Delft University of Technology

Master's Thesis

A Framework for the Implementation and Comparison of Authenticated Data Structures

Author:
Daniël Mast

Supervisors:
Zekeriya Erkin
Thijs Veugen

A thesis written in fulfilment of the requirements
for the degree of Computer Science

in the

Department of Intelligent Systems *of*
Delft University of Technology

March 2016

Abstract

Faculty Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Computer Science

A Framework for the Implementation and Comparison of Authenticated Data Structures

by Daniël Mast

We present the design and implementation of a general framework that enables implementation and performance comparison of Authenticated Data Structures (ADSs). The framework guarantees that an ADS supports initialization, updates, and verification of queries. In the framework, we implemented the hash tree, skip list, and state-of-the-art SeqHash, and extended their functionality to enable insertion and deletion of a list of entries. We alleviate the task of a programmer by reducing insertion and deletion from three core methods: create, merge, and split. We present the results of the performance comparison. Our implementation of the skip list proves to outperform SeqHash, while supporting the same operations, being more intuitive and easier to implement, and is therefore a good alternative for practical use.

Acknowledgements

First of all, I would like to thank my thesis supervisor assistant professor Zekeriya Erkin of the Cyber Security Group at the faculty of EEMCS from the Delft University of Technology. Thank you for the great support throughout the graduation process, your advice, and your valuable time.

Secondly, I want to thank the experts of the department of Cyber Security and Resilience at TNO in The Hague, where I, as a graduate intern, could learn about the practice of Cyber Security. In particular, I want to thank my internship supervisor Thijs Veugen, who was greatly involved in my research.

Finally, I want to express my gratitude to all my friends and family that provided me with support throughout my studies and graduation. I could not have done this without you.

Daniël Mast

Contents

Chapter 1

Introduction

With the emergence of cloud computing, users are increasingly collaborating on remotely stored data, without possessing a local copy of it [1]. A reader obtains data by sending a query to a server, and receives a response. A writer sends an update, for which the server updates the data accordingly. The reader however cannot always be sure that the received response is correct. System failure, in the form of data becoming corrupted in storage or transmission, may cause the data to be different than the writer's original version. This issue can be overcome by hashing the data, which deterministically transforms data of an arbitrary size to a fixed size. The hash values can be used as checksums. With this technique, storage or transmission errors are easily detected [3]. However, if the server owner may choose to deliberately send the wrong data to a reader, the reader will have no possibility of detecting this kind of misbehavior. How can users take advantage of the storage and computational capabilities of a cloud server, without having to trust the server blindly?

Previous research has introduced so-called authenticated data structures (ADSs) [11]. These are structures that enrich the original data with authentication data, which allows the server to prove that he sent the correct response, and allows the reader to verify this proof, being guaranteed that the response is correct. In my literature report [8], I introduce ADSs, and discuss the research where they stem from, including subjects like auditing, provable data possession and proofs of correctness. Practical examples in which ADSs can be applied are: the distribution of stock information, multi-user collaboration on documents in the cloud, cloud email, and all other services on which users communicate indirectly via a (potentially untrusted) cloud provider.

Existing research has come up with a wide variety of ADSs. In most cases, only a single ADS is presented and explained, its time complexity is stated, and performance results are given. Due to the differences in structure and supported operations, one ADS

cannot be easily compared to another. It would however be very convenient to be able to determine which ADS suits or performs best for which application. The research of this thesis aims to answer exactly that question, by presenting a framework that pours the different ADSs into a similar structure, which allows comparison of the performance of the ADSs with it.

In this research, three ADSs from literature are analyzed and compared: the Merkle hash tree [9], the authenticated skip list [2] [4] [5], and SeqHash [12]. The classical Merkle hash tree is a well known and most basic type of ADS, of which the structure is intuitive and straightforward. The authenticated skip list is a similar tree-like structure like the Merkle tree, but instead consists of layers of linked lists, enables more efficient update operations, and is used in applications concerning certificate revocation. SeqHash is a state-of-the-art ADS that is efficient for storing a history of computations and detecting server misbehavior. Its structure uses partial evaluation of the trees of which it consists, to enable efficient data merging. We selected these particular selection of ADSs, as they form a good representation of the existing ADSs, both in terms of complexity and modernity, and therefore serve well to prove the usefulness of the framework. Moreover, we wanted to examine the efficiency of the state-of-the-art SeqHash. To be able to perform a fair comparison, a single framework is created in which all ADSs are implemented. The framework ensures that all ADSs support the following update operations: inserting, deleting, and appending data blocks. Moreover, it guarantees that a proof of correctness can be constructed for a given data block, and that the method to verify this proof is defined. This set of update and verification operations forms the elementary basis of each ADS. The design of each of these operations does not always follow directly from the literature. Therefore, we often extended the existing structure, improving the original design and its efficiency, to support all core operations. The performance of the operations is benchmarked, and the results of the different ADSs are compared. We also compare the ADSs on how complex the operations are, and how many lines of code are needed to implement them. In these ways, our work contributes to the debate on the real-world applicability of ADSs.

Apart from the ability to elegantly compare ADSs, the framework has additional benefits. For the programmer, we provide a clear structure, in which only the *specific* functionality of an ADS has to be implemented, and in which *basic* ADS functionality is already included. This means that the framework is not only useful for comparing the above-mentioned ADSs, but can be used in future research as well to implement and test new ADSs. For the end-user, our framework provides a set of ADSs that he can use instantly, without being required to have any knowledge about the specifics of the ADS itself. Moreover, the framework overcomes the issue of dealing with various forms of input data. The data can consist of numbers, text, DNA sequences, a file hierarchy, or

anything else. When the user defines the blocks that the data consists of, each ADS in the framework will instantly support this data without restrictions, allowing the user to choose the ADS that he finds most suitable for the job.

In summary, our contributions are:

1. the construction of a framework for implementing and comparing ADSs;
2. the implementation and improvement of ADSs from state-of-the-art literature, and extension to fully support update and verification operations;
3. a performance comparison of these ADSs;
4. a discussion on the real-world applicability of ADSs

The rest of this thesis is organized as follows: Chapter 2 explains in detail how an ADS works. It describes the three different ADSs that we selected from literature, in which way we improved them, and how we modified their structure to fit in the framework. Chapter 3 discusses the design and implementation details of the framework and the three ADSs. Chapter 4 presents the experimental results. Chapter 5 concludes with the main outcomes of the research.

Chapter 2

Authenticated Data Structures

Authenticated data structures provide a way to guarantee integrity of outsourced data, in a more efficient way than hashing the data set as a whole [11]. Hashing a data set in this straightforward way would require the reader to download the entire data set, compute its hash, and compare it to the hash received from the writer. ADS provides a more efficient solution, by dividing the data set into blocks, which are hashed separately, and combined in a tree-like structure, from which the root is used as a small-sized authenticator value [6]. This structure allows the reader to verify the integrity of a single queried block, which is much more efficient than having to verify the whole data set every time. We describe the general ADS model in Section 2.1. In Section 2.2, we explain how a reader can verify a publisher's response. In Section 2.3, we discuss how our framework supports updates of the ADS. In Sections 2.4, 2.5, and 2.6 , we discuss the different implementations of this model in the form of the Merkle hash tree, the authenticated skip list, and SeqHash respectively. For each of these ADSs, we describe how they are created, verified, and updated. In Section 2.7, we explain in detail how an arbitrary data set can be mapped to suitable input data for an ADS. We conclude the chapter with Section 2.8, in which confidentiality of ADSs is discussed.

2.1 Model

ADS distinguishes three parties: the writer, the publisher and the readers [11]. The writer maintains the data, and wishes that the readers can access this data. In practice, the writer does not have the capability (or the means) to serve every reader. Therefore, this job is outsourced to the publisher, that does possess these capabilities. The capability issue is now resolved, but the publisher is untrusted. The writer and readers cannot be sure that he will always send the correct data. The publisher may choose

to deliberately respond incorrectly to queries. The readers will then receive data that does not match with what the writer sent to the publisher. ADS provides a scheme that enables detection of this kind of misbehavior. Figure 2.1 shows which data is shared between which parties. The writer maintains a local copy of the data, and performs updates on it. After every update it sends the new authenticator value to the readers. The update information is shared with the publisher, which updates his own copy of the data accordingly. A reader sends a query to the publisher, and receives a response and a proof. The reader uses the writer's authenticator and the proof to verify correctness of the response.

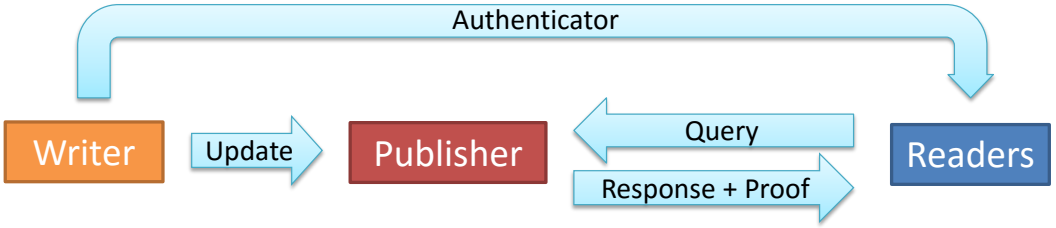

FIGURE 2.1: ADS scheme

2.2 Verification

The verification (Figure 2.2) proceeds as follows: First, the user computes the hash of the response contents. The received proof often consists of a set of hashes of the remaining data set. The response hash and the proof hashes can be combined to compute the authenticator. The details of the proof contents and the combine method differ per ADS. In general, hashing is the main technique used for verification. The main advantage of hashing is that it provides a way to verify data with digests that are constant-sized, independent of the data size. After computing the authenticator, the reader verifies whether it matches with the authenticator received from the writer. The writer's authenticator represents the accumulated hash of the whole data set. If the authenticators match, the reader is certain that the response is correct. If they do not match, the reader knows that the publisher has sent an incorrect response or an invalid proof.

2.3 Updating an ADS

In our framework design, an ADS only has to implement three methods in order to support all update operations: insert, delete, and append. The methods are: *create*,

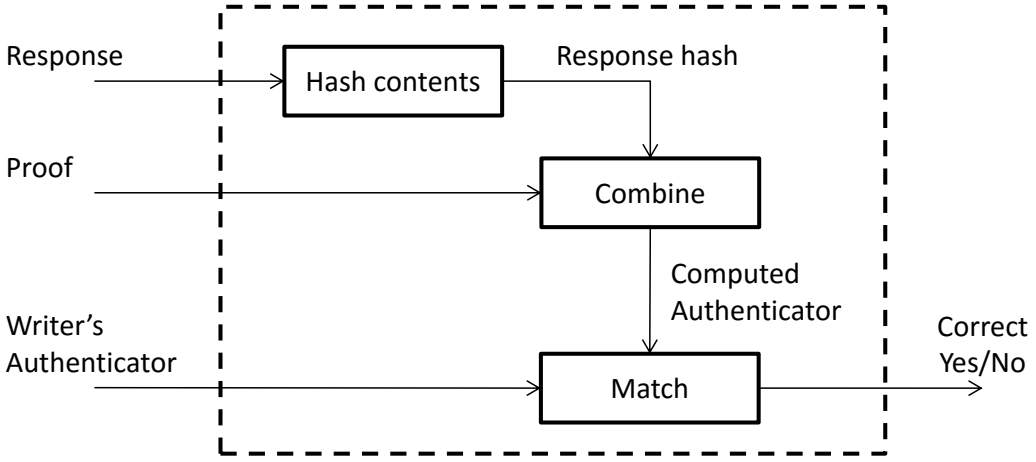

FIGURE 2.2: User's response verification method

merge, and *split*:

- *create([block]* $\rightarrow$ *ads.* Takes a list of blocks, and initializes an ADS for it.
- *merge(ads1, ads2)* $\rightarrow$ *ads.* Merges two ADSs into one.
- *split(ads, index)* $\rightarrow$ *ads1*, *ads2.* Splits one ADS into two at the given index location.

Merge and *split* are exact opposites: if we split an ADS a at any location, and merge the two resulting ADSs, then we will obtain a again.

The following algorithms show how these methods can be used for ADS updates. To be efficient in a practical setting, our design supports the insertion and deletion of a list of blocks at once. Previous research often only supports single block updates. If a user then wishes to insert a sequence of blocks, the ADS will update its hash values after every block, instead of at the end only. Notice that our design supports single blocks as well, as it is possible to provide a list containing a single block.

Insert takes a list of blocks, and inserts it in the given ADS at the given index location.

```
insert(ads, [block], index):
  left, right = split(ads, index)
  middle = create([block])
  result = merge(left, middle)
  result = merge(result, right)
  return result
```

Delete deletes blocks from the given ADS, between *index* and $(index + length)$.

```
delete(ads, index, length):
    left, temp = split(ads, index)
    unused, right = split(temp, length)
    result = merge(left, right)
    return result
```

Append is a special case of insertion, where it is not necessary to provide an index, as the list of blocks are added at the end. This costs less than an insertion, because the given ADS does not have to be splitted first.

```
append(ads, [block]):
  right = create([block])
  result = merge(ads, right)
  return result
```

Throughout this chapter, the implementations of *create*, *merge*, and *split*, will be discussed for the specific ADSs.

2.4 Merkle Hash Tree

The Merkle hash tree is one of the first ADSs in literature, and forms the basis of many other types of ADSs [11]. Variations on this structure exist to be more suitable for certain applications, but in this section, the general version is explained. In Figure 2.3, a Merkle hash tree is depicted. This section describes how such a hash tree is constructed from the input data, and how a queried block can be verified. It will also be explained why a Merkle hash tree is not suitable for updates.

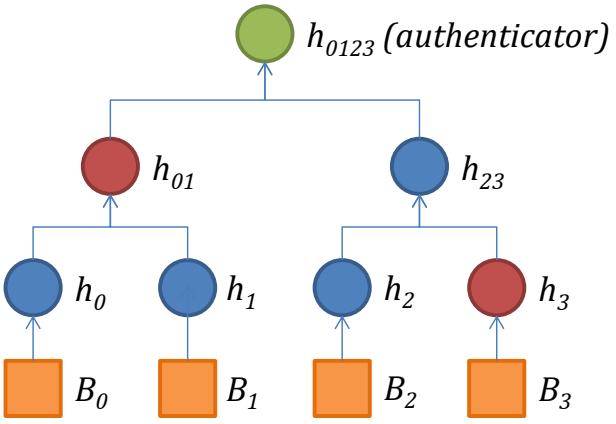

FIGURE 2.3: Merkle hash tree

2.4.1 Creation

We start with the set of input data blocks: $D = \{B_0, ..., B_n\}$. For each block (bottom row in Figure 2.3), the hash is computed, resulting into a new list: $\{h_0, ..., h_n\}$. Every element of this list is set as a leaf node of the hash tree. Next, every two hashes are combined into one hash on a higher level. Combining two hashes into one happens by hashing the concatenation of the two hashes, as shown in Equation 2.1. In this equation, $h()$ denotes the hash function, and the $||$ operator denotes concatenation of two values.

$$h_{01} = h(h_0||h_1) \tag{2.1}$$

This process builds up a binary tree of hash nodes with a single root, the top hash. This is the authenticator for the given set of data blocks. Here we can clearly see that altering one randomly chosen bit in a randomly chosen input data block will always result the authenticator value to change. This feature is very convenient for integrity checking. The tree that is constructed from the data does not contain a reference to the blocks itself anymore. This can clearly be seen in Figure 2.3. Instead, the tree is built *on top of* the data. Only the hashes of the blocks are present in the leaves of the ADS, not the block's contents themselves. This holds for every type of ADS, implemented in the framework. The list of blocks should therefore be stored separately, by the writer and publisher, in order to update the data, and send the contents to the readers.

2.4.2 Verification

Say that a reader sends a query to the publisher, asking for the contents of B_2. Then, the publisher should put these contents in the response. The contents have to be accompanied by a proof of correctness. The proof will consist of the sibling hashes, all the way from B_2 to the root. The proof also contains the relative position of those siblings (left or right), to be able to compute the correct hash. This is important, because a different order will result into a different hash, as regular hashing is not commutative. Equation 2.2 makes this clear:

$$h(a||b) \neq h(b||a) \tag{2.2}$$

In Figure 2.3, the so-called sibling hashes are h_3 and h_{01}. The reader now computes the hash of the received block: $h(B_2)$. If the publisher has sent the correct contents and sibling hashes, then $h(B_2)$ equals h_2. Next, the reader combines $h(B_2)$ with h_3, which

should be equal to h_{23}. That value is combined with h_{01}, resulting in the top hash. The reader should now check whether this final result equals the authenticator he received from the writer. If so, the reader is certain that the response content is correct.

2.4.3 Updating

An ADS should support the same basic update operations as performed on the input data. This means that an ADS should support insertion and deletion of blocks. A Merkle hash tree does not support these operations in an efficient way. We can show this with an example. Say that in Figure 2.3, we want to insert a block between B_0 and B_1. In this case, B_1, B_2, and B_3 will have to shift one place to the right. Due to the binary structure, an insertion of an odd number of blocks will require every block hash to be combined with the other neighbour. This requires recomputation of the hash values of all nodes in the tree structure above these blocks. In the worst case, when a block is inserted before B_0, the cost of this operation is equal to recreation of the entire tree, which is unacceptable. Deletion of an odd number of blocks will have the same issue.

In [13], a method for insertion an deletion of blocks in a Merkle hash tree is proposed, as shown in Figure 2.4. When a new block is inserted between B_2 and B_3, then the leaf node that used to represent B_2 now becomes a parent node of two new nodes, from which the left node now represents B_2, and the right node represents the new node. The parent hashes on the path to the root will have to be updated, but the rest of the tree stays the same. For deletion, the exact opposite happens, as shown in Figure 2.5. A node that used to be a parent of two leaf nodes, becomes a leaf node itself, when one of the leaf nodes (representing a block hash) is deleted. After deletion, the parent hashes are updated. This type of insertion and deletion is easy to implement, but introduces two major issues. First, the tree might become highly unbalanced, due to a sequence of insertions (or deletions) at a concentrated position. In an unbalanced tree, the distance to the root differs greatly between different leaves. This distance affects the time it takes to perform an update on that position, and construct and verify a proof for a block, located in that branch. Also, it affects the proof size, as this depends on the number of hash nodes the block is distanced from the root. This unpredictability of performance time and proof size for a randomly chosen position is undesirable. Second, the structure is not *history independent*, as it reveals information about the order of a sequence of update operations. This is undesirable. In a scenario where the data is static, and never requires updates, the Merkle hash tree serves perfectly as an ADS. However, in a dynamic setting, these two issues make a Merkle hash tree unsuitable for practical use, which created the need of dynamic ADSs.

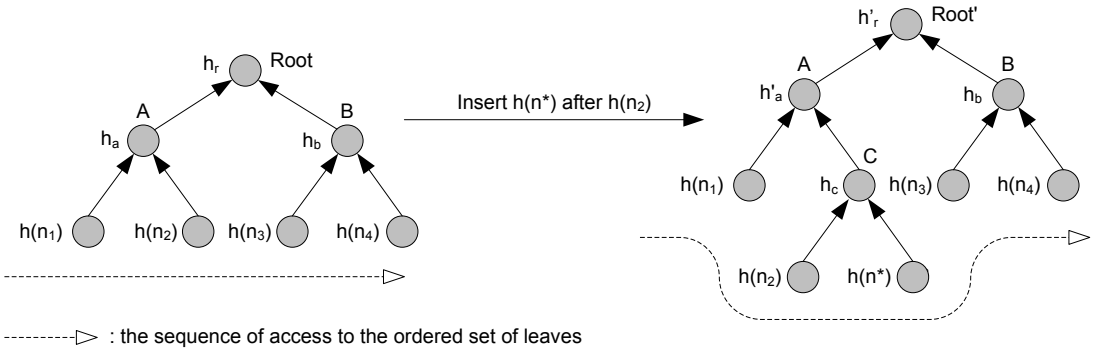

FIGURE 2.4: Inserting a block [13]

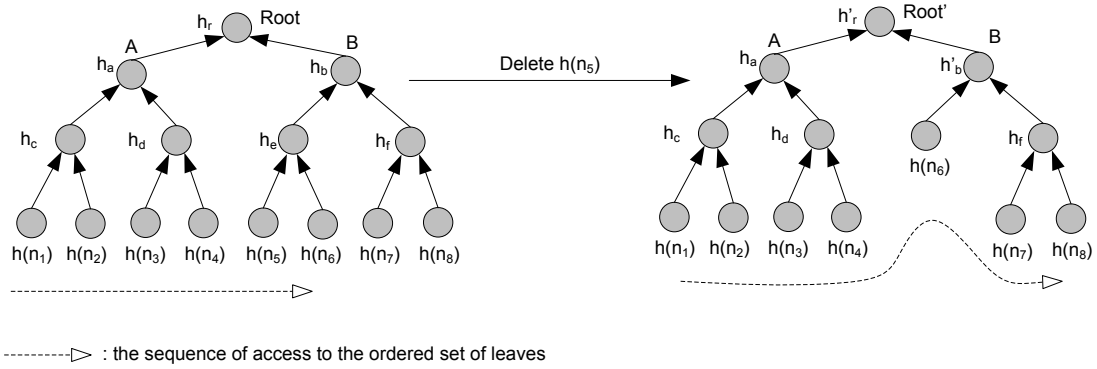

FIGURE 2.5: Deleting a block [13]

2.5 Authenticated Skip List

In [2], a new ADS is introduced, the so-called rank-based skip list. This work is based on previous research [4] [5] that uses skip lists for authentication of data, but is unsuitable for usage with indexed data blocks. The rank-based skip list provides a way to insert and delete a block at a given index more efficiently, without having to recompute as many hash nodes as in the Merkle hash tree. In this section, we will first explain what a skip list is. Then, we will describe the structure of our design of the authenticated skip list. We will point out the elements that are similar to the rank-based skip list, but also the way in which our structure improves it. We will explain how this ADS is created from the input data, how it can be used for verification, and how it can be updated.

2.5.1 Skip list

In contrast to a binary tree, the original skip list design uses probabilistic balancing. The goal of this is to provide an efficient lookup method for sorted entries, and eliminate the need of having to define relatively complex methods to deterministically keep the tree

balanced. Balancing is important for guaranteeing that tree lookups perform similarly well for different locations in the tree. The skip list design shows how a structure can remain balanced in a probabilistic way, making the implementation much less complex. The downside of a probabilistic approach is that the worst case performance is much worse than the deterministic variant, but the probability that this occurs is very low. Similar to a binary tree, a skip list provides a way to lookup values in logarithmic time, not having to visit every entry in a linear way. This is reached by so-called *skipping* of entries. Figure 2.6 depicts a skip list representation of a list of numbers.

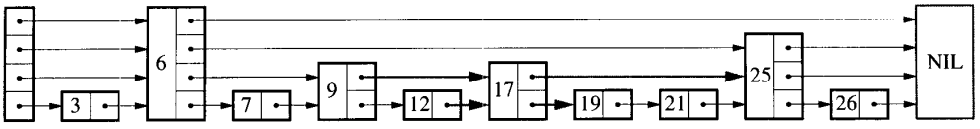

FIGURE 2.6: Skip list [10]

The figure shows that a skip list consists of a set of *towers*. Each tower contains the value of a single entry, together with a stack of leveled nodes. A node contains a reference to its right node on the same level (denoted by an arrow), and its down node (in the same tower). On the left, a dummy tower (without an entry) contains the starting nodes, and on the right, all last references refer to a dummy tower *NIL*, indicating the end. When a search for a specified entry is performed, the pointer starts at the top left node, and traverses to the right as long as the current entry is less than the entry being searched for. If a greater element is spotted, the pointer goes one back, and then one down, and repeats the process, until the entry is found, or the lowest layer is reached. In this way, many entries are skipped, making it more efficient than a linear search. The number of nodes stored in an object differs per entry, giving the towers different *heights*. This height is determined by a sequence of random *coin flips*, starting with a height of 1, and increasing by 1 until the coin lands on tails. Therefore, the probability of a tower getting a certain height is $p(h = x) = 0.5^x$. The expected value of the maximum height of a tower in the skip list equals the depth of a Merkle hash tree with an equal number of entries, providing a similar lookup time with high probability. The advantage of this probabilistic approach is that it simplifies the way in which the structure is kept balanced. The disadvantage is the high variance of performance: in the worst case (with low probability), the structure is highly unbalanced.

2.5.2 Structure

The original skip list keeps the list sorted. This means that adding new entries does not require the user to give a location of where to add the value, because the value itself determines its location, relative to other entries. Therefore, it does not support

an indexed insertion operation. To be able to apply the authenticated skip list on an indexed set of data blocks, the skip list should not sort the entries based on their value. Instead, the writer wants to define an index, deciding where the entry should be put, and knowing where to retrieve it later. The rank-based skip list enables indexed insertion and deletion, by storing additional information in each node. A node v stores the number of leaves that can be reached from that node, denoted as $r(v)$. This is the *rank* of that node. Figure 2.7 shows a rank-based skip list, with the rank of each node shown in its square. With this information, it is possible to traverse from the top left node, referred to as the *start node*, all the way to the leaf that corresponds to the given index, by deciding whether to turn right or down, based on the ranks of the right and bottom nodes.

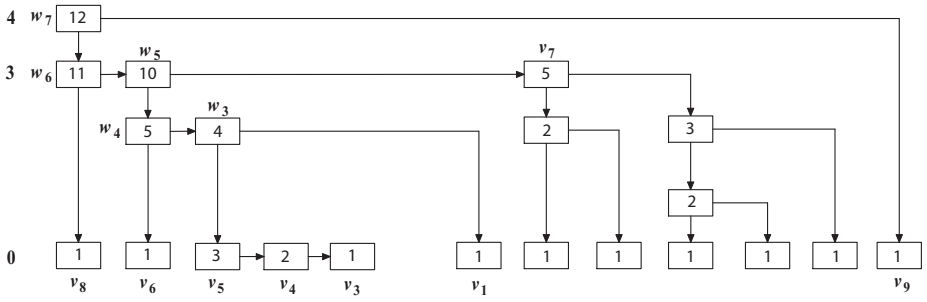

FIGURE 2.7: Rank-based skip list [2]

In our design, we support indexed insertion in a different way. Our design does not require rank information. Instead, each node not only stores a reference to its right and bottom neighbour, but also its top and left neighbour, enabling more flexible node traversal. Also, the ADS stores a list of references to all leaf nodes. When a new node is inserted or deleted, or a proof is constructed, we do not start the traversal at the start node (top down), but rather at the leaf node (bottom up). This takes away the necessity of searching the leaf first.

2.5.3 Creation

First, the start node is created, which at that point is the only node that is contained in the left dummy tower. Then, the tower of the first data block is attached to the dummy tower. The height is determined by a pseudo-random generator, for which the block's contents are used as a seed. This means that the same contents will always result into the same height. This feature is very important, as it guarantees that the writer's ADS will always remain identical to the publisher's copy. If this would not be the case, then verification would be impossible, as the authenticator values would not match. If

a newly created tower is higher than the dummy tower, new nodes are added to the dummy tower until the heights are equal. The start node is now set to the highest node in the left dummy tower. This guarantees that every node in the skip list is reachable from the start node. This process is repeated by adding each data block to the rightmost tower at that point. In the end, the right dummy tower is created, with a height equal to the left dummy tower, referring to all rightmost nodes, and vice versa.

2.5.4 Verification

An authenticated skip list enables verification, by computing a label for each node. A label $f(v)$ for node v represents an accumulating check value that is built up recursively from the nodes that this node refers to. $right(v)$ denotes the node at the right of node v. $down(v)$ denotes the node below node v. $entry(v)$ denotes the entry value of the tower of which node v is a base node.

- if v is a base node
  - if *right(v)* is a tower node: $f(v) = h(entry(v))$
  - if *right(v)* is a plateau node: $f(v) = h(entry(v), f(right(v))$
- else
  - if *right(v)* is a tower node: $f(v) = f(down(v))$
  - if *right(v)* is a plateau node: $f(v) = h(f(down(v)), f(right(v))$

A base node is a node that is located on the lowest level of the skip list. A plateau node is the top node of its tower. The other nodes of this object are tower nodes. Figure 2.8 shows the information flow that determines the contents of the labels for each node. This figure is a modified version of the skip list figure in [5]. The recursive computation of labels guarantees that the top left node of the authenticated skip list represents the authenticator for the entire set of entries, allowing a user to verify correctness of a query response efficiently. Similar as for the Merkle hash tree, the proof contains the labels of other nodes along the path that are necessary to recompute the authenticator.

2.5.5 Updating

To merge two skip lists, the right dummy tower of the left skip list, and the left dummy tower of the right skip list, should be removed. Instead, the nodes that they were connected to, should now be connected to each other. Also, if the two skip lists did not have the same height, the dummy tower's heights should be adjusted accordingly. When the nodes are now connected correctly, the node labels should be updated efficiently, without recomputing hashes for nodes that do not change. We know that the labels of

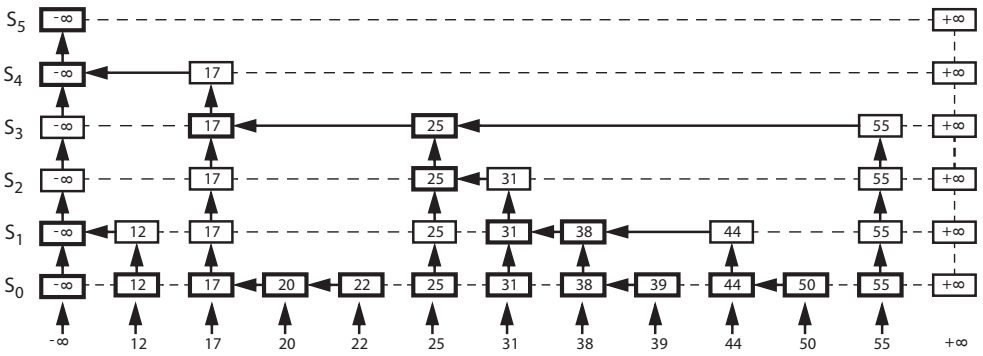

FIGURE 2.8: Authenticated skip list [5]

the nodes in the right skip list do not change, because the nodes that determine their labels have not changed. Only the nodes in the left part that (indirectly) have right neighbours in the right skip list that are plateau nodes, have to be updated. We do this by iterating over the righmost nodes of the left skip list, and checking whether their right neighbour is a plateau node. If so, update the label. If the node is a plateau node itself, its left neighbour should also be updated.

To split one skip list into two, an opposite procedure should be followed. At the given index location, where the skip list should be split, the left part should get a right dummy tower, and the right part should get a left dummy tower. The connections between the left and right part should be removed, and instead connected with the dummy towers. Due to the split, one of the resulting skip lists might have a lower height now. The dummy towers have to be adjusted accordingly. Similar to a merge, the labels in the left skip list have to be updated, as the left part is not connected to the right part anymore, but instead to the created right dummy tower.

2.6 SeqHash

SeqHash is a recent ADS that was first proposed in [12]. The ADS is implemented in a system called *VerSum*, that enables checking the behavior of multiple servers, performing the same outsourced computation for a user. In this system, SeqHash is used to store a history of computations. In a case where the servers return a different outcome, the user can analyze this history to detect where the server went wrong. This history is put in an ADS to make sure that the server cannot cheat by modifying the history. Because a history log is only extended with new computations at the end, SeqHash focused on the efficiency of appending blocks, and not on insertion or deletion. SeqHash reaches this appending efficiency by working with so-called partially evaluated trees. Due to this partial evaluation, it is more efficient to *merge* two SeqHashes, compared to two normal

hash trees. The original Seqhash design does not directly support a *split* method, and the algorithm to do this was not constructed yet. We extended the design with a split method, and the extra storage of values in nodes to make this possible.

2.6.1 Structure

A Merkle hash tree always uses the same method to combine the hashes of the blocks. Every two block hashes are combined, and a binary tree is constructed. In SeqHash however, the merging of two hashes depends on the hash. A sequence of rounds determine with which node a given node will merge. This feature allows SeqHash to be deterministic, and still allow efficient merging of partially evaluated trees.

2.6.2 Creation

Figure 2.9 shows the structure of a SeqHash. The leaf nodes $a - k$ denote the input data blocks. For each of these blocks, the hash is computed. An algorithm uses this hash as a seed for a pseudo-randomly generated sequence of output bits. The contents of a block are used as the seed for this pseudo-random generator, to make sure that the writer's ADS is identical to the publisher's copy. These bits are shown above the nodes. For every node, the first bit is computed. The bits determine whether two nodes are being merged or not. The merging means that a new parent node is created, and that its hash will be the combined hash of its two children. The parent node will be placed in the next level, one level higher. The criterium of two nodes being merged is that the bit of the left node should be 1, and the right node's bit should be 0. On all other combinations, the node are not merged. This ensures that conflicting situations do not occur in which one node should merge with both neighbours. However, it may happen that a node becomes enclosed between two nodes that have merged with their neighbours, making it impossible for the middle node to merge. When this happens, this node is added to the next level, giving it the opportunity to merge in the next round. Besides merging or being enclosed, a third scenario can occur for the rightmost and leftmost node. Namely, if the leftmost node gets an output bit 0, then this means that it can *potentially* get merged with the rightmost node of another SeqHash that is concatenated with this SeqHash in the future. The same holds for the rightmost node if its output bit is a 1. When this happens, these nodes are labeled *unknown*, depicted as a dashed square in Figure 2.9. These unknown nodes are then part of the *fringes* of the SeqHash. The left and right fringe contain the nodes that *might* be merged with nodes of a SeqHash with which this SeqHash is merged in a later stage. The usage of these fringes characterize the partial evaluation of the SeqHash structure.

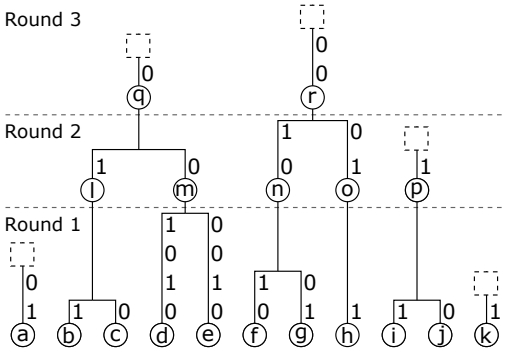

FIGURE 2.9: SeqHash [12]

The process of computing new output bits continues until all nodes have either been merged, enclosed, or labeled *unknown*. Then, a new round starts, attempting to merge the nodes of the next level. The process of rounds stops when at some point, all nodes are labeled *unknown*, causing them to be unable to merge or be enclosed. This scenario is the furthest in which the SeqHash can be evaluated. Due to the partial evaluation, as it does not result into a single tree with a single root. Instead, multiple adjacent trees are constructed, for which each root is an *unknown*-labeled node. If the user wants to fully evaluate the final SeqHash tree, a *finish* method is defined in which the *unknown* nodes are combined. Their fate now becomes known, because there are no other SeqHashes at the left or right of this SeqHash, that will be concatenated with it in a later stage.

2.6.3 Verification

Verification happens similarly to verification in a Merkle hash tree. The sibling hashes are added to the proof, along with their relative position, allowing the reader to compute the correct hash. However, as we have seen, a SeqHash does not consist of a single tree, but rather of a number of adjacent trees. Therefore, the authenticator is not a single hash value, but a list of hashes: the roots of all the trees. The authenticator therefore is not constant-sized, but still much smaller than the data itself. More importantly, the size of the proof is smaller than in a Merkle hash tree, because the partial evaluation produces trees with a lower height. For the same reason, verification is less costly as well, as the hash function has to be executed less. If the reader has computed the root for the tree that contained the hash of the to-be-verified block, he iterates over the list of roots in the authenticator he received from the writer. If one of these values match the computed hash, the reader knows that the response is correct.

2.6.4 Updating

Compared to a Merkle hash tree, SeqHash supports very efficient merging, due to the use of partial evaluation. When two SeqHashes are merged, only the nodes that occur in the right fringe of the left SeqHash, and the left fringe of the right SeqHash, have to be accessed. By design, we know that all other nodes in the structure will remain untouched, and will therefore not change. We do not have to traverse these nodes, which increases the efficiency. Similar to the creation of a single SeqHash, merging two SeqHashes proceeds in rounds, going from the bottom to the top. Every round, the fringe nodes of that level in the left and right SeqHash are added to the pool, and the output bits are computed to construct that part of the SeqHash. This will build up new trees, as nodes are merged, creating parent nodes on higher levels. When all right fringes of the left SeqHash, and left fringes of the right SeqHash are processed, the resulting structure is a SeqHash itself. This is the merged structure of the two given SeqHashes.

To split a SeqHash at a given index into two SeqHashes, the right fringes for the left SeqHash, and the left fringes for the right SeqHash, have to be constructed. We start at the leaf nodes directly left and directly right of the split location. From there, we traverse up and inside the resulting SeqHashes in order to decide which nodes are added to the fringes. To increase efficiency, every node stores a reference to its *previous* and *next* neighbour node. These are the nodes located left and right of the node respectively. Storing these references removes the need of traversing to the root or leaves of the tree and back. Also, each merged node stores the number of output bits that were computed until the node merged with another node. We use this value to determine whether a node can be added to the fringe: if a node, located at the left of the split, returns an output bit 1 faster than originally merged with a node on its left, then we know that this node should not merge with that left node, because it should be labeled *unknown*.

2.7 Construction of Input Data

To support applicability of an ADS on a data set, the writer is required to divide the set into separate blocks. The writer should cleverly decide how to distribute the data. The most important consideration here is that data that is likely to be altered at the same time, should be put in the same block. This will result in the least number of blocks having to be rehashed. For example, if the data consists of files that are edited one at a time, it may be efficient to define one file as one block. The advantage is that this does not mean that all blocks have to be files. The input data for the ADS will be a list

of blocks, from which a number of blocks may be files, but other blocks may represent numbers, DNA sequences or anything.

To give a concrete example, say we have a simple chat application. The user has a username and a profile picture. He has a list of contacts, in which each contact has its own username and profile picture. With each contact, the user can start a chat, which consists of a list of timestamped messages. In this case, one block (probably the first) will be a map, containing information about which information can be found where in the following blocks. The user's username (as a string block) will be stored in block 2, and his profile picture (as an image block) in block 3. Block 4 will serve as a map, indicating where the contacts can be found, and block 5 will be a map for the chats. In this way, the complete set of information of an app can be used for the app itself, and its integrity can at the same time be verified by an ADS.

2.8 Confidentiality

It is important to notice that the main goal of an ADS is to improve integrity, by detecting misbehavior of the publisher. Keeping the data confidential for the publisher is not the goal. In the described scheme, the publisher can read the data that the writer sends. It is however possible to partially achieve confidentiality of the data. The writer should then encrypt the blocks separately, and share the decryption key with the readers. The encrypted blocks are sent to the publisher. The reader decrypts the block after reception. In this way, the only information a publisher can obtain is the number of blocks, and, depending on the encryption scheme used, the varying sizes of block contents. Notice that the data set as a whole cannot be encrypted, as it will lose its block-distributed property, making it unsuitable to be used as ADS input.

Chapter 3

Design and Implementation

The framework, the access to input data, and the implementations of the ADS, are implemented in Java. Its object-oriented characteristic, supporting abstraction and inheritance, combined with the fact that it is widely used, makes it a very suitable language for building a framework which a programmer can use to implement new ADSs. Section 3.1 first discusses the framework, and of which classes and methods it exists. Next, Section 3.2 explains how to implement a new ADS and new input data in the framework. Section 3.3 describes how to use an implemented ADS and input data. Section 3.4 explains how a benchmark for a core method of an ADS is run. The complete implementation code of this work can be found at [7].

3.1 The Framework

Figure 3.1 shows the class diagram of the framework. The classes, contained in the *framework* package, form the core of our implementation, and are referred to when a new ADS is implemented. In the figure, the names of all classes in this package are italized, indicating that they are abstract. An abstract class has to be inherited before it can be instantiated. Italized method names are abstract methods. Abstract methods have to be implemented in a subclass. This abstract property is cleverly used, as it clearly shows the programmer which classes he should extend and which methods to implement in order to build a fully functional ADS. Also, the methods that are not abstract and already implemented, can be used directly by the programmer.

In the *framework* package, we see the classes *Block*, *Data*, *ADS*, *Proof*, and *Authenticator*. As a *Block* may have all kinds of forms (possibly binary), the programmer has to define how it is hashed. Therefore, it contains the method *toString()* and *hash()*. Implementing

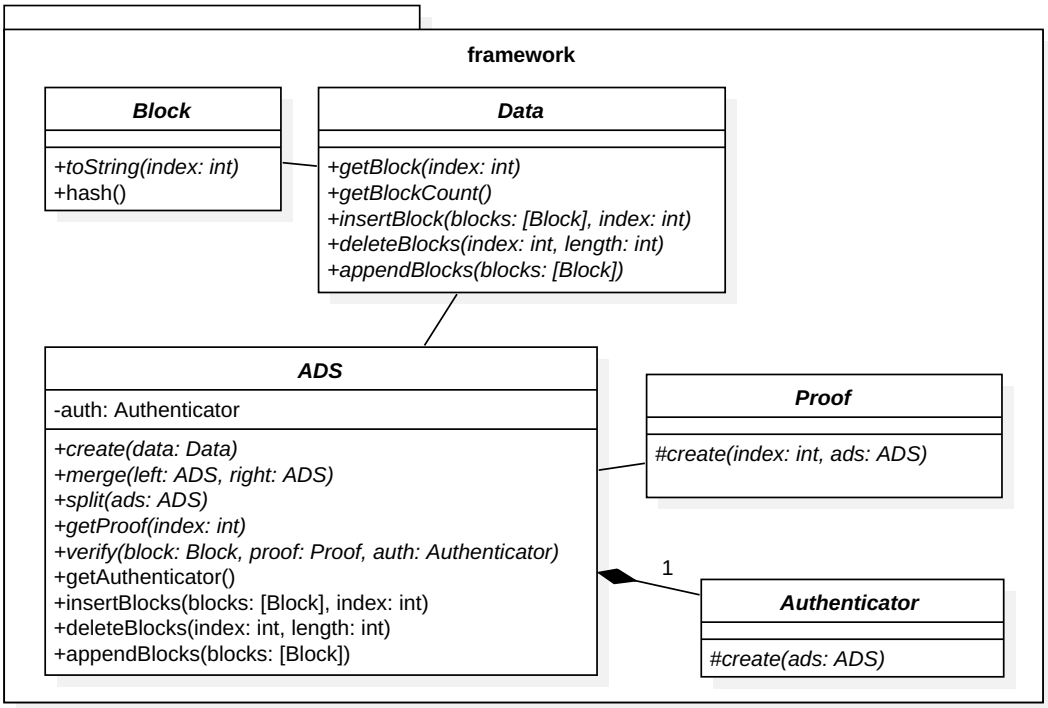

FIGURE 3.1: Class diagram of the ADS framework

toString() tells the program how the contents of the block are converted to a string. *hash()* directly uses this string to create a hash for the block. It is important that *toString()* creates a string that represents the complete content. Otherwise, the publisher could modify parts that will not affect the hash, and therefore the changes will not be detected. For efficient hashing, the string should be as short as possible. An example is depicted in Figure 3.2. On the left, a block representation of a simple file is shown. The file has a title, a creation date, and contents. *toString()* combines these details with the given index, creating a short string, that can be passed to the hash function.

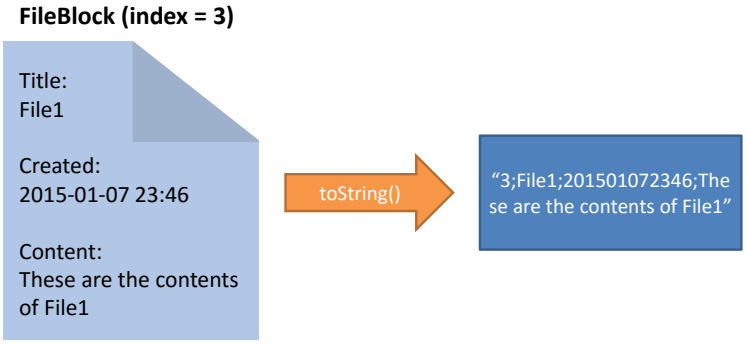

FIGURE 3.2: The *toString()* method

Data stores the references to the input data. In the simplest case, this means that a subclass of *Data* has a field that contains a list of *Block*s in memory. However, this is not

required. The subclass may merely contain a reference to a folder on the hard drive, in which files are stored. In that case, the *Data* subclass serves as an interface, defining how the raw data is translated to separately indexed *Block*s, that are not stored in memory, but are instead created (from the raw data) when queried. To make this interface work, the programmer should implement *getBlock()*, as well as the update methods.

When the *Data* is ready, it can be used to create the *ADS*. The *ADS* requires a reference to the *Data*, and passes it to *create()*. For ADS construction, the hashes of all blocks in the *Data* are computed. After that, the reference to *Data* is not needed anymore. *create()* initializes the fields of *ADS*, including the *Authenticator* field. The in-memory fields contain all properties, defining the ADS. For the *ADS*, the programmer should implement *merge()* and *split()*. These methods are called by the update methods *insertBlocks()*, *deleteBlocks()*, and *appendBlocks()*, so they can be instantly used. When implementing *merge()* and *split()*, the programmer is responsible for updating the *Authenticator* field as well, as the authenticator changes when ADSs merge or split.

To support verification, *getProof()* has to be implemented. *getAuthenticator()* returns the *Authenticator* object, stored in the *ADS*. A subclass of *Authenticator* stores the authenticator information in its fields, and should define methods to access these fields. *getProof()* creates and returns a *Proof* object. The *Proof* constructor uses the given index and the *ADS* to construct the proof, from which the information is stored in its fields. In *ADS*, *verify()* takes a given *Block*, a *Proof*, and an *Authenticator* object, and defines how these are used to verify the block contents with the proof, returning *true* or *false* accordingly.

3.2 Implementation of an ADS with Input Data

Figure 3.3 shows an example of how the previously described framework is used. It shows both how we can define input data as well as an ADS. Keep in mind that the type of data is not related to the type of ADS. The framework focuses on modularity of the two, by distinguishing them programmatically. *IntBlock* and *IntData* are subclasses of *Block* and *Data*, and therefore have to implement the abstract methods of their superclasses. Additionally, an *IntBlock* contains a integer *value*. *IntData* contains a list of *IntBlock*s. In *IntData*, *getBlock()*, and the update operations are defined by simply performing the corresponding update operation on that list. Now we have a fully functioning data instance that consists of a list of integer blocks.

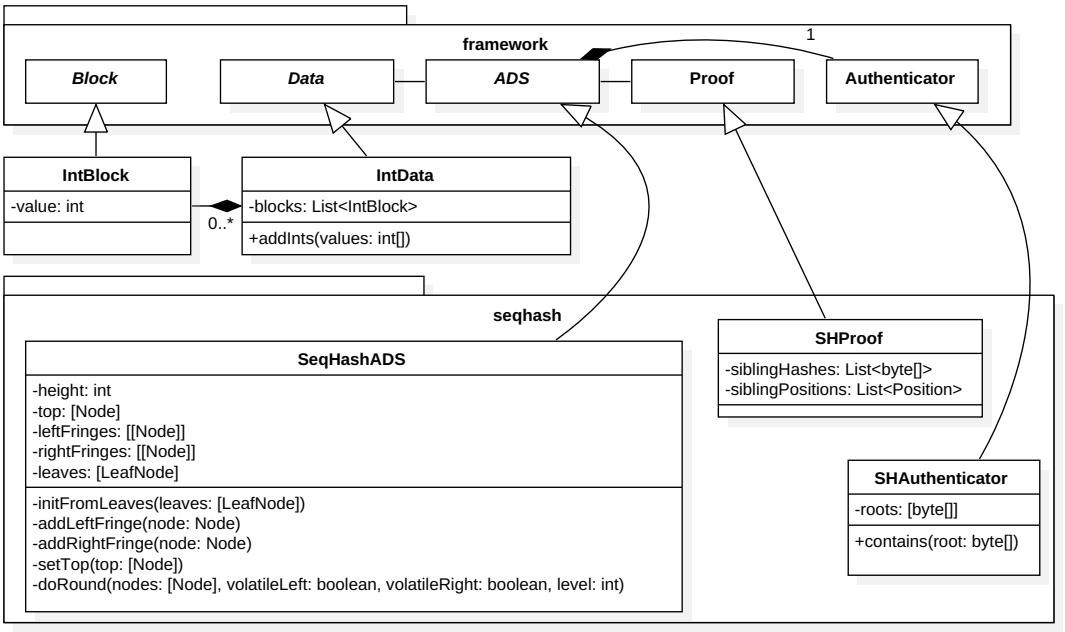

FIGURE 3.3: Class diagram of the SeqHash ADS and *IntData*

Next, we want to implement the SeqHash ADS in the framework. The classes *SeqHashADS*, *SHResponse*, and *SHAuthenticator* represent the SeqHash together. *SeqHashADS* contains a number of fields that represent its contents and enable efficient traversal of the nodes: its *height*, a list of *top* nodes, the left and right *fringes*, which are both a list of node lists, and lastly, a list of *leaf nodes*. *SeqHashADS* also consists of a number of (private) methods to build and update the contents. These are helper methods that are called in *create()*, *merge()* and *split()*. After initialization of *SeqHashADS*, a new instance of *SHAuthenticator* is created. *SHAuthenticator* contains a field *roots*, which is a list of byte arrays. *roots* stores the hash values of the roots of all trees in the SeqHash. The method *contains()* takes a byte array and returns whether this is contained in *roots*, which we need to know for verification. The authenticator object is updated alongside with updates that are performed on the ADS. *getProof()* calls the constructor of *SHProof*, which sets up the sibling hashes and corresponding positions for a given index.

This example has shown how the framework can be used to implement input data and ADSs. In a similar way, other types of data and ADSs can be implemented, as we did for the Merkle hash tree and authenticated skip list.

3.3 Usage of an ADS with Input Data

The previous section explained how *SeqHashADS* and *IntData* were implemented. We will now show these are used. A writer first initializes the data. He adds the integer that the data consists of, and passes the data to the *create()* function of the ADS:

```
data = new IntData()
data.addInts([3, 8, 2, 4, 7])
ads = SplitHashADS.create(data)

sendToPublisher(CREATE, data)

authenticator = ads.getAuthenticator()
sendToAllUsers(AUTH, authenticator)
```

Now both the data and the ADS are initialized. To make sure that the publisher possesses the same ADS, the writer shares the data with the publisher, and sends the authenticator to all users. Now they both have an identical copy of the data and the ADS. In a later stage, the writer wants to perform a number of updates:

```
# Inserting 3 blocks at position 3
index = 3
blocks = [1, 5, 7]
data.insertBlocks(blocks, index)
ads.insertBlocks(blocks, index)

sendToPublisher(INSERT, index, blocks)

# Deleting 3 blocks at position 4
index = 4
length = 3
data.deleteBlocks(index, length)
ads.deleteBlocks(index, length)

sendToPublisher(DELETE, index, length)

# Appending 2 blocks
blocks = [9, 8]
data.appendBlocks(blocks)
ads.appendBlocks(blocks)

sendToPublisher(APPEND, blocks)

authenticator = ads.getAuthenticator()
sendToAllUsers(AUTH, authenticator)
```

As the code shows, each update should be updated on both the original data and the ADS, to make sure they remain consistent. Also, the update information should be shared with the publisher, which should then performs the same updates. Also, the new authenticator should be shared with all users. The code shows the ease of updating the data and the corresponding ADS.

At a certain moment, the reader wants to query a block, and verify it. The reader will send the index of the block he wants to the publisher:

```
index = 3
sendToPublisher(QUERY, index)
```

The publisher will create the response contents:

```
# index = provided by user
block = data.getBlock(index)
proof = ads.getProof(index)

sendToReader(RESPONSE, block, proof)
```

On reception of the block and proof, the reader verifies the contents as follows:

```
# authenticator = provided by writer
result = SeqHashADS.verify(block, proof, authenticator)
```

As shown before, the reader received the writer's authenticator in an earlier stage. Now that he has received the block and proof from the publisher, he can verify the result. Note that although he needs to know the details of the verification method of a specific ADS, he does not need to possess the ADS contents itself. If the result returns true, the reader knows that the contents are correct. If the result is false, the publisher has cheated.

3.4 Benchmarks

To enable performance comparison of the implemented ADSs, we extended the framework with a benchmarking functionality that only requires an ADS as input, and automatically benchmarks the core methods: *create()*, *merge()*, *split()*, *getProof()*, and *verify()*. It is also possible to benchmark other methods that each ADS supports: insertBlocks(), deleteBlocks(), appendBlocks(). However, as the performance of these

update methods is fully dependent of the performance of *merge()* and *split()*, we decided to benchmark those instead. To start a benchmark, the ADSs that need to be benchmarked are first initialized and stored in an array:

```
ADSs = [new HashTreeADS(), new SkipListADS(), new SeqHashADS()]
```

Then, a benchmark is initialized, and starts to run after the array is given:

```
benchmark = new MergeBenchmark()
benchmark.runBatch(ADSs)
```

Inside the *MergeBenchmark* class, details can be set on the size of the data set, and the number of repetitions. Also, the class defines how the data set on which the benchmark is performed is initialized, together with other information that is needed to perform the operation. *runBatch()* repeats the benchmark for a number of different sizes. As soon as a benchmark for one size is finished, the result are printed and saved to a file. The result is computed by dividing the total passed time by the number of repetitions. This example showed *MergeBenchmark*, but the details are the same for the other benchmarks. Notice that when a programmer has implemented a new ADS, he only has to add a reference to the input array, and the benchmarks will automatically output its performance.

Chapter 4

Experimental Results

Apart from studying the three ADSs theoretically, we performed benchmarks on its core elements. The benchmarks measure the execution time of a method. For each ADS, we performed benchmarks on the methods that are specific for this ADS, and need to be implemented by the programmer: *create()*, *merge()*, *split()*, *getProof()*, and *verify()*. In the next sections, we will discuss the results of each of these methods for each ADS. Each figure shows the number of data entries on the x-axis, representing the *size* of the data. The execution time in milliseconds of the performed method is shown on the y-axis. The chapter ends with a code comparison of skip list and SeqHash.

4.1 Creation

Figure 4.1 shows the performance of creation of a new ADS, meaning that a *Data* object is given to the constructor, from which the list of blocks is retrieved, and an ADS is created. For creation, the hash tree performs the best, for every size. This is what we expected, as the hash tree is the most basic ADS, that does not require computation of additional values for future updates, in contrast to the skip list and SeqHash. The skip list turns out to perform almost equally well compared to the hash tree, despite the extra cost of constructing the dummy towers, and the coin tossing for each level of a tower. We see that SeqHash's creation compared to skip list and hash tree has a ratio of approximately 1.8:1. This is explained by the fact that many output bits have to be computed for each node, in each merge round.

The figure shows lines that are almost perfectly linear. However, the complexity for the creation of each ADS is $O(nlog(n))$, of which n equals the number of data entries. This complexity would not give a straight line, but rather a line of which the slope

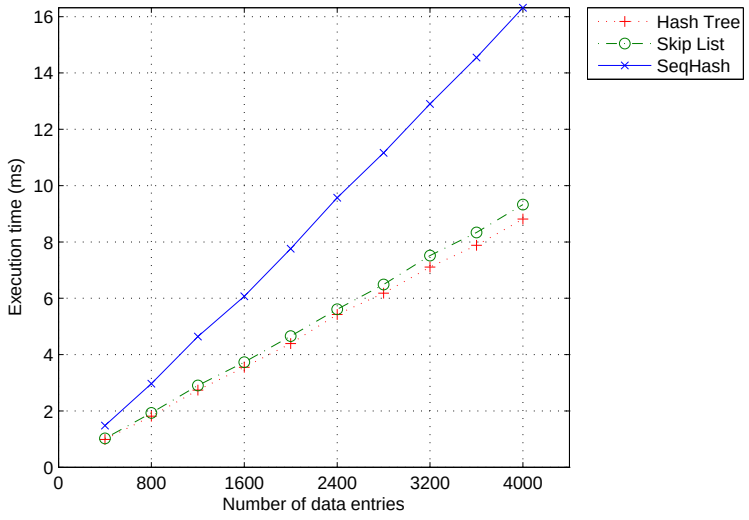

FIGURE 4.1: Creation of a new ADS

slowly increases for an increasing number of data entries. However, in the sizes that are measured, this effect cannot yet be clearly seen, as $log(n)$ is very small. The effect will be visible for a very large number of data entries.

The figure supports the fact, discussed in Chapter 2, that the hash tree is very efficient for static ADSs. If the user never requires updates on its data, he should choose to use a hash tree, rather than a more complex skip list or SeqHash.

4.2 Updating

Figure 4.2 show the performance of *merge()* for the skip list and SeqHash. As discussed, *merge()* and *split()* are not applicable to the hash tree, as its cost is equal to recreating the tree, so the hash tree is not shown in the figures. We see that again, skip list performs better than SeqHash, with a ratio of approximately 2:1. This is explained by the fact that when a SeqHash is merged, the fringe nodes are given to the *round* method, that creates new parents for these nodes. After that, these parents are again used as input for the *round* method, creating a new tree structure. For a skip list, this does not happen. Instead, only the right dummy tower of the left skip list, and the left dummy tower of the right skip list disappear, and the nodes to which they connected, now have to be connected to each other, and the corresponding labels have to be recomputed. The creation of new nodes in the SeqHash explain the difference in merge performance of these two ADSs. The fact that the lines are not straight, but rather a bit tortuous, is caused by the randomness of the input. It may happen that the data with size 2800

require computation of more output bits than the data with size 3200. This cannot be predicted. Despite this fact, the lines clearly show a quasi-linear increasing trend.

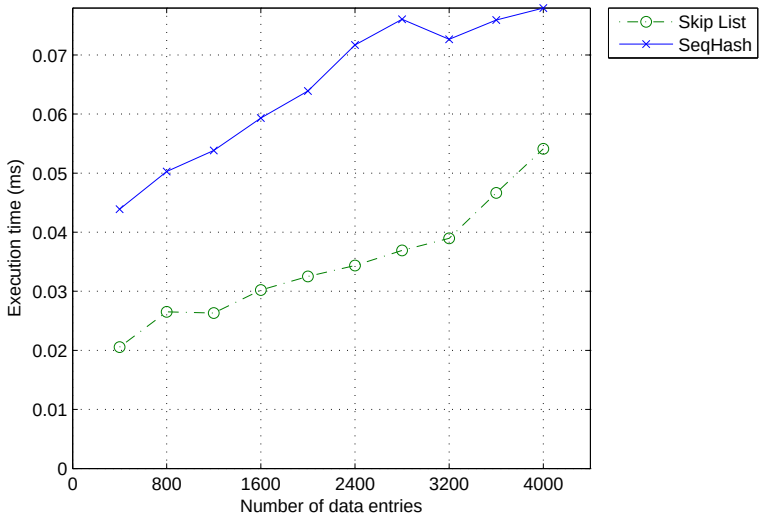

FIGURE 4.2: Merging of two ADSs

For splitting (Figure 4.3), it turns out that skip list is slower than SeqHash, with a ratio of approximately 1.4:1. This is explained by the fact that when a skip list splits, the existing links have to be broken, and those links have to be connected with dummy towers, that need to be created. For SeqHash, splitting only requires traversing the nodes along the split location, and deciding whether to add them to the fringe or not. No new nodes have to be created.

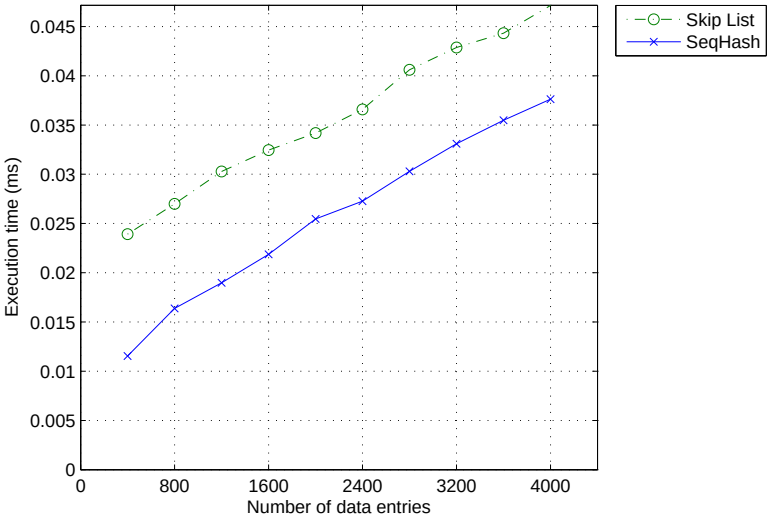

FIGURE 4.3: Splitting of an ADS

We have seen that skip list is more efficient for merging, and SeqHash is more efficient for splitting. It is therefore up to the programmer to analyze the application for which

he will apply the ADS. If the application performs many more merges than splits, he should choose the skip list. If the data has to be split very often, he should choose a SeqHash. If the numbers of merges and splits are about equal, the skip list is a better choice, as the skip list/SeqHash-ratio of merge is higher than split.

4.3 Verification

SeqHash performs better than the hash tree for proof construction (Figure 4.4). This is explained by the fact that SeqHash consists of a number of trees, of which the maximum height is lower than the height of a single hash tree. A lower height incurs that less siblings have to be passed to the proof. Skip list performs worse than the hash tree. A skip list has to decide for each node whether to add the label of its down or right neighbour, whereas a hash tree and SeqHash can directly add their sibling. This causes the performance difference. However, notice that the absolute differences in execution time is very small, with a magnitude of microseconds, so it will require an extremely large data set before these differences will start to become relevant.

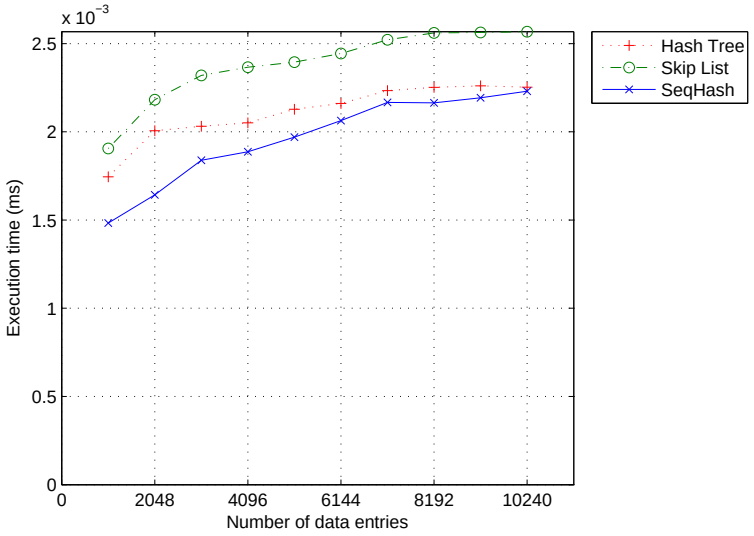

FIGURE 4.4: Construction of an entry proof

The results of verification (Figure 2.2) show the same order of execution time as for proof construction. The difference between SplitHash and hash tree is expected, because a SplitHash proof contains less hashes. Therefore, verification will also have to compute less hashes. However, skip list not only performs worse in proof construction, but also in verification. Verification of a skip list and hash tree are exactly the same, so from this we can conclude that on average, a skip list proof is longer (contains more hashes). This means that the random construction of a skip list does not guarantee that each

leaf node will on average have the same distance to the root. Although, the differences are again very small, so will only be relevant for very large data inputs.

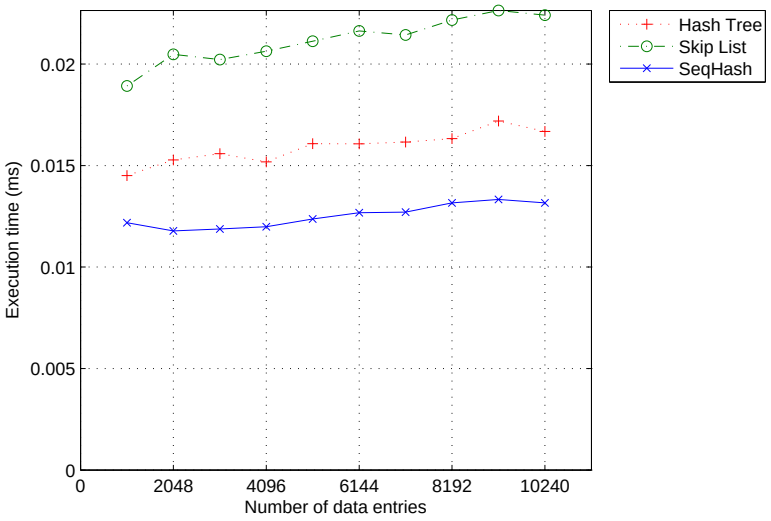

FIGURE 4.5: Integrity verification of entry

4.4 Code comparison

When we look at the two ADSs that support updating, we have seen that on average, skip list performs better than SeqHash. This is an interesting outcome, as in the current literature, SeqHash is a state of the art ADS, that is supposed to merge two ADSs efficiently. However, we have seen that our implementation of the skip list, based on the rank-based skip list, merges more efficiently. Therefore, it is interesting to look at the complexity of the written code. Skip list not only performs better, but also requires less code lines. SeqHash is written in 1009 lines of code, and skip list in 647 lines. Skip list's structure is also more intuitive, and therefore easier to explain. Because it does essentially the same as a SeqHash, we can say that SeqHash's structure is unnecessarily complex. Its goals can be achieved in a simpler and more efficient way, as we have shown.

Chapter 5

Discussion & Conclusion

In this work, we have given a detailed view on Authenticated Data Structures. We have shown how the range of existing ADS designs can be generalized to a model, and how we have implemented this model into a framework. We have explained how this framework can be used to implement new ADSs, without having to re-implement ADS commonalities every time, and how this framework supports benchmarking of an ADS, and performance comparison of different ADSs. We have proposed an intuitive way to generalize update operations, by using a create, merge, and split method as building blocks. To prove the usability of the framework, we implemented and compared the hash tree, skip list, and SeqHash. Without the framework, it would have been much harder to research the differences and similarities of the different ADSs, both qualitatively and quantitatively. Apart from the implementation of specific ADSs, we have also shown that the framework enables an intuitive way of constructing the input data, of every existing type, and explained how this is passed as input to an ADS. Moreover, we have discussed how a user can apply an implemented ADS and input. The most important outcome of the ADS comparison is that our modification of the skip list outperforms the current state-of-the-art SeqHash, and is also easier to implement, while supporting the same functionality.

5.1 Skip list and SeqHash comparison

Our initial goal was to extend the SeqHash design with a split function, as the original SeqHash only supported merging. Our reason for this was to support efficient insertion and deletion of a list of entries. We succeeded in implementing this function, however, the surprising outcome of the comparison with the skip list that we implemented was that the skip list is able to support the same operations as the SeqHash, but performs

better, is more intuitive, easier to implement, and requires less code. With this, we have shown that the structure of SeqHash, consisting of a number of trees, requiring rounds for merging layers of nodes, is unnecessarily complex, and is slower for the same reason. In the background, skip list and SeqHash are essentially the same, but skip list requires less operations to achieve this. Also, SeqHash's authenticator is not constant-sized. Although the size will remain relatively small, it is unpleasant that this value will grow, as this value has to be shared with all users after a writer has performed an update. One of the goals of Authenticated Data Structures is to redeem the writer from executing costly tasks, so it would be desirable the the authenticator is constant-sized. This is the case for skip list. These multiple reasons make the skip list a good alternative for SeqHash.

5.2 General review on use of ADSs

As we have seen, an ADS provides a useful procedure to verify the integrity of queried data. However, there are some disadvantages. First, the verification procedure can only verify the integrity of the block that the reader queried for. In other words, if a publisher maliciously modifies the contents of a block, that is queried much later in time, then there is no way to discover when the publisher applied this modification. For the same reason, there is no possibility to *revert* the data set to a correct version. More generally, an ADS does not provide an error-correcting feature. It only supports error detection. Also, we have seen that the way in which the data has to be distributed into blocks, in order to be able to be inserted into an ADS, reduces the possibility of encrypting the data. It is only possible to encrypt the single blocks. This is less secure compared to encrypting the data set as a whole. Also, even when every single block is encrypted, the structure still reveals information. First, the publisher knows of how many blocks the data set consists. Second, the reader can estimate the size of the entire data set, by looking at the number of hashes that a proof consists of, as this number is logarithmic to the size of the data. Also, the positional information in the proof tells the reader exactly where the queried block is located in the data set.

5.3 Future work

In the current framework, a possibility for the publisher to cheat, would be to send a block to the reader that the reader did not query. If the publisher sends the corresponding proof, then the verification method will return true. If the reader cannot tell by the contents of the block that he received the wrong block, then he cannot know that the

publisher cheated. In [13], a method is proposed that disables this way of cheating by the publisher, by introducing an additional verification of the block hashes. Future work could extend our framework to add this feature to the verification procedure, disabling the publisher from the ability to cheat by sending another block.

Future work of ADSs in general should come up with ways to support error correction after detection, and design structures that are revealing less information about the data contents to the publisher and the reader.

Bibliography

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, 2010. doi: 10.1145/1721654.1721672. URL http://doi.acm.org/10.1145/1721654.1721672.

[2] C. C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia. Dynamic provable data possession. *ACM Trans. Inf. Syst. Secur.*, 17(4):15, 2015. doi: 10.1145/2699909. URL http://doi.acm.org/10.1145/2699909.

[3] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. B. Ferreira, and R. Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12, Salt Lake City, UT, USA - November 11 - 15, 2012*, page 78, 2012. doi: 10.1109/SC.2012.49. URL http://dx.doi.org/10.1109/SC.2012.49.

[4] M. T. Goodrich and R. Tamassia. Efficient authenticated dictionaries with skip lists and commutative hashing. *US Patent App*, 10(416,015), 2000.

[5] M. T. Goodrich, R. Tamassia, and A. Schwerin. Implementation of an authenticated dictionary with skip lists and commutative hashing. In *DARPA Information Survivability Conference & Exposition II, 2001. DISCEX'01. Proceedings*, volume 2, pages 68–82. IEEE, 2001.

[6] M. T. Goodrich, R. Tamassia, and N. Triandopoulos. Efficient authenticated data structures for graph connectivity and geometric search problems. *Algorithmica*, 60 (3):505–552, 2011. doi: 10.1007/s00453-009-9355-7. URL http://dx.doi.org/10.1007/s00453-009-9355-7.

[7] D. Mast. ADS framework code. URL https://github.com/danielmast/ads.

[8] D. Mast. Cryptographic solutions for security and privacy issues in the cloud. Literature Survey, 2015. URL http://cys.ewi.tudelft.nl/content/cryptographic-solutions-security-and-privacy-issues-cloud.

[9] R. C. Merkle. Protocols for public key cryptosystems. In *Proceedings of the 1980 IEEE Symposium on Security and Privacy, Oakland, California, USA, April 14-16, 1980*, pages 122–134, 1980. doi: 10.1109/SP.1980.10006. URL http://dx.doi.org/10.1109/SP.1980.10006.

[10] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990. doi: 10.1145/78973.78977. URL http://doi.acm.org/10.1145/78973.78977.

[11] R. Tamassia. Authenticated data structures. In *Algorithms - ESA 2003, 11th Annual European Symposium, Budapest, Hungary, September 16-19, 2003, Proceedings*, pages 2–5, 2003. doi: 10.1007/978-3-540-39658-1_2. URL http://dx.doi.org/10.1007/978-3-540-39658-1_2.

[12] J. van den Hooff, M. F. Kaashoek, and N. Zeldovich. Versum: Verifiable computations over large public logs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 1304–1316, 2014. doi: 10.1145/2660267.2660327. URL http://doi.acm.org/10.1145/2660267.2660327.

[13] Q. Wang, C. Wang, J. Li, K. Ren, and W. Lou. Enabling public verifiability and data dynamics for storage security in cloud computing. In *Computer Security - ESORICS 2009, 14th European Symposium on Research in Computer Security, Saint-Malo, France, September 21-23, 2009. Proceedings*, pages 355–370, 2009. doi: 10.1007/978-3-642-04444-1_22. URL http://dx.doi.org/10.1007/978-3-642-04444-1_22.