A practical space– and time–efficient algorithm for Merkle tree traversal

Master-Thesis

Author: Markus Knecht
Supervisor: Prof. Dr. Carlo U. Nicola
Advisor: Prof. Dr. Jürg Luthiger

Institute of Mobile and Distributed Systems
University of Applied Science and Arts Northwestern Switzerland
School of Engineering

February 5, 2014

Abstract

We present an algorithm for the Merkle tree traversal problem which combines the efficient space-time trade-off from the fractal Merkle–tree [4] and the space efficiency from the improved log space-time Merkle–trees traversal [9]. We give an exhaustive analysis of the space and time efficiency of our algorithm in function of the parameters H (the height of the Merkle tree) and h ($h = \frac{H}{L}$ where L is the number of levels in the Merkle tree). We also analyse the space impact when a continuous deterministic pseudo–random number generator (PRNG) is used to generate the leaves. We further program a low storage–space and a low time–overhead version of the algorithm in Java and measure its performance with respect to the two different implementations cited above. Our implementation uses the least space when a continuous PRNG is used for the leaf calculation.

Acknowledgements

I would like to express my gratitude to my master thesis supervisor Prof. Dr. Carlo U. Nicola. He has spend much time in instructing me how to write a paper and he read this thesis and gave me useful hints on how to improve it. I would like to thank Prof. Dr. Willi Meier for his support concerning cryptographic questions, which arise during the the time I wrote this thesis. In addition I would like to thank both of them for their support in writing the paper and their indispensable inputs concerning the structure, content and formulations in the paper.
I am also grateful to my friends and family for listening to my never ending talks about post–quantum cryptography and their questions about the subject leading to deeper thoughts about the subject of this thesis.

Statement of Authencity

I confirm that this master thesis research was performed autonomously by myself using only the sources, aids and assistance stated in the report, and that any work adopted from other sources than the paper [1], which was written as part of this thesis, is duly cited and referenced as such.

Windisch, February 5, 2014
Markus Knecht

Contents

List of Figures

List of Tables

List of Algorithms

Glossary

Log algorithm	Merkle tree traversal algorithm from from [9]
Fractal algorithm	Merkle tree traversal algorithm from from [4]
Authentication path	The path that authenticates a leaf in a tree (siblings of the nodes on the path from the leaf to the root)
TreeHash	Algorithm for calculating nodes in a binary hash tree
Improved *TreeHash*	*TreeHash* from [9] (exclusive the scheduling algorithm)
Classical *TreeHash*	*TreeHash* from [7] (exclusive the scheduling algorithm)
TreeHash update	Two step in the classic– or one step in the improved– *TreeHash*
Tail height	Height of the node with the smallest height in a *TreeHash* (see Sec. 4.2)
Exist tree	Part of a subtree that stores the nodes for the upcoming authentication paths
Desired tree	Part of a subtree that stores the nodes for the upcoming *Exist* tree
Bottom level	Lowest level for which a *Desired* tree stores nodes
Bottom level nodes	Nodes in a *Desired* tree with a height equal to bottom level
Non–bottom level nodes	Nodes in a *Desired* tree without the bottom level nodes
Right/Left node	Node which is the right/left child of its parent
Inner node	A node in the Merkle tree which has child nodes (not a leaf node)
Related nodes	Two nodes one in the *Exist* tree and one in the *Desired* tree with the same relative position to the root of their *Exist* or *Desired* tree
$TreeHash_l$	*TreeHash* which calculates nodes up to the level l (it terminates when a node on level l is calculated)
Lower *TreeHash*	*TreeHash* which calculates nodes up to the bottom level ($TreeHash_{Bottomlevel}$) of a *Desired* tree
Higher *TreeHash*	*TreeHash* which calculates the non–bottom level's nodes in a *Desired* tree
Round	Time between the calculation of two successive authentication paths (one run of Alg. 8 and Alg. 9)
H	Height of the Merkle tree
L	Number of subtrees in a Merkle tree
h	Height of a subtree

1. Introduction

Merkle's binary hash-trees are currently very popular, because their security is independent from any number theoretic conjectures [7]. Indeed their security is based solely on two well defined properties of hash functions: (i) Pre-image resistance: that is, given a hash value v, it is difficult to find a message m such that $v = hash(m)$; and (ii) Collision resistance: that is, given two messages $m_1 \neq m_2$, the probability that $hash(m_1) = hash(m_2)$ can be made as small as one wishes by a suitable choice of n, the number of bits of v. It is interesting to note that the best quantum algorithm to date for searching N random records in a data base (an analogous problem to hash pre-image resistance) achieves only a speed up of $\mathcal{O}(\sqrt{N})$ to the classical one $\mathcal{O}(N)$ [2].
A Merkle tree is a complete binary tree (see Fig. 1.1) with a n-bit value associated with each node. Each inner node value is the result of a hash of the node values of its children. Merkle trees are designed so that a leaf value can be verified with respect to a publicly known root value given the authentication path connecting the leaf to the root. The authentication path consists of one node value at each level l, where $l = 0, \cdots, H-1$, and H is the height of the Merkle tree ($H \leq 20$ in most practical cases). The chosen nodes are siblings of the nodes on the path connecting the leaf to the root.(see Fig. 1.2).

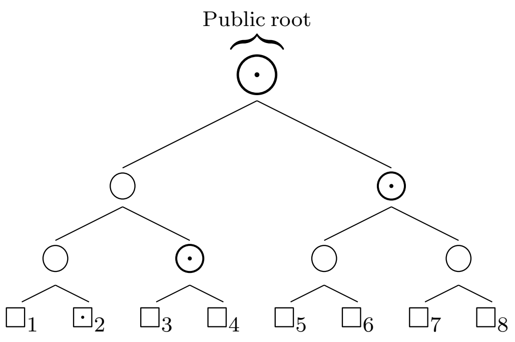


Figure 1.1.: Merkle's binary tree structure for $H = 3$: a) $\square_1 \cdots \square_8$ are the one-time signatures or tokens (leaves) b) Each node's value is built from the hash of its two children c) The root node represents the public key d) Verification path for token $\square_1$ is given by all nodes labelled with a dot.

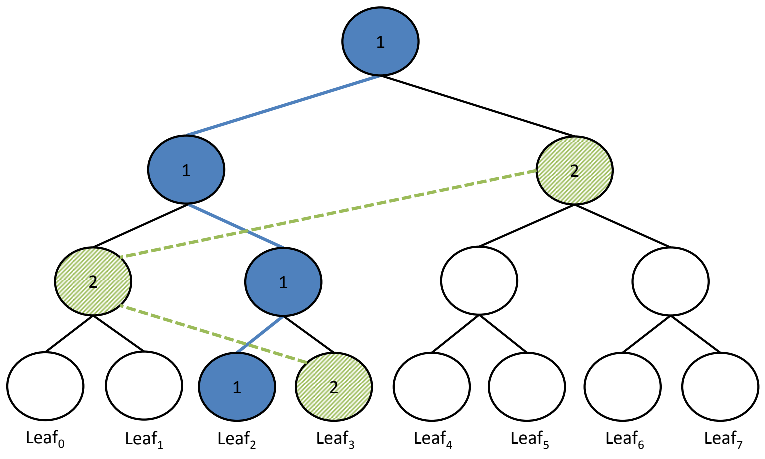


Figure 1.2.: The nodes labled with a 1 are lying on the path from $Leaf_2$ to the root. The nodes labled with a 2 are the nodes of the authentication path for $Leaf_2$; all of them are siblings of a node labled with a 1.

The Merkle tree traversal problem answers the question of how to calculate efficiently[1] the authentication path for all leaves one after the other starting with the first $Leaf_0$ up to the last $Leaf_{2^H-1}$, if there is only a limited amount of storage available (e.g. in smartcards).
The generation of the public key (the root of the Merkle tree) requires the computation of all nodes in the tree. This means a grand total of 2^H leaf evaluations and of $2^H - 1$ hash computations. The root value (the actual public key) is then stored into a trusted database accessible to the verifier.
The leaves of a Merkle tree are used either as a one time token to access resources or as building blocks for a digital signature scheme. In the first case the tokens can be as simple as a hash of a pseudo–random number generated by the provider of the Merkle tree. In the latter, more complex schemes are used in the literature (see for example [3] for a review).

1.1. Related work

Two solutions to the Merkle tree traversal problem exist. The first is built on the classical tree traversal algorithm but with many small improvements [9] (called Log algorithm from now on). The second one is the fractal traversal algorithm [4] (called Fractal algorithm from now on). The Fractal algorithm trades efficiently space against time by adapting

[1]The authors of [8] proved that the bounds of space ($\mathcal{O}(tH/log(t))$) and time ($\mathcal{O}(H/log(t))$) for the output of the authentication path of the current leaf are optimal (t is a freely choosable parameter).

the parameter h (the height of a subtree, see Fig. 2.1), however the minimal space it uses for any given H (if h is chosen for space optimality) is more than what the Log algorithm needs. The Log algorithm cannot as effectively trade space for performance. However, for small H it can still achieve a better time and space trade-off than the Fractal algorithm. But beyond a certain value of H (depending on the targeted time performance) the Fractal algorithm uses less space.

1.2. Our contributions

We developed an algorithm for the Merkle tree traversal problem which combines the efficient space-time trade-off from [4] with the space efficiency from [9]. This was done by applying all the improvements discussed in [9] to the Fractal algorithm [4]. We also analysed the space impact of a continuous deterministic pseudo–random number generator[2] on the algorithms. All these improvements lead to an algorithm with a worst case storage of $[L \times 2^h + 2H - 2h]$ hash values (see Sec. 4.3). The worst case time bound for the leaf computation per round[3] amounts to $\frac{2^h-1}{2^h} \times (L-1)+1$ (see Sec. 4.3).
We further implemented the algorithm in Java with focus on a low space and time overhead and measured its performance (Chapt. 6).

[2] A deterministic pseudo random number generator which can not access any random number in its range without first computing all the preceding ones.
[3] One round corresponds to the calculation of one authentication path

2. Preliminaries

The idea of the Fractal algorithm [4] is to store only a limited set of subtrees within the whole Merkle tree (see Fig. 2.1). They form a stacked series of L subtrees $\{Subtree_i\}_{i=0}^{L-1}$. Each subtree consists of an *Exist* tree $\{Exist_i\}$ and a *Desired* tree $\{Desired_i\}$, except for $Subtree_{L-1}$, which has no *Desired* tree. The *Exist* trees contain the authentication path for the current leaf. When the authentication path for the next leaf is no longer contained in some *Exist* trees, these are replaced by the *Desired* tree of the same subtree and then a new empty *Desired* tree is created. The *Desired* trees are built incrementally after each output of the authentication path algorithm, thus minimizing the operations needed per round to evaluate the subtree.

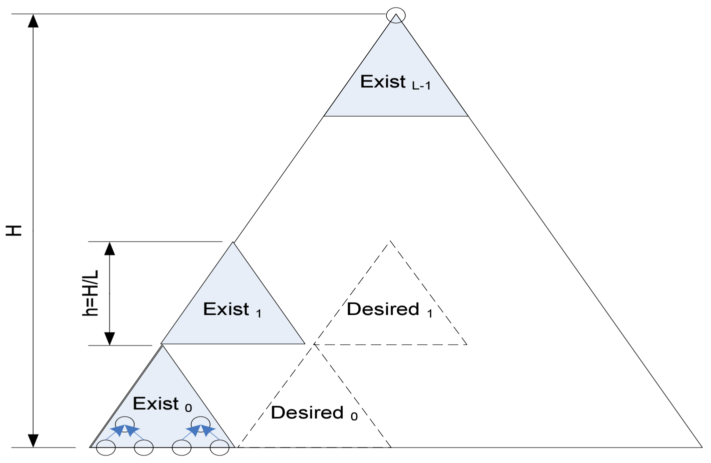


Figure 2.1.: Fractal Merkle tree structure and notation (Figure courtesy of [4]). A hash tree of height H is divided into L levels, each of height h. The leaves of the hash tree are indexed $\{0, 1, ..., 2^H - 1\}$ from left to right. The height of a node is defined as the height of the maximal subtree for which it is the root and ranges from 0 (for the leaves) to H (for the root).

The nodes in a Merkle tree are calculated with an algorithm called *TreeHash*. The classical *TreeHash* algorithm takes as input a stack of nodes, a leaf calculation function and a hash–function and it outputs an updated stack, whose top node is the newly calculated node. Each node on the stack has a height i that defines on what level of the Merkle tree this node lies: $i = 0$ for the leaves and $i = H$ for the root. The *TreeHash* algorithm works in small steps. On each step the algorithm looks at its stack and if the top two elements have the same height it pops them and puts the hash value of their concatenation back onto the top of stack which now represents the parent node of the two popped ones. Its height is one level higher than the height of its children. If the top two nodes do not have the same height, the algorithm calculates the next leaf and puts it onto the stack, this node has a height of zero. To be able to better compare different versions of the $TreeHash$ algorithm we define two steps of the classical $TreeHash$ as an update. For the calculation of all nodes in a tree of height H, the classical $TreeHash$ needs $2^H - 2^{-1}$ updates (2^{-1} because the last update needs only to do one step).

We quickly summarize the three main areas where improvements from [9] were critical for a better space–time performance:

1. Left nodes[1] have the nice property, that when they first appear in an authentication path, their children were already on an earlier authentication path (see Fig. 2.2). For right nodes[2] this property does not hold. We can use this fact to calculate left nodes just before they are needed for the authentication path without the need to store them in the subtrees. So we can save half of the space needed for the subtrees, but one additional leaf calculation has to be carried out every two rounds.

2. In most practical applications, the calculation of a leaf is more expensive than the calculation of an inner node[3]. This can be used to design a variant of the $TreeHash$ algorithm, which has a worst case time performance that is nearer to its average case for most practical applications. The improved $TreeHash$ (see Alg. 6) calculates one leaf and as many inner nodes as possible per update step before needing a new leaf, instead of processing just one leaf or one inner node as in the classical case.

3. In the fractal Merkle tree one $TreeHash$ instance per subtree exists for calculating the nodes of the *Desired* trees and each of them gets an update per round. Therefore all of them have nodes on their stacks which need space of the order of $\mathcal{O}(\frac{H^2}{h})$. We can distribute the updates in another way, so that most $TreeHash$ instances are empty or already terminated. This reduces the space needed by the stacks of the $TreeHash$ instances to $\mathcal{O}(H - h)$.

It is easy enough to adapt point one and two for the Fractal algorithm, but point three needs some changes in the way the nodes in a subtree are calculated (see Sec. 4.1).

[1] A left node is a node witch is the left child of its parent.
[2] A right node is a node witch is the right child of its parent.
[3] A inner node is a node with a height greater than zero.

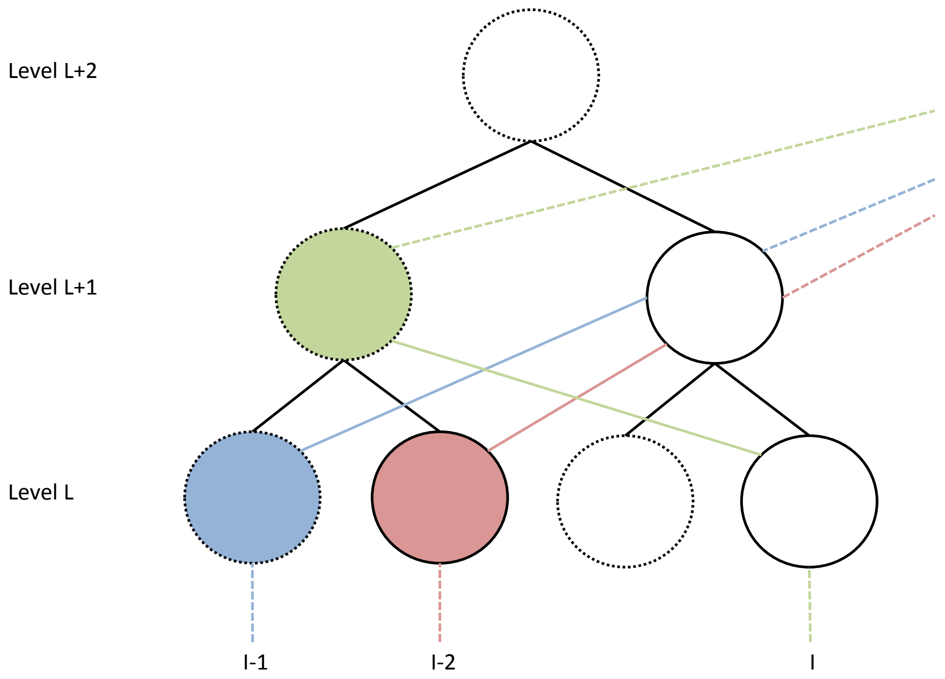


Figure 2.2.: The colored lines mark the different authentication paths. The index I at the start of each line indicates how many times the authentication path has changed on level L. An authentication path that has changed I times on level L has changed $I \times (2^L)$ times on level 0 (which changes each round). The dotted circles are left nodes or the root of a subtree which are not stored in a subtree.

3. Algorithm overview

In this chapter we will give an overview of the whole algorithm and explain how all the parts work together. The algorithm is divided into two phases. The first phase is the initialisation phase in which the public key is calculated (see Alg. 7). To do this the improved $TreeHash$ form [9] is used. The $TreeHash$ algorithm (see Sec. 4.2) is run step by step until the root node is computed. In the initialisation phase each node is computed exactly once. This fact is used to store all right nodes in the first $Exist$ tree of each subtree and the nodes in the authentication path for the $Leaf_0$. The second phase iteratively generates the authentication paths for all the remaining $Leaf_i$ (from left to right) where $i \in \{1, 2, \cdots, 2^H - 1\}$ (see Alg. 8 and Alg. 9). Each authentication path can be computed by changing the previous authentication path [7]. The authentication path for $Leaf_i$ changes on a level k if $2^k | i$[1]. If the node changes to a right node, it can be found in one of the $Exist$ trees. If it changes to a left node, it can be computed from its two children. The right child can be found in the $Exist$ trees and the left child is on the previous authentication path (see Fig. 2.2). When a node in the $Exist$ tree is no longer needed for the computation of any upcoming authentication path, it is removed. To prevent the $Exist$ tree running out of nodes, all the nodes in the $Desired$ tree have to be computed before the $Exist$ tree has no nodes left. This is done with the help of two $TreeHash$ instances per subtree. One, called the lower $TreeHash$, calculates all nodes on the bottom level[2] of a $Desired$ tree (called bottom level nodes) from the leaves of the Merkle tree. The other, called the higher $TreeHash$, calculates all the remaining $Desired$ nodes[3] (called non-bottom level nodes) from the bottom level ones. All the lower $TreeHash$ instances use the scheduling algorithm from [9], whereas the higher $TreeHash$ gets an update every $2^{BottomLevel}$ rounds. The higher $TreeHash$ produces a node on a level k in the $Desired$ tree every 2^k rounds, which corresponds to the rate at which the authentication path changes on that level. When the last node from the $Exist$ tree is removed all the nodes in the $Desired$ tree are computed and the $Exist$ tree can be replaced with the $Desired$ tree. In section 4.2 we will prove, that the lower $TreeHash$es produce the nodes on the bottom level before the higher $TreeHash$es need them if $L - 1$ updates are done per round. A lower $TreeHash$, which has terminated, is initialized again as soon as the generated node is used as input for the higher $TreeHash$. Because only the right nodes are stored in the subtrees, the $TreeHash$es do only have to compute

[1] $m|n$ means that n is a multiply of m

[2] The bottom level is the lowest level in a $Desired$ tree.

[3] $Desired$ nodes are all the nodes stored in a $Desired$ tree.

left nodes which ar used to calculate a right node contained in the *Desired* tree. The only nodes never used to compute a right node in a *Desired* tree is the first left node of each level in this *Desired* tree. To ensure that no unneeded nodes are computed, the lower $TreeHash$ does not compute nodes for its first $2^{BottomLevel}$ updates per *Desired* tree and the higher $TreeHash$ does not compute nodes for its first update per *Desired* tree.

4. Analysis

In this chapter we will analyse the space time performance of our algorithm. The analysis is done as a function of H (height of the tree) and h (height of a subtree). It is assumed that all subtrees have the same height and that $h|H$. In addition we assume $h < H$. These restrictions are not necessary when the algorithm is used in practice. We applied these restrictions to decrease the number of variables (only h instead of $\{h_i\}_{i=0}^{L-1}$ for defining the subtrees) and enhance the comprehensibility. For the storage analysis, it is assumed that if the result of a computation is stored in multiple data structures, it is held in memory only once.

4.1. Computation of the Desired tree

In this section we will explain how the nodes in a *Desired* tree are computed and stored, which hallmarks the main difference of our algorithm compared with the algorithm [8]. In [8], the 2^h nodes with the lowest level in a *Desired* tree are calculated during the first $2^h \times (2^{bottomLevel} - 2^{-1})$ updates[1]. The remaining 2^{h-1} updates are used for calculating the non-bottom level nodes of the *Desired* tree [8]. There is no additional space needed to calculate the non-bottom level nodes from the bottom level ones. This because after calculating a new node, the left child is dropped and the new value can be stored instead [8]. This approach can not be used with the improved $TreeHash$ from [9], because it needs $2^{bottomLevel}$ updates to compute a bottom level node instead of $2^{bottomLevel} - 2^{-1}$. As described in chapter 3 our algorithm uses a lower $TreeHash$ and a higher $TreeHash$ per subtree. All the lower $TreeHash$ instances use the scheduling algorithm from [6], whereas the higher $TreeHash$ gets an update each $2^{bottomLevel}$ rounds. The big difference to [8] is that we compute the nodes in the *Desired* tree continuously during the calculations of the *Desired* tree, and not only at the end. This approach distributes the leaf computations during the computation of a *Desired* tree more equally than the one from [8].

4.1.1. Space analysis for Desired tree computation

We will show that our algorithm needs to store at most $L \times (2^h - 1)$ hash values for the *Exist* and *Desired* tree, when the authentication path is taken into account, instead of

[1]Remember that a classical $TreeHash$ needs $2^{bottomLevel} - 2^{-1}$ updates to compute a bottom level node.

the $L \times (2^h)$ hash values needed by the algorithm [8].
The authentication path is a data structure which can store one node per level. Because the authentication path is contained in all the $Exist$ trees (which store only right nodes), right nodes on the authentication path are contained in both structures and thus have to be held only once in memory.
The authentication path changes on a level k every 2^k rounds, and the higher $TreeHash$ produces a node on a level k' every $2^{k'}$ rounds. Whenever a left node enters the authentication path, its right sibling leaves this path and can be discarded (with the one exception discussed below). The first node on a level in the authentication path no longer needed is a right node, but the first node a higher $TreeHash$ produces is a left node. From this we can conclude, that every 2^{k+1} rounds the $Exist$ tree discards a right node on level k and the higher $TreeHash$ produces a left node on the same level.
We will now look at the exception mentioned above: a right node on level k which has a left node as parent cannot be discarded when it leaves the authentication path, because it is needed for the computation of its parent (see Chapt. 3). It will be discarded 2^k rounds after it left the authentication path. During these 2^k rounds there can be a left node with height k on the higher $TreeHash$, for which storage space must be provided. Fortunately, this situation only occurs if there is a right node in the authentication path. This right node is stored in both, the $Exist$ tree and the authentication path, and must be held in memory only once.
The special scheduling of the lower $TreeHash$ (see Sec. 4.2) may compute a node on the bottom level that is not immediately consumed by the higher $TreeHash$, and therefore should be stored until needed. We can store this node in the space of the higher $TreeHash$, because the left node with the highest level on a higher $TreeHash$ is never stored, for the simple reason that it does not contribute to the calculation of any right node in the subtree (see Fig. 4.1) .
From this we conclude that the authentication path and all the subtrees together use no more than $L \times (2^h - 1) + H$ space, $(2^h - 1)$ is the amount of nodes a tree of height h needs, when it stores only right nodes (see Fig. 4.1), and H is the space needed to store the current authentication path.

4.1.2. Sharing the same data structure in both Exist and Desired trees

We now show that we can store the nodes of the $Exist$ tree and the $Desired$ tree in one single tree data structure. This is the case because we can store two related[2] nodes in the same slot. We can do this, because when a node in the $Desired$ tree is stored, its related node in the $Exist$ tree was already discarded. This is trivial for left nodes because they are never stored in the $Exist$ or $Desired$ tree.

[2]Two nodes of a subtree one from the *Desired* and the other from the $Exist$ tree are said to be related nodes if they have the same position relative to their root in the subtree.

In the previous section, we showed that, with one exception, the *Exist* tree discards a right node on a level in the same round the higher *TreeHash* computes a left node on that level. The sibling of the left node a higher *TreeHash* computes every 2^{l+1} rounds on a level l, is the node related to the right node the *Exist* tree discards at level l in the same round. The right node which is computed 2^l rounds later on the level l, is the node related to the discarded one, and so it can be stored in the same slot of the data structure. We now look at the special case: right nodes with a left node as parent (see Sec. 4.1.1). Such a right node on level k will be discarded 2^k rounds later as the other right nodes. It will be discarded in the same round as the higher *TreeHash* produces its related node. We ensure that the slot in the data structure is free by calculating left nodes in the authentication path before we update the higher *TreeHash* (see Alg. 9).

This leads to the following order of steps in our algorithm (see Alg. 9):

1. Calculate new left nodes in the authentication path (see Alg. 9);
2. Update higher *TreeHash*es (see Alg. 9);
3. Calculate new right nodes in authentication path (see Alg. 9).

In Fig. 4.1 we show how the different nodes of the *Desired* and *Exist* trees are managed.

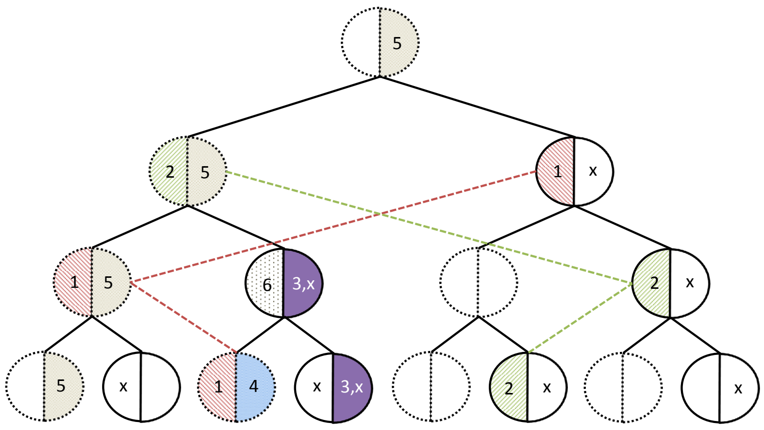


Figure 4.1.: The left half of each circle represents the *Exist* tree and the right half the *Desired* tree. The nodes with dotted lines are left nodes, or the root, and thus are not stored in the subtree, but they may be stored in the authentication path or on the higher *TreeHash*. The markings on the nodes have the following meanings: Label x: nodes already discarded (in case of *Exist* tree) or not yet computed (in case of *Desired* tree). Label 1: nodes lying on the current authentication path. Label 2: nodes lying on the upcoming authentication path. Label 3: nodes computed by the next higher *TreeHash* update. Label 4: left node on the higher *TreeHash*. Label 5: nodes which never contribute to a right node calculation (they are not stored in higher *TreeHash*). Label 6: node which could not yet be discarded, because it is needed for calculating a left node in the upcoming authentication path.

4.1.3. Space used for the key generation of the leaves

We will now analyse the space used by the deterministic pseudo–random number generators (PRNG), which calculate the private keys used in the leaf calculations. Suppose the PRNG algorithm can generate any random number within its range without first calculating all the preceding ones (indexed PRNG), then only one instance of the PRNG would be needed to calculate the private keys for all the leaves. No PRNG's currently recommended by NIST [10] have this property. For both, the Log and the Fractal algorithms, solutions exist that use a PRNG which calculates the leaves' private keys in a sequential order (continuous PRNG). This requires storing multiple instances of the continuous PRNG during the generation of the authentication paths. The Fractal algorithm stores as many continuous PRNG instances as it has subtrees, whereas the Log algorithm stores two continuous PRNG instances per $TreeHash$ [9] and one for calculating the leaves which are left nodes. Our algorithm uses the same PRNG approach as the Fractal one. When it skips a leaf calculation (because it would not contribute to the calculation of a right node stored in a subtree), it still calculates the leaf's private key and thus advances the state of the PRNG. Therefore, our algorithm and the Fractal one store L additional continuous PRNG states, whereas the Log algorithm needs to store $2 \times (H - K) + 1$ continuous PRNG states [9]. The internal state of a PRNG has to be at least the size of the output length of the hash function used, otherwise its security would be weaker than the security of the rest of the tree.

4.2. The TreeHash algorithm

In this section we will explain the reason why we use the $TreeHash$ scheduling from [6] together with the improved $TreeHash$ from [9] and what impact this has on the performance of the algorithm. A $TreeHash$ instance which calculates a node on height i and all its children is called $TreeHash_i$. For each *Desired* tree in a subtree we need a lower $TreeHash_{bottomLevel}$ instance. Each of these instances have up to $bottomLevel + 1$ nodes on their stack in the classical $TreeHash$. If we compute them simultaneously, as it is done in [7], it can happen that each instance has its maximum amount of nodes on their stack. The update scheduling algorithm proposed in [6] uses less space by computing the $TreeHash$ instances, so that most of them are not yet started or are already terminated. The former means no nodes, the latter only one node on the stack. The basic idea is to start a freshly initialized $TreeHash_k$ only if there is no $TreeHash$ with nodes of height smaller than k on their stack. This is achieved by assigning each update to the $TreeHash$ instance with the smallest tail height (see Alg. 1). The tail height is the smallest height for which there is a node on the stack of the $TreeHash$. An empty $TreeHash_k$ is considered to have a tail height of k and a terminated one is considered to have a infinite tail height. Furthermore, the improved $TreeHash$ from [9] we use, changes the definition of a update compared to the classical $TreeHash$. Originally in [7], a update was considered two steps each of them calculating either a leaf node or an inner node. This is fine as long as a hash computation can be considered as expensive as a leaf calculation. More often though, a leaf computation is significantly more expensive than the one of an inner node. This leads to a larger difference between the average and worst case time needed for a update. The update of the improved $TreeHash$ instead calculates one leaf and all the right node parents of this leaf (see Alg. 6).

Nodes' supply for the higher TreeHash

We wish to prove that when we spend $L - 1$ updates on the lower $TreeHash$es they produce nodes before the higher $TreeHash$ needs them. Our proof will be based on the one in [9] which states that in the Log algorithm the $TreeHash$es produce the nodes before they are needed for the authentication path. We focus on a subtree ST_j with a lower $TreeHash_k$ where k is the bottom level of ST_j. We consider a time interval starting at the initialization of $TreeHash_k$ and ending when the next node at height k is required by the higher $TreeHash$ of ST_j. We call this node $Need_k$. The higher $TreeHash$ requires a node every 2^k rounds. In the time considered we execute $(L - 1) \times 2^k$ updates. On a lower level, the authentication path changes more often and thus a higher $TreeHash$ of a subtree containing lower levels needs new nodes more frequently. For any given $TreeHash_i$ with $i < k$, $\frac{2^k}{2^i}$ nodes are needed during the time interval defined above. Each of them needs 2^i updates. Thus $TreeHash_i$ needs $\frac{2^k}{2^i} \times 2^i = 2^k$ updates to produce all nodes needed. If there are N $TreeHash_i$ with $i < k$, then all of them together need at most $N \times 2^k$ updates to compute all their nodes needed. They may need less because they may already have nodes on their stack. There may be a partial contribution to any $TreeHash_j$ with $j > k$. But they can only receive updates as long as they have nodes at height $< k$ (tail height $< k$). A $TreeHash_j$ needs at most 2^k updates to raise its tail height to k. There are $(L - N - 2)$ $TreeHash_j$ with $j > k$ (the top subtree has no $TreeHash$). Together they need at most $(L - N - 2) \times 2^k$ updates. All $TreeHash_s$ with $s \neq k$ need at most $(L - N - 2) \times 2^k + N \times 2^k = (L - 2) \times 2^k$ updates. This leaves $(L - 1) \times 2^k - (L - 2) \times 2^k = 2^k$ updates for $TreeHash_k$, which are enough to compute $Need_k$.

4.2.1. Stack sharing in TreeHash algorithm

We will show, that when we use the improved scheduling algorithm from [9] and the node with height h of a $TreeHash_h$ is stored outside the stack, we can share a stack between all the lower $TreeHash$es. Besides this, we will analyse how much space this shared stack needs. This was already analysed in [9] for the Log algorithm. We explain it from another perspective better suited for our needs.
Consider $TreeHash_i$ and $TreeHash_k$ with $i < k$. We will prove that when $TreeHash_k$ gets an update, $TreeHash_i$ has no nodes stored on its stack (tail height i or ∞) and thus the nodes of $TreeHash_k$ will be on top of the shared stack. In Figure 4.2 we see all the relevant states these two $TreeHash$ instances can be in, with all the possible transitions between them.

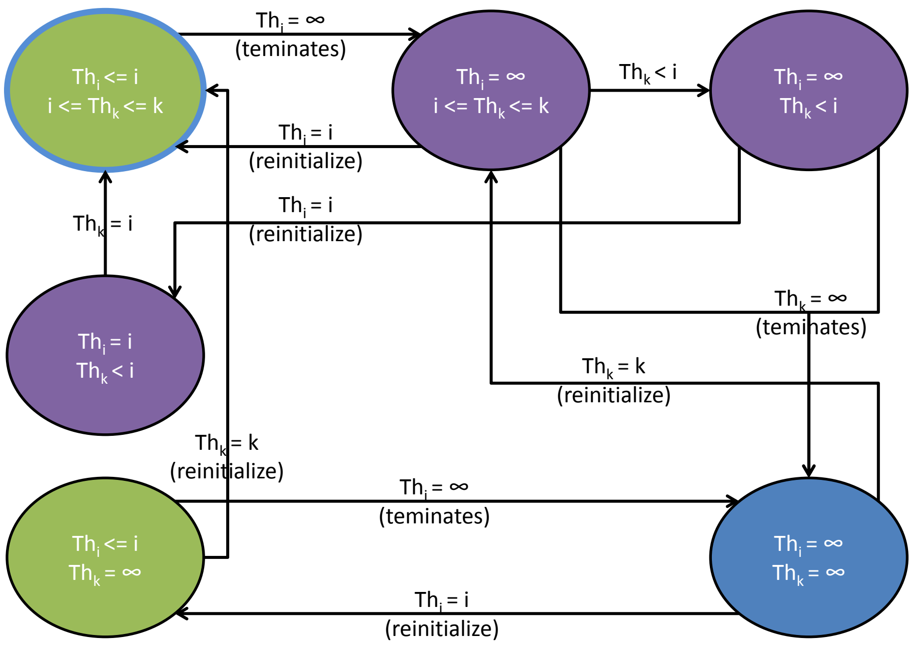


Figure 4.2.: A state diagram displaying how the tail heights Th_i and Th_k of two $TreeHash$es $TreeHash_i$ and $TreeHash_k$ changes over time when $i < k$. Each state has some conditions restricting the values of the tail heights of the two $TreeHash$es. When a change of a tail height violates one of these conditions, a transition leads to a new state, where none of the conditions are violated. The green circles represent states where $TreeHash_i$ will get updates. The lila circles represent states where $TreeHash_k$ will get updates. The blue circle represents the state where none of the two $TreeHash$es gets an update. The state with the blue border is the start state.

We see that in each state where $TreeHash_k$ gets an update (lila colored states), $TreeHash_i$ has a tail height of i or ∞ and thus has no nodes on its stack. In addition, we see that in the only state where none of the two $TreeHash$es is terminated or empty, $TreeHash_k$ has a tail height $>= i$ and thus no nodes on a level smaller than i. $TreeHash_i$ can only have nodes with a level $< i$ on its stack. This proves that there is at most one node of each height at any time on the shared stack.
The highest subtree (bottom level: $H - h$) does not have a lower $TreeHash$ instance, thus the highest level on which ever a node has to be computed by a lower $TreeHash$ is the bottom level of the second highest subtree (with bottom level: $H - h - h$). The shared stack needs to store hash values on $H - 2h$ levels. In total we need to store $H - 2h$

nodes for all lower TreeHash–instances.

4.3. The space and time gains of our approach

In this section we will give the total space–and time–bounds of our algorithm, and compare them with the Log and Fractal ones under the condition that a continuous PRNG with an internal state equal in size of the hash value is used. We obtain the total space needed by our algorithm by summing up the contributions of its different parts: $L \times (2^h - 1) + H$ from the subtrees and authentication path (see Sec. 4.1.1), $H - 2h$ from the lower $TreeHash$es (see Sec. 4.2) and L from the PRNG internal states (see Sec. 4.1.3). This sums up to $L \times 2^h + 2H - 2h$ times the hash value size.
For the time analysis we look at the number of leaves' calculations per round. The improved $TreeHash$ makes one leaf calculation per update and we make at most $(L-1)$ lower $TreeHash$ updates per round. The higher $TreeHash$ never calculates leaves. So in the worst case all $TreeHash$es together need $(L-1)$ leaves' calculations per round. We need an additional leaf calculation every two rounds to compute the left nodes of the authentication path (see Chapt. 3). In the worst case we need L leaves' calculations per round. In the average case we need less, since the first node of the 2^h bottom level nodes of a $Desired$ tree is not computed because it is not needed to compute any right node in the $Desired$ tree.
This reduces the average case time by the factor $\frac{2^h-1}{2^h}$ and leads to a total of $\frac{2^h-1}{2^h} \times (L-1) + \frac{1}{2}$ leaves' computations per round. The term $\frac{1}{2}$ enters the expression because the left node computation needs a leaf every two rounds. The average case time bound considers only the first $2^H - 2^{H-h}$ rounds. Thereafter, less leaf computations would be needed on average because some subtrees no longer need a $Desired$ tree. Table 4.1 summarizes our results and Table 4.2 does the same for the Log and Fractal algorithm, where all uses a continuous PRNG with an internal state equal to the size of a hash value.

Bounds	$h = 1,\ L = H$	$h = 2,\ L = \frac{H}{2}$	$h = log(H),\ L = \frac{H}{log(H)}$
Worst case: space	$4H - 2$	$4H - 4$	$\frac{H^2}{log(H)} + 2H - 2log(H)$
Average case: time	$\frac{H}{2}$	$\frac{3H-2}{8}$	$\frac{H-1}{log(H)} + \frac{1}{2}$
Worst case: time	H	$\frac{H}{2}$	$\frac{H}{log(H)} - 1$

Table 4.1.: Space–time trade–off of our Merkle tree traversal algorithm as a function of H (height of the tree) with h (height of a subtree) as parameter.

Bounds	Log [9] $K = 2$	Log [9] $K = 4$	Fractal [4] $h = \log(H)$
Worst case: space	$5.5H - 7$	$5.5H - 5$	$\frac{5H^2+2H}{2log(H)}$
Average case: time	$\frac{H}{2} - \frac{1}{2}$	$\frac{H}{2} - \frac{3}{2}$	$\frac{H}{log(H)} - 1$
Worst case: time	$\frac{H}{2}$	$\frac{H}{2} - 1$	$\frac{2H}{log(H)} - 2$

Table 4.2.: Space–time trade–off of Log algorithm [9] and Fractal algorithm [4] optimized for storage space. The values in the table include the space needed by the continuous PRNG.

For the Log algorithm we included $K = 4$ in the Table 4.2 because if the continuous PRNG is taken into account, the Log algorithm achieves best space with $K = 4$ instead of $K = 2$, as proposed in [9]. For the Log algorithm it is assumed that H is even. When $h = 2$, our algorithm has better worst case space and similar time bounds than the Log algorithm [9]. When we choose the same space–time trade-off parameter as in the Fractal algorithm [4] (column $h = log(H)$ in Table 4.1), our algorithm needs less storage space.

4.4. Side channel attack resistance of our approach

A modified Log algorithm was discussed in [15] which showed a higher resistance against side channel attacks. They used the so called bounded leakage methode [16]. The idea behind this methode is that even when a cryptographic function is vulnerable to side channel attacks in general, the vulnerability can only be exploited if the cryptographic function is executed enough times against the same input. Thus when reducing the number of times the function is applied on the same input the resistance against side channel attacks is increased. It was shown in [15] that the leakage is directly dependent on the number of times a leaf is calculated. By reducing the number of times a leaf is calculated, the algorithm leaks less information and thus its side channel attack resistance is increased. The Log algorithm needs to compute a leaf at most $(H - K)$ times during the authentication path generation phase and once during the initialisation phase. The modification in [15] reduced the times a leaf has to be computed during authentication path computations to $(H - K)/2$ but the price to pay for this reduction are $\binom{H-K}{2} = ((H - K)^2 - (H - K))/2$ additional hash values to be stored [15].
In total, our algorithm computes a leaf at most $L = H/h$ times. Table 4.3 and 4.4 depicts the worst case leakage and the space usage (for our space formula see Sec. 4.3) for our, the Log and the improved algorithm from [15]:

Parameters	Log [9]	modified Log [15]	Our
$h = K = 2$	$H - 2$	$H/2 - 1$	$H/2$
$h = K = 4$	$H - 4$	$H/2 - 2$	$H/4$
$h = K = 6$	$H - 6$	$H/2 - 3$	$H/6$

Table 4.3.: Worst–case number of computations of a leaf for different parameters, and for the three algorithms.

Parameters	Log [9]	modified Log [15]	Our
$h = K = 2$	$5.5H - 7$	$\frac{H^2}{2} + 3H - 4$	$4H - 4$
$h = K = 4$	$5.5H - 5$	$\frac{H^2}{2} + H + 5$	$6H - 8$
$h = K = 6$	$5.5H + 33$	$\frac{H^2}{2} - H + 21$	$12H + \frac{2}{3}H - 12$

Table 4.4.: Worst–case number of hash values stored for different parameters, and for the three algorithms. PRNG internal state accounted with one hash value

As the tables 4.3 and 4.4 show, the reduced leakage is bought in with a quadratic term in the modified Log algorithm [15]. In our algorithm, the cost for decreasing the leakage is bought in with a constant factor. If a configuration is needed with small leakage and acceptable space, our algorithm with $h = 4$ is a good candidate.

5. Implementation

We implemented our algorithm in Java. There are several aspects which are by purpose unspecified by the Merkle tree traversal algorithms. These are the hash function, the deterministic pseudo–random number generator and the algorithm used for the leaf calculation. The latter is defined by the usage of the tree. The hash function and PRNG on the other hand are independent of the trees' usage, but these have nevertheless an impact on the cryptographic strength and the performance. The hash function used for the traversal algorithm must be collision-resistant as shown in [14]. Good performance and strong security are main selection criteria for this function. A suitable candidate is BLAKE [5].
We implemented BLAKE in a way to minimalize overhead for the two most common cases:

1. Concatenate two hash values and take the hash of them: $hash(hash1||hash2)$ becomes $hash(hash1, hash2)$
2. Iteratively hash a hash value: $hash(hash(hash1))$ becomes $hashIteratively(hash1,2)$.

Case one is used to calculate a node in the Merkle tree from its children, and case two for calculating a Winternitz[1] signature and public key. The overhead reduction in both cases is achieved through the fact that the size of the input is known. Additionally, some intermediate transformations can be skipped in the second case.
As a PRNG we chose an algorithm based on a hash function. In [10], NIST has recommended two continuous hash–based PRNGs, named Hash_DBRG and HMAC_DBRG. Both of them have an internal state composed of two values with the same length as the output length of the used hash function. Hash_DBRG has the advantage that one of its two internal values solely depends on the seed and does not change until a reseeding occurs. For Merkle trees, there is no reseeding necessary as long as less than 2^{48} leaves exist [10]. Hence, in our application one of its two internal values is the same for all Hash_DBRG instances used within the same Merkle tree. We prefer Hash_DBRG over HMAC_DBRG because it uses less space and is more performant.

[1]Winternitz is one of the most popular one time signature schemes

5.1. Subtree implementation

In this section we will show what data structure is used for a subtree and how it is managed. It was shown, in the section 4.1.2 that we can share a data structure for both, the *Exist* and the *Desired* tree. We use an array which only contains the hash value of the nodes. The array index where a node is stored is computed from the nodes position in the *Exist* or *Desired* tree: $index = 2^{h-k-1} + \frac{i}{2} - 1$, where k is the level in the *Exist* or *Desired* tree and i the index on this level (from left to right). Thus the highest level $k = h - 1$ is stored at the start of the array and the lowest $k = 0$ at the end. Using this encoding, only right nodes can be stored (the term $\frac{i}{2}$ requires that $2|i$). Both, level and index, are relative to the *Exist* or *Desired* tree and independent of where in the Merkle tree the subtree is located. Two nodes which are related (see Sec. 4.1.2) always have the same i and k and thus share a slot in the array. This index calculation can be realized very efficiently. The 2^x term is implemented as a left shift by x, and the $\frac{y}{2}$ term as a logical right shift by one. If we know the level and index of a node in the Merkle tree, we can compute k and i: $k = level - bottomLevel$ and $i = index \bmod 2^{h-level}$. The latter is equivalent to the more efficient $i = index \wedge (1 << (h - level) - 1)$, where $<<$ is a logical left shift and $\wedge$ the bitwise AND–operation.

5.2. TreeHash implementation

Normally the *TreeHash* algorithm is described and implemented with a stack storing the hash value and the level of the nodes. The level is needed to decide on each step if a new leaf or a new inner node has to be calculated. For our algorithm we use an array of hash values instead of a stack. For a $TreeHash_i$, an array with a length of i is needed. We showed in section 4.2.1 that when a stack is shared for all lower *TreeHash*es and the improved scheduling algorithm from [9] is used, then at most one node per level is stored in all lower *TreeHash* instances. We use this fact to store the hash value of a node with level i at index i in the array. The level of the node is not stored because it is given by the index. The following Java code demonstrates the lower *TreeHash* algorithm (see Listing 5.1):

```
DataArray node = calcNewLeaf();
int level = 0;

while(lowerArray[level] == null) {
        node = hash(lowerArray[level], node);
        lowerArray[level++] = null;
}

if(level == h) {
        topNode = node;
        tailHeight = Integer.MaxValue;
} else {
        lowerArray[level] = node;
        tailHeight = level;
}
```

Listing 5.1: *lowerArray* is the array used to store the nodes. *tailHeight* as it is defined in section 4.2 and *topNode* is the node which has to be stored outside of the array to be able to share it (see Sec. 4.2.1). *calcNewLeaf*() calculates the hash value of the next leaf and *hash*(a, b) returns the hash of the concatenation of a and b. *DataArray* is the intreface type representing a hash value (see Sec. 5.3).

This implementation has multiple benefits:

- Only the the hash values are stored no additional information nor structural overhead[2] are needed.
- Nearly no computational overhead due to the data structure (just an array access). The array access is generally fast because for moderate sized tree it fits in one cash line.

One aspect not shown in Listing 5.1 is that the first node on the bottom level of a *Desired* tree is not needed to compute any node that is stored in the *Desired* tree. If a lower *TreeHash* would receive an update which would contribute to the calculation of the first bottom level node, the update will be skipped. The code for the higher *TreeHash* is slightly different because the nodes eventually need to be stored in the *Desired* tree (see Listing 5.2). In addition we can use a single array for all higher *TreeHash* instances. This array has H slots. Each higher *TreeHash* uses a number of slots from this array equal to their height. A node on the higher *TreeHash* then is stored at the array slot

[2]Structural overhead in this context means references and other information needed to manage the data structure.

with the index equal to the nodes level in the Merkle tree. With this approach one slot per higher $TreeHash$ is not used because the root node of a subtree is not computed by their higher $TreeHash$. In this slot the $topNode$ of the lower $TreeHash$ can be stored (not shown in Listing 5.1 and 5.2). The benefit of this approach are less array overhead and more cache locality.

```
DataArray node = topNode;
topNode = null;
int level = bottomLevel;

if (!processNode(level, node)) return;
while(higherArray[level] == null) {
        node = hash(higherArray[level], node);
        higherArray[level++] = null;
        if (!processNode(level, node)) return;
}

higherArray[level] = node;
```

Listing 5.2: $higherArray$ is the array used to store the nodes. $topNode$ is the node produced by the lower $TreeHash$, and $processNode$ stores the node into the subtree if it is a right node. It returns true if the node is needed to compute any further nodes stored in the subtree, false otherwise.

5.3. Representation of hash values

Hash values are represented as byte arrays in most implementations. This has a performance impact in the case of BLAKE. Dependent on the BLAKE version used, 32bit integers (int) or 64bit integers (long) arrays are used internally. Coping with the different representations introduces some overhead. To prevent these conversions whenever possible, we define an interface called $DataArray$. This interface has different implementations for different types from a byte array up to a long array. The hash algorithm now decides, at compile time, which implementation to use.

5.4. Representation of the authentication path

The verification of a leaf against the root of a Merkle tree needs to iteratively hash the concatenation of a node with a authentication node. Because $hash(y||x) \neq hash(x||y)$ if $x \neq y$ the verifier has to know whether x is the left or the right node. Often the index of the leaf is passed in addition to the authentication path. It could be calculated from the leaf index whether the value has to be concatenated from the right or the left side. Our implementation already differentiate between left and right nodes. Instead of storing the index in the authentication path, we keep track if a authentication node is a left or right node and store this for each entry. We thus represent the authentication path as an array with H entries where each entry consists of a hash value and the left–right node information.

5.5. Memory management

There are several situations where a node needs to be de–allocated when a programming environment with no garbage collector is used. A right node stored in the current authentication path should be de–allocated as soon as it is removed from the *Exist* tree. This is safe because as long a right node is present in the current authentication path, it is stored in the *Exist* tree too. When a node is no longer in the *Exist* tree, it is no longer on the authentication path either and thus is never needed again.
A left node on the authentication path should be de–allocated as soon as it leaves the path. This can be done because these left nodes are separately calculated and never stored anywhere else. Nodes on a lower *TreeHash* should be de–allocated as soon as they are used to calculate their parent. The last node a lower *TreeHash* computes is used only as input for the higher *TreeHash*. All other nodes will be used only to compute their parent node and thus are no longer needed afterwards. The higher *TreeHash* is responsible for the de–allocation of the last node produced by a lower *TreeHash*. Left nodes on the higher *TreeHash* should be de-allocated in the same way as in the lower *TreeHash*. Right nodes on the other hand are not de–allocated by the higher *TreeHash*. They are de–allocated when they leave the *Exist* tree. During the initialisation phase all nodes which are not stored in a subtree or in the authentication path could be de–allocated as soon as they are used to calculate their parent node.
When no garbage collector is used, it is important, that the nodes in the current authentication path are copied before the next authentication path is generated (see Algorithm 9). If this is not done its nodes would be de–allocated by the authentication path generation algorithm during later rounds.

6. Results

We measured performance of our algorithm in two different ways: (i) hardware independent which mainly measures the algorithmic properties and (ii) hardware dependent. We compare our algorithm with our implementation of the Log and Fractal ones. When implementation details were left open in their description, we used similar techniques their as with our algorithm. As an example, all three algorithms use an array instead of a stack as described in section 5.2. All measurements are done on a round basis, where one round corresponds to the generation of one authentication path. The figures do not show every single round, but an aggregate over several rounds. This enabled us to cogently display measurements for trees of different sizes in a way that can be easily interpreted and compared. The chosen aggregation method is dependent on the aspect of the algorithm we want to visualize. The following aggregation methods are used for aggregating n rounds:

- Maximum: $\underset{i=1,\cdots,n}{max}\{m_i\}$. The maximum value during the n aggregated rounds.
- Average: $\frac{1}{n}\sum_{i=1}^{n} m_i$. The average of the n aggregated round values.
- Sum: $\sum_{i=1}^{n} m_i$. The sum of all values measured during the aggregated rounds.

We used BLAKE with 64 byte output length as hash and Hash_DBRG as PRNG–function for all measurements.

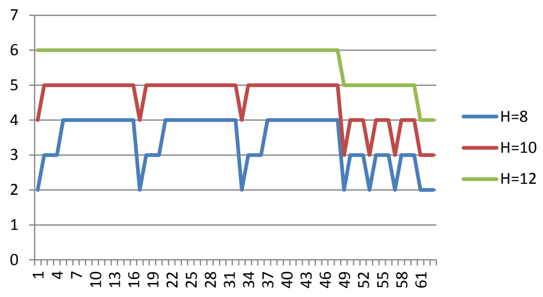
6.1. Hardware independent measurements

We measured the number of leaf computations as well as the amount of stored hash values. The leaf computations were used as an indicator for time performance because when the tree is used for signing, the time spent outside of leaf computations is negligible. These measurements were performed by inserting special code into the function which computes the leaf. This code counts how many times the function is called per round.
A different approach was used to measure the stored hash values. To find out how many hash values need to be stored we have to count all hash values which are stored in at least one data structure. This is done by creating a new implementation of the *DataArray* interface which has a flag. Each time the algorithm stores or releases a hash value, we

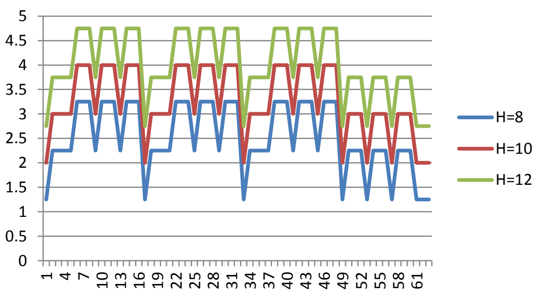
first unset this flag on all hash values stored in any data structure. In a second phase, we set this flag on all hash values stored in any of these data structures. Whenever we set a flag that previously was unset, we increased a counter. At the end the value of the counter represents the number of hash values stored in the Merkle tree. This technique counts each hash value only once even if it is stored in multiple data structures. This may produce multiple values per round, which were aggregated to one value per round depending on what we wanted to measure.

6.1.1. Our algorithm

We first show the results for different H values whereby the parameter h is fixed to 2 (see Fig. 6.1 and Fig. 6.2). The analysis of our algorithm predicts an upper space bound of $4H-3$, and a time bound (leaf calculations) of $H/2$ for the worst–case and $(3H-2)/8$ average case.



(a) Evaluation method: Maximum.



(b) Evaluation method: Average.

Figure 6.1.: Time measurements for $h=2$ with $H=8$, $H=10$ and $H=12$: Number of calculated leaves as a function of blocks of 4 rounds for H=8, 16 rounds for H=10 and 64 rounds for H=12.

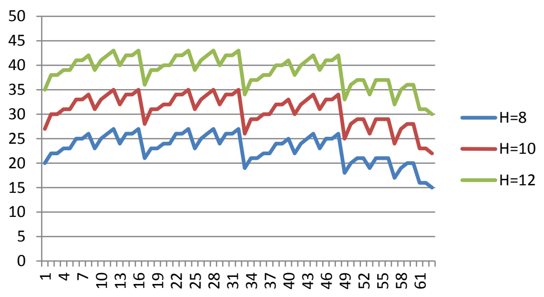
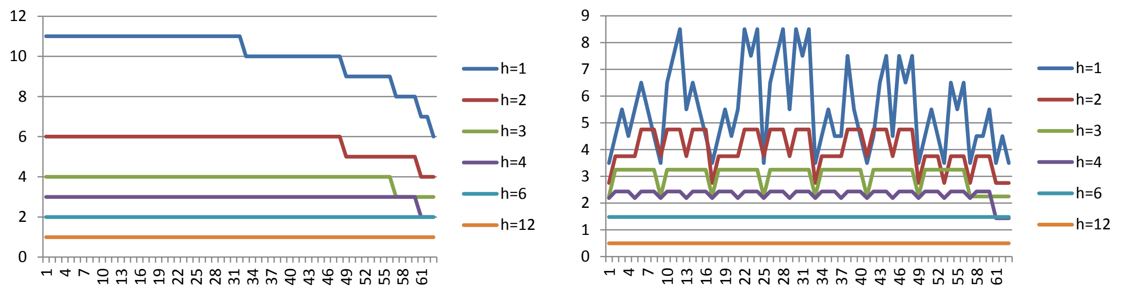


Figure 6.2.: Space measurements for $h=2$ with $H=8$, $H=10$ and $H=12$: Number of stored hash values as a function of blocks of 4 rounds for H=8, 16 rounds for H=10 and 64 rounds for H=12. Evaluation method: Maximum.

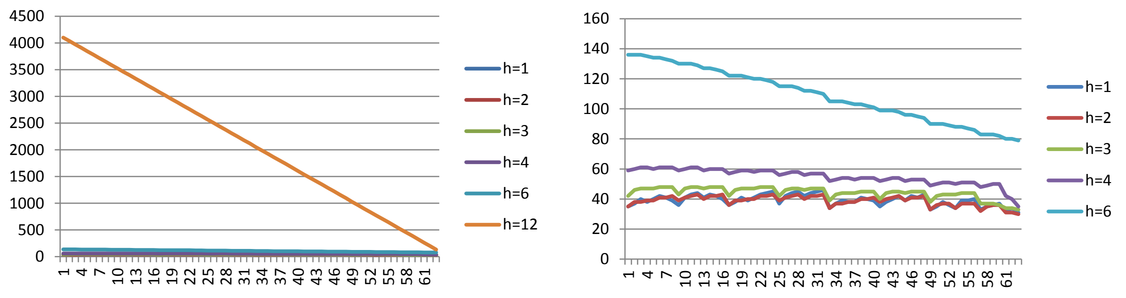
We see in Figure 6.2 that our algorithm stores 43 hash values at its peak for $H = 12$, which is under the predicted upper bound. This reduction is due to the fact that their is no state during the authentication path generation where all analysed contributions use their worst case space. If we reduce H by one, we see that we need four hash values less to store at the peak. This corresponds to the predicted reduction. The predicted worst case leaf calculations are in good agreement for $h = 2$ (see Fig. 6.1a). In addition we see that less leaf calculations per round are needed later on. This is because some subtrees do no longer need updates. The prediction for the average case are in the predicted range for the the first 3072 rounds[1] (around 4.25 for H=12, see Fig. 6.1b). Moreover it seems, that lower H values have a higher instability; this impression results from aggregations over a smaller number of rounds.
Next we measured how our algorithm behaves for $H = 12$ and different values of h (see Fig. 6.3 and Fig. 6.4). For the upper–space bound we expect: $((12 \times 2^h)/h) - 2h + 25$ and for the worst case time: $(12/h)$.



(a) Evaluation method Maximum. (b) Evaluation method Average.

Figure 6.3.: Time measurements for $H = 12$ with $h = 1$, $h = 2$, $h = 3$, $h = 4$, $h = 6$ and $h = 12$: Number of calculated leaves as a function of blocks of 64 rounds.

[1] 3072 rounds are the first $2^H - 2^{H-h}$ when $h = 2$ and $H = 12$ (see Sec. 4.3).



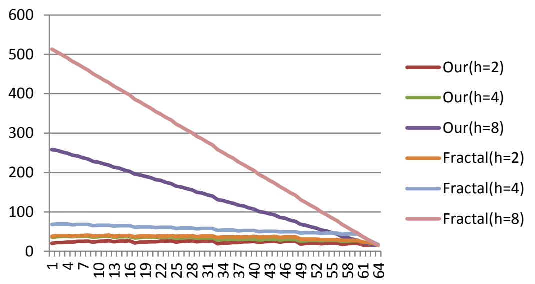
(a) Full. (b) Without h=12.

Figure 6.4.: Space measurements for $H = 12$ with $h = 1$, $h = 2$, $h = 3$, $h = 4$, $h = 6$ and $h = 12$: Number of stored hash values as a function of blocks of 64 rounds. Evaluation method Maximum.

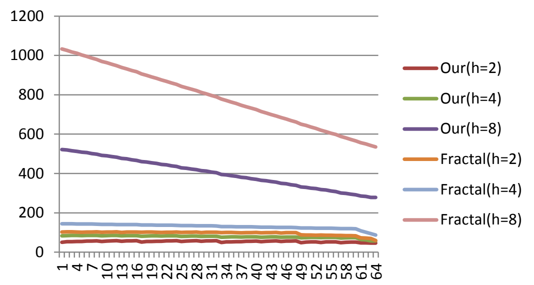
In Figure 6.4a we have a worst case value of 4098 for $h = 12$ but the value predicted by the analysis is only 4097. This disparency is due to the assumption in our analysis that $h < H$. This problem occurs because the contribution to the upper bound from the shared stack is $H - 2h$. For $H = h = 12$ this would be -12. The correct contribution of the shared stack in trees with only one level would be $H - h$ and thus 0.
For the remaining values of h the bound does hold. For example the predicted upper bound for $h = 6$ is 141 and the maximal measurement is 136. In Figure 6.3 we see that except for $h = 1$ the upper leaf calculation bounds predicted by the analysis are in good agreement. For $h = 1$ we need one less leaf calculation per round as the bound would predict. This is the case because in the same rounds where a left leaf is calculated for the authentication path a leaf calculation is skipped on a lower $TreeHash$.

6.1.2. Compared with Fractal algorithm

Here we will show the measurements comparing the space and time performance of our algorithm with the Fractal one. Two different trees were analysed: one with 2^8 leaves, the other with 2^{16} leaves. For the heights of the subtrees (parameter h) we use 1,2,4 and 8 (see Fig. 6.5 and Fig. 6.6).

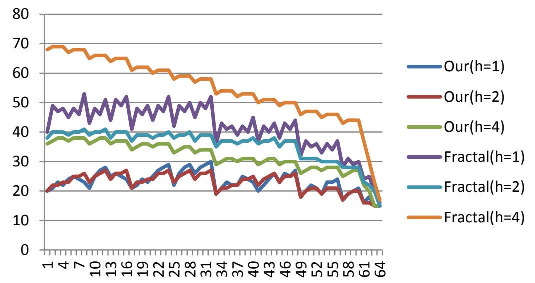


(a) H=8. Blocks of 4 rounds.

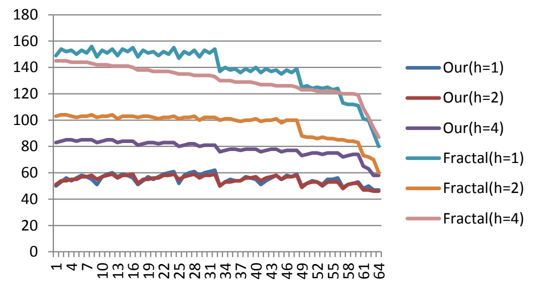


(b) H=16 . Blocks of 1024 rounds.

Figure 6.5.: Space of our algorithm compared with the Fractal one: Number of stored hash values as a function of multiple rounds. Without h=1. Evaluation method Maximum.



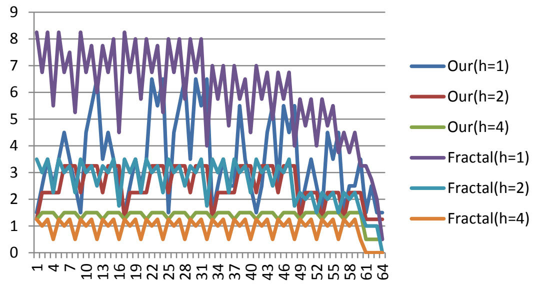
(a) H=8. Blocks of 4 rounds.



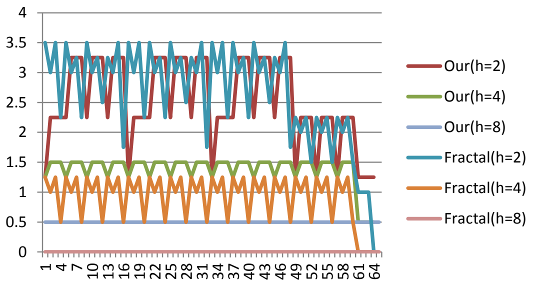
(b) H=16 . Blocks of 1024 rounds.

Figure 6.6.: Space of our algorithm compared with the Fractal one: Number of stored hash values as a function of multiple rounds. Without h=8. Evaluation method Maximum.

We see in Figure 6.5 and Figure 6.6 that within the measured parameter range our algorithm with $h <= 4$ uses less space than the Fractal algorithm independently of its parameter h. For bigger h values we still use less space than the Fractal algorithm with the same h.
Next we examine the values for the average case leaf computations with the same parameters as in the space measurements (see Fig. 6.7 and Fig. 6.8).



(a) Without h=8.



(b) Without h=1.

Figure 6.7.: Time performance of our algorithm compared with the Fractal one for $H = 8$: Number of calculated leaves as a function of blocks of 4 rounds. Evaluation method Average.

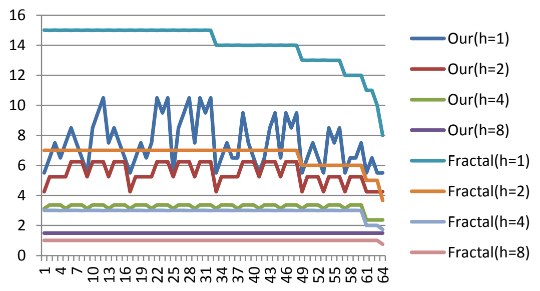
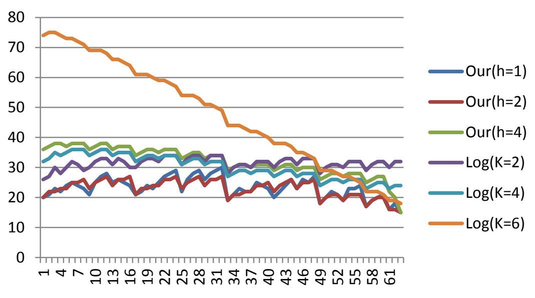


Figure 6.8.: Time performance of our algorithm compared with the Fractal one for $H = 16$: Number of calculated leaves as a function of blocks of 1024 rounds. Evaluation method Average.

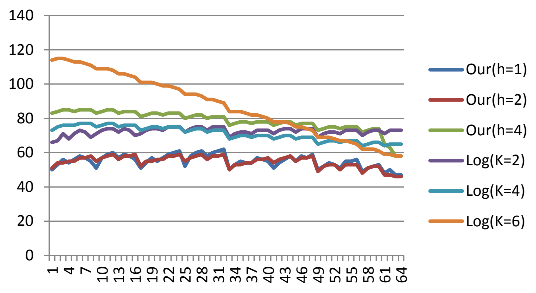
We see in Figure 6.7 and Figure 6.8 that in the measured parameter range our algorithm with $h <= 2$ uses less leaf's calculations than the Fractal algorithm with the same h parameter. For values of $h > 2$, our algorithm uses slightly more leave calculations, but on average at most $\frac{1}{2}$ more per round. These $\frac{1}{2}$ correspond to the left leaf calculations, which are necessary in order to store only right nodes. As a summary we can say that our algorithm uses less space than the Fractal one and has approximately the same time performance.

6.1.3. Compared with Log algorithm

Here we show the measurements comparing the space and time performance of our algorithm with the Log one. Two different trees were analysed: one with 2^8 leaves, the other with 2^{16} leaves. For the heights of the subtrees (parameter h) we use 1, 2 and 4. The values used for the space–time trade–off parameter K of the Log algorithm are 2, 4 and 6 (see Fig. 6.9).



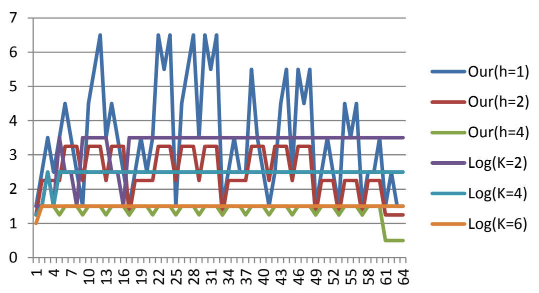
(a) H=8. Blocks of 4 rounds.



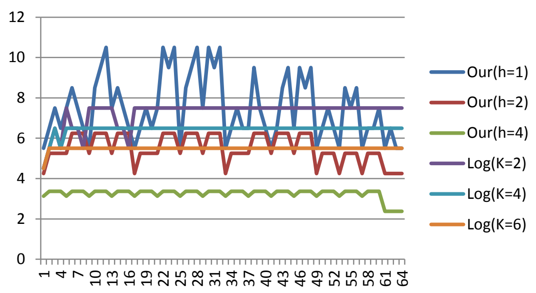
(b) H=16. Blocks of 1024 rounds.

Figure 6.9.: Space of our algorithm compared with the Log one: Number of stored hash values as a function of multiple rounds. Evaluation method Maximum.

We see in Figure 6.9 that within the measured parameter range our algorithm with $h <= 2$ use less space than the Log algorithm independently of its parameter K. For $h = 4$ we need more space than the Log algorithm for $K = 2$ and $K = 4$, but can still compete.
Next we will look at the values for the average case leaf computations for the same parameters as in the space measurements (see Fig. 6.10).



(a) H=8. Blocks of 4 rounds.



(b) H=16. Blocks of 1024 rounds.

Figure 6.10.: Time performance of our algorithm compared with the Log one: Number of calculated leaves as a function of multiple rounds. Evaluation method Average.

We see in Figure 6.10 that within the measured parameter range our algorithm with $h = 4$ achieves the best time performance. The Log algorithm with $K = 6$ performs nearly as well for trees with 2^8 leaves. For 2^{16} leaves our algorithm with $h = 2$ and the Log algorithm with $K = 6$ achieve similar time performance. We can conclude that the

Log algorithm can achieve similar time performance, but needs a lot more space than our algorithm needs for achieving the same time performance.

6.2. Hardware dependent measurements

We only measure the time in the hardware dependent measurements. This because space measurements in a Java virtual machine is dependent on too many uncontrollable factors, like object overhead, escape analysis and garbage collection. The time spent during a round is measured with the Scala code in Listing 6.1.

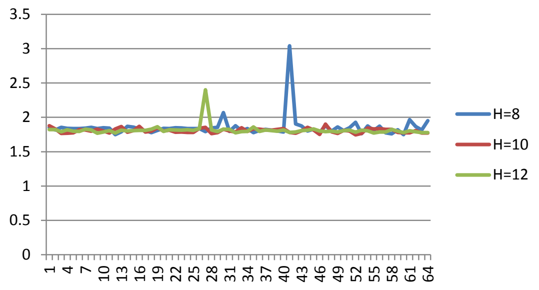
```
recorder.startInitPhase()
var tlast = System.nanoTime()
while(tree.initStep()){
        val tn = System.nanoTime()
        recorder.initPhaseMeasure(tn - tlast)
        tlast = System.nanoTime()
}

recorder.startAuthPhase()
tlast = System.nanoTime()
while(tree.canAuthMore){
        tree.createSignature(msg)
        tree.updateStep()
        val tn =  System.nanoTime()
        recorder.authPhaseMeasure(tn - tlast)
        tlast = System.nanoTime()
}
recorder.finished()
```

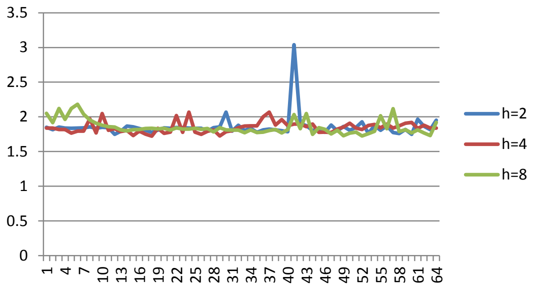
Listing 6.1: Time measurements of a merkle tree. *tree* is the merkle tree instance with the functions: *initStep*() executes one initialisation round and returns false if the whole tree is initialised. *canAuthMore*() returns true if their are more leaves available for signing messages. *createSignature*(*msg*) signs the message *msg* with the current leaf. *updateStep*() computes the authentication path for the next leaf. *recorder* is an object used for gather the results with functions to measure during the initialisation phase (*startInitPhase*() and *initPhaseMeasure*(*time*)) and the authentication phase (*startAuthPhase*() and *authPhaseMeasure*(*time*)).

The code in Listing 6.1 is executed twice in a row per tree. The first run is used to warm up the JVM and the second to produce the better measurements. This approach is not optimal. It would be better to repeat the measurement until n measurements in

a row would differ by no more then ϵ. In our case this approach is not usable because a single measurement for a tree with 2^{16} leaves needs up to 15 minutes. This sums up to multiple hours when all the measurements are done twice. Until n measurements in a row would differ by no more then ϵ for all 2^H rounds of a tree, a huge amount of runs had to be made probably resulting in a runtime of multiple weeks or even months. The time measurements are done with BLAKE as Hash and Winternitz as OTS[2] function. All the measurements were done on a laptop with an i7-2630QM CPU @ 2.00GHz and 8GB ram with a HotSpot JVM version 1.7.0 running on a Windows 7. Influences from other processes and tasks can not be excluded and may result in some unexpected spikes in the measurements (see Fig. 6.11, Fig. 6.12 and Fig. 6.15).
We first look at the time measurements results during the initialisation phase (see Fig. 6.11 and Fig. 6.12).



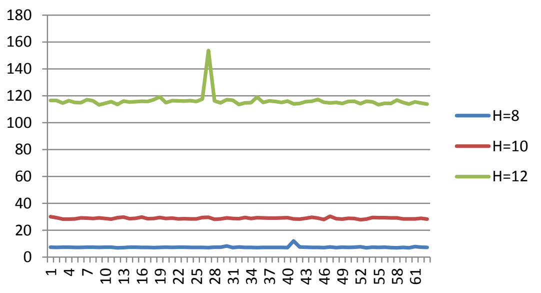
(a) h=2. Blocks of 4 rounds for H=8, 16 rounds for H=10 and 64 rounds for H=12.



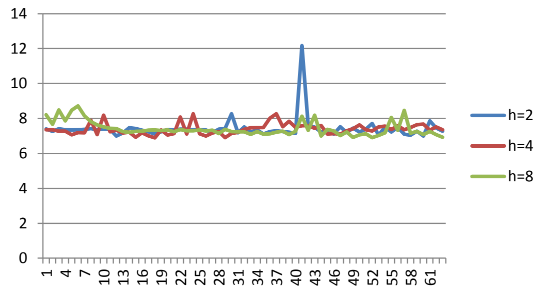
(b) H=8. Blocks of 4 rounds.

Figure 6.11.: Average execution time measurement for initialisation phase: millisecond passed as a function of multiple rounds. Evaluation method Average.

[2]OTS means one time signature scheme.



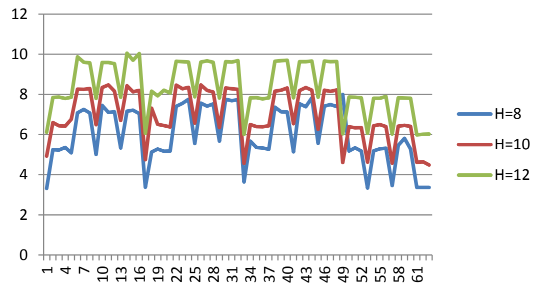
(a) h=2. Blocks of 4 rounds for H=8, 16 rounds for H=10 and 64 rounds for H=12.



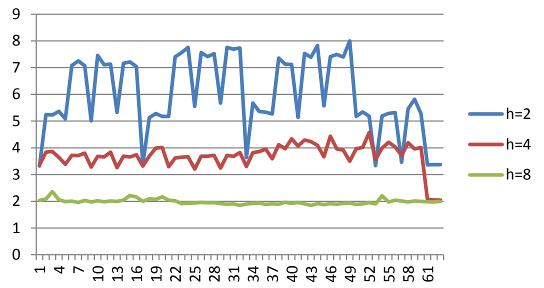
(b) H=8. Blocks of 4 rounds.

Figure 6.12.: Execution time measurements for initialisation: millisecond passed as a function of multiple rounds. Evaluation method Sum.

We can see in Figure 6.11 that the average time needed per initialisation step is not dependent on the Merkle tree height or the subtree height. This is the case because in each initialisation step one leaf is calculated independently from the chosen parameters. We see in Figure 6.12 that the total time is dependent on the parameter H but independent of h.
Next we will look at the signing process which consists of the generation of the authentication paths and the signing of the message. In each round the same message is signed, which is the hash of the empty string.



(a) h=2. Blocks of 4 rounds for H=8, 16 rounds for H=10 and 64 rounds for H=12.



(b) H=8. Blocks of 4 rounds.

Figure 6.13.: Excecution time measurements for message signing: millisecond passed as a function of multiple rounds. Evaluation method Average.

We can see a correlation between Figure 6.13a and Figure 6.1b. This confirms that the inner nodes have nearly no impact on the time performance when an expensive leaf calculation function like Winternitz is used. The average signature generation time and the total initialisation time are listed in Table 6.1.

	Total initialisation	Average signature
$H = 12,\ h = 2$	7.31s	8.37ms
$H = 10,\ h = 2$	1.82s	6.98ms
$H = 8,\ h = 2$	466.93ms	6.00ms
$H = 8,\ h = 4$	465.35ms	3.70ms
$H = 8,\ h = 8$	466.81ms	1.98ms

Table 6.1.: Time needed for initialisation and signature generation.

We will now compare the execution time of our algorithm with the Log and the Fractal one. We examined all three algorithms with the configuration using the least space (according to the analysis) and with 2^{16} leafs (see Fig. 6.14).

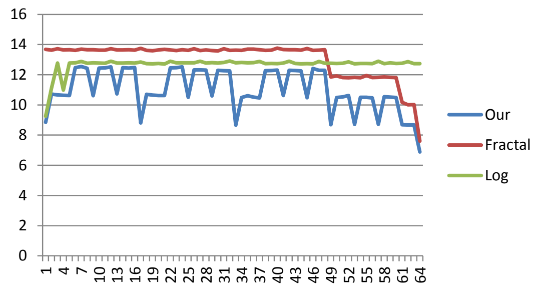


Figure 6.14.: Time needed for signature generation as a function of blocks of 1024 rounds for H=16. Evaluation method Average.

The implementations of the algorithms were designed with a low computational overhead as target. With computational overhead we mean the computation time spent outside the cryptographic primitives (the OTS and hash function). This overhead was measured by replacing the hash and Winternitz functions with a constant function[3]. Figure 6.15 shows the percental overhead in the measurements of the three algorithms when 2^{16} leaves and the configuration resulting in the least used space were used.

[3]A constant function is a function which always returns the same value.

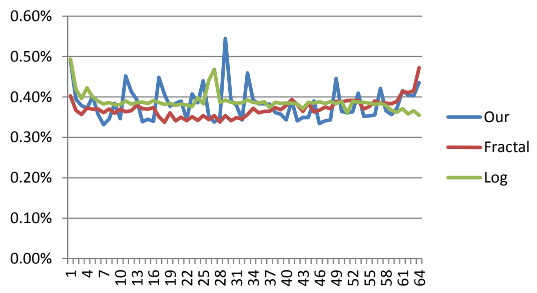


Figure 6.15.: Percental time overhead for signature generation as a function of blocks of 1024 rounds for H=16. Evaluation method Average.

We see in Figure 6.15 that the computational overhead is rather small for all the three implementations, most of the time under 0.5% of the total time spend in the same rounds.

7. Future work

The first bottom level node which is generated by each lower $TreeHash$ could be set in the initialisation phase instead of computing it with the lower $TreeHash$ during the authentication path generation. This would spare some hash and leaf calculations (especially for small values of h).
When a signature is generated, it has to be serialized if the verifier is executed on another computer. It would be beneficial if there were a specialized serialisation for this purpose. At the moment Java serialisation or a custom serialisation format has to be used. If the initialisation phase is executed on another computer as the signature generation (i.e target device has limited computational power), a compact serialisation format for an initialized tree would be useful.
The measurements suggest that the space used by our algorithm is at least $(h-1)$ less than what the analysis predicts. Further investigation would be necessary to find out if this holds for all combinations of H and h, and why this is the case.

8. Conclusions

When a continuous PRNG is used, our algorithm has a space advantage over the Log and Fractal ones and a time advantage over the Log algorithm. Ours, as well as the other two suffer from a long initialisation time for large values of H. This problem was solved by the CMSS [11] and GMSS [12] algorithms. These two algorithms use a stacked series of Merkle trees where the higher trees sign the roots of their child trees and the lowest tree is used for the real cryptographic purpose. Both of them thus rely on a solution of the Merkle tree traversal problem for each layer, for which our algorithm could be used instead of the current ones. It is possible to use different parameters for different layers in the CMSS or GMSS. In addition, the higher trees used in these schemes favour Winternitz as leaf calculation function, which is significantly more expensive than an inner node computation, and thus can profit from the improved $TreeHash$ used in our algorithm. The XMSS [13] is an extension to the Merkle signature scheme (MSS), which allows to use a hash function which is only second-pre–image resistant instead of collision resistant. It is based on the Log algorithm and the usage of a forward secure continuous PRNG. Under these circumstances, our algorithm would be a good replacement for the Log algorithm: it would use less space and provide greater flexibility.

A. Appendix

A.1. Algorithms

Our algorithm uses the following data structures:

1. $Auth_h$, $h = 0, ..., H-1$. An array of nodes that stores the current authentication path.
2. $Subtree_h$, $h = 0, ..., L-1$. An array of subtree structures with the following properties:
   a) bottomLevel: the minimal height for which the subtree stores nodes.
   b) rootLevel: the height of the root of the subtree.
   c) tree: the data structure for the *Exist* and *Desired* tree with the following functions:
      i. get(j, k): get kth node (from left to right) with height j in the subtree
      ii. add(node): store node in the subtree
      iii. remove(j, k): remove kth node (from left to right) with height j in the subtree
   d) stackHigh: the stack for the higher $TreeHash$.
   e) nextIndex: the index of the next leaf needed by the lower $TreeHash$.
   f) bottomLevelNode: the node of lower $TreeHash$ which is stored outside the shared stack [9].
   g) stackLow: the stack for the lower $TreeHash$ (the part of the shared stack currently containing nodes for this *Subtree* [9]).

Our algorithm has the following phases:

1. Initialisation of the subtrees and computation of the public key and first authentication path (Alg. 7).
2. Generation of the $2^H - 1$ remaining authentication paths: repeat 2^H times:
   a) Output current authentication path $Auth_h$, $h = 0, ..., H-1$
   b) Update lower $TreeHash$es (Alg. 8)
   c) Compute next authentication path (Alg. 9)

Algorithm 1 TailHeight: Calculation of the height of the lowest node on a stackLow

```
INPUT : subtree index $i$
OUTPUT : $height$
if Subtree$_i$ has a stackHigh $\land$ $\neg$(Subtree$_i$.bottomLevelNode) then
  if Subtree$_i$.stackLow == empty then
    $height \leftarrow$ Subtree$_i$.bottomLevel
  else
    $height \leftarrow$ Subtree$_i$.stackLow.peek.height
  end if
else
  $height \leftarrow \infty$
end if
return $height$
```

Algorithm 2 SubTreeForLevel: Find subtree which contains level i

```
INPUT : $i$
OUTPUT : $SubTree$
$s \leftarrow \underset{l=0,\cdots,L-1}{max} \{l : \mathrm{Subtree_l.bottomLevel} \le i\}$ return $Subtree_s$
```

Algorithm 3 Process$_0$

```
INPUT : Node, index $j$
OUTPUT : $continue$
if Node.index $\le$ $2^{\mathrm{SubTreeForLevel(Node.height).rootLevel-Node.height}}$ $\land$ Node.index (mod 2) == 1 then
  SubTreeForLevel(Node.height).tree.add(Node)
end if
if Node.index == 1 then
  Auth$_{\mathrm{Node.height}}$ $\leftarrow$ Node
end if
return $continue \leftarrow 1$
```

Algorithm 4 Process$_1$:

```
INPUT : Node; subtree index $j$
OUTPUT : $continue$
if Node.height == $Subtree_j.bottomLevel$ then
  $\text{Subtree}_\text{j}$.bottomLevelNode ← Node
  $continue$ ← 0
else
  $continue$ ← 1
end if
return $continue$
```

Algorithm 5 Process$_2$:

```
INPUT : Node; subtree index $i$
OUTPUT : $continue$
if Node ≠ $dummy$ then
  $continue$ ← 1
  if Node.index (mod 2) == 1 then
    $\text{Subtree}_\text{i}$.tree.add(Node)
    if Node.index/2 $\pmod{2^{\text{Subtree}_\text{i}.\text{rootLevel}-\text{Node.height}-1}}$ == 0 then
      $continue$ ← 0
    end if
  end if
  if $Node.height$ == $\text{Subtree}_\text{i}$.rootLevel − 1 then
    if $\text{Subtree}_\text{i}$.nextIndex + 1 >= $2^H$ then
      $\text{Subtree}_\text{i}$.stackHigh ← $remove$
    end if
  end if
else
  $continue$ ← 0
end if
return $continue$
```

Algorithm 6 Listing: Generic version of *TreeHash* that accepts different types of $\mathrm{Process_i}$. A node has a height and an index, where the index indicates where a node in relation to all nodes with the same height is positioned in the Merkle tree.

```
INPUT : StackOfNodes, Leaf, $\mathrm{Process_i}$, SubtreeIndex
OUTPUT : updated StackOfNodes
Node $\leftarrow$ Leaf
if Node.index $\pmod 2 == 1$ then
  $continue \leftarrow \mathrm{Process_i}(\mathrm{Node}, \mathrm{SubtreeIndex})$
else
  $continue \leftarrow 1$
end if
while $continue \neq 0 \wedge (\mathrm{Node.height} == \mathrm{StackOfNodes.peek.height})$ do
  Node $\leftarrow$ hash(StackOfNodes.pop||Node)
  $continue \leftarrow \mathrm{Process_i}(\mathrm{Node}, \mathrm{SubtreeIndex})$
end while
if $continue \neq 0$ then
  StackOfNodes.push(Node)
end if
```

Algorithm 7 Key generation (PK) and Merkle tree setup.

```
INPUT :
OUTPUT : PK
    {Initialize L-1 subtrees}
for all Subtree_i with $i \in \{0, \cdots, L-1\}$ do
  Subtree_i.tree ← empty
  if i < (L-1) then
    Subtree_i.stackHigh ← empty
    Subtree_i.stackLow ← empty
  end if
  Subtree_i.bottomLevel ← $i \times h$
  Subtree_i.rootLevel ← Subtree_i.bottomLevel + $h$
  Subtree_i.nextIndex ← $2^{Subtree_i.rootLevel} - 1$
end for
    {Initialize Stack, set Leaf level $k = 0$}
$k \leftarrow 1$
$Stack \leftarrow empty$
$Stack.push(LeafCalc(0))$
while Stack.peek.height < $H$ do
  TreeHash(Stack, LeafCalc(k), $\mathrm{Process}_0$, null)
  $k \leftarrow k + 1$
end while
$PK \leftarrow$ Stack.pop
return $PK$
```

Algorithm 8 Distribution of updates to the active lower *TreeHash* instances:

```
INPUT : leaf index $i \in \{1, \cdots, 2^H - 1\}$
$updates \leftarrow$ number of desiredTree in SubTrees
repeat
    {Find TreeHash instance with lowest tail height, on a tie use the one with lowest index}
  $s \leftarrow \min_{i=0,\cdots,L-2}\{l : \forall \text{TailHeight(l)} == \min_{j=0,\cdots,L-2}\{\text{TailHeight(j)}\}\}$
  $\text{Subtree}_\text{s}\text{.nextIndex} \leftarrow \text{Subtree}_\text{s}\text{.nextIndex} + 1$
  if $\text{Subtree}_\text{s}\text{.nextIndex} \pmod{2^{\text{Subtree}_\text{s}\text{.rootLevel}}} \geq 2^{\text{Subtree}_\text{s}\text{.bottomLevel}}$ then
    $TreeHash(\text{Subtree}_\text{s}\text{.stackLow}, LeafCalc(\text{Subtree}_\text{s}\text{.nextIndex}), Process_1, s)$
  else
    if $\text{Subtree}_\text{s}\text{.nextIndex+1} \pmod{2^{\text{Subtree}_\text{s}\text{.rootLevel}}} == 2^{\text{Subtree}_\text{s}\text{.bottomLevel}}$ then
      $\text{Subtree}_\text{s}\text{.bottomLevelNode} \leftarrow dummy$
    end if
  end if
  $updates \leftarrow updates - 1$
until $updates == 0$
```

Algorithm 9 Generation of the next authentication path

```
INPUT : leaf index $i \in \{1, \cdots, 2^H - 1\}$
    {$k$ is 0 if leaf $i$ is a righ node and $k \neq 0$ means the height of the first parent of leaf $i$ that is a right node}
$k \leftarrow \max_{m=0,\cdots,H} \{m : i \mod 2^m == 0\}$
if $k == 0$ then
  $\mathrm{Auth_0} \leftarrow \mathrm{LeafCalc(i-1)}$
else
  $\mathrm{leftNode} \leftarrow \mathrm{Auth_{k-1}}$
  $\mathrm{rightNode} \leftarrow \mathrm{SubTreeForLevel(k-1).tree.get(leftNode.index \oplus 1, k-1)}$
  $\mathrm{Auth_k} \leftarrow \mathrm{hash(leftNode||rightNode)}$
  $\mathrm{SubTreeForLevel(k-1).tree.remove(j, k-1)}$
end if
    {Remove sibling of $\mathrm{Auth_k}$}
if $\mathrm{Auth_k.index/2 \pmod 2} == 1$ then
  $\mathrm{SubTreeForLevel(k).remove(Auth_k.index \oplus 1, k)}$
end if
    {Run through stackHigh in all Subtrees whose $Auth_{bottomLevel}$ changed}
for all $r \in \{0 \cdots L-2\}$ where $\mathrm{Subtree_r.bottomLevel} \leq k$ do
  if SubTree[r] has a desiredTree then
    $TreeHash(\mathrm{Subtree_r.stackHigh, Subtree_r.bottomLevelNode}, Process_2, r)$
    $\mathrm{Subtree_r.bottomLevelNode} \leftarrow remove$
  end if
end for
for all $t \in \{0 \cdots k-1\}$ do
  $\mathrm{Auth_t} \leftarrow \mathrm{SubTreeForLevel(t).tree.get(n, (i/2^t) \oplus 1)}$
end for
return $\mathrm{Auth_j}\ \forall j \in \{0, \cdots, H-1\}$
```

Bibliography

[1] Markus Knecht, Willi Meier, and Carlo U. Nicola. A space–and time–efficient Implementation of the Merkle Tree Traversal Algorithm, 2014. 4

[2] Lov K. Grover (1996) A fast quantum mechanical algorithm for database search. *Proc. of the 28th Ann. Symp. on the Theory of Computing*, 212 – 219. 11

[3] J. Buchmann, E. Dahmen, Michael Szydlo. Hash-based Digital Signatures Schemes *Post-Quantum Cryptography 2009, Springer Verlag*, 35–93. 12

[4] Markus Jakobson, Frank T. Leighton, Silvio Micali, Michael Szydlo (2003). Fractal Merkle Tree representation and Traversal. *Topics in Cryptology - CT-RSA 2003*, 314 – 326. Springer 2, 10, 12, 13, 14, 27

[5] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, Raphael C.-W. Phan (2010). SHA-3 proposal *BLAKE*. `http://www.131002.net/blake/` 29

[6] Michael Szydlo, Merkle tree traversal in log space and time.In C. Cachin and J. Camenisch (Eds.): Eurocrypt 2004, LNCS 3027, pp. 541–554, 2004. 19, 23

[7] Ralph C. Merkle. A certified digital signature. In Gilles Brassard, editor, CRYPTO, volume 435 of LNCS, 218–238. Springer, 1989. 10, 11, 17, 23

[8] Piotr Berman, Marek Karpinski, Yakov Nekrich. Optimal trade-off for Merkle tree traversal. Theoretical Computer Science, volume 372, 26–36. 2007. 12, 19, 20

[9] J. Buchmann, E. Dahmen, M. Schneider. Merkle tree traversal revisited. PQCrypto '08 Proceedings of the 2nd International Workshop on Post-Quantum Cryptography Pages 63 - 78, 2008 2, 10, 12, 13, 15, 17, 19, 23, 24, 27, 28, 30, 48

[10] Elaine Barker and John Kelsey, NIST Special Publication 800-90A: Recommendation for Random Number Generation Using Deterministic Random Bit Generators, 2012. 22, 29

[11] J. Buchmann, C. Coronado, E. Dahmen, M. Dörig, E. Klintsevich CMSS: an improved Merkle signature scheme. *INDOCRYPT 2006, LNCS 4329*, 349 – 363. 47

[12] J. Buchmann, E. Dahmen, E. Klintsevich, K. Okeya, C. Vuillaume . Merkle Signatures with Virtually Unlimited Signature Capacity. ACNS 2007, pp. 31–45 47

[13] J. Buchmann, E. Dahmen, A. Hülsing. XMSS – A Practical Forward Secure Signature Scheme based on Minimal Security Assumptions. PQCrypto 2011, pp. 117–129. 47

[14] Luis Carlos Coronado Garcia. On the security and the efficiency of the Merkle signature scheme. Tatra Mt. Math. Publ., 37, 2005, pp.1–21 29

[15] Thomas Eisenbarthy, Ingo von Maurichz, Xin Yey. Faster Hash-based Signatures with Bounded Leakage. SAC 2013. 27, 28

[16] S. Faust, E. Kiltz, K. Pietrzak, and G. N. Rothblum. Leakage-Resilient Signatures. In Theory of Cryptography, volume Springer LNCS, Volume 5978 of LNCS 5978, pages 343–360. Springer, 2010. 27