Sparse Merkle Trees: Definitions and Space-Time Trade-Offs with Applications for Balloon

Rasmus Östersjö
Sparse Merkle Trees: Definitions and Space-Time Trade-Offs with Applications for Balloon

Rasmus Östersjö

© 2016 The author and Karlstad University
This dissertation is submitted in partial fulfillment of the requirements for the Bachelor’s degree in Computer Science. All material in this dissertation which is not my own work has been identified and no material is included for which a degree has previously been conferred.

Rasmus Östersjö

Approved, June 8, 2016

Advisor: Tobias Pulls

Examiner: Thijs J. Holleboom
Abstract

This dissertation proposes an efficient representation of a sparse Merkle tree (SMT): an authenticated data structure that supports logarithmic insertion, removal, and look-up in a verifiable manner. The proposal is general in the sense that it can be implemented using a variety of underlying non-authenticated data structures, and it allows trading time for space by the use of an abstract model which represents caching strategies. Both theoretical evaluations and performance results from a proof-of-concept implementation are provided, and the proposed SMT is applied to another authenticated data structure referred to as Balloon. The resulting Balloon has preserved efficiency in the expected case, and is improved with respect to worst case scenarios.
Acknowledgements

First and foremost, I would like to thank my advisor Tobias Pulls for his support throughout the entire project. Not only did he provide invaluable feedback in terms of technical expertise and report writing, but he also invested time to prepare me for future challenges that goes far beyond the scope of this dissertation. Secondly, I would like to thank Roel Peeters for his valuable insights. He suggested an improved layout of the dissertation, and shared his thoughts on the subject. Finally, I sincerely thank my brother Victor Östersjö. He supports me in every project that I participate in, be that with either proof reading, grammar, or in-depth discussions.
# Contents

1 Introduction 1

1.1 Motivation and Challenges 1

1.2 Expected Outcome and Scope 2

1.3 Contribution 2

1.4 Roadmap 2

2 Background 3

2.1 Cryptographic Primitives 3

2.1.1 Hash Functions 3

2.1.2 Digital Signatures 4

2.2 Tree-Based Data Structures 5

2.2.1 Merkle Tree 6

2.2.2 History Tree 7

2.2.3 Binary Search Tree 8

2.2.4 Heap 9

2.2.5 Treap 9

2.2.6 Hash Treap 10

2.3 Balloon 11

2.3.1 Key Building Blocks 12

2.3.2 Updating 12

2.3.3 Snapshots 13

2.3.4 Algorithms 13

2.4 Summary 16

3 Sparse Merkle Tree 17

3.1 Notion 17

3.2 Definition 18
3.3 Approach ......................................................... 19
3.4 Recursive Relationships ................................. 21
3.5 Relative Information ........................................ 23
3.6 Final Proposal ................................................... 26

4 Analysis ......................................................... 28
4.1 Space Complexity ............................................. 28
4.2 Time Complexity .............................................. 31
4.3 Performance .................................................... 34
  4.3.1 Setup ......................................................... 35
  4.3.2 Space Benchmarks ...................................... 35
  4.3.3 Time Benchmarks ...................................... 36
4.4 Minding the Adversary .................................... 38
4.5 Summary ....................................................... 39

5 Applications to Balloon .................................... 40
5.1 Overview ....................................................... 40
5.2 Extension of the Sparse Merkle Tree .................. 40
5.3 Integration ..................................................... 42
  5.3.1 Key Setup .................................................. 42
  5.3.2 Update ...................................................... 42
  5.3.3 Query and Verify ....................................... 42
  5.3.4 Pruned Algorithms .................................... 43
5.4 Evaluation of the New Balloon ......................... 43

6 Conclusion .................................................... 44
6.1 Project Evaluation ........................................... 44
6.2 Future Work .................................................. 45
References 46

A Memory Requirements for the BD-Hybrid Oracle 48
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Security properties for cryptographic hash functions.</td>
<td>4</td>
</tr>
<tr>
<td>2.2</td>
<td>Shapes of binary trees.</td>
<td>6</td>
</tr>
<tr>
<td>2.3</td>
<td>A perfect Merkle tree containing four leaves with attributes $a$–$d$. The Merkle audit path</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>for the left-most leaf includes $H(b)$ and $H(H(c)</td>
<td></td>
</tr>
<tr>
<td>2.4</td>
<td>Two history trees representing distinct views. The earlier view is reconstructed</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>from the newer one by “forgetting” circled nodes.</td>
<td></td>
</tr>
<tr>
<td>2.5</td>
<td>A simplified hash treap with associated values omitted. Dashed edges represent paths $P$</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>determined by binary searches for keys $k$, and circled nodes constitute the corresponding</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(non-)membership proofs.</td>
<td></td>
</tr>
<tr>
<td>2.6</td>
<td>Balloon viewed in the three party setting.</td>
<td>11</td>
</tr>
<tr>
<td>2.7</td>
<td>A pruned hash treap for the non-membership proofs generated by $3 \leftarrow H(k_1)$</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>and $9 \leftarrow H(k_2)$. Circled nodes would be redundant without applying the prune</td>
<td></td>
</tr>
<tr>
<td></td>
<td>operation.</td>
<td></td>
</tr>
<tr>
<td>3.1</td>
<td>A simplified SMT with values inserted into the leaves with indices $H(k_1) = 5$ and</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>$H(k_2) = C$.</td>
<td></td>
</tr>
<tr>
<td>3.2</td>
<td>An illustration of a recursive traversal that obtains the root hash. Dashed components</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>need not be visited.</td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td>Hashes recorded by the branch, depth, and BD-hybrid oracles. Circled nodes represent</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>branches, and diamonds are recorded by depth-based oracles.</td>
<td></td>
</tr>
<tr>
<td>4.1</td>
<td>Two branch paths down to the first and the last leaves, respectively.</td>
<td>34</td>
</tr>
<tr>
<td>4.2</td>
<td>The sizes of sparse Merkle audit paths as a function of existing key-value pairs.</td>
<td>36</td>
</tr>
<tr>
<td>4.3</td>
<td>The memory used by different oracle models as a function of existing key-value pairs.</td>
<td>36</td>
</tr>
<tr>
<td>4.4</td>
<td>The time used to query 1000 (non-)membership proofs as a function of oracle model and</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>existing key-value pairs.</td>
<td></td>
</tr>
</tbody>
</table>
4.5 The time used to insert 1000 key-value pairs as a function of oracle model and existing key-value pairs. .................................................. 37

4.6 The time required to verify (non-)membership proofs as a function of existing key-value pairs. .................................................. 38
List of Tables

3.1 The notation used when describing recursive relationships. . . . . . . . . . . 22
1 Introduction

Consider a trusted author who maintains a collection of data on an untrusted server. Further suppose that clients issue queries to the untrusted server of the form “what is the value associated with key k”. This forms the notion of a (non-)membership query; a kind of query that results in either a retrieved value or false. Regardless of the answer, however, what stops the untrusted server from lying when responding to queries? Existing data could be modified, or even claimed to be non-existing [3]. Therefore clients need proofs, succinct pieces of information, that undoubtedly prove query responses correct with respect to the trust of the author. The field of authenticated data structures and dictionaries proves useful in such settings, combining regular data structures and cryptographic primitives [27].

Dating back to the year of 1987, Merkle [18] pioneered a tree-based data structure that incorporates the use of hash functions. It was intended as an integral part of a digital signature system, but today it is widely deployed in a large variety of other applications. These include Bitcoin [19, 21], certificate issuance [14, 15, 16, 22], and authenticated data structures and dictionaries [8, 20, 24]. The idea of a Merkle tree is general in the sense that it can be applied to any tree, thereby allowing regular data structures such as red-black trees [28], AVL-trees [10, 13], and treaps [2, 4] to be transformed into authenticated constructions. Further these transformations can be implemented persistently [8], meaning that a client can verifiably query past versions of the data structure. This comes at the price of additional storage and/or query time, however. Therefore another type of append-only Merkle tree was proposed [6, 17] which resulted in a naturally persistent construction.

1.1 Motivation and Challenges

Recently, Pulls and Peeters [24] presented an authenticated data structure referred to as Balloon. It is the composition of a hash treap [8, 24] and a history tree [6], both of which are authenticated data structures on their own. This work intends to replace the
hash treap with a sparse Merkle tree (SMT) \cite{3, 16}, presumably reducing implementation complexity while preserving efficiency in terms of time and space. The main challenge with this seemingly straight forward task is not the replacement of the hash treap, however. Since the proposal of the SMT by Laurie and Kasper \cite{16}, there have been no further publications that outline an efficient approach. Different researchers have claimed it to be both inefficient \cite{25} and promising \cite{24}, thereby making it an exciting topic to explore.

1.2 Expected Outcome and Scope

With emphasis on preserved efficiency, the goal is to determine if a SMT can replace the hash treap in Balloon. It is considered out of scope to perform a rigorous security evaluation of the new Balloon, but it is likely that similar evaluations as those provided by Pulls and Peeters \cite{24} apply (hash treaps and SMTs are all based on Merkle trees).

1.3 Contribution

This dissertation formalises the SMT and proposes an efficient representation that is based on the use of an abstract oracle model. Five different oracles are considered, and two of these are proven efficient. The proposed SMT is also applied to Balloon, replacing the hash treap with preserved (improved) efficiency in the expected (worst) case.

1.4 Roadmap

The remainder of this dissertation is structured as follows. Section 2 describes relevant background in terms of preliminaries and terminology. Section 3 provides an in-depth study on SMTs. Section 4 analyses the efficiency of the SMT by means of theoretical evaluations and performance results from a proof-of-concept implementation. Section 5 applies the SMT to Balloon. Finally, Section 6 concludes the dissertation, evaluates the project, and presents future work.
2 Background

The purpose of this section is to equip the reader with necessary background. Section 2.1 introduces cryptographic primitives. Section 2.2 describes data structures related to SMTs and Balloon. Section 2.3 presents Balloon. Finally, Section 2.4 provides a summary.

2.1 Cryptographic Primitives

Cryptographic hash functions and digital signatures are the fundamental building blocks when transforming regular data structures into authenticated constructions. The former provides a secure compression mechanism, and the latter the means for authentication.

2.1.1 Hash Functions

Similar to a regular hash function, a cryptographic hash function compresses an arbitrary large message $m$ into a fixed sized digest $h$. Thereby a larger space of messages are mapped into a smaller space of digests, implying that collisions are inevitable. However, in a cryptographic context, such collisions must be computationally hard to find [1]. Therefore we deal with with computational security, as oppose to information-theoretical security.

More generally there are multiple security properties that a cryptographic hash function is expected to conform with. The most common ones include preimage, second preimage, and collision resistance [1]. These are illustrated in Figure 2.1 and interpreted as follows:

- Figure 2.1a: given the digest $h \leftarrow H(m)$ for message $m$, it must be computationally hard to find a preimage $m'$ generating $h$ without knowledge of $m$;

- Figure 2.1b: given a fixed preimage $m_{\text{fixed}}$, it must be computationally hard to find another preimage $m \neq m_{\text{fixed}}$ such that $H(m) = H(m_{\text{fixed}})$;

- Figure 2.1c: it must be computationally hard to find any distinct preimages $m_1$ and $m_2$ such that $H(m_1) = H(m_2)$. 


When being exposed to cryptographic hash functions for the first time, a common mistake is not noticing the difference between second preimage and collision resistance. If we assume there are no internal weaknesses in the hash function $H$ producing digests of $N \leftarrow |H(\cdot)|$ bits, an adversary must exhaustively search a collision on attempts to break second preimage resistance. Approximately that requires computing $2^N$ hashes [1], which makes collision resistance an easier target. Relying on the birthday paradox, an adversary can cache all message-digest pairs and compare newly derived ones to those in the cache. Collisions are expected after computing $2^{N/2}$ hashes [1].

2.1.2 Digital Signatures

The notion of a digital signature was first introduced by Diffie and Hellman [9]. It is used for the purpose of authenticity, non-repudiation, and integrity of a message, meaning that the signature binds the signer (or more accurately, the signer’s secret key) to a message that cannot be tampered with. A general digital signature scheme consists of a security parameter $\lambda$, a message space $M$, and three algorithms defined as follows [11]:

- $\{vk, sk\} \leftarrow KGen(1^\lambda)$: On input of a security parameter $\lambda$, the key generation algorithm $KGen$ outputs a verification key $vk$ and a signing key $sk$;

- $s \leftarrow Sig(sk, m)$: On input of a signing key $sk$ and a message $m$, the signing algorithm $Sig$ outputs a signature $s$ for $m$;
– \{\text{true}, \text{false}\} \leftarrow Vf(vk, s, m): On input of a verification key \(vk\), a signature \(s\), and a message \(m\), the verification algorithm \(Vf\) outputs \text{true} if \(s\) is a valid signature for \(m\), else \text{false}.

As a security notion, a digital signature scheme can be existentially unforgeable under known message attack \[11\]. In this model the adversary is given the capability to request arbitrary messages and obtain valid signatures for those before attempting to break the scheme. Without access to the signing key \(sk\), the adversary wins if she can generate one valid tuple \((s, m)\) for \(m \in M\) with non-negligible probability in the security parameter \(\lambda\). Else, the adversary loses and the scheme is said to be secure.

### 2.2 Tree-Based Data Structures

In computer science, a tree \[12\] is an (un)ordered collection of entities, not necessarily unique, that has a hierarchical parent-child relationship defined between pairs of entities. Every tree has a single root node designating the start of the tree, and each descendant is recursively defined as a tree. A node is said to be an ancestor to all its descendants, and a parent to its concrete children. All children that have the same parent are referred to as siblings, and every node without children is referred to as a leaf. Also related to tree terminology is level, height, and depth. Henceforth the root is found at level one, the height is the number of levels in the tree, and the depth of a subtree rooted at a leaf is zero.

A binary tree is closely related to trees. Informally it can be viewed as a tree where each node is restricted to at most a left child and a right child, but as pointed out by Knuth \[12\] that is not entirely correct. For instance, a tree can neither be empty nor distinguish between two children. The informal notion suffices for our purposes, however, and Figure 2.2 proceeds to illustrate three different types of binary trees. For a binary tree of height \(h\) to be perfect, the tree must contain exactly \(2^h - 1\) nodes. For a binary tree to be full, all nodes must have two or no children. Finally, for a binary tree to be complete, it must be filled left-to-right at the lowest level of the tree, and entirely at the level above.
2.2.1 Merkle Tree

Storing values at the lowest level of the tree, a Merkle tree [18] is a binary tree that utilises cryptographic hash functions. While leaves compute the hash of their own attributes, parents derive the hash of their children’s hashes concatenated left-to-right. Therefore the hash rooted at a particular subtree is recursively dependent on all its descendants, effectively serving as a succinct summary for that subtree.

Using the prevalent feature of subtree summaries, a Merkle tree can prove values to be present by constructing efficient membership proofs. Each proof must include a Merkle audit path [17], and it is verified by recomputing all hashes, bottom up, from the leaf that the proof concerns towards the root. The proof is believed to be valid if the recomputed root hash matches that of the original Merkle tree, but to be convincing it requires a trustworthy root (e.g., signed by a trusted party or published periodically in a newspaper).

As a concrete example, consider how to generate a membership proof for the left-most leaf in Figure 2.3. The Merkle audit path is composed of grey hashes, and it is obtained by traversing the tree down to the left-most leaf (dashed edges), requesting the hash of each sibling along the way. Assuming knowledge of the attribute \(a\) and the authentic root hash \(r\), the proof is verified by recomputing\(^1\) the root hash \(r'\), testing if \(r' = r\).

---

\(^1\) Start by computing \(H(a)\) and use the proof to learn \(H(b)\). Proceed to derive \(H(H(a)||H(b))\) and use the proof to learn \(H(H(c)||H(d))\). Finally, derive \(r' \leftarrow H(H(H(a)||H(b)||H(H(c)||H(d))))\).
Figure 2.3: A perfect Merkle tree containing four leaves with attributes a–d. The Merkle audit path for the left-most leaf includes $H(b)$ and $H(H(c)||H(d))$.

Following from the work by Merkle [18] and Blum et al. [5], the security of a Merkle audit path reduces to the collision resistance of the underlying hash function. In order to yield unambiguous hashes, however, these must be encoded such that the structure of the tree is captured. An invalid encoding could, for instance, replace the hash of a non-existing child with the empty string, thereby making it impossible to distinguish between a (non-)empty subtree. In order to solve such encoding issues, it is common to add constants [8, 17].

2.2.2 History Tree

A history tree [7] is an append-only Merkle tree that stores events left-to-right at the lowest level of the tree. Therefore it is not lexicographically sorted, and unable to generate efficient non-membership proofs. Nevertheless, the history tree is interesting due to its persistent nature, support of efficient membership proofs, and ability to generate incremental proofs.

To clarify what makes the history tree naturally persistent and how an incremental proof is generated, consider Figure 2.4. If there is only access to the version 5 view, the version 2 view can be reconstructed by temporarily “forgetting” events. Hence, without explicit versioning, the history tree is persistent in the sense that past versions of the tree can be efficiently reconstructed and queried for membership. It is also possible to show
consistency between root hashes for different views, and that requires proving all events in the earlier view present in the newer view. It is achieved by returning just enough information to reconstruct both root hashes (the filled node in Figure 2.4a, along with the circled filled node and the right child of the root in Figure 2.4b), checking if expected roots are obtained.

2.2.3 Binary Search Tree

A binary search tree (BST) [13] is a binary tree that requires the value of each node to be greater (lesser) than the value of its left (right) child. This property, referred to as the BST property, implies a lexicographical order and allows every look-up operation to use a divide-and-conquer technique known as binary search.

In the context of BSTs, a binary search refers to the process of traversing the tree starting at the root down to the node holding a particular value \( v \). At each step, the algorithm checks if the value of the current node is lesser, greater, or equal to \( v \). In the two former cases, the binary search continues recursively in the left/right subtree. In the latter case, \( v \) was found and the algorithm halts. Halting also occurs when an empty subtree is reached, indicating that \( v \) is a non-member.
Because the time required to complete a binary search is bounded by the height of the BST, it is important that the tree structure remains balanced. The paths starting from the root down to the leaves should differ by at most one in length, thereby allowing insertion, removal and look-up in time $O(\log n)$. Some of the traditional balancing techniques include AVL [10, 13] and red-black trees [28], and as described in Section 2.2.5, a BST can also be probabilistically balanced.

2.2.4 Heap

Most commonly used in the context of priority queues, a heap [13] is a tree-based data structure associating with each node a priority. At all times, it preserves two important properties:

- The shape property, requiring that the heap is a complete binary tree;
- The heap property, requiring that every node has a lower or equal priority with respect to its parent.

Traditional heap operations are not in the scope of this dissertation, and Section 2.2.5 will merely refer to the heap property when describing the treap.

2.2.5 Treap

Being a probabilistic data structure, a treap is a randomized search tree associating with each entity a key and a randomly selected priority [2, 4]. Treaps enforce the BST property with respect to keys, the heap property with respect to priorities, and are also set-unique. Set-uniqueness ensures the tree structures of identical collections to be equivalent, thereby implying history independence if priorities are assigned deterministically. Because history independence is an appealing property, cryptographic hash functions have previously been used to transform\(^2\) treaps into deterministic treaps [8, 24].

\(^2\) Simply apply the hash function $H$ to the key $k$, yielding priority $H(k)$. 

9
When an entity is inserted into or removed from a treap, its location is first determined by a binary search. In the case of insertion, the node is inserted in place and then rotated upwards in the tree until the heap property is fulfilled. In the case of removal, the node is instead rotated downwards and discarded once it is a leaf. As priorities are assigned randomly, this construction yields a probabilistic balance. Referring to the average case, expected time for key look-up, insertion, and removal is $O(\log n)$, and expected amount of rotations during insertion and removal are $O(1)$ [2]. The worst case is $O(n)$, however [8].

2.2.6 Hash Treap

Representing entities with keys $k$ and values $v$, a hash treap [8, 24] is a lexicographically sorted history independent key-value store combining a regular Merkle tree and a deterministic treap. Yielding a construction where each node is associated with an entity and every (non-)member has a unique position, hash treaps support efficient (non-)membership proofs. These are represented by means of pruned hash treaps, as shown in Figure 2.5.

![Figure 2.5](image)

(a) Membership proof for key $k = 15$.  
(b) Non-membership proof for $k = 9$.

Figure 2.5: A simplified hash treap with associated values omitted. Dashed edges represent paths $P$ determined by binary searches for keys $k$, and circled nodes constitute the corresponding (non-)membership proofs.
Similar to regular Merkle trees, a proof must necessarily include a Merkle audit path. Due to encoding hashes differently\footnote{Setting the hash of an empty child to a string of consecutive zeros, the hash of each node is computed as $H(k||v||LeftHash||RightHash)$.}, however, information regarding key-value pairs are also required when recomputing the root hash. Denoting by $P$ the path starting at the root down to node $x$ being verified, the minimal (non-)membership proof includes the Merkle audit path, the hash of each child of $x$, and all $(k,v) \in P$.

### 2.3 Balloon

Viewed as a persistent authenticated data structure in the \textit{three-party setting} \cite{27}, Balloon \cite{24} is an append-only key-value store. As shown in Figure 2.6, a trusted \textit{author} maintains a Balloon at an untrusted \textit{server}. Clients issue (non-)membership queries to the server, and are able to verify the correctness of each query based only on public information and the trust of the author. The construction is persistent due to allowing queries to past versions of the Balloon, and it is notably achieved while keeping memory requirements minimal \cite{24}.

Figure 2.6: Balloon viewed in the three party setting.

Balloon relies on the modest assumption of a collision resistant hash function, an existentially unforgeable digital signature scheme under known message attack, and a \textit{perfect}
gossiping mechanism. That is, similar to related work [17, 25], public information referred to as snapshots must be exchanged between clients, auditors and monitors to ensure that the untrusted server is not presenting contradicting views to different clients. Provided this setting, Balloon is provably correct and secure [24] with respect to the authenticated data structure scheme defined by Papamanthou et al. [23].

2.3.1 Key Building Blocks

Balloon combines a hash treap and a history tree. The hash treap serves the purpose of proving non-membership, and it also determines if an event is included in the version \( v \) Balloon \( B_v \). The latter requires encoding extra information referred to as epochs [24], which are basically intervals of events. That is, the hash treap can store the index \( i \) of a leaf in the history tree containing the digest of an event \( e \), and if \( i \) is included in the first \( v \) epochs then \( e \in B_v \). Whenever \( e \in B_v \) has been proven using the hash treap, the role of the history tree is to provide a Merkle audit path for the event leaf. The history tree also serves the purpose of showing consistency between different versions of the Balloon.

2.3.2 Updating

Inserting an event into the Balloon is generally the process of updating a non-authenticated data structure \( D \), the hash treap, and the history tree. The hash treap and the history tree are collectively denoted by \( \text{auth}(D) \), and they merely serve the purpose of authentication. Hence, inserting an event \( e \) containing key \( k \) and value \( v \) is a three-step procedure defined as follows [24]:

1. The key-value pair \((k, v)\) is inserted into \( D \);

2. The digest \( H(k \| v) \) is inserted into the history tree at leaf \( i \);

3. The index \( i \) is inserted into the hash treap with key \( H(k) \).
2.3.3 Snapshots

In the context of Balloon, a snapshot is defined as the root hash of the hash treap, the root hash of the history tree, and a digital signature for those [24]. The signature is issued by the author, and it is used by the untrusted server to prove correct behaviour.

2.3.4 Algorithms

The **genkey**, **setup**, **refresh**, **update**, **query**, and **verify** algorithms are used during setup, initialization and maintenance of the Balloon. They are inherited from the authenticated data structure scheme defined by Papamanthou *et al.* [23], and Pulls and Peeters [24] extend this scheme by also defining **query** (Prune), **verify** (Prune), update*, and **refreshVerify**. The three former algorithms concern efficiency in terms of space, and the last algorithm enforces *publicly verifiable consistency*. It is a property that allows anyone to verify whether a set of events was correctly inserted into the Balloon, but because the algorithm itself is neither interesting for the purpose of this dissertation nor the understanding of Balloon it is considered no further.

Using **KGen** as a subroutine, the **genkey** algorithm generates the author’s verification key \( vk \) and secret key \( sk \). In addition, it defines the authors public key \( pk = \{vk, \Omega\} \) after picking a deterministic function \( \Omega \) that orders a set of events when inserting into the history tree. The **setup** and **update** algorithms are also intended for the author. Given a set of events they initialise and update the Balloon respectively, generating snapshots using \( sk \).

The **refresh** algorithm is intended for the untrusted server, and it is closely related to the **update** algorithm. Given a set of events it updates the Balloon, but due to having no access to the author’s secret key it is unable to generate the next snapshot. Hence, the next snapshot must always be provided as input by the author, thereby allowing the **update** algorithm to generate the new state of the Balloon in a verifiable manner.

The **query** algorithm is used to query version \( v \) of the Balloon for (non-)membership with respect to an event key \( k \). The proof consists of up to three parts:
1. A (non-)membership proof in the hash treap for key $H(k)$;

2. The index $i$ in the history tree (if applicable);

3. A membership proof for leaf $i$ in the history tree (if applicable).

If no index with key $H(k)$ is found, the query result is \texttt{false} and the proof of correctness consists of the non-membership proof in the hash treap. If an index with key $H(k)$ is found but it is not contained within the first $v$ versions, the query result is also \texttt{false}. The proof of correctness consists of the membership proof for key $H(k)$ in the hash treap, along with the retrieved index $i$. Finally, if an index with key $H(k)$ is found and it is also contained within the first $v$ versions, the query result is \texttt{true}. The proof of correctness consists of the membership proof for key $H(k)$ in the hash treap, the retrieved index $i$, and the membership proof for leaf $i$ in the history tree.

The \texttt{verify} algorithm determines if a query result and its proof of correctness are valid. Given a snapshot it checks if the (non-)membership proof in the hash treap is authentic, if the index $i$ is in the claimed version (if applicable), and if the membership proof in the history tree is authentic (if applicable). It outputs \texttt{true} if the answer is correct with respect to the proof and the snapshot, else \texttt{false}.

The algorithms used to increase efficiency concern \texttt{pruning} of $\text{auth}(D)$. Given a set of events $u$ that are non-members, $\text{pruned}(\text{auth}(D), u)$ denotes the pruned Balloon for $u$. It includes just enough information to insert all events $e \in u$ into $\text{auth}(D)$. This is useful because the author can safely discard both $D$ and $\text{auth}(D)$ and query for the essential parts in a verifiable manner before generating the next snapshot as follows:

1. Use \texttt{query (Prune)} to obtain non-membership proofs for all $e \in u$, or a proof that at least one $e \in u$ is already a member of the Balloon;

2. Verify the correctness of the result returned by \texttt{query (Prune)} using \texttt{verify (Prune)};
3. Let \texttt{update*} build $\text{pruned}(\text{auth}(D), u)$ based on the output from \texttt{query (Prune)}, insert each event $e \in u$ into $\text{pruned}(\text{auth}(D), u)$, and sign the new snapshot using the author’s secret key $sk$.

The \texttt{query (Prune)} algorithm uses \texttt{query} as a subroutine, and similarly \texttt{verify (Prune)} uses \texttt{verify}. The pruned Balloon for $u$ consists of a pruned hash treap and history tree respectively, and in the former case all non-membership proofs for $e \in u$ are included with redundant nodes removed. In the latter case, a single membership proof for the final inserted event suffices (this is due to the append-only nature of the history tree). As an example of a pruned hash treap that refers to Figure 2.5, consider pruning for the two non-membership proofs generated by $3 \leftarrow H(k_1)$ and $9 \leftarrow H(k_2)$. The result is shown in Figure 2.7, and notably all but one node would have been redundant if two separate proofs were sent from the server to the author. More generally, Pulls and Peeters [24] show that pruning roughly reduces the information sent from the server by a factor of two, and that it is linear in the size of $u$.

Figure 2.7: A pruned hash treap for the non-membership proofs generated by $3 \leftarrow H(k_1)$ and $9 \leftarrow H(k_2)$. Circled nodes would be redundant without applying the prune operation.
2.4 Summary

A Merkle tree is the key building block when discussing authenticated data structures and dictionaries. It is a binary tree that incorporates the use of cryptographic hash functions, and its most prevalent feature is that of subtree summaries. Using these, it is possible to generate efficient membership proofs that show particular values as present in the Merkle tree. A proof is composed of a Merkle audit path, verified by reconstructing the root hash, and believed to be valid if it matches a trustworthy root (e.g., one that is digitally signed).

History trees and hash treaps are two types of Merkle trees that are used in the context of Balloon. A history tree is naturally persistent in the sense that it can reconstruct past views by “forgetting” events, and it also supports efficient generation of incremental proofs. Unlike a hash treap, however, it is not lexicographically sorted. Therefore a history tree cannot generate efficient non-membership proofs, whereas a hash treap can. This follows from the observation that a treap is probabilistically balanced, and a binary search can determine if an entity is a (non-)member in expected logarithmic time. Hence, after also defining a collection of algorithms that outline how the two authenticated data structure should collaborate, Balloon supports efficient (non-)membership queries to current and past versions of the Balloon.
3 Sparse Merkle Tree

In this section the sparse Merkle tree (SMT) is incrementally described. Section 3.1 begins with a brief overview, introducing the reader to the SMT. Section 3.2 proceeds to formally define the SMT. Section 3.3 describes the rationale behind the approach. Section 3.4 defines recursive relationships, and for completeness the concept of relative information is used and later described in Section 3.5. The final proposal is presented in Section 3.6.

3.1 Notion

As proposed by Bauer [3] and Laurie and Kasper [16], a SMT is a Merkle tree of intractable size. The depth is fixed in advance with respect to the underlying hash function $H$, meaning there are always $2^{|H(\cdot)|}$ leaves. These are referred left-to-right by indices, and are associated with either default or non-default values. In the latter case the hash of a key determines the index, which implies there is a unique leaf reserved for every conceivable digest $H(k)$. This is the novelty of the SMT, and allows generation of (non-)membership proofs using regular Merkle audit paths.

Considering that the output of a standardized cryptographic hash function is generally 256 bits or more [1], the idea of a tree containing at least $2^{257} - 1$ nodes does indeed sound crazy\(^4\). The key observation, however, is that the SMT is sparse. The large majority of all leaves will be empty, and consequently most nodes rooted at lower levels of the tree derive identical default hashes. Each empty leaf associated with the default value $\Gamma$ computes $h \leftarrow H(\Gamma)$, all parents whose children are empty leaves derive $h' \leftarrow H(h||h)$, and so on. These default hashes must not be derived explicitly, and are preferably cached in advance.

Figure 3.1 illustrates a simplified SMT that has $2 \leftarrow n$ entities, a cache with default hashes, and a hash function $H$ producing digests of $4 \leftarrow |H(\cdot)|$ bits. When deriving the root

\(^4\) For instance, the Shannon number that describes a conservative lower bound for the complexity of chess is $10^{120} \approx 2^{360}$. Using a machine that operates at one variation per microsecond, Shannon [26] estimates that it takes $10^{90}$ years to determine the first move in a perfect game.
hash for the first time, all white nodes that depend on non-empty leaves must be processed. Grey nodes, however, have associated default hashes and can always be requested from the cache. Therefore the SMT has at least one tractable representation that derives root hashes in time $O(n)$. As brief proof sketch, this result is attained by observing there are $n$ paths down to the leaves that are non-default (i.e., grey subtrees need not be visited).

The problem with naïve approaches that visit all non-empty leaves is inefficiency [25]. What we need is some mechanism that prevents such visits, presumably allowing an efficient (logarithmic) representation of the SMT. In subsequent sections this challenge will be addressed using relative information, an approach that provides non-default hashes in various parts of the SMT. Before exploring that in detail, however, let us formalise the SMT and further discuss the rationale behind the approach taken here.

### 3.2 Definition

Let $H$ denote a preimage and collision resistant hash function, $\Gamma$ a constant value, $k$ a key, and $v$ a value. The SMT can be defined in terms of Definitions 1–3 as follows.

**Definition 1 (Sparse Merkle tree).** *Serving as a history independent key-value store, a SMT is an authenticated data structure based on a perfect Merkle tree of fixed size that has $2^{\lvert H(\cdot) \rvert}$ leaves and five associated methods:*
- \( r \leftarrow \text{SMT.Init}(L) \): On input of a list \( L \) containing pairs of distinct keys \( k \) and arbitrary values \( v \), the \text{Init} method extracts all \((k,v) \in L\), inserts each \( v \) into the leaf with index \( H(k) \), computes the cache \( \xi \) with default hashes, prepares some relative information \( \delta \), and outputs the first root hash \( r \);

- \( r \leftarrow \text{SMT.Add}(k,v) \): On input of a key \( k \) and an associated value \( v \), the \text{Add} method inserts \( v \) into the leaf with index \( H(k) \), updates the relative information \( \delta \), and outputs the new root hash \( r \);

- \( r \leftarrow \text{SMT.Remove}(k) \): On input of a key \( k \), the \text{Remove} method resets the leaf with index \( H(k) \) to the default value \( \Gamma \), updates the relative information \( \delta \), and outputs the new root hash \( r \);

- \( \{\alpha_k, P\} \leftarrow \text{SMT.Query}(k) \): On input of a key \( k \), the \text{Query} method outputs an answer \( \{\text{true}, \text{false}\} \leftarrow \alpha_k \) and a proof \( P \) proving this claim. In case of membership, \( \alpha_k \) also contains the associated value \( v \);

- \( \{\text{true}, \text{false}\} \leftarrow P.\text{Verify}(\alpha_k, r) \): On input of an answer \( \alpha_k \) and a root hash \( r \), the \text{Verify} method outputs \text{true} if \( P \) proves \( \alpha_k \) correct with respect to \( r \), else \text{false}.

**Definition 2 (Computing hashes).** An empty leaf computes \( H(\Gamma) \), a non-empty leaf computes \( H(H(k) || v) \), and a parent computes \( H(\text{LeftHash} || \text{RightHash}) \).

**Definition 3 (Sparse Merkle audit path).** Excluding the root, denote by \( P_k \) the list of nodes down to the leaf with index \( H(k) \) and \( s(x) \) the sibling of node \( x \). A sparse Merkle audit path is defined as the minimal proof of (non-)membership for an answer \( \alpha_k \), comprising only non-default hashes associated with nodes in \( P_k^* = \{s(x) \mid x \in P_k\} \).

### 3.3 Approach

Unlike the proposal by Bauer [3], the approach taken here is not based on storing a pruned tree structure in memory. Instead a non-authenticated data structure \( D \) is used for the
purpose of maintaining all key-value pairs inserted into the SMT, and a cache with default hashes is also used to attain a tractable representation. The data structure $D$ supports insertion in time $O(C_a)$, removal in time $O(C_r)$, look-up in time $O(C_l)$, and splitting\(^5\) in time $O(C_s)$. Thereby the challenge with this seemingly odd approach is to map intervals of keys (or more accurately, hashes of keys) to their respective subtrees. One way to achieve that has already been proposed by Laurie and Kasper [16], and this dissertation proposes yet another related solution. However, regardless of how mappings are implemented, the recursive nature of deriving the root hash boils down to four cases:

- **an empty leaf is reached**: apply Definition 2;
- **a non-empty leaf is reached**: apply Definition 2;
- **an empty subtree is reached**: request a cached hash;
- **a non-empty subtree is reached**: ask recursively for the hash of each child, then apply Definition 2.

In order to distinguish between different subtrees, parameters regarding the depth $d$ and the relative position of a node can be used. The base is a bitmap initialised as all zeros containing $|H(\cdot)|$ bits. It refers to the index of the first leaf in a subtree, and combined with the depth it uniquely determines a node in the SMT. As such, the base remains the same on left traversals, but on right traversal the appropriate bit must be flipped to a one. This process can informally be viewed as starting from an empty string, and for each left (right) traversal a zero (one) is appended. Moreover, by setting the $d - 1$ least significant bits in the base to ones, the index of the last leaf in the left subtree can be determined.

Thus, the combination of base, depth, and splitting allow dividing $D$ appropriately for any two subtrees that are rooted at the children of a node.

\(^5\) Splitting refers to the ability of dividing $D$ into two data structures containing all key-value pairs that compare lesser than (greater than or equal to) a split value. Another way to approach splitting is to use references that refer to the start and the end of the data structure, respectively.
A concrete example that derives the root hash is provided in Figure 3.2. A recursive traversal starts at the root with depth $d \leftarrow 3$, base $b \leftarrow 000_2$, and non-authenticated data structure $D \leftarrow \{(k_1, v_1), (k_2, v_2)\}$. All but the first and the last leaves are empty, thereby implying that $H(k_1) = 000_2$ and $H(k_2) = 111_2$. At each left (right) traversal the depth is reduced by one, the base remains the same (is updated), and $D$ is split into new data structures based on the index of the last leaf in the left subtree. A traversal stops when either a (non-)empty leaf or an empty subtree is reached, and the root hash is eventually obtained as the recursion wraps around. The formalised recursion will be presented next.

**Figure 3.2:** An illustration of a recursive traversal that obtains the root hash. Dashed components need not be visited.

### 3.4 Recursive Relationships

Recursions 1–4 outline the derivation of a cache with default hashes, subtree root hashes, and sparse Merkle audit paths. The notation in Table 3.1 is used, and the concept of relative information $\delta$ is included for completeness. Simply view it as an oracle: a system entity that provides normally unavailable information regarding non-default hashes.
Table 3.1: The notation used when describing recursive relationships.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D[n]$</td>
<td>An ordered data structure containing entities $(D_0, D_1, \ldots, D_{n-1})$</td>
</tr>
<tr>
<td>$D[a : h]$</td>
<td>New ordered data structure containing entities $(D_a, D_b, \ldots, D_{h-1})$</td>
</tr>
<tr>
<td>$D[n] : D'[m]$</td>
<td>New ordered data structure containing entities $(D_0, \ldots, D_{n-1}, D'<em>0, \ldots, D'</em>{m-1})$</td>
</tr>
<tr>
<td>$D^&lt;_s$</td>
<td>New ordered data structure containing all entities $D_i$ such that $\text{true} \iff D_i &lt; s$</td>
</tr>
<tr>
<td>$D^&gt;_s$</td>
<td>New ordered data structure containing all entities $D_i$ such that $\text{true} \iff D_i &gt; s$</td>
</tr>
<tr>
<td>$\beta^n$</td>
<td>An $n$-bit string containing $\beta\ldots\beta$, where $\beta \in {0, 1}$</td>
</tr>
<tr>
<td>$a</td>
<td>\beta_i$</td>
</tr>
<tr>
<td>$a</td>
<td>\beta^n$</td>
</tr>
<tr>
<td>$a|b$</td>
<td>Concatenates $a$ and $b$</td>
</tr>
<tr>
<td>$\Lambda$</td>
<td>The empty string</td>
</tr>
</tbody>
</table>

Recursion 1 (Deriving default hashes). Starting from the predefined constant $\Gamma$, the cache $\xi[N + 1]$ with default hashes $(\xi_0, \ldots, \xi_N)$ is derived as follows:

$$
\xi_i = \begin{cases} 
H(\Gamma), & i = 0 \\
H(\xi_{i-1}\|\xi_{i-1}), & i \in [1, N]
\end{cases}
$$

where $N \leftarrow |H(\cdot)|$ is henceforth the size of a digest outputted by the underlying hash function $H$ in bits.

Recursion 2 (Root hash). Starting from the base $b \leftarrow 0^N$, depth $d \leftarrow N$, and data structure $D$ containing key-value pairs $(D^<_i, D^>_i)$, the root hash is derived as follows:

$$
\text{RH}(D[n], d, b) = \begin{cases} 
\delta_{d,b}, & \text{if available} \\
\xi_d, & n = 0 \\
H(b\|D^>_0), & d = 0, n = 1 \\
\text{SubRH}(D[n], d, b), & \text{else}
\end{cases}
$$

$$
\text{SubRH}(D[n], d, b) = H(\text{RH}(D^<_s, d - 1, b)\|\text{RH}(D^>_s, d - 1, b|1_j))
$$
where \( s \leftarrow b|1^{d-1} \) is the split index, \( j \leftarrow (N - d) \) the bit in the base that must be set to a one on right-traversals, and \( D_s^* \) applies to the hash of the keys.

**Recursion 3 (Merkle audit path).** Traversing the SMT down to the leaf with index \( h^k \leftarrow H(k) \), all hashes in the Merkle audit path are grouped into a list as follows:

\[
\text{MAP}(D[n], d, h^k) = \begin{cases} 
\Lambda, & d = 0 \\
\text{MAP}(D_s^{\leq}, d - 1, h^k) : RH(D_s^{\geq}, d - 1, b|1_j), & d \neq 0, h^k_j = 0 \\
\text{MAP}(D_s^{>}, d - 1, h^k) : RH(D_s^{\leq}, d - 1, b), & \text{else}
\end{cases}
\]

where \( d, D, D_s^*, j, s \) are interpreted as in Recursion 2, \( b \leftarrow h^k|0^d \) is the base, and \( h^k_j \) is the \( j^{th} \) most significant bit in \( h^k \).

**Recursion 4 (Sparse Merkle audit path).** Starting from a Merkle audit path \( M \) that was returned by Recursion 3, all redundant hashes are removed as follows:

\[
\text{SMAP}(M[n]) = \begin{cases} 
\Lambda, & n = 0 \\
\text{SMAP}(M[0 : n - 1]) : \Lambda, & n \neq 0, M_{n-1} = \xi_{n-1} \\
\text{SMAP}(M[0 : n - 1]) : M_{n-1}, & \text{else}
\end{cases}
\]

where an \( N \)-bitmap specifying the (non-)default hashes must also be encoded; setting the \( d^{th} \) bit in the map to a one indicates the hash at depth \( d \) to be an included non-default hash.

### 3.5 Relative Information

While modelling relative information, it is intuitive to introduce an abstract model. If we proceed to use an oracle abstraction, the aim is to find one or many different representations that provide hashes on request. That is, the purpose of an oracle is to add shortcuts in the
SMT, effectively preventing recursive traversals down to the leaves. These models can be implemented as two-dimensional data structures that are indexed by depth and base. They should support querying in time $O(1)$ and must not become the bottle-neck upon updates on use in recursions. Now, in order to get the discussion started, consider Oracles 1–2.

**Oracle 1 (Memoryless oracle).** *A memoryless oracle maintains no relative information.*

**Oracle 2 (Perfect oracle).** *A perfect oracle records every non-default hash in the SMT.*

The memoryless oracle is implicitly chosen if no relative information is maintained. Clearly, it is not very helpful and the perfect oracle seems to be much more promising. Although it puts a great burden on memory, it is the perfect representation in terms of time. That is, using Oracle 2, any non-default hash is provided on request. Therefore Recursion 2 runs in time $O(1)$, and consequently Recursion 3 runs in time $O(C_s)$. Unfortunately, recording every non-default hash is exhausting (imagine being the oracle). Attempting to find a reasonable trade-off between time and space, Oracles 3–5 are defined.

**Oracle 3 (Branch oracle).** *Referring to a node as a branch when each child derive a non-default hash, a branch oracle records every hash rooted at a branch.*

**Oracle 4 (Depth oracle).** *Denoting by $d$ the depth that may fluctuate over time, a depth oracle records all non-default hashes rooted at the first $d$ levels of the SMT.*

**Oracle 5 (BD-hybrid oracle).** *A BD-hybrid oracle combines Oracle 3 and Oracle 4, recording every non-default hash rooted at a branch and at the first $d$ levels of the SMT.*

Figure 3.3 depicts the hashes recorded by Oracles 3–5. The depth of the depth-based oracles is set to three, grey (white) nodes derive default (non-default) hashes, circled nodes represent branches, and diamond nodes are recorded by depth-based oracles. The example is particularly interesting due to *clumping* at the leaves, however. The branch oracle records

---

Do not be confused by the depth being variable; it merely refers to the fact that the oracle need not decide statically in advance how many levels to record. That is, the depth of the SMT is always fixed.
Figure 3.3: Hashes recorded by the branch, depth, and BD-hybrid oracles. Circled nodes represent branches, and diamonds are recorded by depth-based oracles.

An essential hash that is not covered by the depth oracle at level four, and the result is a depth oracle that cannot be fully utilised (one grey node at level three).

More generally, it is a good idea to set the depth of a depth-based oracle to \( \lceil \log n \rceil + 1 \). This follows from the observation that the output of a cryptographic hash function is statistically independent \([1]\), thereby implying that non-empty leaves will be evenly spread out. Thus, there will be roughly one non-empty leaf per subtree at level \( \log n + 1 \). Further, the sparse part of the SMT starts at levels greater than \( \log n + 1 \), and recording additional levels would only prevent recursive visits to the leaves in rare cases.

Querying the relative information is trivial when there is access to the base and the depth. Hence, after obtaining some prevalent oracle model, the final piece of the puzzle is to maintain and create it in the first place. Recursion 5 describes how to update the relative information on insertion and removal, and it uses the hash maybe cache (HMC) routine. Given the left and the right child hash, it computes the hash rooted at a node and gives the oracle an opportunity to update the relative information accordingly.

**Recursion 5 (Update).** Traversing the SMT down to the leaf with index \( h^k \leftarrow H(k) \) and
value \( v \), the update routine derives the new root hash as follows:

\[
\text{Upd}(D[n], d, h^k, v') = \begin{cases} 
H(v'), & d = 0 \\
\text{HMC}(\text{Upd}(D_s^≤, d - 1, h^k, v'), \text{RH}(D_s^>, d - 1, b|1_j), d, b), & d \neq 0, h^k_j = 0 \\
\text{HMC}(\text{RH}(D_s^≤, d - 1, b), \text{Upd}(D_s^>, d - 1, h^k, v'), d, b), & \text{else}
\end{cases}
\]

where \( b, d, D, D^*_s, h^k_j, j, s \) are interpreted as in Recursion 3, \( v' \) is either \( \Gamma \) or \( h^k||v \) depending on if the leaf has been reset or associated with a new value, and \( \text{HMC}(\text{LHash}, \text{RHash}, d, b) \) computes the new hash rooted at depth \( d \) with base \( b \), also updating the relative information \( \delta_{d,b} \) according to the oracle model.

The update routine must always be invoked after a key-value pair has been inserted into or removed from the non-authenticated data structure \( D \). It returns the new root hash, and ensures that the relative information is in a correct state. Depending on the oracle model, however, the \( \text{HMC} \) routine may also have to be integrated into \( \text{SubRH} \) in Recursion 2. For instance, that is the case when maintaining depth-based oracles in a plug-and-play manner (hashes are cached as they are derived for the first time).

### 3.6 Final Proposal

Consider the current view of the SMT. It does not maintain an explicit tree structure, and it uses a non-authenticated data structure to keep track of key-value pairs, succinct recursions to derive (non-)membership proofs, and relative information to increase the overall efficiency. As a final proposal, to clearly state how each of these components are combined to satisfy Definitions 1–3, Algorithms 1–5 are defined.

**Algorithm 1** \((r \leftarrow \text{Init}(L))\). On input of a list \( L \) containing pairs of distinct keys \( k \) and arbitrary values \( v \), the \text{Init} algorithm chooses an oracle model, derives the cache \( \xi \) with default hashes using Recursion 1, invokes Algorithm 2 for all \((k, v) \in L\) to initialise the non-authenticated data structure \( D \), and outputs the first root hash \( r \) using Recursion 2.
Algorithm 2 \((r \leftarrow \text{Add}(k,v))\). On input of a key-value pair \((k,v)\), the \text{Add} algorithm inserts \((k,v)\) into the non-authenticated data structure \(D\) (overwriting any previous pair with the same key), gives the oracle an opportunity to perform initial adjustments to the relative information \(\delta\), and uses Recursion 5 to refresh \(\delta\) and output the new root hash \(r\).

Algorithm 3 \((r \leftarrow \text{Remove}(k))\). On input of a key \(k\), the \text{Remove} algorithm removes the key-value pair \((k,v)\) from the non-authenticated data structure \(D\) (if applicable), gives the oracle an opportunity to perform initial adjustments to the relative information \(\delta\), and uses Recursion 5 to refresh \(\delta\) and output the new root hash \(r\).

Algorithm 4 \(\{\alpha_k, P\} \leftarrow \text{Query}(k)\). On input of a key \(k\), the \text{Query} algorithm determines the answer \(\alpha_k \leftarrow k \notin D\), uses Recursion 3 and 4 to obtain the (non-)membership proof \(P\) comprising a sparse Merkle audit path and an \(|H(\cdot)|\)-bitmap specifying the non-default hashes, appends the associated value \(v\) to \(\alpha_k\) if \(\alpha_k \neq \text{true}\), and outputs \(\{\alpha_k, P\}\).

Algorithm 5 \(\{\text{true}, \text{false}\} \leftarrow \text{Verify}(\alpha_k, r)\). Given a proof \(P\), an answer \(\alpha_k\), and an authentic root hash \(r\), the \text{Verify} algorithm derives the cache with default hashes \(\xi\) using Recursion 1, reconstructs the root hash \(r'\) using the knowledge of \(P\), \(\alpha_k\), and the associated value \(v\) in case of membership, and outputs \(r' \overset{?}{=} r\).

Apart from a minor detail, recomputing the root hash in Algorithm 5 is analogous to recomputing the root hash in a regular Merkle tree (see Section 2.2.1). The one difference is that the \(|H(\cdot)|\)-bitmap must always be consulted before using the next hash in the sparse Merkle audit path, indicating whether the next hash is implicit or explicit. Per definition, referring to Recursion 4, the next hash at depth \(d\) is implicit and obtained from \(\xi_d\) whenever the \(d^{th}\) bit in the map is zero. In all other scenarios, when the \(d^{th}\) bit is one, the next hash in the sparse Merkle audit path is used. Note also that the order in which hashes are concatenated need not be provided by the proof; knowledge of the key suffices to derive the index of the leaf in question.
4 Analysis

This section analyses the efficiency of the SMT. Section 4.1 proves memory requirements for each oracle model and (non-)membership proofs. Section 4.2 proceeds with a similar analysis concerning time. Section 4.3 presents performance results. Section 4.4 considers how and if an adversary can cause inefficiency. Finally, Section 4.5 provides a summary.

4.1 Space Complexity

Throughout the remainder of section 4, let \( n \in \mathbb{N} \) denote the number of non-empty leaves in the SMT, \( d \leftarrow (\lceil \log n \rceil + 1) \) the depth of all depth-based oracles, and \( N \leftarrow |H(\cdot)| \) the size of a digest in bits. Further suppose that \( n \) is a power of two to simplify the analysis and interpretation of Theorems 1–11.

**Theorem 1 (Memoryless oracle).** Oracle 1 records no hashes.

*Proof.* This follows directly from the definition of Oracle 1. \( \blacksquare \)

**Theorem 2 (Perfect oracle).** Oracle 2 records at most \( n(N - \log n + 2) - 1 \) hashes.

*Proof.* In order to yield the upper bound, the idea is to have as little overlap as possible between paths of non-default hashes. This is achieved by inserting values such that they are evenly spread out at the leaves, meaning there will be \( n \) paths starting at level \( \log n + 1 \) without overlap. One way to view this scenario is as follows: \( n \) paths start at the root down to the leaves, overlapping entirely at the root node. At the children of the root, the overlap is reduced to 50%. Similarly, at the children of the children of the root, the overlap is reduced to 25%. When level \( \log n + 1 \) is reached the overlap stops, and visually the upper part of the SMT is a perfect binary tree of height \( \log n \) with overlapping paths. Thus, the upper part of the SMT has \( 2^{\log n} - 1 = n - 1 \) non-default hashes.

All remaining non-default hashes are found at levels greater than \( \log n \). Therefore we consider how many hashes are contained in the remaining \( n \) non-overlapping paths that
start from level \( \log n + 1 \). A path from the root contains \( N + 1 \) hashes. Consequently, removing the first \( \log n \) hashes leaves only \( N + 1 - \log n \) hashes. Thus the lower part of the SMT contains \( n(N - \log n + 1) \) hashes, and when also considering the hashes from the upper part the total amount is \( n - 1 + n(N - \log n + 1) = n(N - \log n + 2) - 1 \).

Theorem 3 (Branch oracle). Oracle 3 records \( n - 1 \) hashes.

Proof. Let \( S(n) \) be a function that determines how many hashes are recorded by the branch oracle. The claim \( S(n) = n - 1 \) is proven correct by induction for all \( n > 0 \) as follows.

In the base case, there is only a single value associated with leaf \( x \). Hence, there is one path down to \( x \) that contains non-default hashes, and all remaining nodes have associated default hashes. Thus, there are no two siblings that both derive non-default hashes, thereby implying there are no branches and \( S(1) = 0 \).

During the induction step, let \( n = k \) and suppose that \( S(k) = k - 1 \) is true. When a value is associated with a previously empty leaf \( y \), a new branch is created at the first ancestor shared by \( y \) and any of the other \( k \) existing non-empty leaves. Because no branches are removed by inserting another value into the SMT, the new branch count will be \( S(k) + 1 \). Thus, \( S(k + 1) = S(k) + 1 = k \).

Theorem 4 (Depth oracle). Oracle 4 records \([\log n + 1, 2n - 1]\) hashes.

Proof. The lower bound is determined by considering how \( n \) values can be associated with the SMT. On insertion of the first value, \( \log n + 1 \) non-default hashes must always be recorded due to the inevitable path of non-default hashes down to leaf \( x \). Succeeding values, however, can be inserted such that their leaves share the ancestor of \( x \) at level \( \log n + 1 \). Hence, no additional hashes are recorded and the lower bound is \( \log n + 1 \).

The upper bound requires non-empty leaves to be evenly spread out, meaning that each subtree rooted at level \( \log n + 1 \) contains a single non-default value. Visually this implies the depth oracle to be a perfect binary tree of height \( \log n + 1 \), effectively recording \( 2^{\log n+1} - 1 = 2n - 1 \) non-default hashes.
Theorem 5 (BD-hybrid oracle). Oracle 5 records at most $2n - 1$ hashes.

Proof sketch. The upper bound for the BD-hybrid can be proven using a greedy algorithm. The idea is to insert values one at a time, and for each insertion the aim is to always have the maximum number of hashes inserted into the relative information. The approach is sufficient because trading the locally best option with one that is not part of the set of greedy solutions is never worthwhile, meaning that a better option never arises due to first placing a value strategically. The full proof can be found in Appendix A.

The lower bound for the BD-hybrid is nontrivial to derive. With reasonable accuracy, however, it is approximated by $n$. This follows from the observation that all values can be inserted into one subtree at level $\log n + 1$, thereby only utilising the depth oracle on the first insertion.

Theorem 6. A membership proof is expected to contain $\log n$ hashes, and a non-membership proof is expected to contain $\log n + 1$ hashes.

Proof. Since the output of a cryptographic hash function is statistically independent [1], each insertion yield a random walk down to some leaf. As $n$ such inserts are performed, expected case refers to the scenario where each subtree at level $\log n + 1$ contains a single non-empty leaf. Therefore a sparse Merkle audit path includes $\log n$ hashes from the first $\log n + 1$ levels (the root is excluded), and in the case of membership that suffices. Let us use a proof of contradiction to show why that is.

Suppose that a membership proof includes more than $\log n$ hashes. When the traversal down to the non-empty leaf proceeds past level $\log n + 1$, there are $\log n$ hashes included in the sparse Merkle audit path. Consequently, one or more of the remaining siblings must be the roots of non-empty subtrees. If that ever occurs, however, a contradiction is attained due to multiple non-empty leaves in the subtree rooted at level $\log n + 1$. Therefore our assumption must be wrong, and $\log n$ hashes suffices in the expected case.
In order to show that a non-membership proof requires one more hash, consider the traversal down to the empty leaf \( x \). The initial \( \log n \) hashes are first included as for membership, and at each level greater than \( \log n + 1 \) two situations can appear. Either the traversal proceeds towards the non-empty leaf, or it moves into an empty subtree towards \( x \). In the former case the sibling has a default hash and the same situation appears again at the next level. In the latter case the sibling is the non-empty subtree, the hash of the sibling is included, and thereafter the subtree is always empty. As such, a single additional hash must always be included when moving into the empty subtree for the first time. ■

4.2 Time Complexity

It is convenient to analyse time in terms of how many hashes must be computed during generation of (non-)membership proofs. This is the only factor that differs depending on the oracle model, and it is independent of the underlying non-authenticated data structure. In addition, such an analysis is similar to the case of insertion and removal. Disregarding the \( N + 1 \) hashes that must change on updates, associating a value with an empty leaf is equivalent to the generation of a non-membership proof. Similarly, updating the value of a non-empty leaf is equivalent to the generation of a membership proof. These observations follow from the fact that Recursion 3 and 5 does identical recursive traversals whenever the key is the same.

Theorem 7 (Memoryless oracle). Denote by \( x \) the leaf being authenticated, \( P \) the path down to \( x \), \( m \) the total number of non-default hashes, and \( m' \) the number of non-default hashes rooted at nodes in \( P \). Using Oracle 1, generating a (non-)membership proof for \( x \) requires \( m - m' \) hashes to be computed.

Proof. Recall that a Merkle audit path contains the minimal amount of hashes required to recompute the root hash. It is constructed by requesting all hashes rooted at siblings of nodes in \( P \), and given these the root hash can be derived by computing the remaining
$N + 1$ hashes (these are rooted at the nodes in $P$). Thus, none of the $m'$ hashes need be computed on generation of (non-)membership proofs, but all the other $m - m'$ hashes must be computed due to the Merkle audit path being a minimal representation and there being no relative information available.

**Theorem 8 (Perfect oracle).** Using Oracle 2, no hashes are computed on generation of (non-)membership proofs.

*Proof.* Since all non-default hashes are recorded by the perfect oracle, the hash of an arbitrary node can always be determined by mere table look-up. That is, if the hash rooted at a node is default it is already precomputed, and if it is non-default it is inserted into the relative information. In either case the hash is available, implying that no further computation is ever needed.

**Theorem 9 (Branch oracle).** Using Oracle 3, $N - \log n + 1$ hashes are expected to be computed on generation of membership proofs, and $2(N - \log n) + 1$ hashes are expected to be computed on generation of non-membership proofs.

*Proof.* Recall that expected behaviour implies non-empty leaves to be evenly spread out. Therefore all branches are formed at the first $\log n$ levels, and the analysis of generating (non-)membership proofs can start the traversal down to leaf $x$ at level $\log n + 1$.

In the case of membership, the traversal proceeds towards $x$ along a path of non-default hashes. The sibling at level $\log n + 1$ has a non-default hash, and it is not available in the relative information. Thereby it must be computed, requiring a traversal down to the non-empty leaf $y$ and computing $N - \log n + 1$ hashes. These are all rooted at the path down to $y$, and no additional hashes are computed due to the SMT being sparse. Once the hash at level $\log n + 1$ is obtained, the original traversal continues down to $x$ and all remaining siblings are associated with default hashes. Thus, $N - \log n + 1$ hashes are computed during generation of membership proofs.
In the case of non-membership, the traversal also continues onto level \( \log n + 1 \). The hash rooted at the sibling is non-default, and is derived by computing \( N - \log n + 1 \) hashes. Unlike the case of membership, however, the traversal down to the non-empty leaf must deviate from the path of non-default hashes at some point (refer to the discussion in Theorem 6). For the purpose of being conservative, suppose that such deviation appears at level \( \log n + 2 \). Then the subtree rooted at the sibling will be non-empty, requiring \( N - \log n \) hashes to be computed (one less than that of level \( \log n + 1 \)). No additional hashes are computed since the remainder of the subtree is empty, and \( (N - \log n + 1) + (N - \log n) = 2(N - \log n) + 1 \) thus serves as a conservative bound for the amount of hashes computed on generation of non-membership proofs.

Theorem 10 (Depth oracle). Using Oracle 4, no hashes are expected to be computed on generation of membership proofs, and \( N - \log n \) hashes are expected to be computed on generation of non-membership proofs.

Proof. The proof is almost identical to the case of a branch oracle. The one difference is that the first \( \log n + 1 \) levels are recorded, meaning that an additional hash is available in the relative information. Fortunately, this is the one non-default hash that had to be derived in the case of membership, and one of two non-default hashes that had to be derived in the case of non-membership. Thus, adjusting Theorem 9 by subtracting the \( N - \log n + 1 \) hashes computed at level \( \log n + 1 \), this theorem is proven correct.

Theorem 11 (BD-hybrid oracle). Using Oracle 5, the same number of hashes as computed by a regular depth oracle is expected on generation of (non-)membership proofs.

Proof. When leaves are evenly spread out, the branch oracle is unable to record any hash that is not already covered by the depth oracle. Thus, the BD-hybrid is expected to behave as a depth oracle.

In order to motivate the BD-hybrid, note that the worst case of the depth oracle occurs when the lower space bound is achieved. That is, the majority of the SMT must be
processed when generating (non-)membership proofs for the non-empty subtree at level \(\log n + 1\). With the BD-hybrid, however, the branch oracle records roughly \(n\) hashes and the overall efficiency is presumably good. The catch is that the branch oracle also has a worst case. When there are branch paths down to the leaves, as shown in Figure 4.1, generating (non-)membership proofs can require processing up to \(N\) leaves (one for each sibling). Fortunately this problem is limited by the fixed height of the SMT, and as \(n\) grows large it is also hard to find the right key-value pairs causing such paths. Thereby we conclude that the branch and BD-hybrid oracles are efficient regardless of how key-value pairs are chosen, whereas the depth oracle struggles in the worst case.

![Figure 4.1: Two branch paths down to the first and the last leaves, respectively.](image)

### 4.3 Performance

The SMT was implemented using the C++ programming language and SHA-256 as the underlying hash function. The non-authenticated data structure \(D\) is a \texttt{std::map} indexed by the hash of a key, and relative information is implemented as a \texttt{std::vector} containing a separate \texttt{std::unordered_map} for each level of the SMT. Therefore insertion, removal, look-up and splitting\(^7\) are performed in time \(O(\log n)\), and relative information is expected to be maintained and queried in time \(O(1)\).

\(^7\) Instead of creating new data structures, splitting refers to the use of iterators that specify the first and the last entity contained in a subtree.
4.3.1 Setup

All benchmarks are the average of 20 independent runs using an Intel Core i5-2500 3.3GHz quad core CPU and 2x8GB DDR3 RAM. The execution environment is a VMware virtual machine, and it is allotted 8GB of RAM. Using this setup, key-value pairs with unique keys are inserted into the SMT. Keys used to query for (non-)membership are also chosen randomly, and a new SMT is initialised whenever the oracle model changes or the number of existing key-value pairs is updated to the next power of two. Moreover, the depth-based oracles are maintained such that they are always up-to-date.

Benchmarks concerning memory and verification time of (non-)membership proofs are executed as a single program, and for each independent run the average of 1000 queries is computed. Remaining benchmarks are executed as a different program, and it starts by querying for 1000 (non-)membership proofs. Then the memory used by the oracle is logged, and finally 1000 distinct key-value pairs are inserted one at a time into the SMT.

4.3.2 Space Benchmarks

Figure 4.2 shows the average size of a sparse Merkle audit path. The logarithmic curve is included as a reference, and both membership and non-membership proofs appear to require $O(\log n)$ space. This is indeed expected considering Theorem 6, but there is no indication whatsoever that a sparse Merkle audit path would be larger in the case of a non-membership proof. It is unknown why this difference could not be measured.

The memory used by each oracle model is shown in Figure 4.3. A depth-based oracle records roughly twice as many hashes as a branch oracle, and a BD-hybrid consistently records more hashes than a regular depth oracle. No data was gathered for the memoryless oracle since it records no hashes, and the perfect oracle was excluded due to excessive memory usage.
4.3.3 Time Benchmarks

Figure 4.4 shows the time used when querying for (non-)membership proofs. As is evident when observing a seemingly linear growth on a logarithmic x-axis, all of the observed oracles allow queries to run in time $O(\log n)$. The perfect oracle provides the shortest query times, but since it is unable to complete the benchmark (hence the drop after $2^{17}$ inserted key-value pairs) it is not of practical interest. Therefore the BD-hybrid proves most efficient when considering the remaining oracles, and a regular depth oracle is almost as fast. The branch oracle is the least efficient, but nevertheless logarithmic in $n$.

The insertion benchmark is shown in Figure 4.5. It looks very much like Figure 4.4, thereby implying that all observed oracles allow insertion in time $O(\log n)$. Due to the
necessity of updating underlying data structures, however, extra overhead is added. That is most evident for the perfect oracle, which significantly outgrows the other oracles due to a larger and more frequently updated relative information.

Figure 4.5: The time used to insert 1000 key-value pairs as a function of oracle model and existing key-value pairs.

Figure 4.4: The time used to query 1000 (non-)membership proofs as a function of oracle model and existing key-value pairs.

The final benchmark concerns the time required when verifying a non-membership proof. As clearly depicted in Figure 4.6, it is constant and performed in roughly 200 $\mu$s. This is indeed no surprise, considering that $N + 1$ hashes are computed regardless of how many of those are (non-)default. Thus, a membership proof is also verified in time $O(1)$. 
Figure 4.6: The time required to verify (non-)membership proofs as a function of existing key-value pairs.

4.4 Minding the Adversary

Consider the most powerful adversary that is given the capability to insert values into leaves with arbitrary indices (implies breaking preimage resistance). The three main attacks that can be launched are consuming oracle memory, increasing proof sizes, and increasing query times. The first is not really an attack, however, considering that the oracle model is not chosen by the adversary (achieving the upper space bound is generally a good thing). As such, let us consider only the latter two approaches that are harmful.

Regardless of the oracle model, the adversary is in complete control of which nodes contain (non-)default hashes. Therefore a sparse Merkle audit path need not be sparse, and reduces to a regular Merkle audit path with \( N \) hashes. This is significantly larger than the expected case of \( \log n \), but nevertheless it is bounded by a fixed constant.

Attacks aimed to increase the query time are more interesting to consider. Since the adversary has complete control over the (non-)default leaves, the worst case for each oracle model can always be triggered. Therefore it is necessary to consider such cases carefully, as done in Section 4.2, and determine whether the oracle model is resilient or not. That is the case for the branch and BD-hybrid oracles, but not the case for the depth oracle. It is not particularly relevant to consider the perfect oracle due to its excessive use of memory, and the memoryless oracle has no worst case.
4.5 Summary

The memoryless and perfect oracles are the most efficient in terms of space and time respectively, but their optimality comes at the price of complete inefficiency in the other extreme of time or space. Thereby the branch, depth, and BD-hybrid oracles are more promising, requiring $O(n)$ space while also supporting insertion, removal\textsuperscript{8}, and generation of (non-)membership proofs in time $O(\log n)$. Moreover, regardless of the oracle model, (non-)membership proofs are expected to require space $O(\log n)$. In the worst case they are bounded by the fixed depth of the SMT, and verification time is always $O(1)$.

This section was concluded by considering the most powerful adversary that is given the capability to insert values into leaves with arbitrary indices. Both the branch and the BD-hybrid oracles are resilient against such an adversary, but a regular depth oracle struggles due to inefficiency in the worst case scenario (all non-empty leaves are in the same subtree rooted at level $\log n + 1$). Thus, the branch and BD-hybrid oracles are the most prevalent models when considering space, time, and security.

\textsuperscript{8} Because removal uses the same update recursion as insertion, no explicit benchmark was shown.
5 Applications to Balloon

This section describes how the SMT is applied to Balloon. Section 5.1 introduces an overview of the approach. Section 5.2 adjusts the current view of the SMT to meet the requirements of Balloon. Section 5.3 describes how the Balloon algorithms are altered. Finally, Section 5.4 evaluates efficiency of the new Balloon.

5.1 Overview

Recall that Balloon comprises a hash treap, a history tree, and a collection of algorithms. The hash treap is mainly used for the purpose of non-membership proofs, and combining it with the history tree allows Balloon to be an interesting hybrid. Since the objective in this section is to replace the hash treap with a SMT, there will be little focus on the history tree. Instead, let us study how the hash treap is used to understand its replacement.

On insertion of a new event \( e \), the digest \( H(k) \) serves as a key when inserting index \( i \) into the hash treap. If \( i \) is inserted into the SMT with key \( H(k) \), the exact same operation can be mimicked. The SMT can also mimic the role of the hash treap when Balloon is queried for (non-)membership, and when the outputted proofs are verified. That is, the hash treap uses Merkle audit paths as (non-)membership proofs, and so does the SMT.

Since the trusted author is able to safely discard \( \text{auth}(D) \) and query the untrusted server for a pruned Balloon, the hash treap supports pruning. As of now there is no such operation supported by the SMT, and thus it must be defined. Similarly, the author must also be able to use the pruned representation when inserting events to generate the next snapshot. These two operations will be defined next.

5.2 Extension of the Sparse Merkle Tree

Definition 4 extends the current view of the SMT to support pruning. A minor reduction is also added in Definition 5 to attain an append-only representation that suits Balloon.
Definition 4 (Extension of the sparse Merkle tree). Building on Definition 1, the following methods are added:

- \textbf{SMT.Prune}(S, u): On input of a set \( S \) containing non-membership proofs \( P \) for events \( u \), the \texttt{Prune} method removes all redundant hashes, encodes additional information \( p \) to determine which hashes were removed, and outputs the pruned representation \( P^S_u \);

- \( P^S_u \texttt{.Root}() \): On invocation of the \texttt{Root} method, all events \( e \in u \) are inserted into the pruned representation and the new root hash \( r \) is outputted.

Definition 5 (Reduction of the sparse Merkle tree). Building on Definition 4, the \texttt{Remove} method is forbidden and the \texttt{Add} method rejects all attempts to change the value of a non-empty leaf.

Considering that many sparse Merkle audit paths are \textit{broken} when removing shared non-default hashes, it is essential to encode additional information \( p \) that determines how to properly reconstruct them. One approach is to order the non-membership proofs \( P \in S \) increasingly with respect to the leaves’ indices, and for each proof also encode one more bitmap specifying if a hash is shared. The first time a shared hash is included in a sparse Merkle audit path the \( i^{th} \) bit can be set to one in both bitmaps, and indicating that a hash is included in a previous proof is specified by setting the \( i^{th} \) bit in the original (additional) bitmap to zero (one). Thereby a search backwards suffices to find a shared hash, stopping when the \( i^{th} \) bit is set in both bitmaps.

On implementation of \( P^S_u \texttt{.Root}() \), all proofs \( P \in P^S_u \) can be “verified” as if they concern membership for the appropriate events’ indices. However, as each root hash is constructed, affected hashes in \( P^S_u \) must also be updated. Then the first root hash captures insertion of the first event, the second root hash insertion of the first two events, and so on. Eventually the root hash that captures all insertions is obtained, and it is outputted as the final result.
5.3 Integration

The nine algorithms described in Section 2.3.4 will now be revisited to show how a SMT can replace the hash treap. The new Balloon uses the revised SMT, as per Definitions 4–5.

5.3.1 Key Setup

Considering that the genkey algorithm merely concerns the author’s keys, there is no interaction with the hash treap. Thus it is not altered.

5.3.2 Update

Inserting an event into the Balloon is essentially equivalent to the procedure described in Section 2.3.2. The one difference is that the final step is redefined as follows: the index \( i \) is inserted into the SMT with key \( H(k) \). Hence, considering that setup, refresh, and update merely use the hash treap via the update procedure, these algorithms remain unaltered.

5.3.3 Query and Verify

Disregarding the role of the history tree, querying for (non-)membership with respect to the version \( v \) Balloon is the process of looking for index \( i \) in the hash treap and proving either its existence or absence. Thus, denoting by \( \ell \) the leaf in the SMT with key \( H(k) \), the following adjustments are required:

- The first part of the proof consists of a (non-)membership proof for \( \ell \);
- The query result is false if \( \ell \) is empty;
- The query result is false if \( \ell \) is non-empty and the retrieved index \( i \) is included after version \( v \);
- The query result is true if \( \ell \) is non-empty and the retrieved index \( i \) is in the first \( v \) versions of the Balloon.
In all of these scenarios, a (non-)membership proof in the SMT is generated and returned as the proof of absence or existence of the index $i$.

Considering that hashes are encoded differently for hash treaps and SMTs, the verify algorithm must be redefined to properly check the validity of a (non-)membership proof. For that purpose it uses Algorithm 5, and no further changes are necessary.

### 5.3.4 Pruned Algorithms

Since `query (Prune)` and `verify (Prune)` use only subroutines, these algorithms remain unaltered. A minor adjustment to the `update*` algorithm is necessary, however. Instead of using the operations defined for the hash treap, it must refer to Definition 4 when processing pruned representations.

### 5.4 Evaluation of the New Balloon

The main issues with the hash treap is implementation complexity and $O(n)$ worst case behaviour. The former is addressed by the SMT using succinct recursions, and the latter by defining a prevalent oracle model (e.g., the branch or the BD-hybrid). Thereby the efficiency of the new Balloon does not depend on a probabilistic balance, and it is also enabled to trade time for space using different oracle models. Moreover, the expected proof size is preserved as $O(\log n)$, but in the worst case the SMT offers a tighter bound than that of the hash treap. This follows from the observation that a sparse Merkle audit path cannot include more than a fixed amount of hashes, whereas the hash treap is linearly bounded.
6 Conclusion

This dissertation proposed a general approach that simulates a SMT. It uses the concept of relative information to attain an efficient representation, and it is maintained using an abstract oracle model. The memoryless, perfect, and depth oracles are not practical due to consuming too much time, requiring excessive amounts of memory, and having an unfortunate worst case behaviour, respectively. The branch oracle, however, provides an efficient representation that is also resilient against an adversary with the capability of inserting values into leaves with arbitrary indices. Similarly, the BD-hybrid oracle offers the same resilience while also improving efficiency in terms of time. This comes at the price of consuming roughly twice as much memory, which is a good example of how an abstract oracle model can be used to trade time for space.

Using either a branch or a BD-hybrid oracle, the performance benchmark shows that insertion, removal, and generation of (non-)membership proofs require time $O(\log n)$. In order to achieve that, the oracles use space $O(n)$. The expected size of a non-membership proof is $O(\log n)$, and in the worst case it is bounded by the depth of the SMT. Proofs are always verified in time $O(1)$, notably regardless of their size.

Apart from formally defining and evaluating the proposed representation of the SMT, this dissertation outlined how to use it in the context of Balloon. The new Balloon has preserved time and space complexities in expected cases, and in worst cases it is improved. As such, a SMT is a viable option when implementing Balloon.

6.1 Project Evaluation

During this project I have had the chance to practice project planning, problem solving, reading academic research papers, and writing a larger documentation. I used the suggested work flow where one starts to write the dissertation as soon as possible, and in the end that worked out without any major issues. It did necessitate that some of the earlier parts
be rewritten later, but considering that writing is generally the process of rewriting that is to be expected.

In terms of how I spent my time, the objective throughout the entire project was to work continuously. The initial time plan was a little bit too optimistic, however. Therefore the work rate was increased during the last two months, which allowed me to complete the project in time.

The next time I am in the position of doing a similar project, I expect that my initial time plan will be better. I would also like to try a different work flow, not because the one used here was bad, but because it is generally a good idea to widen your perspective. Personally, I also view it as more natural to wait with the formal write up until one is deeper into the project.

6.2 Future Work

Currently there is no recursion capable of handling batch inserts. Instead of inserting values one at a time, it would be more efficient to use one recursion that inserts all values. Not only would that reduce the amount of reads from and writes to the relative information, but it also minimizes the amount of visited nodes. Considering that splitting is a costly operation, this could be a key-recursion. Similarly, it may be interesting to consider batch removes and batch queries.

The oracles presented thus far serve the main purpose of proving the proposal efficient. The next step is to consider other oracles that trade time versus space more efficiently. These may either be based on entirely new ideas, or build on top of those provided here. For instance, how is time affected by caching every other level in a depth-based oracle? It would also be interesting to consider persistent and/or consistent representation of the SMT. It appears to be an open problem whether any lexicographically sorted data structure supports incremental proofs [24].
References


A Memory Requirements for the BD-Hybrid Oracle

Theorem 5 (BD-Hybrid Oracle). Oracle 5 records at most $2n - 1$ non-default hashes.

Proof. The idea is to use a greedy algorithm that inserts $n$ values into the SMT. On each insertion, the aim is to have the maximum amount of non-default hashes added to the relative information. The algorithm is described below, and it is proven sufficient by considering what happens if one tries to deviate from the set of greedy solutions.

The first value is arbitrarily associated with a leaf $x$, and the depth oracle records $d$ non-default hashes. Referring to the level below the root, the next insertion can be performed either in the same subtree or the opposite subtree with respect to the first value. The locally best solution is to record $d - 1$ hashes in the empty subtree, as any insertion in the non-empty subtree produces at most $d - 2$ new hashes. During the next $2^1$ insertions, the same greedy approach can record at most $d - 2$ hashes per insertion, and is achieved by choosing leaves such that the paths deviate from the already recorded hashes after the first two levels. Similarly, the succeeding $2^2$ insertions can at most record another $d - 3$ new hashes per insertion, and so on. Eventually the point where $d - \log n = 1$ is reached, and the remaining $\frac{n}{2}$ insertions will cause either the depth or the branch oracle to record at most one hash. Hence, the final result is the same regardless of if the branch or the depth oracle does the recording. Choosing to let the depth oracle do the work implies that the branch oracle records nothing. Therefore one of the solutions provided by the greedy algorithm is equivalent to the upper bound of the depth oracle, and the analysis reduces to Theorem 4.

The final part of the proof is to prove the greedy algorithm sufficient. In other words, it must be shown that it is not possible to record more hashes when also utilising the branch oracle. In order to do just that, consider what happens if the branch oracle records a hash that would not be covered by the depth oracle. Clearly it must be in the sparse part of the SMT, meaning that the branch is formed below level $d$. Furthermore, it records at most a
single hash per insertion. Such manoeuvres may be performed multiple times, and comes at the cost of not utilising the depth oracle. The final result will either be a subset of the greedy solutions, or there will be at least one possible insertion left that would have caused the depth oracle to record multiple hashes. In the latter case, many hashes have been traded for a single hash and is thereby never worthwhile. As such, the greedy algorithm is optimal and the upper limit for the BD-hybrid oracle is $2^n - 1$. ■