Execution model	Single-threaded and in-memory	Isolation of cost at the structure level
Hash function	SHA-256 for both trees	Same cryptographic work and fixed 256-bit digest width
Key format	Fixed 32-byte keys	Stable key width and common hashing pattern on the caller side
Value format	Fixed 32-byte values	Stable payload width across all scenarios
Benchmark interface	Build by insert, update existing entries, generate proof, and verify proof	Same measured actions for both trees
AVL design	Balanced search tree with cached heights, node hashes, and subtree hashes	Canonical AVL authenticated behavior
SMT design	Sparse Merkle Tree with path compression, branch nodes, and leaf nodes	Sparse Merkle semantics with a reduced explicit node count
Included scope	Build, update, root maintenance, proof generation, and proof verification	Direct support for RQ1 to RQ3
Excluded scope	Persistence, networking, consensus, concurrent mutation, history, and disk layout tuning	Removal of benchmark influences unrelated to structure

The fixed 32-byte value format reflects a common pattern in authenticated state systems.

The raw payload is often hashed before insertion into the tree. The tree then stores the SHA-256 digest rather than raw bytes, strings, JSON documents, or other application data. This made the benchmark value format representative while keeping payload width constant across both implementations.

3.2.1 Shared Measurement Interface

The data collection process used one benchmark framework for both trees. The framework did not compare unrelated public APIs. It compared equivalent actions defined by the benchmark. Build scenarios inserted new key and value pairs into empty structures. Update scenarios first created a known initial state and then either inserted new keys or reapplied existing keys with replacement values. Proof scenarios operated on a prebuilt state. They selected targets from present keys for inclusion proofs and from absent keys for exclusion proofs. The benchmark then generated proof objects and verified them against the recorded root.

Root recomputation was part of the same measurement flow. After each structural change, both implementations propagated hash updates to the root according to their own internal rules. The benchmark did not treat root maintenance as an optional background task. It treated it as intrinsic to authenticated dictionary behavior. This point was important because an authenticated structure that updates values but delays root maintenance would not be equivalent to one that maintains an immediately verifiable commitment after every change.

The framework also separated the work performed by the prover and the verifier. Proof generation was timed as the cost of walking the structure and constructing the proof material. Proof verification was timed as a separate step that consumed the recorded root and the generated proof. This separation made it possible to compare not only how expensive proofs were to construct, but also how expensive they were for a verifier to check. That distinction mattered because a structure could produce compact proofs yet still impose higher verification cost, or vice versa.

The raw benchmark result rows were captured before summary statistics were computed. Each repetition produced timing, memory, and heap results for the measured phase. Representative runs also produced diagnostic profiles for later inspection. The test result

CSV files, and related benchmark artifacts were kept in the repository, so the underlying data can be inspected directly.

3.2.2 AVL Implementation Evaluated in This Study

The AVL implementation evaluated in this study was an authenticated balanced binary search tree. Each node stored a key, a value representation, height metadata related to balance, a node hash for the local key and value tuple, and a subtree hash for the authenticated subtree rooted at that node. This design meant that an insertion or update could change the tree shape, balance metadata, and authenticated hashes at the same time. When an insertion or update modified one part of the tree, hashes on the affected path had to be recomputed before a new root commitment could be reported.

Balance maintenance was a central part of the measured behavior. The AVL tree used standard rotations to restore balance after structural changes. A rotation changed relationships between parent and child nodes and therefore changed the subtree boundaries whose hashes had to be recomputed. The benchmark therefore measured more than plain binary search tree lookup and replacement cost. It also measured the structural repair work that an authenticated AVL tree had to perform to preserve logarithmic height and a valid root hash at the same time.

The AVL implementation was methodologically useful because its behavior depended directly on key ordering and current balance. Its search paths depended on key ordering and current balance. For that reason, the AVL prototype was particularly relevant to the ordered prehashed build variants and to existing element updates in prebuilt states.

3.2.3 SMT Implementation Evaluated in This Study

The SMT implementation evaluated in this study was an authenticated Sparse Merkle Tree with path compression over logical 256-bit key paths. Conceptually, each hashed key defined a route through a binary space. In a naive sparse Merkle structure, every bit position could correspond to an explicit branch. The implementation used in this study did not materialize the full sparse shape. Instead, it represented only the branch nodes and leaf nodes required by the inserted key set. This kept the explicit in-memory structure smaller than the full logical key space.

Path compression was a critical property of the implemented SMT. Compressed path segments allowed one node to capture a portion of the path without branching that would otherwise require many explicit intermediate nodes. This reduced the number of materialized nodes while preserving the commitment structure required by a sparse Merkle tree. In methodological terms, path compression mattered because the study did not compare an AVL tree against a worst case textbook SMT. It compared the AVL tree against an implemented sparse Merkle design that already incorporated one of the most important practical ideas for saving space from the SMT literature [18, 8].

The SMT structure responded differently to updates than the AVL structure. Updates were driven by bitwise path decisions rather than by ordered search tree position. In a standard SMT, proofs follow a fixed key depth, so proof size is constant for a fixed hash length. In the path compressed SMT, long unbranched paths are compressed and only materialized branches are stored. As a result, proof size is not fixed by the 256-bit key space alone and can grow with the number of stored elements. The compressed branching logic and path materialization rules also affect how many nodes are visited and recomputed during updates and proof construction. Those properties made the SMT implementation a useful counterpoint to the AVL implementation in analysis oriented toward proofs and memory.

3.2.4 Go Tooling and Profiling

Go was selected for both implementations because it was relevant to software related to blockchain and because it offered consistent runtime tooling for measurement. Timing data were collected with Go based benchmark instrumentation and Go runtime time measurement. Allocation and heap data were collected from runtime benchmark outputs and profile capture. This kept the measurement stack within the same runtime environment as the compared implementations.

The `pprof` profiler was used as a diagnostic tool rather than as a primary metric. In practical terms, `pprof` recorded sampled CPU profiles and heap profiles. Those profiles helped identify where time and memory were concentrated inside each implementation. They were useful for explaining major differences after the quantitative results were collected. They were not themselves the main evidence for RQ1 to RQ3. The primary evidence remained the benchmark timings, proof sizes, and memory statistics produced by the structured

benchmark campaign.

3.2.5 Benchmark Environment

All benchmarks were run on one dedicated machine. The machine used an AMD Ryzen 7 9800X3D processor with 8 cores and 16 threads. Its maximum clock was 4.7 GHz. The processor had 640 KB of L1 cache, 6 MB of L2 cache, and 96 MB of L3 cache. The system had 48 GB of DDR5 memory at 5600 MHz in a two module configuration. Storage used an ADATA LEGEND 860 1 TB NVMe SSD with sequential read throughput of 6,000 MB/s and sequential write throughput of 4,000 MB/s. The software environment was documented as well. Windows 11 Pro version 25H2 (build 26200.8037) served as the host operating system. Benchmark binaries were compiled and executed inside Ubuntu 24.04.4 LTS running under Windows Subsystem for Linux 2 (WSL2). The WSL2 kernel was Linux 6.6.87.2-microsoft-standard-WSL2, and the implementations were built and tested with the standard Go 1.25.6 tools for `linux/amd64`.

These details mattered because the compared structures were pointer heavy authenticated in-memory data structures. Cache sizes influenced how often traversals, rotations, hash propagation, and proof walks could reuse recently touched nodes. This was especially relevant during ordered construction, insertion after the build, existing element updates, and proof generation over large in-memory states. A processor with a large shared L3 cache could reduce some memory latency penalties that would otherwise dominate tree traversal cost.

Main memory characteristics also mattered. The benchmark campaign included large prebuilt states, repeated structural updates, and proof construction over live in-memory dictionaries. Memory capacity had to be large enough that all tested states fit comfortably in RAM without forced paging. Memory bandwidth influenced how quickly the runtime could move data through allocation, pointer chasing, and garbage collection activity during long benchmark runs. For that reason, documenting RAM capacity and speed was part of the methodological record.

The SSD specifications were recorded even though the measured tree operations were in-memory. The storage device still shaped the surrounding research environment. The same machine handled source code, test execution, profile capture, and CSV result storage.

Fast storage therefore contributed to the practicality of the benchmark workflow and artifact handling, even though the direct benchmark focus remained in-memory execution rather than disk I/O.

The machine was kept otherwise idle during measurements. This reduced interference from unrelated background work and helped keep the timing environment stable across repetitions. The benchmark campaign also remained strictly single-threaded in scope. It measured one operation stream at a time and did not evaluate concurrent mutation or distributed execution. That choice matched the intended comparison at the structure level and kept the environment consistent with the defined research questions.

3.3 Benchmark Input Generation and Workload Selection

The study generated benchmark inputs and operation sequences in a controlled and reproducible way. The workload design covered keys, values, build orders, update targets, and proof targets. All of these elements were prepared so that both trees received equivalent workloads.

3.3.1 Key and Value Generation

Deterministic generation was used to create reproducible benchmark inputs. Keys were derived by hashing an incrementing counter with SHA-256 and then using the resulting digest as the benchmark key. This produced a fixed 32-byte key representation throughout the benchmark campaign and kept key preparation identical across scenarios. SHA-256 was used because it has fixed width digest, is collision resistant, and designed to behave like a uniformly distributed value over the 256-bit output space. This made the generated keys distributed over the 256-bit output space while still being fully reproducible.

The same generated key sets were used for both structures. In the baseline build scenarios, the generated hashes were inserted directly as they were produced.

For the ordered prehashed variants, the benchmark did not generate a second key set. It reused the same hashed keys and changed only their insertion order. Before insertion, the keys were sorted using the ordering rule of the tree being tested. This preserved the same 32-byte key representation that was used elsewhere in the campaign. It allowed insertion

order effects to be tested without introducing a second key format or a different external key preparation step. Deterministic replay of these generated key sets and ordered variants was therefore a direct support for reliability.

Values were also fixed at 32 bytes. This kept payload width stable throughout the campaign. A variable payload size would have changed node size, hash input size, and allocation behavior at the same time as the tree workload. That would have weakened the ability to attribute observed differences to the tree structures themselves. Using a constant payload size therefore improved comparability across both build and update workloads.

The benchmark inputs were generated before the measured phases and were then replayed consistently. This mattered for both fairness and interpretability. A repeated build or update campaign should measure the same logical work on both trees. If input generation were interleaved with measurement in a way specific to one structure, the comparison would become unfair and incorrect.

3.3.2 State Sizes and Preloaded States

The benchmark campaign used a stepped size series that covered very small, medium, and high (millions of entries) states. The main build scenarios covered 10,000, 50,000, 100,000, 200,000, 300,000, 500,000, 700,000, 900,000, 1 million, 2 million, 5 million, 10 million, and 25 million entries. This gave denser coverage in the lower and middle ranges and larger jumps once the benchmark reached million scale states. For both implementations, insert only builds and ordered prehashed insert only builds were executed across that same size series. This made it possible to compare ordinary construction and construction under tree specific key ordering across the full range of evaluated states.

This size design was methodologically useful because authenticated tree behavior can change with scale even when theoretical asymptotic complexity remains the same. A structure that looks competitive at tens of thousands of elements may behave differently at hundreds of thousands or millions of elements once cache pressure, heap growth, and path length effects become more visible. Using a stepped progression therefore made it possible to observe both local trend changes and broader scaling behavior without making the benchmark campaign impractically large.

Preloaded scenarios reused that stepped size series according to the measured task. The proof only benchmarks were executed after prebuilds of 10,000, 50,000, 100,000, 200,000, 300,000, 500,000, 700,000, 900,000, 1 million, 2 million, 5 million, 10 million, and 25 million entries. The insertion and existing element update scenarios after the build also used that stepped size series for their prebuilt states, and each measured workload applied additional operations equal to 20% of the prebuilt state size. This separated construction cost from maintenance and proof cost on a prebuilt structure. A proof or modification scenario after the build should measure behavior on an already formed authenticated state rather than rebuild cost from empty.

3.3.3 Update and Proof Workload Selection

The insertion scenarios after the build added new keys to a prebuilt structure. Their purpose was to measure how each authenticated dictionary behaved when it had to continue growing after the initial construction phase.

The existing element update scenarios revisited keys that were already present in the current state. In the implemented benchmark suite, these keys were revisited from the deterministically generated key set rather than from a separate skewed or random update distribution. The measured work therefore included the search path, the value replacement, and the required hash propagation along the affected path. Because these scenarios rewrote existing keys, final tree size did not grow during the measured phase.

Proof targets were selected separately from update targets. Inclusion proof measurements used keys sampled from the current tree state, while exclusion proof measurements used absent keys derived outside that state. Proof sample count was calculated from the current state size and the configured sample ratio, rounded to the nearest whole proof with a minimum of one sampled key and capped by the number of available keys. The dedicated proof only scenarios used sample ratios rather than fixed proof counts. From 10,000 entries through 1 million entries, 5% of the current tree keys were sampled. For example, a 10,000 entry tree used 500 proof targets. At 2 million and 5 million entries, 3% of keys were sampled. At 10 million and 25 million entries, 2% of keys were sampled. These ratios kept the absolute number of proof queries large enough to stabilize timing results while keeping very large proof campaigns computationally manageable.

3.4 Benchmark Scenarios

The final scenario set was chosen to cover the main benchmark questions raised by the literature and by the thesis research questions. The same scenario definitions were then applied to both trees so that they could be compared fairly under the same workloads and operational situations. The benchmark campaign therefore combined build, update, and proof scenarios rather than assigning different workflows to different structures.

Table 2. Benchmark scenarios and their rationale.

Scenario	**Workload Definition**	**Reason for Inclusion**
Insert only random build	Hash keys with SHA-256 and insert the resulting hashes directly	Provides the baseline construction case
Ordered prehashed insert only build	Use the same hashed keys, sort them by the internal comparison order of the evaluated tree, and then insert them	Tests sensitivity to insertion order
Post build insertion	Insert new keys into a preloaded tree	Measures continued growth cost after initial construction
Existing element update	Update values for keys already present in a preloaded tree	Measures update cost for already present keys
Inclusion proof	Generate and verify proofs for present keys sampled from the current prebuilt state	Measures inclusion proof efficiency
Exclusion proof	Generate and verify proofs for absent keys sampled outside the current prebuilt state	Measures exclusion proof efficiency

The build scenarios isolated construction behavior under two different insertion orders. The insert only full build used SHA-256 hashes generated from the deterministic key source and inserted those hashes directly. The ordered prehashed build reused the same hashed keys. Before insertion, it sorted those keys using the ordering rule of the tree being tested.

Taken together, these scenarios separated baseline construction from deliberately ordered insertion while keeping the generated key material fixed.

The two modification scenarios after the build isolated different forms of work on a prebuilt structure. Post build insertion measured the cost of extending an existing authenticated state with new keys. Existing element update measured the cost of revisiting keys that were already present in that state. That distinction was relevant in a comparison between a balanced search tree and a Sparse Merkle structure with path compression.

The proof scenarios isolated verifier relevant behavior. Inclusion proofs showed how efficiently each structure could prove the presence of a current key. Exclusion proofs showed how efficiently each structure could prove the absence of a queried key. These scenarios were necessary because proof behavior does not always track update behavior. A structure could update quickly but still produce larger or slower proofs, or the reverse. Measuring proof generation and verification for present and absent keys therefore addressed the third research question more directly than build or update workloads alone could.

3.5 Data Analysis Methods

The collected data were analyzed quantitatively. The primary aim was comparative interpretation rather than formal hypothesis testing. The analysis therefore focused on comparing the AVL tree and the SMT across matched scenarios and state sizes.

Table 3. Primary benchmark metrics.

Metric Group	**Operational Definition**	**Purpose**
Operation time	Total measured phase time and average time per build or update operation within a scenario	Primary CPU efficiency measure for RQ1
Memory behavior	Change in allocated memory, total allocated memory, and heap object count after the phase	Main measures related to RAM for RQ2
Proof size	Average serialized proof bytes	Main space measure for RQ3
Proof generation time	Total and average time to construct sampled inclusion or exclusion proofs	Efficiency measure for the prover
Proof verification time	Average time to verify an inclusion or exclusion proof against the recorded root hash	Efficiency measure for the verifier
Supporting diagnostics	Average durations for ten equal insert progress buckets and representative `pprof` profiles	Explains variation within a measured phase

Total measured phase time and average time per operation were treated as the primary CPU efficiency measures. CPU percentage was not sufficient on its own because the benchmark ran as a single-threaded process that drove one CPU core to full utilization during the measured interval. In that CPU-bound setting, the program's working time effectively matched the busy time of that core and was not materially limited by disk I/O or similar external factors. Thus total CPU utilization percentage alone could not distinguish how efficiently each implementation converted CPU time into completed work. Average time per build or update operation provided a normalized view of computational cost across scenarios of different sizes. Total phase time complemented that view by showing the aggregate cost of completing the measured workload. Together they aligned directly with

the practical question behind RQ1, which was how much authenticated work each structure required to perform the same logical task.

Memory analysis used two complementary views. Change in allocated memory measured the net difference in live allocated memory between the start and end of the measured phase. Total allocated memory measured the cumulative amount of memory allocated during the phase, including memory later reclaimed by garbage collection. Heap object count recorded how many heap objects remained after the phase finished. These views need to be interpreted together. A structure could allocate heavily but release many temporary objects quickly, or it could end the phase with a larger retained heap and more remaining objects.

Proof analysis also required a combined reading of several metrics. Proof size describes how much authenticated material has to be transmitted or stored. Proof generation time describes the effort required from the prover. Proof verification time describes the effort required from the verifier. None of these metrics alone is sufficient. A small proof that is expensive to generate or verify is not necessarily better. Likewise, a fast proof routine that produces much larger serialized proofs could shift cost to storage or transport. Because total proof time also depends on sample count, cross scenario comparisons emphasized average generation time, average verification time, and average serialized proof size. For that reason, proof efficiency in this study was interpreted as the joint behavior of size, generation time, and verification time rather than as a single value.

Average values were interpreted together with insert timing buckets. For insertion phases, the measured work was divided into ten equal progress segments covering 0–10%, 10–20%, and so on up to 90–100% of the phase, and an average duration was recorded for each segment. These buckets were based on progress through the measured phase rather than on sorted latency percentiles. This made it possible to observe whether work remained stable, became faster, or became slower as a large build or existing element update campaign progressed. The bucketed view therefore supplemented the overall averages by showing timing trends within a phase rather than only one aggregate value.

Supporting diagnostics had a secondary role. They were used to explain observed differences, not to replace the primary metrics. The benchmark framework did not instrument internal counters such as rotations, visited nodes, or hash recomputation counts. For example, a

higher update time might instead be interpreted alongside slower insert buckets from a later phase or a `pprof` hotspot in insertion, path traversal, or hash maintenance code. A `pprof` profile could then show where time or memory was concentrated inside that implementation. This explanatory layer was useful because a benchmark chapter should describe not only what differed, but also which internal mechanisms most plausibly produced the difference.

3.6 Experimental Procedure

Each benchmark scenario was repeated 10 times. The same repetition count was used for both trees in every scenario and at every state size. This kept the comparison symmetric and ensured that one implementation did not benefit from a larger or smaller observation set than the other.

Each repetition began from a known input sequence. Fresh structures were created for every repetition. Build scenarios started from an empty tree and inserted the scenario specific key order. Update and proof scenarios first constructed the required preloaded state from the fixed input sequence and only then measured the target phase. This separation prevented earlier repetitions from contaminating later ones and kept the measured work aligned with the stated scenario definitions.

The benchmark framework recorded raw benchmark result rows before any aggregation into means, medians, standard deviations, and percentile ranges. Timing, allocation, heap, and proof related measurements were captured for each repetition. After the measured phase, results were written to the raw output set and later combined into summary tables and CSV artifacts. This order was important. It preserved the original measurements and allowed later checks of overall averages, bucketed timing trends, and any unusual runs without reconstructing the benchmark campaign from summaries alone.

Proof generation and proof verification were executed as distinct measured steps. For a proof scenario, the prebuilt structure first generated proof material for the selected targets. Verification then consumed the generated proof and the recorded root as a separate verifier task. This procedure prevented prover work and verifier work from being conflated in one aggregate timing. It also matched the way authenticated proofs are used in practice, where the party generating a proof and the party verifying it often have different roles and cost

concerns.

Garbage collection and runtime cleanup were performed between major repetitions so that each measured phase started from a consistent runtime state as far as practical. Representative runs also produced CPU and heap profiles for diagnostic analysis. These profiles were collected after the primary benchmark design had already defined what the main metrics would be. They supported interpretation, but they did not alter the repetition structure or replace the raw benchmark outputs.

Scenario execution followed a consistent fixed order across the benchmark campaign. The same order was used for both trees. This kept the procedural treatment equal across implementations and simplified reproduction of the campaign. The tradeoff was that fixed ordering could not remove every change in measurement conditions over time. That limitation was accepted because procedural symmetry across the two trees was more important than an unverified alternative ordering policy.

3.7 Validity, Reliability, and Methodological Limitations

3.7.1 Internal Validity

Internal validity was strengthened by keeping the core measurement conditions equal across both implementations. The two trees were written in the same language. They used the same SHA-256 hash function. They were exercised through the same benchmark framework on the same hardware. They used the same generated keys, the same generated values, the same state sizes, and the same scenario definitions. These controls did not guarantee perfect neutrality, but they reduced several major threats that often complicate performance comparisons across systems.

Correctness validation also supported internal validity. Benchmarking a broken authenticated structure would make any performance result meaningless. For that reason, both implementations were subjected to correctness checks before the final benchmark campaign. These checks covered key handling, proof verification behavior, and structure specific properties in both implementations. This reduced the risk that performance was being measured on logically invalid states.

The study also improved internal validity by deliberately excluding broader system variables.

Persistence, networking, consensus, concurrent execution, historical version management, and storage tuning were outside scope [21, 22, 13, 20]. This narrow focus meant that measured differences were more likely to arise from authenticated structure behavior itself rather than from unrelated system architecture choices. The cost of that choice was reduced breadth, which is discussed later under external validity.

3.7.2 Construct Validity

Construct validity concerns whether the recorded metrics actually represent the concepts named in the research questions. For RQ1, total phase time, average time per build or update operation, and insert timing buckets were treated as the observable representation of computational cost. This was a reasonable mapping because the benchmark measured the time required for authenticated work under controlled conditions. That work included traversal, structural repair, and hash maintenance. This mapping was appropriate for the benchmark because it expressed how much time each structure required to complete the same authenticated work.

For RQ2, memory behavior was represented through the change in allocated memory during the measured phase and total allocated memory during the phase. It was also represented through heap object count recorded after the phase and the counts of created, freed, and net live heap objects. These metrics captured related but distinct aspects of RAM usage. The change in allocated memory represented the net difference in live allocated memory between the start and end of the phase. Total allocated memory represented cumulative allocation activity, including memory later reclaimed by garbage collection. Heap object count represented how many heap objects remained after measurement. Created and freed heap object counts described allocation activity during the measured phase. Net live heap object change summarized the balance between object creation and reclamation. Taken together, they provided a more complete view of memory behavior than any single memory counter would have provided alone.

For RQ3, proof efficiency was represented through serialized proof size, proof generation time, and proof verification time for inclusion and exclusion proofs. This mapping was appropriate because authenticated proofs are useful only when they can be generated, transmitted or stored, and verified. A measure that ignored any one of these dimensions

would have described only part of the proof problem. The chapter therefore interprets proof efficiency as a combined property rather than as a judgment based on a single metric.

Supporting diagnostics such as time buckets at the phase level and `pprof` profiles had limits of construct validity that were acknowledged explicitly. They described externally observed timing distribution and hotspot concentration rather than direct algorithm step counts. They did not directly answer the research questions by themselves. Their role was explanatory. They helped interpret why a difference appeared, but they did not substitute for the primary time, memory, and proof metrics.

3.7.3 External Validity

External validity was necessarily narrower than the internal validity of the benchmark design. The results generalized most directly to the two measured prototypes and to similar in-memory authenticated dictionaries implemented under comparable assumptions. They also informed engineering decisions about which authenticated data structure better matched particular system requirements.

The results did not automatically generalize to every authenticated dictionary derived from AVL or every authenticated dictionary derived from SMT. Different variants may use alternative hash schedules, node layouts, cache strategies, persistence schemes, concurrency models, or proof encodings [19, 13, 8]. A distributed SMT with heavy caching or a multiversion balanced tree optimized for batch updates could behave differently from the prototypes used here. The study therefore supported comparative reasoning about implemented designs, not universal claims about whole data structure families.

The same limitation applied at the application level. The benchmark did not model a full blockchain node, a disk backed key value store, or a production network for serving proofs. Those systems introduce additional costs such as I/O scheduling, serialization layers, persistence policies, transport overhead, and concurrent access patterns [21, 22]. The thesis results described authenticated structure behavior inside a constrained in-memory setting rather than total system throughput.

3.7.4 Reliability

Reliability was supported through repeated execution and artifact retention. Each scenario was repeated 10 times for both trees. The same repetition count, state sizes, and scenario definitions were applied consistently across the campaign. Deterministic workload generation made it possible to replay the same logical workloads. Raw benchmark result rows were captured before summary tables were produced and were retained with the thesis artifacts. These steps improved the chance that another researcher could reproduce the same benchmark campaign or at least audit how the reported summaries were derived.

Reliability was also improved by using one benchmark framework for both implementations. The framework applied the same procedural structure to each tree, including prebuild logic, measured phases, proof handling, and output recording. This reduced the risk that one implementation benefited from a more favorable measurement script than the other. Consistent execution order across both trees also contributed to procedural stability, even though it did not eliminate every change in measurement conditions over time.

A further reliability issue arose from implementation bias associated with a single author. Both compared structures were implemented and benchmarked within one thesis project. That created a risk that one implementation could accidentally receive cleaner engineering attention, more effective low level optimizations, or more debugging effort than the other. This threat could not be removed completely. It was reduced by using the same language, the same hash function, the same interface defined by the benchmark, the same tooling, and the same review standard for both trees. Both implementations were developed with the same optimization intent and were given comparable optimization effort before the final benchmark campaign. An independent code review before the final benchmark run also helped identify correctness issues, asymmetric optimizations, and benchmark unfairness that a single author could overlook. These safeguards did not make the comparison free from bias, but they made the remaining risk explicit and more manageable.

3.8 Ethical Considerations

The ethical considerations in this study centered on research integrity and transparent reporting. Benchmarking work still imposes clear ethical duties. Methods have to be

reported accurately. Results have to be presented without selective omission of inconvenient runs or scenarios. Limitations have to be stated plainly so that readers would not mistake a controlled in-memory comparison for a full evaluation of a production system. This was especially important in performance research, where persuasive but incomplete reporting can easily mislead later design decisions.

Artifact preservation also has ethical value. The code, benchmark configuration, and raw outputs were archived in the repository referenced in Appendix 2 so that the reported claims could be checked by other researchers. This reduced the risk of irreproducible or overstated conclusions. Accurate citation of related work, transparent reporting of the benchmark design, and complete presentation of results were therefore treated as part of the methodology itself rather than as afterthoughts.

4 Results

This chapter presents the empirical findings of the benchmark campaign described in Chapter 3. The reported results include only workload and metric combinations present for both compared implementations, the AVL Hash Tree and the Sparse Merkle Tree. Figures show median values across repeated runs. The p10 and p90 bands mark the 10th and 90th percentiles of those repeated run values. The p10 band marks the lower run range, while p90 marks the upper run range. For these cost metrics, p10 is more optimistic, while p90 is more pessimistic. The compact chapter tables supplement these plots with mean and standard deviation values computed from the raw repeated runs.

4.1 Benchmark Scope and Reporting Conventions

The benchmark outputs mix totals, averages per operation, and phase progress bucket summaries, so the interpretation rules matter for correct reporting. Insert totals and proof totals remain useful when describing individual scenarios. For proof comparisons across scales, this chapter uses average proof generation time, average proof verification time, and proof size. This avoids distortion from changing proof sample fractions. Insert buckets represent progress through the measured phase rather than sorted latency percentiles.

Table 4. Benchmark coverage used in the results chapter.

Workload family	**Scale range**	**Metrics used**	**Ordered comparison**
Full build	10K–25M	Average insert latency, total insert time, insert phase allocation, number of created heap objects, insert buckets	Yes
Proof only	10K–25M	Proof generation time, proof verification time, proof size	No
Post build add	10K–25M	Average insert latency and total volume of heap allocation performed during the measured phase after growth of a prebuilt state	No
Existing element update	10K–25M	Average update latency, total volume of heap allocation performed during the measured phase, update/add new ratio	No

The coverage in Table 4 defines the scope of all later comparative statements. The current exported suites provide matching AVL and SMT rows for each reported default workload family over the listed scale ranges. Ordered comparisons are narrower and are therefore discussed only for the full build baseline.

4.2 Full Build Insertion and Memory Results

Figure 1 shows the average insert latency for full build workloads. AVL Hash Tree had lower median insert latency than SMT at every tested scale from 10,000 to 25 million elements. The AVL/SMT ratio ranged from 0.787x to 0.940x, and no full build insert crossover appeared in the measured range. At 100,000 elements, the median latency was 2.519 µs for AVL and 2.900 µs for SMT. At 1 million, it was 3.758 µs versus 4.609 µs. At 25 million, it was 7.212 µs versus 8.200 µs.

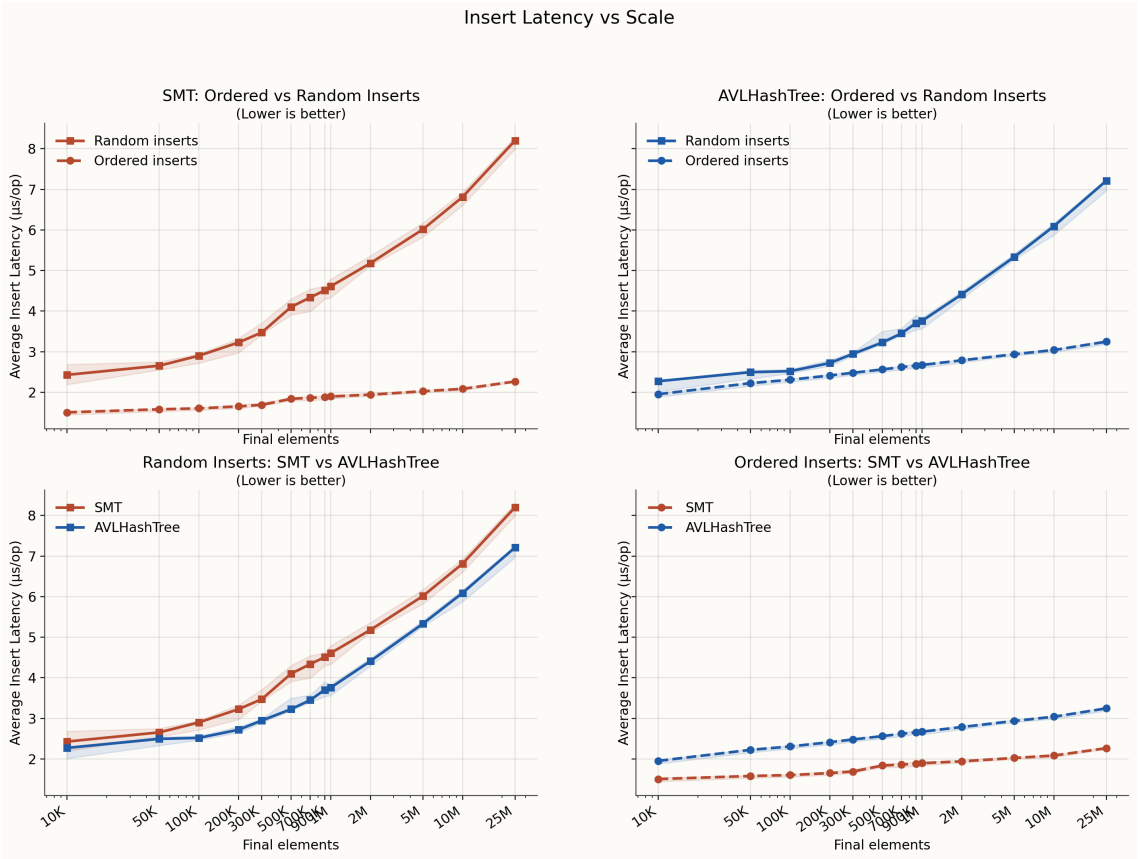

Figure 1. Median full build average insert latency with p10 and p90 run level bands.

The same ranking is visible in total build time in Figure 2. This confirms that the full build result is not an artifact of normalization by operation count. The AVL lead remains visible from the smallest to the largest tested scale. The separation also becomes clearer in the middle and upper range.

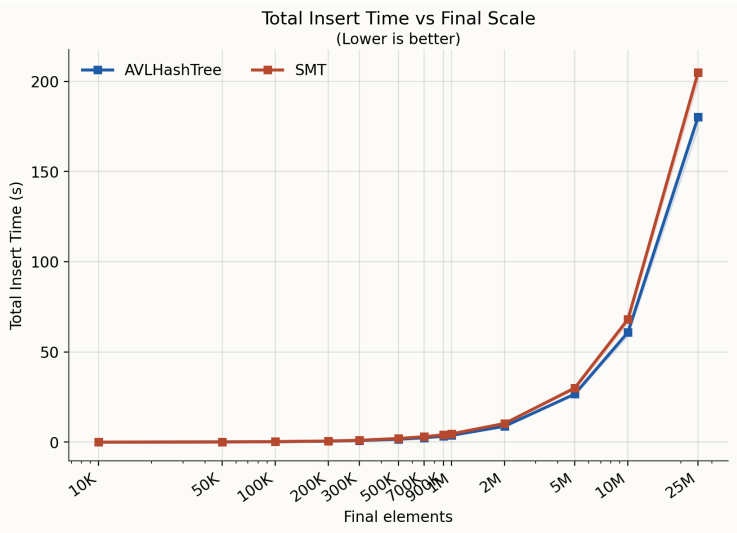

Figure 2. Median total full build insert time.

Figure 3 adds the insert phase memory and heap object perspective for the same full build scenarios. The net change in live heap memory during the measured phase reflects retained live memory growth. The total volume of heap allocation performed during that phase reflects allocation traffic, including short lived objects.

This distinction is practically important. A workload can show a small net change in live heap memory during the measured phase but a large total volume of heap allocation performed during that phase. That pattern means many temporary allocations were created, while only a small amount of memory remained live at phase end.

The heap object count after the measured phase shows how many heap objects remained after the measured phase. Created heap objects show how many heap objects were allocated during the phase. Freed heap objects show how many heap objects were released during the same phase. Net live heap object change is the difference between created and freed heap objects. It therefore shows how much the live heap object count grew during the phase.

AVL allocated less memory than SMT at every tested scale. For the net change in live heap memory during the measured phase, the AVL/SMT ratio ranged from 0.436x to 0.473x. For the total volume of heap allocation performed during the phase, the ratio remained between about 0.423x and 0.426x across the entire measured range.

The same asymmetry is visible in the number of created heap objects. AVL created about 0.444x as many heap objects as SMT, while the net live heap object change stayed between about 0.432x and 0.494x of SMT. In other words, the measured memory related work of AVL remained consistently below that of SMT over the full build workload family.

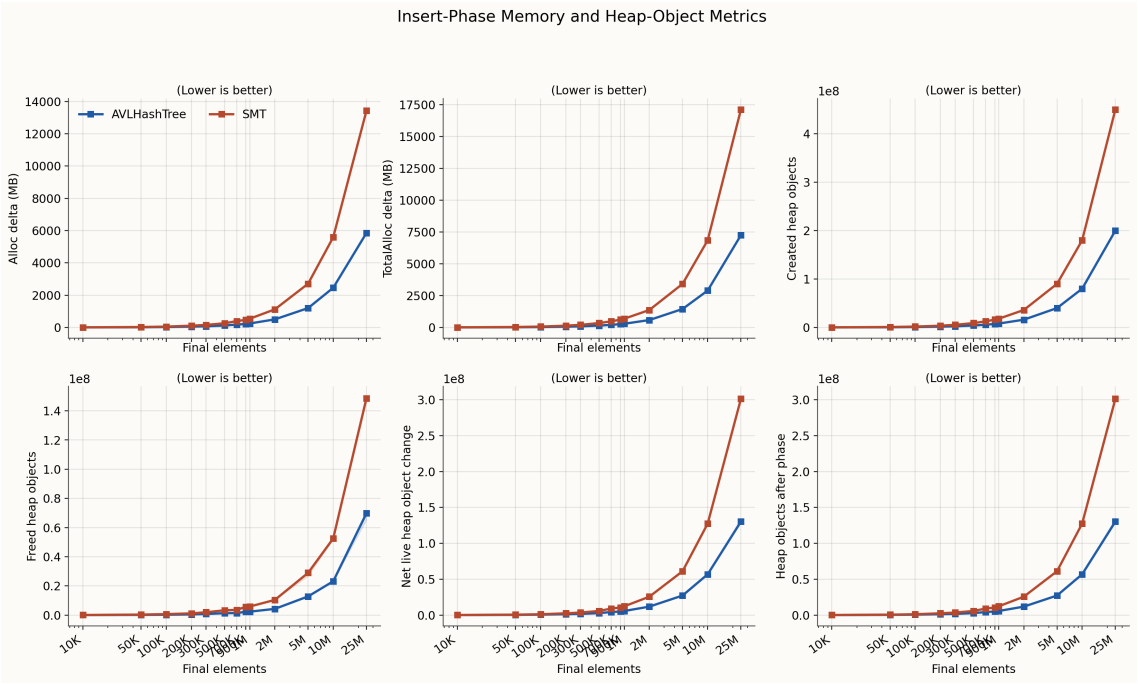

Figure 3. Insert phase allocation and heap object metrics for full build workloads.

Table 5. Selected descriptive statistics for full build results. Insert latency is in µs per operation, allocation in MB, and object counts are absolute counts.

Scale	Metric	Tree	Mean	SD	Median	p10	p90
100K	Insert latency	AVLHashTree	2.528	0.092	2.519	2.464	2.564
		SMT	2.867	0.103	2.900	2.715	2.943
	Live allocation	AVLHashTree	24.659	0.018	24.660	24.639	24.672
		SMT	52.445	1.092	52.140	51.958	52.659
	Total allocation	AVLHashTree	29.047	0.008	29.050	29.039	29.051
		SMT	68.522	0.004	68.520	68.520	68.530
	Created heap objects	AVLHashTree	800094.3	8.9	800096	800083	800103
		SMT	1800035.4	5.1	1800037	1800029	1800040
	Net live heap change	AVLHashTree	571317.4	702.2	571330	570387	572233
		SMT	1165713.7	34419.5	1156164	1150391	1172600
1M	Insert latency	AVLHashTree	3.728	0.134	3.758	3.562	3.832
		SMT	4.569	0.183	4.609	4.332	4.789
	Live allocation	AVLHashTree	245.716	0.900	245.475	244.825	246.971
		SMT	540.570	1.411	540.585	539.224	541.841
	Total allocation	AVLHashTree	290.078	0.004	290.080	290.070	290.080
		SMT	685.079	0.007	685.080	685.070	685.090
	Created heap objects	AVLHashTree	8000137.4	12.4	8000135	8000123	8000151
		SMT	18000059.1	3.3	18000060	18000055	18000060
	Net live heap change	AVLHashTree	5685016.4	36948.1	5674994	5648586	5736504
		SMT	12163710.4	44664.8	12164086	12121190	12203895
10M	Insert latency	AVLHashTree	6.048	0.122	6.090	5.873	6.162
		SMT	6.777	0.135	6.812	6.603	6.925
	Live allocation	AVLHashTree	2456.977	4.848	2457.990	2451.278	2460.477
		SMT	5593.140	14.307	5591.640	5578.539	5610.138
	Total allocation	AVLHashTree	2901.207	0.027	2901.215	2901.170	2901.231
		SMT	6851.338	0.029	6851.325	6851.310	6851.381
	Created heap objects	AVLHashTree	80000422.6	10.1	80000420	80000414	80000434
		SMT	180000324.5	4.2	180000324	180000319	180000328
	Net live heap change	AVLHashTree	56812369.5	198361.3	56852834	56579865	56956848
		SMT	127529790.2	450967.6	127482072	127069688	128066381

Figure 4 isolates the effect of ordered arrivals for the full build baseline. Ordered arrivals improved full build insert latency for both trees at every tested scale from 10,000 to 25 million. The ordered/default full build insert ratio ranged from 0.450x to 0.916x for AVL and from 0.276x to 0.619x for SMT. This means that insertion order remained operationally relevant, and the measured effect was much stronger for SMT than for AVL.

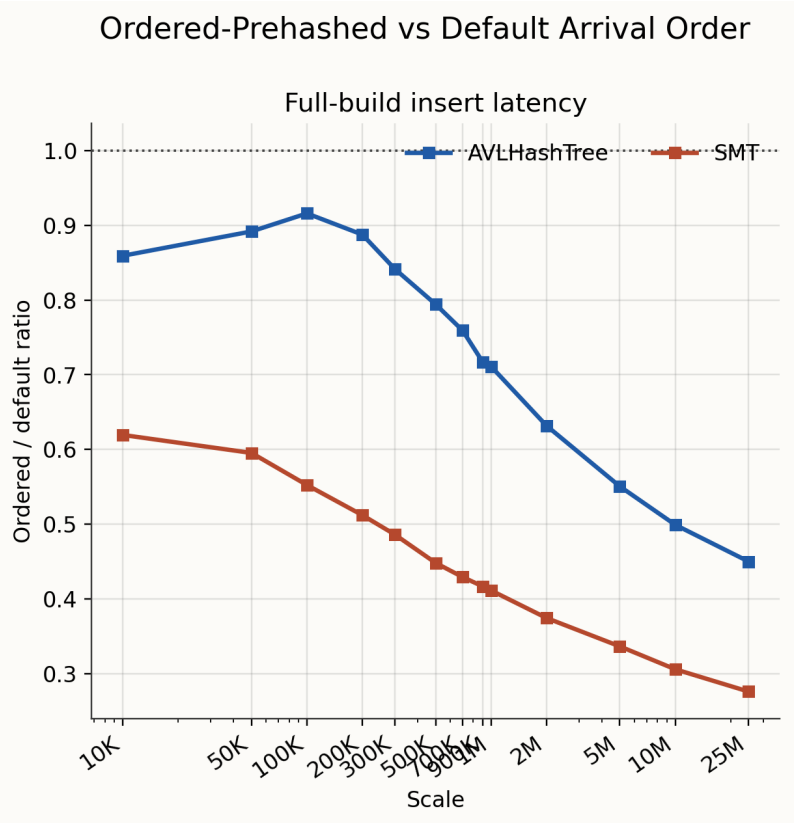

Figure 4. Ordered versus default arrivals for full build insert latency.

Figure 5 shows how insert timing evolves over the build phase for representative full builds. Both trees slowed down as the build phase progressed under the default arrival order. The median last/first bucket ratio was 1.715x for AVL and 1.817x for SMT. Under ordered arrivals, the increase across buckets became flatter for both trees, with median last/first ratios of 1.216x for AVL and 1.176x for SMT. The insert bucket view therefore supports the main latency curves by showing that the measured cost increases were not confined to a single bucket or outlier scale.

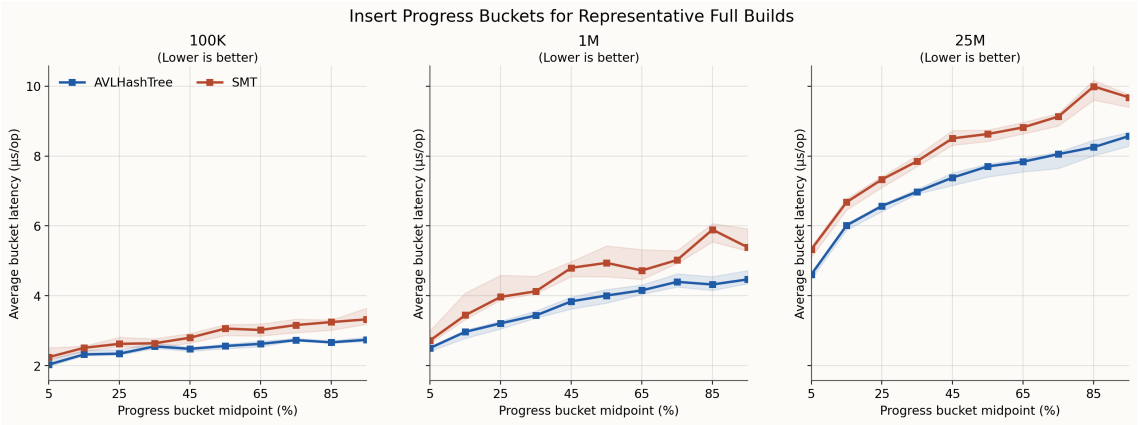

Figure 5. Insert progress buckets for representative full build scenarios.

4.3 Post Build Update Results

Figure 6 compares the two post build update families. One adds new keys to a prebuilt state. The other updates values for existing keys. In the post build scenarios, garbage collection had already run before insertion of the measured new elements. This can lower the net change in live heap memory during the measured phase and the total volume of heap allocation performed during that phase. It can also improve insert performance. The full build case differs here. It inserts data continuously, without an explicit garbage collection run before the measured inserts.

For post build "add new" scenarios, AVL had lower median latency than SMT at every tested scale from 10,000 through 25 million. The AVL/SMT ratio ranged from 0.706x to 0.897x. At 100,000 prebuild, the median latency was 2.932 µs for AVL and 3.822 µs for SMT. At 1 million, it was 5.051 µs versus 5.950 µs. At 25 million, it was 8.673 µs versus 10.202 µs.

For existing element updates, AVL also had lower median latency than SMT at every tested scale from 10,000 through 25 million. The AVL/SMT ratio ranged from 0.579x to 0.764x, which is a larger separation than in the "add new" case. At 100,000, the median update latency was 2.154 µs for AVL and 3.718 µs for SMT. At 1 million, it was 3.668 µs versus 5.452 µs. At 25 million, it was 7.053 µs versus 9.280 µs. The middle panel in Figure 6 shows the same pattern. Updating existing elements is cheaper for both structures. The savings are much larger for AVL. The update/add new ratio ranged from 0.726x to 0.813x for AVL and from 0.884x to 0.973x for SMT. The panel for the total volume of

heap allocation performed during the phase shows the same asymmetry for allocation, with larger update savings on AVL.

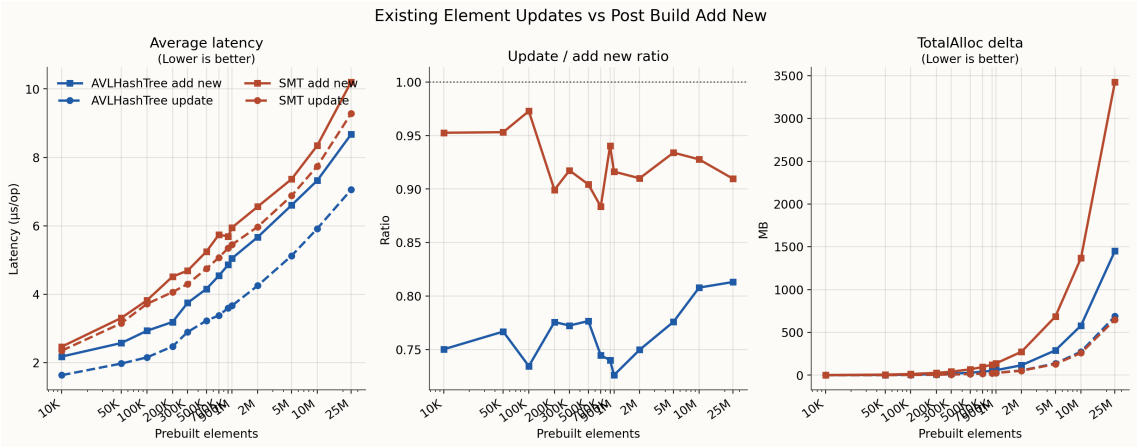

Figure 6. Post build update results: add new versus update of existing elements.

Table 6. Selected descriptive statistics for post build update results. Latencies are in µs per operation and values for the total volume of heap allocation performed during the measured phase are in MB.

Prebuild scale	Metric	Tree	Mean	SD	Median	p10	p90
100K prebuild	Add new latency	AVLHashTree	2.911	0.075	2.932	2.803	2.981
		SMT	3.829	0.161	3.822	3.668	3.999
	Add new total allocation	AVLHashTree	5.810	0.000	5.810	5.810	5.810
		SMT	13.720	0.000	13.720	13.720	13.720
	Update latency	AVLHashTree	2.165	0.055	2.154	2.106	2.243
		SMT	3.690	0.094	3.718	3.593	3.789
	Update total allocation	AVLHashTree	2.760	0.000	2.760	2.760	2.760
		SMT	2.610	0.000	2.610	2.610	2.610
1M prebuild	Add new latency	AVLHashTree	5.040	0.255	5.051	4.783	5.309
		SMT	5.946	0.215	5.950	5.720	6.139
	Add new total allocation	AVLHashTree	58.030	0.000	58.030	58.030	58.030
		SMT	137.042	0.004	137.040	137.040	137.050
	Update latency	AVLHashTree	3.681	0.076	3.668	3.608	3.785
		SMT	5.456	0.190	5.452	5.268	5.582
	Update total allocation	AVLHashTree	27.510	0.000	27.510	27.510	27.510
		SMT	25.990	0.000	25.990	25.990	25.990
10M prebuild	Add new latency	AVLHashTree	7.328	0.091	7.322	7.239	7.422
		SMT	8.294	0.138	8.344	8.128	8.408
	Add new total allocation	AVLHashTree	580.352	0.006	580.350	580.349	580.360
		SMT	1370.351	0.007	1370.350	1370.340	1370.360
	Update latency	AVLHashTree	5.859	0.114	5.916	5.674	5.953
		SMT	7.701	0.106	7.741	7.559	7.801
	Update total allocation	AVLHashTree	275.068	0.009	275.065	275.060	275.080
		SMT	259.919	0.003	259.920	259.919	259.920

4.4 Inclusion Proof Results

Figure 7 provides the joint overview of the proof only scenarios. The three proof metrics behave differently enough that they need to be read together rather than collapsed into a single notion of proof efficiency. In the current data, proof generation, proof verification, and proof size do not all favor the same structure.

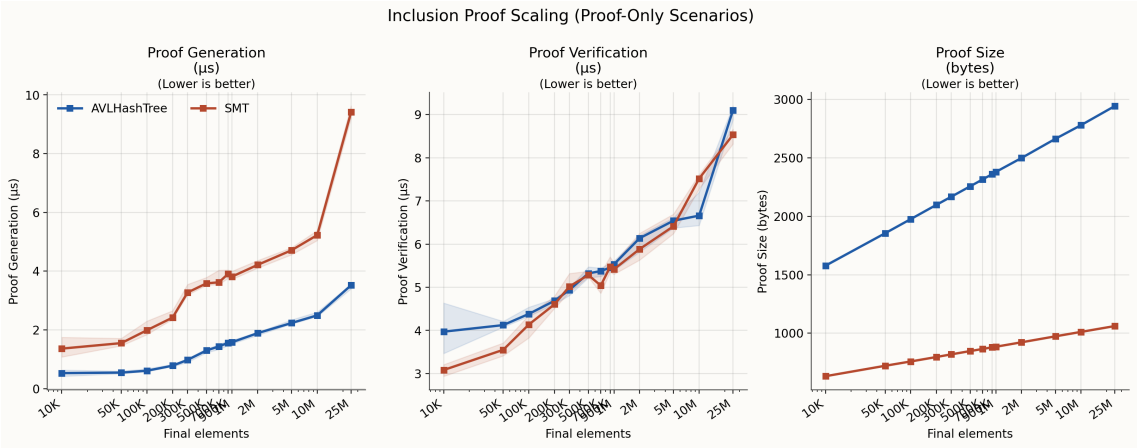

Figure 7. Overview of proof only scaling results for generation, verification, and proof size.

On the left, figure 7 shows proof generation. AVL had lower median proof generation time than SMT at every tested scale from 10,000 through 25 million. The AVL/SMT ratio ranged from 0.299x to 0.477x. At 100,000, the median proof generation time was 0.608 µs for AVL and 1.988 µs for SMT. At 1 million, it was 1.580 µs versus 3.815 µs. At 25 million, it was 3.520 µs versus 9.414 µs.

In the middle, 7 shows proof verification. This result is much tighter than proof generation and should be described as near parity with a slight SMT tendency rather than as a strong SMT separation. The AVL/SMT ratio ranged from 0.886x to 1.288x. SMT had lower median verification time at 10 of the 13 tested scales, while AVL was lower at 300,000, 900,000, and 10 million. At 100,000, the median proof verification time was 4.382 µs for AVL and 4.136 µs for SMT. At 1 million, it was 5.532 µs versus 5.416 µs. At 10 million, it was 6.659 µs versus 7.518 µs. At 25 million, it was 9.102 µs versus 8.534 µs.

On the right, figure 7 shows proof size. SMT produced smaller proofs at every tested scale. AVL proofs were between 2.50x and 2.77x larger than SMT proofs over the measured range. At 100,000, the median proof size was 1976 bytes for AVL and 758 bytes for SMT. At 1 million, it was 2380 bytes versus 884 bytes. At 25 million, it was 2944 bytes versus 1061 bytes.

The proof balance view in Figure 8 shows how generation and verification compare for each structure. AVL verification remained substantially slower than AVL generation at every tested scale. SMT generation stayed much closer to SMT verification and exceeded

it only at the largest 25 million point. This means that the verifier side burden dominates the proof cost profile more strongly for AVL than for SMT in the current experiment. The same figure also shows that smaller proof size did not coincide with faster proof generation. Verification values clustered more tightly than generation values, and proof size per million elements decreased sharply with scale for both trees.

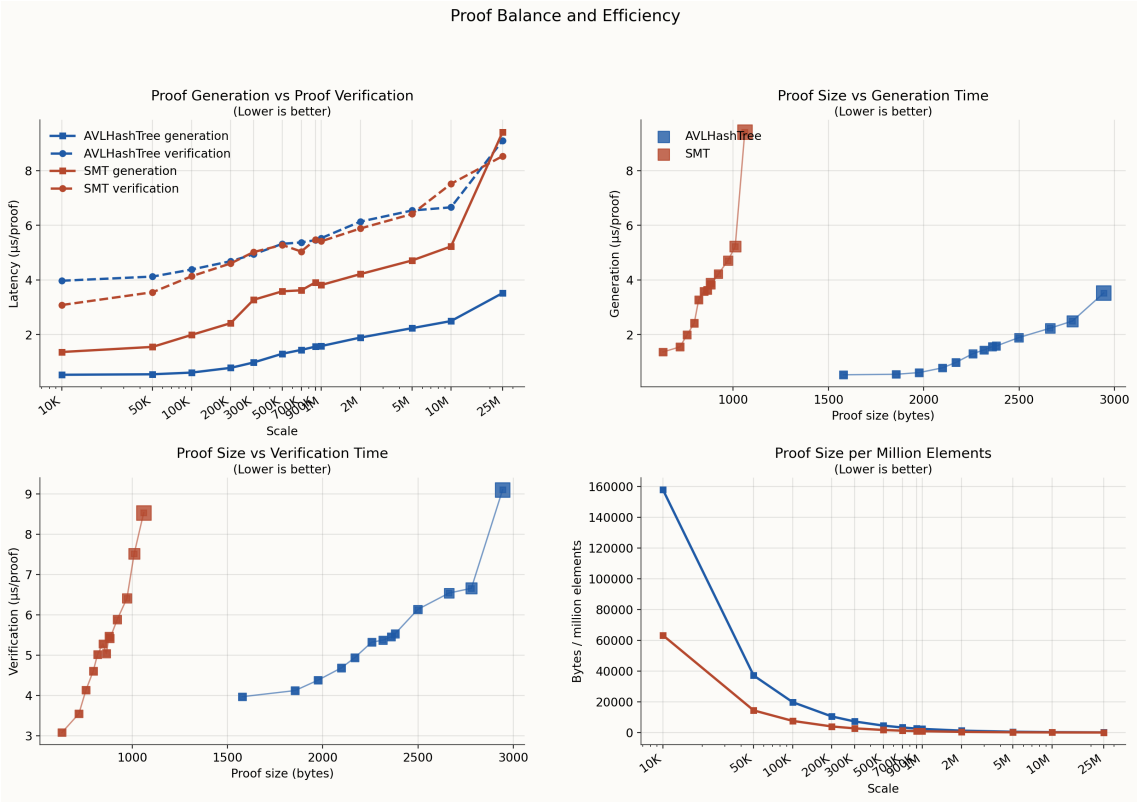

Figure 8. Proof balance and proof size comparisons.

Table 7. Selected descriptive statistics for proof only results. Generation and verification are in µs per proof, proof size is in bytes.

Scale	Metric	Tree	Mean	SD	Median	p10	p90
100K	Proof generation	AVLHashTree	0.616	0.037	0.608	0.589	0.676
		SMT	2.038	0.240	1.988	1.838	2.298
	Proof verification	AVLHashTree	4.407	0.101	4.382	4.326	4.538
		SMT	4.126	0.279	4.136	3.827	4.352
	Proof size	AVLHashTree	1976.2	1.7	1976	1974	1978
		SMT	758.3	0.7	758	758	759
1M	Proof generation	AVLHashTree	1.588	0.054	1.580	1.526	1.649
		SMT	3.872	0.182	3.815	3.719	3.997
	Proof verification	AVLHashTree	5.508	0.098	5.532	5.391	5.606
		SMT	5.444	0.153	5.416	5.300	5.564
	Proof size	AVLHashTree	2379.0	1.2	2380	2378	2380
		SMT	884.4	0.5	884	884	885
10M	Proof generation	AVLHashTree	2.514	0.064	2.494	2.452	2.604
		SMT	5.226	0.153	5.227	5.044	5.383
	Proof verification	AVLHashTree	6.709	0.302	6.659	6.432	7.199
		SMT	7.484	0.148	7.518	7.277	7.609
	Proof size	AVLHashTree	2780.5	0.5	2780	2780	2781
		SMT	1011.0	0.0	1011	1011	1011

4.5 Exclusion Proof Results

Figure 9 provides the joint overview of the exclusion proof only scenarios. The pattern is more one sided than in the inclusion proof results. AVL had lower proof generation and verification time across the measured range, while SMT produced smaller proofs.

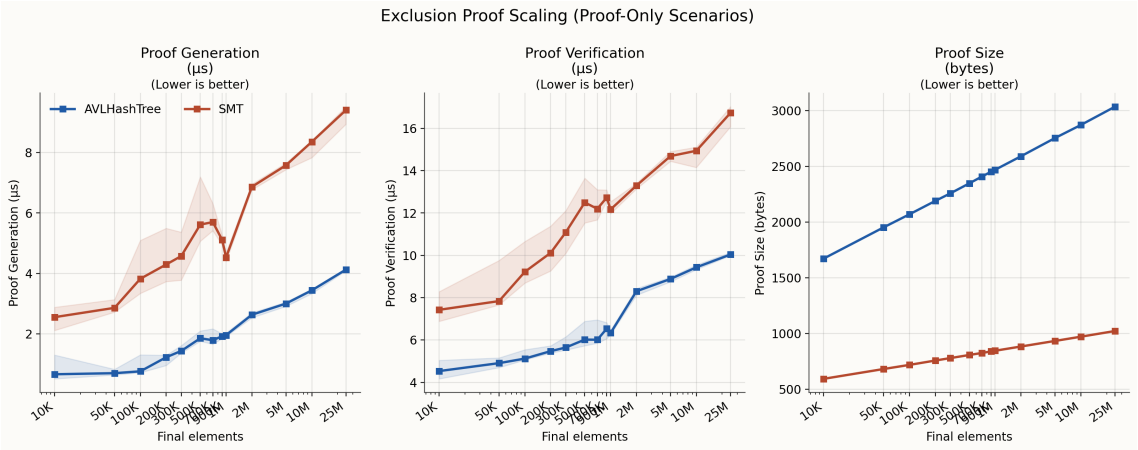

Figure 9. Overview of exclusion proof only scaling results for generation, verification, and proof size.

On the left, figure 9 shows exclusion proof generation. AVL had lower median exclusion proof generation time than SMT at every tested scale from 10,000 through 25 million. The AVL/SMT ratio ranged from 0.199x to 0.439x. At 100,000, the median proof generation time was 0.763 µs for AVL and 3.825 µs for SMT. At 1 million, it was 1.960 µs versus 4.530 µs. At 25 million, it was 4.128 µs versus 9.406 µs.

In the middle, figure 9 shows exclusion proof verification. AVL had lower median verification time than SMT at every tested scale. The AVL/SMT ratio ranged from 0.482x to 0.631x. At 100,000, the median proof verification time was 5.123 µs for AVL and 9.224 µs for SMT. At 1 million, it was 6.332 µs versus 12.170 µs. At 25 million, it was 10.040 µs versus 16.729 µs.

On the right, figure 9 shows exclusion proof size. SMT produced smaller exclusion proofs at every tested scale. AVL proofs were between 2.82x and 2.97x larger than SMT proofs over the measured range. At 100,000, the median proof size was 2069 bytes for AVL and 718 bytes for SMT. At 1 million, it was 2469 bytes versus 845 bytes. At 25 million, it was 3035 bytes versus 1022 bytes.

The exclusion proof balance view in Figure 10 shows how generation and verification compare for each structure. Verification remained slower than generation for both structures at every tested scale. The gap was larger for SMT, whose verification time stayed above AVL verification time across the full measured range. The same figure also shows that smaller exclusion proof size did not coincide with faster generation or verification. AVL

produced larger exclusion proofs, but it had lower generation and verification latency in these scenarios.

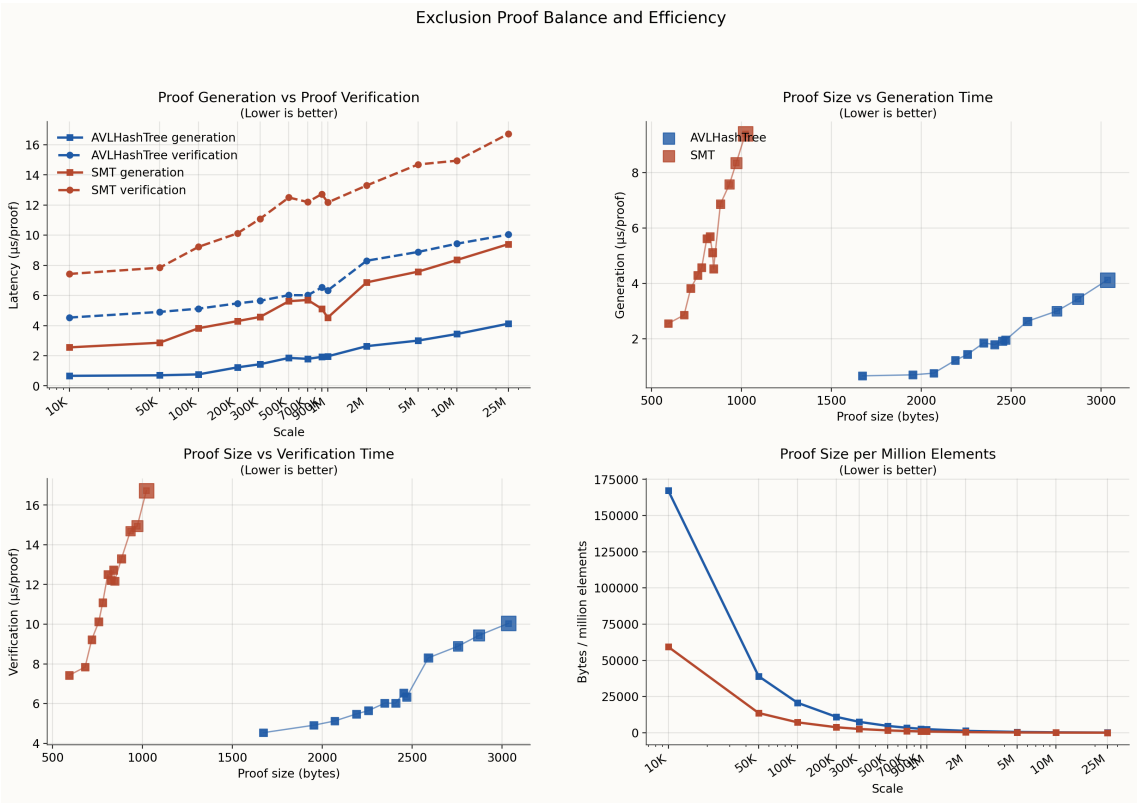

Figure 10. Exclusion proof balance and proof size comparisons.

Table 8. Selected descriptive statistics for exclusion proof only results. Generation and verification are in µs per proof, proof size is in bytes.

Scale	Metric	Tree	Mean	SD	Median	p10	p90
100K	Proof generation	AVLHashTree	0.912	0.277	0.763	0.719	1.308
		SMT	3.973	0.706	3.825	3.342	5.094
	Proof verification	AVLHashTree	5.239	0.232	5.123	5.059	5.541
		SMT	9.480	0.890	9.224	8.691	10.654
	Proof size	AVLHashTree	2069.0	0.0	2069	2069	2069
		SMT	718.0	0.0	718	718	718
1M	Proof generation	AVLHashTree	1.973	0.048	1.960	1.917	2.032
		SMT	4.545	0.141	4.530	4.404	4.691
	Proof verification	AVLHashTree	6.353	0.088	6.332	6.261	6.459
		SMT	12.203	0.296	12.170	11.970	12.531
	Proof size	AVLHashTree	2469.0	0.0	2469	2469	2469
		SMT	845.0	0.0	845	845	845
10M	Proof generation	AVLHashTree	3.428	0.058	3.442	3.335	3.484
		SMT	8.215	0.277	8.356	7.831	8.391
	Proof verification	AVLHashTree	9.450	0.117	9.434	9.331	9.550
		SMT	14.756	0.469	14.941	14.151	15.112
	Proof size	AVLHashTree	2871.0	0.0	2871	2871	2871
		SMT	971.0	0.0	971	971	971

4.6 Results Summary

Across the workload families, AVL led every update side metric reported in this chapter. This included full build insertion, post build new element additions, existing element updates, insert phase allocation, number of created heap objects, and proof generation for both proof types. SMT led inclusion and exclusion proof size. Inclusion proof verification was near parity with a slight SMT tendency, while AVL led exclusion proof verification. Ordered arrivals benefited both trees in the full build comparison. The measured benefit was much larger for SMT.

5 Discussion

5.1 Interpretation of RQ1: Computational Cost

The first research question asked which structure had lower computational cost during build and update workloads. Within the measured single-threaded in-memory setting, the answer is consistently the AVL Hash Tree. AVL had lower median latency in the full build baseline, in post build insertion of new keys, in existing element updates, and in proof generation. The ratio ranges summarize the result. AVL/SMT was 0.787x to 0.940x for full build insertion and 0.706x to 0.897x for post build new element additions. It was 0.579x to 0.764x for existing element updates, 0.299x to 0.477x for inclusion proof generation, and 0.199x to 0.439x for exclusion proof generation. No full build insertion crossover and no proof generation crossover appeared in the measured range.

This matters because the thesis compares authenticated work rather than plain dictionary operations. The reported update costs already include the structure specific maintenance required to keep the root commitment valid after each modification. In this setting, the AVL implementation consistently completed the measured workloads with lower time than the SMT implementation. The result therefore supports the claim that, for these two concrete Go implementations, AVL is the faster updater and the faster prover under the selected workload families.

5.2 Interpretation of RQ2: Memory Behavior

The second research question asked how RAM related behavior differed between the two trees. The measured answer again favors AVL. In the full build results, the net change in live heap memory during the measured phase for AVL remained between 0.436x and 0.473x of SMT. The total volume of heap allocation performed during the measured phase stayed between about 0.423x and 0.426x of SMT across the full range. The same gap was visible in the number of created heap objects. AVL created about 0.444x as many heap objects as SMT, and the net live heap object change remained between approximately

0.432x and 0.494x of SMT.

These results do not measure peak resident set size, and they do not establish causality on their own. However, they are consistent with the latency advantage observed for AVL in the same workloads. In other words, the faster structure was also the structure that allocated less and materialized fewer heap objects during the measured insert phases. For the scope of this thesis, that makes the memory result practically important rather than merely secondary.

5.3 Interpretation of RQ3: Proof Efficiency

The third research question concerned inclusion and exclusion proof efficiency as a joint property of proof generation time, proof verification time, and proof size. The current benchmark results show that no single structure dominates all proof dimensions. AVL is the faster prover, SMT gives the smaller proof, and verification depends on proof type.

The inclusion proof generation result is clear. AVL stayed faster at every proof only scale, with an AVL/SMT ratio between 0.299x and 0.477x. Exclusion proof generation followed the same direction, with an AVL/SMT ratio between 0.199x and 0.439x. Proof size points in the opposite direction. SMT proofs were smaller at every tested scale for both proof types. Inclusion proof verification is the least decisive dimension. SMT was lower at most scales, but the ratio stayed in the relatively narrow band of 0.886x to 1.288x, and AVL was still lower at three scales. Exclusion proof verification was more one sided, with AVL lower at every tested scale.

The proof balance figures add a useful interpretive detail. For AVL inclusion proofs, verification remained much slower than generation across the whole range, so verifier side work dominated the proof cost profile more strongly. SMT was more balanced for inclusion proofs. Its generation and verification curves stayed closer together, and only at the largest scale did generation rise above verification. For exclusion proofs, AVL had lower generation and verification time, while SMT retained the smaller proof size. This clarifies the practical tradeoff in the current data. The smaller proof structure was not the faster prover, and the faster prover did not also minimize transmitted proof material.

5.4 Arrival Order and Update Type Sensitivity

Arrival order remained important in the full build ordered/default comparison. Ordered arrivals improved both trees at every measured scale. The ordered/default ratio range nevertheless differed substantially between the two structures: 0.916x to 0.450x for AVL, versus 0.619x to 0.276x for SMT. The insert bucket progression results reinforce this interpretation by showing that ordered arrivals also flattened the within phase cost growth for both trees.

Update type also mattered. Both implementations handled existing element updates more cheaply than adding new keys into an established state, but the size of that benefit was very different. AVL had an update/add new ratio between 0.726x and 0.813x, whereas SMT ranged between 0.884x and 0.973x. This means that existing key updates reduced measured work more strongly for AVL than for SMT. In practical terms, the relative ranking of the two trees did not invert across these update families, but the margin of AVL's advantage changed with workload type.

5.5 Implications for Authenticated Data Structure Selection

For the two implementations studied here, the engineering choice becomes clearer when the main bottleneck is known. If update throughput, prover side work, and measured memory footprint are the dominant constraints, AVL is the stronger candidate. It led the build, both post build update families, allocation, and proof generation results for both proof types. If proof compactness is the dominant constraint, SMT is the stronger candidate because it consistently produced much smaller proofs.

The verification result needs more caution in system level reasoning. For inclusion proofs, SMT usually verified slightly faster, but the difference was small compared with the much larger gaps observed for proof generation and proof size. For exclusion proofs, AVL verified faster across the measured range. A system should therefore not select a structure on verification assumptions alone. The present data support a narrower claim: verification depends on proof type, AVL has a larger prover side advantage, and SMT has a proof size advantage.

5.6 Future Work

Future work should test whether the observed ranking remains stable once both structures receive optimization strategies that were intentionally outside the scope of the present benchmark. One useful direction is batch inserts for both trees. Grouping updates can reduce repeated work through shared search paths and hash reuse, and for AVL it may also reduce rebalance overhead. Future SMT implementations could additionally use parallel hash recomputation across independent branches and could process many nodes from the same depth together before propagating hash updates toward the root [13, 8]. Parallelization is also possible for AVL, for example through locking based designs, but it is more difficult to implement well and may result in smaller efficiency gains than on SMT [12, 13]. Those kinds of changes could change both latency and memory behavior enough to narrow or even reverse parts of the current gap.

A second extension is to compare these optimized workflows in a setting with retained state in a database rather than only in memory execution. A database backed design introduces cache effects, serialization cost, write amplification, and recovery concerns that may favor different structures than the current benchmark [21, 22]. Future work could therefore evaluate batch and parallel workflows for both AVL and SMT together with database backed retention so that update cost, proof cost, and storage behavior are measured within the same system level setting.

5.7 Limitations of the Present Comparison

These conclusions are limited to the workloads and metrics present in the current benchmark exports. The current results provide matching AVL and SMT coverage for the default full build, post build add, existing element update, and proof only families across the reported scale range. Ordered comparisons were available only for full builds, while ordered post build and ordered proof only scenarios were not part of the exported dataset.

The benchmark itself also remains limited to a single-threaded, in-memory, Go specific comparison of two concrete implementations. The results do not directly generalize to all AVL derived authenticated dictionaries or all SMT derived authenticated dictionaries. Alternative node layouts, caching policies, proof encodings, persistence strategies, or

concurrency models could change the balance.

5.8 Discussion Summary

Taken together, the results show a stable tradeoff rather than a universal winner on every axis. In this benchmark campaign, AVL dominated the update side and prover side results, while SMT dominated proof size and only slightly tended to lead proof verification. The main design tradeoff exposed by the experiment is therefore update and prover efficiency versus proof compactness.

6 Conclusion

This thesis studied the comparative performance of two authenticated data structures, the AVL Hash Tree and the Sparse Merkle Tree. The work started from a clear gap in the literature. Existing studies optimize and evaluate structures within the same family, but they do not provide a direct and controlled comparison between these two widely used authenticated dictionaries. Because both structures are relevant to blockchain state management and verifiable storage, this gap creates a practical problem for engineers who must choose between them without direct empirical evidence.

The goal of the thesis was therefore to provide a systematic comparison under equivalent implementation conditions. To achieve that goal, both structures were implemented in Go and evaluated under the same single-threaded, in-memory benchmark framework. The comparison used the same SHA-256 hash function, the same fixed 32-byte key and value format, the same hardware, and the same measured workloads. The benchmark campaign covered full builds, ordered builds, post build insertion of new keys, existing element updates, and inclusion and exclusion proof generation and verification. Each scenario was repeated ten times, and the analysis focused on medians, variability across runs, proof size, memory allocation behavior, and the number of created heap objects.

The results show a consistent pattern on the update side. The AVL Hash Tree had lower insertion latency than the Sparse Merkle Tree across the full build workload and across both post build update families. This result held from the smallest tested states to the largest measured states. No crossover was observed in full build insertion or in proof generation. Updating existing elements was cheaper than insertion of new keys for both structures, but the benefit was much stronger for AVL. Ordered arrivals improved both structures, yet the improvement was substantially larger for SMT. This indicates that arrival order remained operationally important, even though the benchmark used fixed width hashed keys.

The memory results also favored AVL. During full builds, AVL showed lower live allocation,

lower total allocation traffic, fewer created heap objects, and lower net live heap object growth than SMT. This does not prove that memory behavior alone caused the latency gap, but it is consistent with the timing results. In practical terms, the faster structure in this benchmark was also the structure that allocated less and created fewer heap objects during the measured insert phases.

The proof results were mixed and do not admit declaring a single winner. AVL was clearly faster at inclusion and exclusion proof generation across the full measured range. SMT, however, produced substantially smaller proofs at every tested scale. Inclusion proof verification remained comparatively close. SMT was usually slightly faster, but the gap was narrow and did not support a claim of strong SMT dominance on the verifier side. Exclusion proof verification was different, with AVL faster at every tested scale. The proof side tradeoff was therefore clear: AVL minimized prover work, while SMT minimized proof size. Verification cost depended on proof type.

Taken together, the benchmark answers the research questions as follows. For RQ1, the AVL Hash Tree had lower computational cost during the measured build and update workloads. For RQ2, the AVL Hash Tree also showed lighter memory behavior in the measured allocation and heap object metrics. For RQ3, the answer split across dimensions and proof types: AVL was the faster prover, SMT produced the smaller proofs, inclusion proof verification was close, and AVL led exclusion proof verification. These findings imply that structure choice should depend on the dominant system constraint. If the main bottleneck is update throughput, prover work, or measured memory footprint, AVL is the stronger choice in the present comparison. If the main bottleneck is proof compactness, SMT is the stronger choice.

The contribution of this thesis is therefore twofold. First, it closes an identified empirical gap by providing a direct comparison between AVL Hash Trees and Sparse Merkle Trees under controlled and comparable conditions. Second, it translates that comparison into practical guidance. The results show no universal advantage of one structure over the other, but rather a tradeoff between update and prover efficiency versus proof compactness. This is a more useful conclusion than relying on asymptotic complexity alone, because both structures have logarithmic behavior in theory while still differing substantially in measured constant factors and memory behavior.

The thesis also has clear limits. The comparison concerns two concrete Go implementations, not every possible AVL derived or SMT derived authenticated dictionary. The benchmark remained single-threaded, in-memory, and intentionally isolated from persistence, networking, consensus, and distributed execution. For that reason, the conclusions are strongest at the authenticated structure level rather than at the level of complete blockchain nodes or storage engines. Future work could extend the comparison to concurrent settings, persistent storage backends, broader proof workloads, and alternative implementation strategies. Even with those limits, the present study provides a defensible and practically useful baseline for selecting between the two compared authenticated data structures.

References

[1] OpenAI. *Codex: AI Coding Partner from OpenAI*. Online product documentation. 2026. URL: https://openai.com/codex/ (visited on 20.05.2026).

[2] OpenAI. *ChatGPT: AI Chatbot to Discover, Learn, and Create*. Online product documentation. 2026. URL: https://chatgpt.com/overview/ (visited on 20.05.2026).

[3] Cosmos SDK Documentation. *Store*. Online documentation. 2026. URL: https://docs.cosmos.network/sdk/latest/learn/concepts/store (visited on 04.05.2026).

[4] Cosmos IAVL. *IAVL: Immutable AVL+ Tree*. GitHub repository. 2026. URL: https://github.com/cosmos/iavl (visited on 04.05.2026).

[5] Tendermint. *Merkle Trees*. GitHub wiki. 2026. URL: https://github.com/tendermint/tendermint/wiki/Merkle-Trees (visited on 04.05.2026).

[6] BNB Chain. *Tendermint IAVL*. GitHub repository. 2026. URL: https://github.com/bnb-chain/bnc-tendermint-iavl (visited on 04.05.2026).

[7] Diem. *Authenticated Data Structures*. GitHub repository documentation. 2026. URL: https://github.com/diem/diem/blob/main/specifications/common/authenticated_data_structures.md (visited on 04.05.2026).

[8] Z. Gao, Y. Hu, and Q. Wu. *Jellyfish Merkle Tree*. Online document. Version dated 2021-01-14. Jan. 2021. URL: https://developers.diem.com/papers/jellyfish-merkle-tree/2021-01-14.pdf (visited on 20.02.2026).

[9] Aptos Documentation. *Blockchain Deep Dive*. Online documentation. 2026. URL: https://aptos.dev/network/blockchain/blockchain-deep-dive (visited on 04.05.2026).

[10] Celestia. *Celestia Specifications*. GitHub repository. 2026. URL: https://github.com/celestiaorg/celestia-specs (visited on 04.05.2026).

[11] W. Zhang et al. "TICK: Tiny Client for Blockchains". In: *IEEE Internet of Things Journal* 9.16 (Aug. 2022), pp. 14172–14184. DOI: 10.1109/JIOT.2020.3015476.

[12] Y. Qi, X. Tang, and Y. Huang. "Enabling Efficient Verification of Dynamic Data Possession and Batch Updating in Cloud Storage". In: *KSII Transactions on Internet and Information Systems* 12.6 (June 2018), pp. 2429–2449. DOI: 10.3837/tiis.2018.06.001.

[13] Y. Qi, X. Tang, and Y. Huang. "Enabling Efficient Batch Updating Verification for Multi-versioned Data in Cloud Storage". In: *Chinese Journal of Electronics* 28.2 (2019), pp. 377–385. DOI: 10.1049/cje.2018.02.007.

[14] Q. Wang et al. "A Verifiable Symmetric Searchable Encryption Scheme Based on the AVL Tree". In: *The Computer Journal* 66.1 (Jan. 2023), pp. 174–183. DOI: 10.1093/comjnl/bxab152.

[15] Reza Curtmola et al. “Searchable symmetric encryption: Improved definitions and efficient constructions”. In: *J. Comput. Secur.* 19.5 (Sept. 2011), pp. 895–934. ISSN: 0926-227X.

[16] L. Reyzin et al. “Improving Authenticated Dynamic Dictionaries, with Applications to Cryptocurrencies”. In: *Financial Cryptography and Data Security*. Ed. by A. Kiayias. Cham: Springer International Publishing, 2017, pp. 376–392. DOI: 10.1007/978-3-319-70972-7_21.

[17] R. Dahlberg, T. Pulls, and R. Peeters. “Efficient Sparse Merkle Trees”. In: *Secure IT Systems*. Ed. by B. B. Brumley and J. Röning. Cham: Springer International Publishing, 2016, pp. 199–215. DOI: 10.1007/978-3-319-47560-8_13.

[18] R. Östersjö. “Sparse Merkle Trees: Definitions and Space-Time Trade-Offs with Applications for Balloon”. Bachelor’s dissertation. Karlstad, Sweden: Karlstad University, June 2016. URL: https://www.diva-portal.org/smash/record.jsf?pid=diva2%5C%3A936353%5C&dswid=2978 (visited on 20.02.2026).

[19] J. Kalidhindi et al. *Angela: A Sparse, Distributed, and Highly Concurrent Merkle Tree*. Project Report. Berkeley, CA, USA: University of California, Berkeley, 2018. URL: https://www.academia.edu/download/121564245/project1%5C_report%5C_ver3.pdf (visited on 20.02.2026).

[20] Z. Chen et al. “PEEP: A Parallel Execution Engine for Permissioned Blockchain Systems”. In: *Database Systems for Advanced Applications*. Ed. by C. S. Jensen et al. Cham: Springer International Publishing, 2021, pp. 341–357. DOI: 10.1007/978-3-030-73200-4_24.

[21] Q. Burke et al. “On Scalable Integrity Checking for Secure Cloud Disks”. In: *23rd USENIX Conference on File and Storage Technologies (FAST 25)*. Santa Clara, CA, USA, 2025, pp. 391–405. URL: https://www.usenix.org/conference/fast25/presentation/burke (visited on 20.02.2026).

[22] I. Zhang et al. *QMDB: Quick Merkle Database*. arXiv preprint arXiv:2501.05262. 2025. arXiv: 2501.05262. URL: https://arxiv.org/abs/2501.05262 (visited on 20.02.2026).

[23] R. M. Shadab et al. “HMT: A Hardware-centric Hybrid Bonsai Merkle Tree Algorithm for High-performance Authentication”. In: *ACM Transactions on Embedded Computing Systems* 22.4 (July 2023), pp. 1–28. DOI: 10.1145/3595179.

[24] S. Ponnapalli. “Minimizing I/O Bottlenecks to Achieve Scalable and High-Throughput Systems”. PhD thesis. Austin, TX, USA: The University of Texas at Austin, 2023. DOI: 10.26153/tsw/55494. URL: https://doi.org/10.26153/tsw/55494.

[25] I. Tzialla et al. *Transparency Dictionaries with Succinct Proofs of Correct Operation*. Cryptology ePrint Archive, Report 2021/1263. 2021. URL: https://eprint.iacr.org/2021/1263 (visited on 20.02.2026).

[26] I. Tzialla. “Efficient Verification of Untrusted Services”. PhD thesis. New York, NY, USA: New York University, Sept. 2022. URL: https://search.proquest.com/openview/ce64840ae57ab9c405198fd4a3a93cc7/1?pq-origsite=gscholar%5C&cbl=18750%5C&diss=y (visited on 20.02.2026).

[27] A. Buldas et al. *Alphabill Yellowpaper*. Technical Report AB-1383-remove-fee-credit-bills/fad407ad. Version: TestNet V1. Guardtime, Apr. 2025.

[28] A. Buldas et al. *The Unicity Execution Layer*. Online document (PDF). Sept. 2025. URL: https://github.com/unicitynetwork/execution-model-tex/releases/download/latest/unicity-execution-layer.pdf (visited on 20.02.2026).

[29] E. Stefanov et al. "Iris: a scalable cloud file system with efficient integrity checks". In: *Proceedings of the 28th Annual Computer Security Applications Conference*. Orlando, Florida, USA: ACM, 2012, pp. 229–238. DOI: 10.1145/2420950.2420985.

[30] A. Anagnostopoulos, M. T. Goodrich, and R. Tamassia. "Persistent Authenticated Dictionaries and Their Applications". In: *Information Security*. Ed. by G. I. Davida and Y. Frankel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 379–393. DOI: 10.1007/3-540-45439-X_26.

[31] H. H. Pham, V. K. Nguyen, and T. Nguyen. "Optimization Methods for Merkle Tree in Blockchain". In: *Multi-disciplinary Trends in Artificial Intelligence*. Ed. by C. Sombattheera, P. Weng, and J. Pang. Singapore: Springer Nature Singapore, 2025, pp. 127–139. DOI: 10.1007/978-981-96-0695-5_11.

[32] A. Miller et al. "Authenticated data structures, generically". In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2014)*. San Diego, California, USA: ACM, 2014, pp. 411–423. DOI: 10.1145/2535838.2535851.

Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis[1]

I Nikolai Ovtšinnikov

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis "Performance Comparison of AVL Hash Trees and Sparse Merkle Trees", supervised by Ahto Truu
    1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
    1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright
2. I am aware that the author also retains the rights specified in clause 1 of the nonexclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

20.05.2026

[1]The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive licence shall not be valid for the period.

Appendix 2 – Source Code Repository

The source code repository containing the two Go implementations developed for this thesis is available at:

`https://github.com/NickOvt/go-chain-trees`