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

Merkle's binary hash-trees are one of the most interesting cryptographical building blocks currently available, because their security is independent from any number theoretic conjectures [6]. The security is based solely on two well defined mathematical 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-bit-random records in a data base (an analogous problem to hash collision) achieves only a speed up of $O(\sqrt{n})$ to the classical one $O(n)$, where n is the number of records [1].

A Merkle tree is a complete binary tree with an n-bit value associated with each node. Each internal node value is the result of a hash of the node values of its children (n is the number of bits returned by the hash function). 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 $l = 0,\ldots,H-1$, 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).

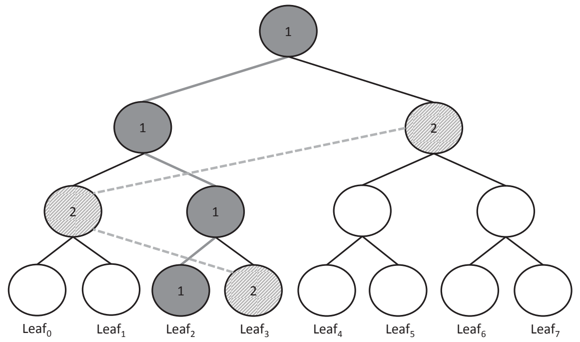

Figure 1: The nodes marked with a 1 are lying on the path from $Leaf_2$ to the root. The nodes marked with a 2 are the nodes of the authentication path for $Leaf_2$; all of them are siblings of a node marked with a 1.

The Merkle tree traversal problem answers the question of how to calculate efficiently[1] the authentication path for all leafs one after another starting with the first $Leaf_0$ up to the last $Leaf_{2^H-1}$, if there is only a limited amount of storage available (i.e. 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 leaves evaluations and of $2^H - 1$ hash computations. In this process the value of the public key is kept, the rest is discarded. 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 onetime token to access resources or as building block 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).

The nodes in a Merkle tree are calculated with an algorithm called *TreeHash*. The 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 pushes the hash value of their concatenation

1 Two children of the same node of a binary tree

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 pushes it onto the stack, this node has a height of zero. To calculate all nodes in a tree of height H, the *TreeHash* needs $2^{H+1} - 1$ steps, where one step is either a leaf or a hash calculation.

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 $\{Subtree_i\}_{i=0..L-1}$. Each subtree consists of an *Exist* tree $\{Exist_i\}$ and a *Desired* tree $\{Desired_i\}$, except for $Subtree_L$, 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. The *Desired* trees are built incrementally after

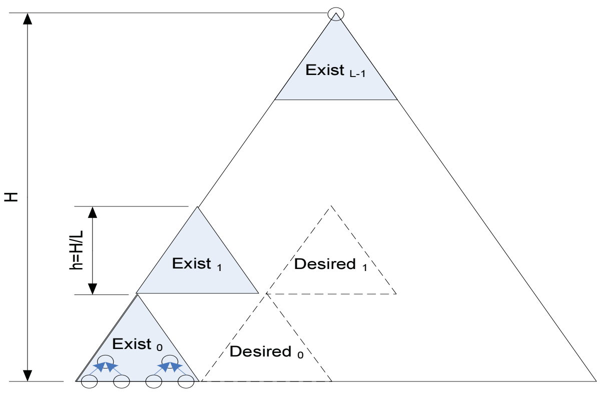

Figure 2: Fractal Merkle tree structure and notation (Figure courtesy of [2]). A hash tree T of height H is divided into L levels, each of height h. The leaves of the hash tree are indexed {0, 1, …, $2^H - 1$} from left to right. The height of a node is defined as the height of the maximal subtree for which it is the root and ranges from 0 (for the leaves) to H (for the root). An h-subtree is "at level i" when the altitude of its root is h(i + 1) for some i ∈{0, 1, …, L – 1}

each output of the authentication path algorithm, thus amortizing the operations needed to evaluate the subtree. During the initialization the values of the leftmost $Exist_i$ 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 *TreeHash* algorithm, which has a worst case time performance that is nearer to its average case for most practical applications. The modified *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 *TreeHash* instance per subtree exists for calculating the nodes of the *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 *TreeHash* instances are empty or already finished. This reduces the space needed by the stacks of the *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 $(L - 1)\ (2^h - 1)/2^h + 1$.

2 The authors of [7] proved that the bounds of space O(t·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.

```
INPUT : stack<node>; leaf; process_i
OUTPUT: updated stack

node := leaf
if (node.index (mod 2) = 1) then
  continue := process_i(node)
else
  continue := 1
end if
while (continue != 0) &
 (node.height = stack.top.height) do
  node := hash(stack.pop || node)
  continue := process_i(node)
end while
if (continue != 0) then
  stack.push(node)
end if
```

Listing 1: Generic version of TreeHash that accepts different types of $Process_i$. $Process_i$ manages nodes in dependence of the traversal algorithm used and returns true if more nodes are needed and false if the current iteration is finished. || is the concatenation operator.

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[4] of a *Desired* tree lower *TreeHash*. Another improved *TreeHash* instance, called higher *TreeHash*, calculates all non-bottom[5] 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 $2^{BottomLevel}$ 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 $2^{BottomLevel}$ 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[6] 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.

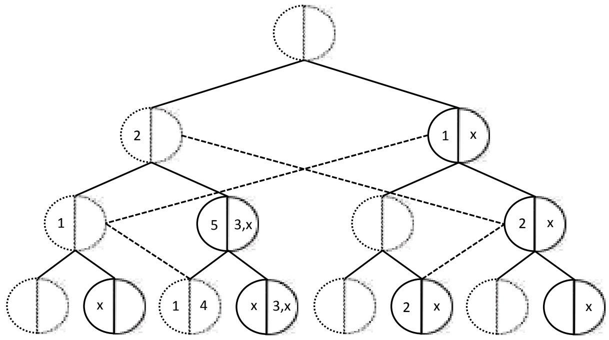

Figure 3: The left half of each circle represents the Exist tree and the right half the Desired tree. The dotted nodes are left nodes or the root and thus are not really stored in the subtree, but they may be stored in the authentication path or on the higher TreeHash. Nodes marked with an x are already discarded (in case of Exist tree) or not yet computed (in case of Desired tree). The nodes marked with a 1 are lying on the current authentication path; nodes marked with a 2 are lying on the upcoming authentication path. Nodes marked with a 3 are the nodes which are computed next by the higher TreeHash. The node marked with a 4 is a left node on the higher TreeHash. The node marked with a 5 is needed for calculating a left node in the upcoming authentication path.

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 $2^{bottomLevel}$ updates in each *Desired* tree do not compute any leaves. This because it would produce a left 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 ½ + $(L - 1)(2^h - 1)/2^h$ leaves' computations per round on the average (the ½ 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.

	h = 1, L = H	h = 2, L = H/2	h = log(H), L = H/log(H)
Worst case space bound	3 H – 2	3.5 H – 4	H(H – 1)/log(H) + 2H – 2 log(H)
Average case time bound	H/2	(3H – 2)/8	(H – 1)/log(H) + ½
Worst case time bound	H	H/2	H/log(H) – 1

Table 1: Space-time tradeoff of our Merkle tree traversal algorithm as function of h (number of levels) and H (height of the tree).

	Log space-time [8]	Fractal [4]
Worst case space bound	3.5 H – 4	2½ H^2/log(H)
Average case time bound	H/2 – ½	H/log(H) – 1
Worst case time bound	H/2	2 H/log(H) – 2

Table 2: Space-time tradeoff of log space-time algorithm [8] and fractal algorithm [4] optimized for storage space.

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

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

[2] Dalit Naor, Amir Shenhav, Avishai Wool. One-time signatures revisited: Practical fast signatures using fractal Merkle tree traversal. IEEE 24th Convention of Electrical and Electronics Engineers in Israel, 255 – 259, 2006.

[3] J. Buchman, E. Dahmen, Michael Szydlo. Hash-based Digital Signatures Schemes. Post-Quantum Cryptography, 35 – 93, 2009.

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

[5] 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.

[6] Ralph C. Merkle. A certified digital signature. In Gilles Brassard, editor, CRYPTO, volume 435 of LNCS, 218–238, 1989.

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

[8] J. Buchman, E. Dahmen, M. Schneider. Merkle tree traversal revisited. PQCrypto '08 Proceedings of the 2nd International Workshop on Post-Quantum Cryptography Pages 63 – 78, 2008.

[9] Markus Knecht. A space- and time-efficient Implementation of the Merkle Tree Traversal Algorithm. Master Thesis MSE FHNW, 2014.

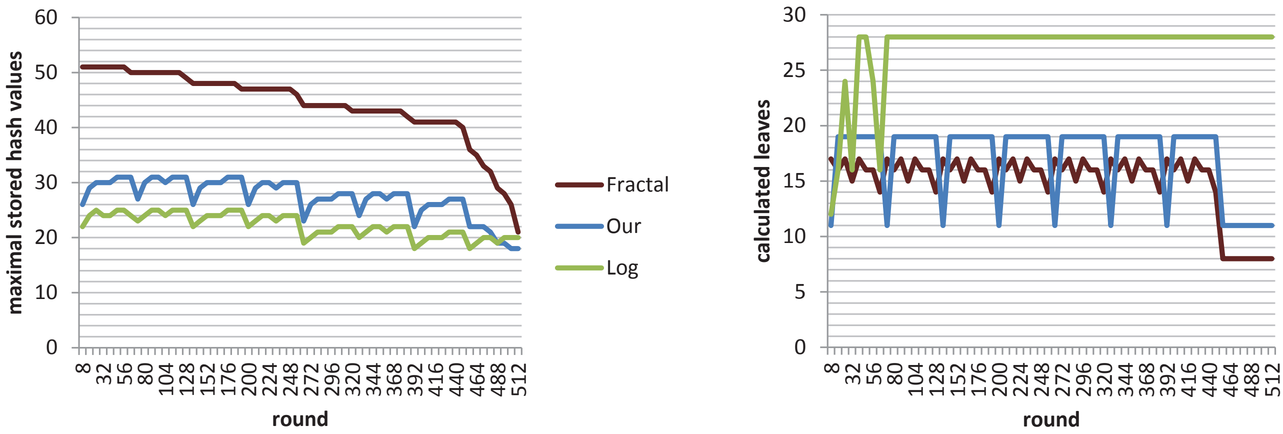

Figure 4: Left: Number of stored hash values as function of rounds for similar space-time trade-off.

Right: Number of calculated leaves as function of rounds for similar space-time trade-off.

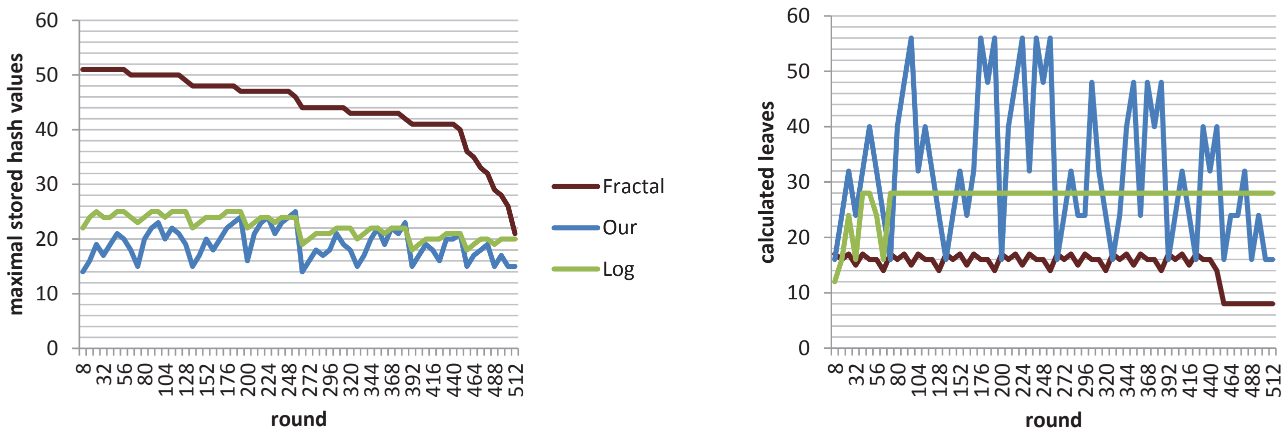

Figure 5: Left: Number of stored hash values as function of rounds for minimal space.

Right: Number of calculated leaves as function of rounds for minimal space

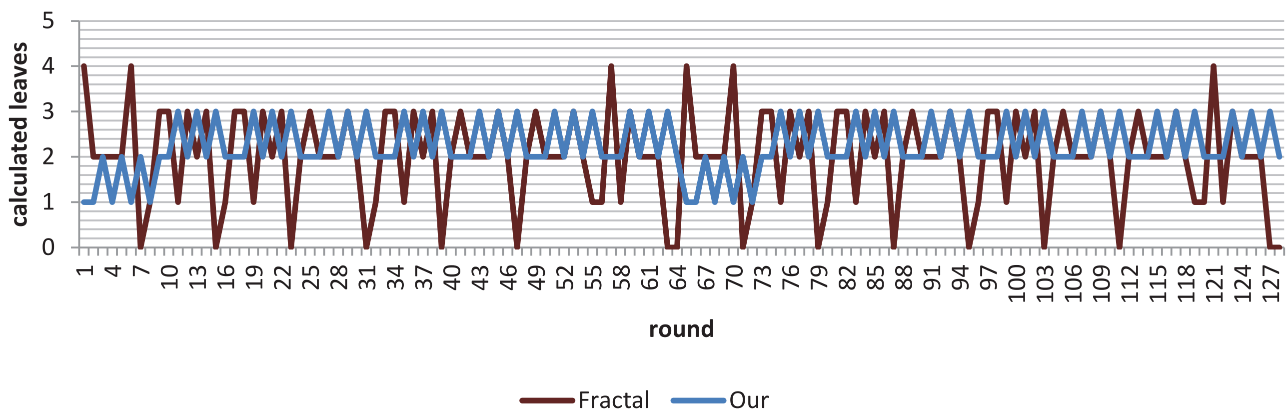

Figure 6: Number of calculated leaves as function of rounds for similar space-time trade-off (first 128 rounds in detail)