A space- and time-efficient Implementation of the Merkle Tree Traversal Algorithm

We have developed an algorithm for the Merkle tree traversal problem which combines the efficient space-time trade-off from the fractal Merkle-trees [4] and the space efficiency from the improved log space-time Merkle-trees traversal [8]. We further programmed a low storage space and a low time overhead version of the algorithm in Java and measured its performance with respect to two different implementations.

Markus Knecht, Carlo U. Nicola | carlo.nicola@fhnw.ch

Figure 1: The nodes marked with a 1 are lying on the path from Leaf₂ to the root. The nodes marked with a 2 are the nodes of the authentication path for Leaf₂; all of them are siblings of a node marked with a 1.
Overview

Two solutions to the Merkle tree traversal problem exist. The first is built on the classical tree traversal algorithm but with many small improvements [8]. The second one is the fractal traversal algorithm [4]. The fractal traversal algorithm trades efficiently space against time by adapting the parameter \( h \) (the height of a subtree, see Fig. 2), however the minimal space it uses for any given \( H \) (if \( h \) is chosen for space optimality) is more than what the improved classical algorithm needs. The improved classical 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 traversal algorithm. But beyond a certain value of \( H \) (depending on the targeted time performance) the fractal traversal algorithm uses less space.

The idea of the fractal tree’s traversal algorithm [4] is to store only a limited set of subtrees within the whole Merkle tree (see Fig. 2). They form a stacked series of \( L \) subtrees \( \{\text{Subtree}\}_i \) each \( \{0, \ldots, L - 1\} \). Each subtree consists of an \( \text{Exist tree} \) \( \{\text{Exist}\}_i \) and a \( \text{Desired tree} \) \( \{\text{Desired}\}_i \), except for \( \{\text{Subtree}\}_i \), which has no \( \text{Desired} \) tree. The \( \text{Exist} \) trees contain the authentication path for the current leaf. When the authentication path for the next leaf is no longer contained in some \( \text{Exist} \) trees, these are replaced by the \( \text{Desired} \) tree of the same subtree. The \( \text{Desired} \) trees are built incrementally after each output of the authentication path algorithm, thus amortizing the operations needed to evaluate the subtree. During the initialization the values of the leftmost \( \text{Exist} \) trees are kept in addition to the root value.

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 [8]. This was done by applying all the improvements discussed in [8] to the fractal tree’s traversal algorithm [4].

The enhancements are as follows:

- Left nodes\(^2\) have the nice property, that when they first appear in an authentication path, their children were already on an earlier authentication path. For right nodes\(^3\) this property does not hold. We can use this fact to calculate left nodes as soon as they are needed in 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.

- In most practical applications, the calculation of a leaf is more expensive than the calculation of an inner node. This can be used to design a variant of the \( \text{TreeHash} \) algorithm, which has a worst case time performance that is nearer to its average case for most practical applications. The modified \( \text{TreeHash} \) (see Algorithm 1) calculates in an update step, one leaf and as many inner nodes as possible before needing a new leaf, instead of processing just one leaf or one inner node as in the normal case.

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

It is easy enough to adapt point one and two for the approach discussed in [4], but point three needs some changes in the way the nodes in a subtree are calculated. All these improvements lead to an algorithm with a worst case storage of \( |(H/h)(2^h - 1) + 2H - 2h| \) hash values. The worst case time bound for the leaf computation per round amounts to \( (L - 1)(2^L - 1)/2^h + 1 \).

---

\(^2\) The authors of [7] proved that the bounds of space \( O(|H|/\log t) \) and time \( O(|H|/\log t) \) for the output of the authentication path of the current leaf are optimal (\( t \) is a freely choosable parameter).

\(^3\) The left child of its parent node.
We further implemented the algorithm in Java with focus on a low space and time overhead and we measured its performance.

**Computation of the Desired Tree**

The main difference between our algorithm and the one described in [4], is in the way we compute the nodes in the Desired tree. In our algorithm we use the improved TreeHash from [8] which needs $2^h$ steps to calculate a node on level $h$ (instead of $2^{h+1} - 1$ as in [4]). We call the TreeHash instances which calculate the nodes on the bottom level of a Desired tree lower TreeHash. Another improved TreeHash instance, called higher TreeHash, calculates all non-bottom level nodes, by using the bottom level nodes as leaves in $2^h$ updates. In our case we do an update on the higher TreeHash all $2^{\text{BottomLevel}}$ rounds. The updates on the lower TreeHash are distributed with the algorithm described in [5]. The bottom level nodes which are produced by the lower TreeHash are ready when the higher TreeHash needs it as a leaf of the Desired tree (which happens every $2^{\text{BottomLevel}}$ rounds) [9]. In Figure 3 we show how the different nodes of the Desired and Exist trees are managed.

**The space and time gains of the new algorithm**

It can be shown that a subtree\(^4\) needs $2^h - 1$ hash values storage space when the authentication path is taken into account [9]. All subtrees together with the authentication path thus need $L(2^h - 1)$ for the subtrees (where $L = H/h$) and $H$ hash values for the authentication path. The lower TreeHashes need to store at most one node per level up to the bottom level of the second highest subtree which sums up to $H - 2h$ hash values [9]. Taking all together our algorithm needs $L(2^h - 1) + 2H - 2h$ hash values storage space.

For the time analysis we look at the number of leaves’ calculations per round, which often are way more expensive than the hash calculation. 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 both TreeHash need $(L - 1)$ leaves’ calculations per round. We need an additional leaf calculation every two rounds to compute the left nodes as shown in [8]. If we sum this up, we need $L$ leaves’ calculations per round in the worst case. In the average case we need less, since the first $2^{\text{BottomLevel}}$ updates in each Desired tree do not compute any leaves. This because it would produce a leaf node which never is needed for the computation of a right node in the Desired tree. There are $2^h$ nodes on the bottom level of a subtree from which one node is not computed. In the average case the computation of the Desired trees is reduced by the factor $(2^h - 1)/2^h$. This leads to a total of $\frac{H}{2} + (L - 1)(2^h - 1)/2^h$ leaves’ computations per round on the average (the $\frac{H}{2}$ term enter the equation because the left node computation needs a leaf only each two rounds). This average case analysis does not take into account that later some subtrees have no Desired trees and thus will no longer need an update on their TreeHash. Table 1 summarizes our results and Table 2 does the same for the log space- and fractal-algorithm.

When $h = 2$ our algorithm can compete with the log space-time [8] one, in the case of optimized storage space. Since our algorithm has the same improved space-time trade-off as the fractal one [4], it can handle more efficiently space vs. time

---

\(^4\) The right child of its parent node.

\(^5\) Lowest level for which a Desired tree stores nodes.

\(^6\) We mean with subtree the set of Existi, Desiredi and the higher TreeHashi currently processed.
trade-offs in all other cases, where an optimized storage space is not required. When we choose the same space-time trade-off parameter as in the fractal algorithm [4] (column \( h = \log(H) \) in Table 1), our algorithm needs less storage space.

**Results**

We compared the performance of our algorithm with both the algorithm from [8] (referred as Log from now on) and the algorithm from [4] (referred as Fractal from now on). We chose as performance parameters the leaves computations and the number of stored hash values. This choice is reasonable because the former is the most expensive operation if the Merkle tree is used for signing, and the latter is a good indicator of the storage space needed. Operations like computing a non-leaf node or generating a pseudo random value, have nearly no impact on the runtime in the range of \( H \) values of practical interest. A leaf computation is exactly the same in each of the three algorithms and therefore only dependent on the underlying hardware for its performance.

To be able to present cogently the results, each data point represents an aggregation over eight rounds. In the case of storage measurements one point represents the maximal amount of stored hash values at any time during these eight rounds. In the case of the leaf computation one point represents the number of leaves’ computations done during the eight rounds.

For the log- and fractal-algorithms the parameters are chosen so that the algorithms use minimal storage space. The log algorithm is a good choice if a small memory footprint is needed. The fractal algorithm on the other hand, is a good choice if a better space time trade-off is needed. Our algorithm can be used in both cases but with different parameter settings. We have measured it once with the parameter chosen for a similar space time trade-off as the fractal tree (same number of levels) and once with a parameter setting for minimal storage space requirements. The results of these measurements for trees with 512 leaves \((H = 9)\) are shown in the Figure 4 for a similar space-time trade-off as the fractal tree and in Figure 5 for minimal storage space.

We see that in a setting where a good space time trade-off is needed, our algorithm uses less space than the Fractal algorithm and this with just a small constant amount of additional leaves calculations. It uses more space as the log algorithm but it can reduce the number of leaves’ calculations. For the case where a small memory footprint is needed, our algorithm uses most of the time less memory as the log algorithm, but computes more leaves.

The plots show in addition a weak point of our algorithm compared with the log algorithm: the leaves’ calculations have larger deviations. The fractal algorithm has for similar parameter even greater deviations, but they are not visible in the Figure 5, because they cancel each other out over the eight rounds. If we measure the first 128 rounds with no aggregation we see, that the deviations of our algorithm decrease markedly (see Fig. 6).

**References**


Figure 4: Left: Number of stored hash values as function of rounds for similar space-time trade-off. Right: Number of calculated leaves as function of rounds for similar space-time trade-off.

Figure 5: Left: Number of stored hash values as function of rounds for minimal space. Right: Number of calculated leaves as function of rounds for minimal space.

Figure 6: Number of calculated leaves as function of rounds for similar space-time trade-off (first 128 rounds in detail)