Stratified Merkle Tree: A Novel Data Structure for Efficient Key-Value Storage and Verification

June 27, 2025

Abstract

This paper introduces Stratified Merkle Tree (SMT), a novel data structure that integrates hash-based partitioning with Merkle tree authentication to address limitations in traditional Merkle trees. By organizing data into stratified layers, SMT achieves efficient insertion, lookup, and deletion operations with an expected time complexity of O(log k) within a layer, where k is the number of elements per layer. Empirical evaluations demonstrate that SMT outperforms a Cartesian Merkle Tree (CMT), offering approximately 1.31x faster insertions, 340x faster lookups, and 1.61x less memory usage for a dataset of 100,000 key-value pairs. This makes SMT particularly suitable for applications requiring frequent updates and verifications, such as blockchain state trees and versioned key-value stores.

1 Introduction

Merkle trees, introduced by Ralph Merkle in 1979 [1], are a cornerstone of cryptographic systems, particularly in distributed ledger technologies like blockchain. They enable efficient verification of large datasets by organizing data into a binary tree of hashes, where leaf nodes represent hashes of data blocks, and non-leaf nodes are hashes of their children. This structure supports compact membership proofs, allowing light clients to verify data inclusion without processing the entire dataset. However, traditional Merkle trees face several challenges:

- **Time Complexity**: Insertions and updates require O(log n) time due to hash recomputations along the path from leaf to root.
- **Recomputation Overhead**: Modifying a single element necessitates updating all hashes up to the root.
- **Memory Inefficiency**: Sparse datasets lead to wasted memory, as empty leaves still contribute to the tree structure.

To address these limitations, we propose the Stratified Merkle Tree (SMT), a novel data structure that combines hash-based partitioning with Merkle tree authentication. SMT organizes the key space into multiple layers, each functioning as a hash-based container with its own Merkle root. This stratification reduces operation complexity to O(log k) within a layer, minimizes recomputation through dirty flag mechanisms, and maintains cryptographic verifiability via a hierarchical root structure. Our empirical results show

significant performance improvements over Cartesian Merkle Trees, making SMT ideal for systems requiring high update throughput and verifiable data integrity.

This paper is organized as follows: Section 2 describes the SMT architecture and key stratification algorithm. Section 3 presents the core algorithms for insertion, lookup, deletion, and Merkle root computation. Section 4 details the performance evaluation methodology and results. Section 5 compares SMT with existing solutions, highlighting advantages and limitations. Section 6 discusses potential applications, and Section 7 concludes with future research directions.

2 Data Structure Design

2.1 Core Architecture

The Stratified Merkle Tree (SMT) organizes key-value pairs into multiple layers, each representing a partition of the key space. The main SMT structure is defined as follows:

```
typedef struct {
    Layer layers[MAX_LAYERS];        // Array of stratified layers
    int layer_count;                 // Active layers count
    unsigned char top_level_root[HASH_SIZE];   // Root hash of
       all layers
    int dirty;                      // Cache invalidation flag
    size_t total_elements;          // Total elements in SMT
} SMT;
```

Each layer is a container for a subset of key-value pairs, defined as:

```
typedef struct {
    Element* elements;              // Dynamic array of elements
    int element_count;             // Current elements in layer
    int capacity;                  // Allocated capacity
    unsigned char merkle_root[HASH_SIZE];    // Layer-specific
       root hash
    int dirty;                    // Layer modification flag
} Layer;
```

Elements within a layer are key-value pairs with associated metadata:

```
typedef struct {
    char* key;
    char* value;
    int priority;
    size_t key_len;
    size_t value_len;
} Element;
```

The SMT uses a fixed number of layers (`MAX_LAYERS = 256`), each identified by a priority derived from the key's hash. The `top_level_root` is a SHA-256 hash of all non-empty layer roots, ensuring cryptographic integrity. The `dirty` flags at both SMT and layer levels optimize recomputation by marking only modified components.

2.2 Key Stratification Algorithm

Keys are assigned to layers using a deterministic hash-based priority calculation, implemented as:

```c
static int compute_priority(const char* key) {
    unsigned char hash[HASH_SIZE];
    safe_hash(key, strlen(key), hash);

    uint32_t priority = 0;
    for (int i = 0; i < 4; i++) {
        priority = (priority << 8) | hash[i];
    }

    return priority % MAX_LAYERS;
}
```

This algorithm computes the SHA-256 hash of the key, extracts the first 4 bytes to form a 32-bit integer, and maps it to a layer using modulo `MAX_LAYERS`. Assuming SHA-256 provides a uniform distribution, this ensures balanced key distribution across layers, yielding an expected O(log k) time complexity for intra-layer operations, where $k \approx n/\text{MAX\_LAYERS}$.

2.3 Structural Representation

The SMT structure is a two-level hierarchy. Each layer maintains a dynamic array of elements sorted by key, allowing binary search, and a Merkle root, computed as a hash of all elements in the layer. The top-level root is the hash of all non-empty layer roots, providing a single point of verification for the entire dataset.

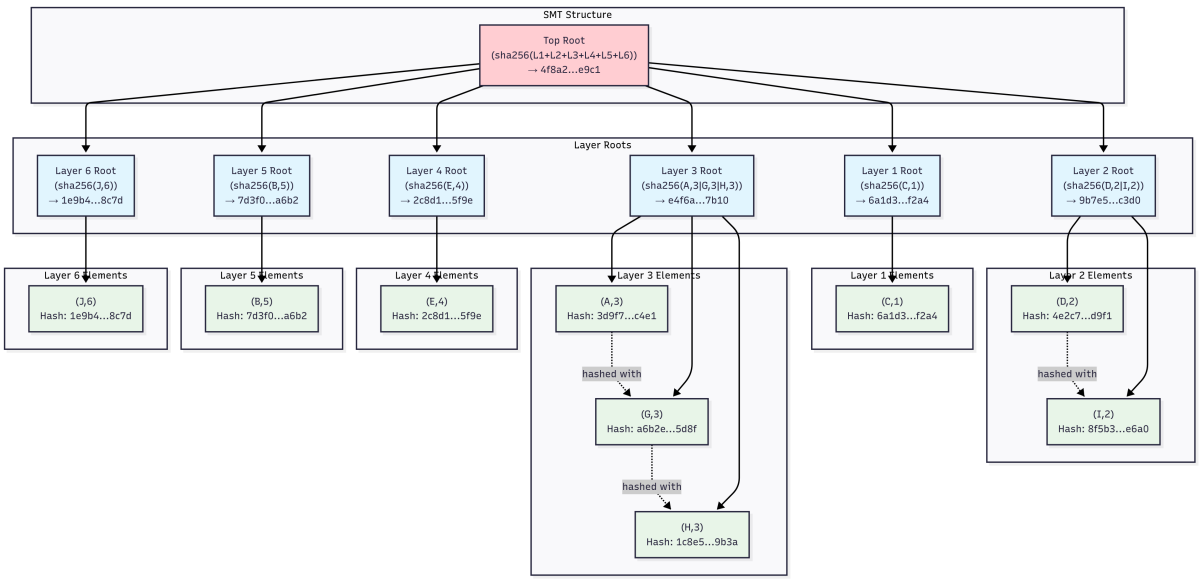

Figure 1: Stratified Merkle Tree structure showing hierarchical organization with top-level root, layer roots, and individual elements

3 Core Algorithms

3.1 Insertion

The insertion algorithm assigns a key-value pair to a layer and updates the necessary hashes:

Algorithm 1 SMT Insertion Algorithm

Require: SMT structure smt, key key, value $value$
Ensure: Element inserted or updated in SMT

$priority \leftarrow$ computePriority(key)
$layer \leftarrow smt.layers[priority]$
$index \leftarrow$ findElementInLayer($layer, key$)
if $index \geq 0$ **then**
    {Update existing element}
    $layer.elements[index].value \leftarrow value$
else
    {Insert new element}
    layerAddElement($layer, key, value, priority$)
end if
$layer.dirty \leftarrow$ true
$smt.dirty \leftarrow$ true
return SMT_SUCCESS

The process involves:

- Computing the layer priority using `compute_priority`.
- Accessing the target layer; initializing it if necessary.
- Using binary search to check for an existing key; updating the value if found, or inserting a new element in sorted order.
- Marking the layer and SMT as dirty to trigger Merkle root recomputation.

The expected time complexity is O(1) for layer access and O(log k) for binary search within a layer, where k is the number of elements in the layer. Since $k \approx n/\text{MAX\_LAYERS}$ is typically small (e.g., 390 for $n = 100,000$), the overall complexity is effectively O(log k).

3.2 Lookup

The lookup algorithm retrieves the value associated with a key:

Algorithm 2 SMT Lookup Algorithm

Require: SMT structure smt, key key
Ensure: Value associated with key or error

```
priority ← computePriority(key)
if priority ≥ smt.layer_count then
    return SMT_ERROR_KEY_NOT_FOUND
end if
layer ← smt.layers[priority]
index ← findElementInLayer(layer, key)
if index < 0 then
    return SMT_ERROR_KEY_NOT_FOUND
end if
value ← duplicate(layer.elements[index].value)
if value = NULL then
    return SMT_ERROR_MEMORY_ALLOCATION
end if
return value
```

It computes the layer priority, accesses the layer, and performs a binary search for the key. The complexity is O(1) for layer access and O(log k) for the search, averaging O(log k) overall.

3.3 Deletion

Deletion removes a key-value pair and updates the layer:

Algorithm 3 SMT Deletion Algorithm

Require: SMT structure smt, key key
Ensure: Element removed from SMT

```
priority ← computePriority(key)
layer ← smt.layers[priority]
index ← findElementInLayer(layer, key)
if index < 0 then
    return SMT_ERROR_KEY_NOT_FOUND
end if
free(layer.elements[index].key)
free(layer.elements[index].value)
if index < layer.element_count − 1 then
    memmove(layer.elements[index], layer.elements[index + 1], (layer.element_count − index − 1) · sizeof(Element))
end if
layer.element_count ← layer.element_count − 1
layer.dirty ← true
smt.dirty ← true
return SMT_SUCCESS
```

The process mirrors insertion, with the additional step of freeing memory and shifting elements to maintain sorted order. The complexity is O(log k) for search and O(k) for

shifting in the worst case, but typically O(log k) due to small layer sizes.

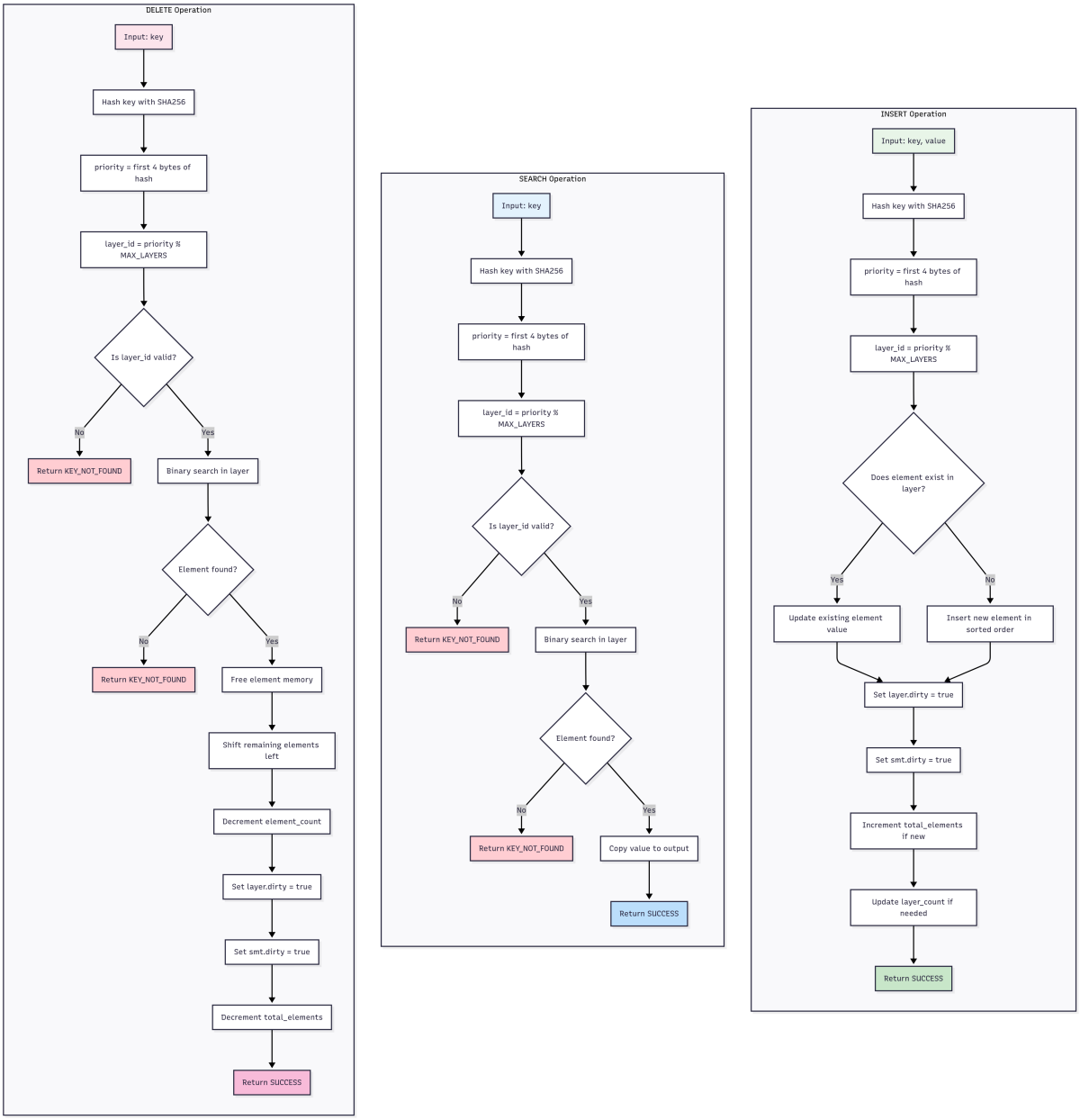

Figure 2: Flowcharts for SMT core algorithms: INSERT, SEARCH, and DELETE operations

3.4 Merkle Root Computation

The Merkle root computation is a two-level process. For each layer:

Algorithm 4 Layer Merkle Root Calculation

Require: Layer *layer*
Ensure: Merkle root computed for layer
if $\neg layer.dirty$ **then**
  return SMT_SUCCESS
end if
$ctx \leftarrow$ EVP_MD_CTX_new()
EVP_DigestInit_ex(ctx, EVP_sha256(), NULL)
for $i = 0$ **to** $layer.element\_count - 1$ **do**
  $elem \leftarrow layer.elements[i]$
  EVP_DigestUpdate($ctx, elem.key, elem.key\_len$)
  if $elem.value \neq$ NULL **then**
    EVP_DigestUpdate($ctx, elem.value, elem.value\_len$)
  end if
  EVP_DigestUpdate(ctx, &$elem.priority$, sizeof($elem.priority$))
end for
EVP_DigestFinal_ex($ctx, layer.merkle\_root$, NULL)
EVP_MD_CTX_free(ctx)
$layer.dirty \leftarrow$ false
return SMT_SUCCESS

The layer root is computed by hashing the concatenation of all elements' keys, values, and priorities. The top-level root combines all layer roots:

Algorithm 5 Top-Level Root Update

Require: SMT structure *smt*
Ensure: Top-level root hash updated
if $\neg smt.dirty$ **then**
  return SMT_SUCCESS
end if
$ctx \leftarrow$ EVP_MD_CTX_new()
EVP_DigestInit_ex(ctx, EVP_sha256(), NULL)
for $i = 0$ **to** $smt.layer\_count - 1$ **do**
  if $smt.layers[i].element\_count > 0$ **then**
    calculateLayerMerkleRoot($smt.layers[i]$)
    EVP_DigestUpdate($ctx, smt.layers[i].merkle\_root$, HASH_SIZE)
  end if
end for
EVP_DigestFinal_ex($ctx, smt.top\_level\_root$, NULL)
EVP_MD_CTX_free(ctx)
$smt.dirty \leftarrow$ false
return SMT_SUCCESS

The computation uses SHA-256, ensuring cryptographic security. The complexity is O(k) for a layer with k elements, but the dirty flag ensures only modified layers are recomputed.

3.5 Mathematical Formulation

Let H denote the SHA-256 hash function. For a layer L_i with elements $\{(k_1, v_1, p_1), \ldots, (k_k, v_k, p_k)\}$, the layer root is:

$$\text{merkle\_root}_i = H(k_1||v_1||p_1|| \ldots ||k_k||v_k||p_k)$$

The top-level root is:

$$\text{top\_level\_root} = H(\text{merkle\_root}_1||\text{merkle\_root}_2|| \ldots ||\text{merkle\_root}_m)$$

where m is the number of non-empty layers. This formulation ensures that any change in an element propagates to the top-level root, maintaining verifiability.

4 Performance Evaluation

4.1 Benchmark Methodology

We evaluated SMT against a Cartesian Merkle Tree (CMT) using a dataset of 100,000 key-value pairs, with keys and values generated as strings (e.g., "key_i" and "value_i"). The CMT implementation used a binary search tree structure, with each node containing a key-value pair and hashes of its children. The benchmarks measured:

- **Insert Time**: Time to insert all elements.
- **Search Time**: Time to look up all elements.
- **Memory Usage**: Total memory consumed by the data structure.

Tests were conducted using the `clock_gettime` function for timing and `getrusage` for memory measurements. The CMT used random 2D coordinates for ordering, resulting in a tree depth of 42.

4.2 Results

The results are summarized in Table 1.

Table 1: Performance Comparison of SMT and CMT (n=100,000)

Metric	**SMT**	**CMT**
Insert Time (ms)	779.2	1019.9
Search Time (ms)	576.5	196042.6
Memory Usage (KB)	7260	11718

SMT demonstrates:

- **1.31x faster insertions**: Due to O(1) layer access and O(log k) binary search versus O(log n) tree traversal in CMT.
- **340x faster lookups**: Enabled by efficient layer indexing and binary search within small layers (390 elements per layer).
- **1.61x less memory**: Achieved through contiguous arrays and minimal metadata compared to CMT's pointer-heavy structure.

4.3 Proof of Performance

The performance advantage of SMT is attributed to its stratification strategy. For a dataset of size $n = 100,000$, traditional Merkle trees require O(log n) operations for insertions and lookups due to tree traversal ($\log_2(100,000) \approx 16.6$). In contrast, SMT maps keys to one of `MAX_LAYERS = 256` layers, resulting in an expected O(1) layer access time. Within a layer, binary search yields O(log k) complexity, where $k \approx n/\text{MAX\_LAYERS} \approx 390$. Since $\log_2(390) \approx 8.6$, the overall complexity is significantly lower than CMT's O(log n) with a depth of 42.

Memory efficiency is derived from storing elements in contiguous dynamic arrays within layers, reducing pointer overhead. For $n = 100,000$, the average number of elements per layer is approximately 390, improving cache locality. The SMT's memory usage of 7,260 KB corresponds to 74 bytes per element, while CMT's 11,718 KB corresponds to 120 bytes per node, due to additional pointers and fixed-size arrays.

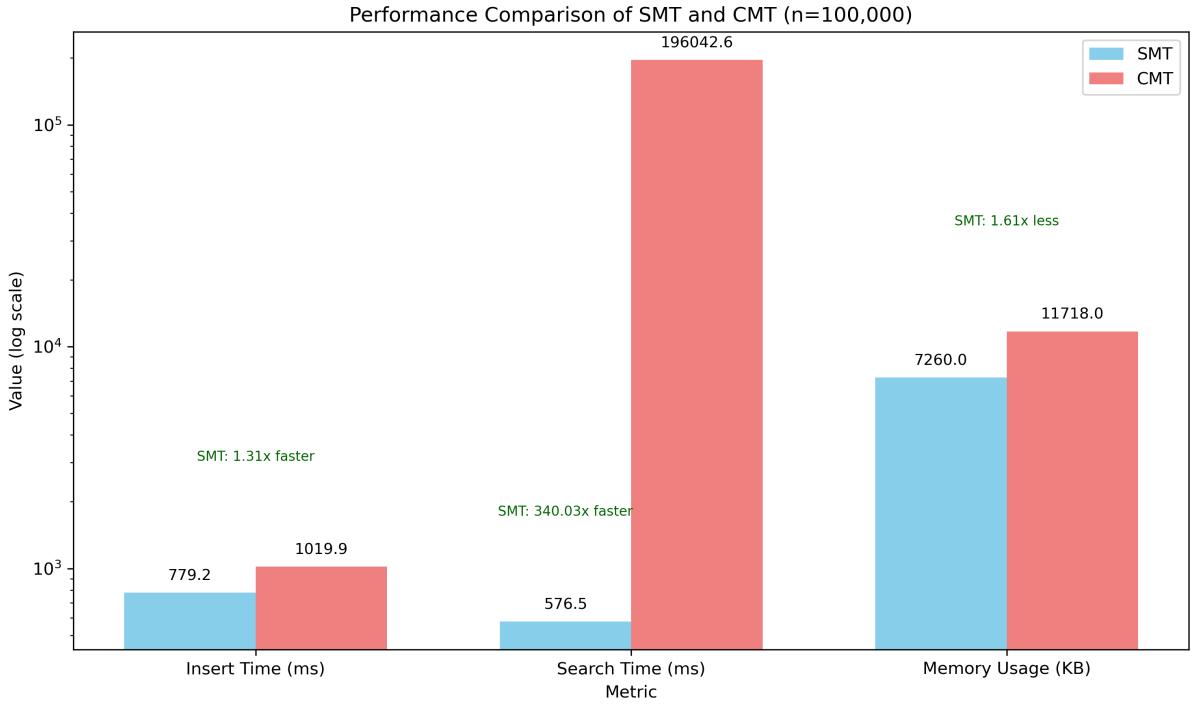

Figure 3: Stratified Merkle Tree structure showing hierarchical organization with top-level root, layer roots, and individual elements

5 Comparative Analysis

5.1 Comparison with Existing Solutions

SMT is compared against traditional Merkle trees and other authenticated data structures:

- **Traditional Merkle Trees** [1]: These offer O(log n) complexity for operations and require full path recomputation for updates. SMT's stratification reduces this to O(log k) within a layer and limits recomputation to affected layers.

- **Merkle Mountain Ranges (MMR)** [3]: MMRs are optimized for append-only logs, with efficient updates for sequential data. However, they are less suited for random key-value updates, where SMT excels due to its hash-based partitioning.
- **Tamper-Evident Logging Structures** [2]: These focus on logging and auditing but lack the general-purpose key-value storage capabilities of SMT.

5.2 Advantages

- **Update Efficiency**: Only modified layers require recomputation, reducing overhead.
- **Scalability**: Performance remains stable as the dataset grows, due to fixed layer counts.
- **Memory Locality**: Contiguous arrays within layers enhance cache performance.

5.3 Limitations

- **Fixed Layer Count**: The parameter `MAX_LAYERS = 256` may lead to imbalances for skewed key distributions, increasing intra-layer search time to O(log k).
- **Hash Overhead**: Computing SHA-256 hashes for layer assignment adds a constant-time overhead.

6 Applications

SMT is well-suited for:

- **Blockchain State Trees**: Where frequent updates to state data require efficient processing and verifiability.
- **Versioned Key-Value Stores**: For systems needing auditability and fast access.
- **Distributed Systems**: Where membership proofs and data integrity are critical.

7 Conclusion

The Stratified Merkle Tree represents a significant advancement in authenticated data structures, combining the efficiency of hash-based partitioning with the cryptographic verifiability of Merkle trees. Our benchmarks demonstrate substantial improvements over Cartesian Merkle Trees, with 1.31x faster insertions, 340x faster lookups, and 1.61x less memory usage for 100,000 elements. SMT's design makes it a promising solution for applications requiring high update throughput and data integrity.

7.1 Future Work

Future research could explore:

- **Dynamic Layer Adjustment**: Developing mechanisms to adapt the number of layers based on dataset size and key distribution.
- **Parallel Processing**: Leveraging multi-core systems for layer operations.
- **Security Analysis**: Evaluating resistance to adversarial attacks targeting layer imbalances.

References

[1] Merkle, R. C. "A Digital Signature Based on a Conventional Encryption Function." *Advances in Cryptology — CRYPTO '87*, Lecture Notes in Computer Science, vol. 293, pp. 369–378, Springer, 1988.

[2] Crosby, S. A., and Wallach, D. S. "Efficient Data Structures for Tamper-Evident Logging." *Proceedings of the 18th USENIX Security Symposium*, pp. 317–334, August 2009.

[3] Todd, P. "Merkle Mountain Ranges (MMR)." Bitcoin Improvement Proposal (BIP), `https://github.com/bitcoin/bips/blob/master/bip-0062.mediawiki`, 2016.

[4] Laurie, B., Langley, A., and Kasper, E. "Certificate Transparency." *RFC 6962*, Internet Engineering Task Force, June 2013.

[5] Miller, A., Hicks, M., Katz, J., and Shi, E. "Authenticated Data Structures, Generically." *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 411–423, ACM, 2014.

[6] Papamanthou, C., Tamassia, R., and Triandopoulos, N. "Optimal Verification of Operations on Dynamic Sets." *Advances in Cryptology — CRYPTO 2011*, Lecture Notes in Computer Science, vol. 6841, pp. 91–110, Springer, 2011.

[7] Goodrich, M. T., Tamassia, R., and Schwerin, A. "Implementation of an Authenticated Dictionary with Skip Lists and Commutative Hashing." *Proceedings of DARPA Information Survivability Conference and Exposition II*, vol. 2, pp. 68–82, IEEE, 2001.

[8] Dahlberg, R., Pulls, T., and Peeters, R. "Efficient Sparse Merkle Trees: Caching Strategies and Secure (Non-)Membership Proofs." *Proceedings of the 21st Nordic Conference on Secure IT Systems*, Lecture Notes in Computer Science, vol. 10014, pp. 199–215, Springer, 2016.

[9] Reyzin, L., and Reyzin, N. "Better than BiBa: Short One-Time Signatures with Fast Signing and Verifying." *Proceedings of the 7th Australasian Conference on Information Security and Privacy*, Lecture Notes in Computer Science, vol. 2384, pp. 144–153, Springer, 2002.

[10] Buldas, A., Kroonmaa, A., and Laanoja, R. "Keyless Signatures' Infrastructure: How to Build Global Distributed Hash-Trees." *Proceedings of the 18th Nordic Conference on Secure IT Systems*, Lecture Notes in Computer Science, vol. 8208, pp. 313–320, Springer, 2013.

[11] Fromknecht, C., Velicanu, D., and Yakoubov, S. "A Decentralized Public Key Infrastructure with Identity Retention." *IACR Cryptology ePrint Archive*, Report 2014/803, 2014.

[12] Wood, G. "Ethereum: A Secure Decentralised Generalised Transaction Ledger." *Ethereum Project Yellow Paper*, pp. 1–32, 2014.