Merkle tree traversal revisited

Johannes Buchmann, Erik Dahmen, and Michael Schneider

Technische Universität Darmstadt
Department of Computer Science
Hochschulstraße 10, 64289 Darmstadt, Germany
{buchmann,dahmen,mischnei}@cdc.informatik.tu-darmstadt.de

Abstract. We propose a new algorithm for computing authentication paths in the Merkle signature scheme. Compared to the best algorithm for this task, our algorithm reduces the worst case running time considerably.

Keywords: *Authentication path computation, digital signatures, Merkle signatures, Merkle tree traversal, post-quantum cryptography.*

1 Introduction

Digital signatures are extremely important for the security of computer networks such as the Internet. For example, digital signatures are widely used to ensure authenticity and integrity of updates for operating systems and other software applications. Currently used signature schemes like RSA and ECDSA base their security on the hardness of factoring and computing discrete logarithms. In the past 20 years, there has been significant progress in solving these problems which is why the key sizes for RSA and ECDSA are constantly increased [9]. The security of RSA and ECDSA is also threatened by large quantum computers that, if built, are able to solve the underlying problems in linear time and thus are able to completely break RSA and ECDSA [12]. The research on alternative signature schemes, so-called post quantum signature schemes, is therefore of extreme importance.

One of the most interesting post-quantum signature schemes is the Merkle signature scheme (MSS)[10]. Its security can be reduced to the collision resistance of the used hash function [4]. The best known quantum algorithm to find collisions of hash functions achieves only a square root speed-up compared to the birthday attack [6]. Therefore, the security of MSS is only marginally affected if large quantum computers are built. If a specific hash function is found to be insecure, MSS is easily saved by using a new, secure hash function. This makes MSS an intriguing candidate for a post-quantum signature scheme. It is therefore important to implement the Merkle signature scheme as efficiently as possible. In recent years, many improvements for MSS were proposed [2, 3, 5, 11]. With those improvements, the performance of MSS is now competitive. However, signing with MSS is in most cases still slower than signing with ECDSA. This paper proposes an MSS improvement that reduces the signing time.

The time required for generating a Merkle signature is dominated by the time for computing the authentication path, that later allows the verifier to deduce the validity of the one-time verification key from the validity of the MSS public key. Current algorithms [1, 10, 13, 14, 7] for computing authentication paths have fairly unbalanced running times. The best case runtime of those algorithms is significantly shorter than the worst case runtime. So the computation of some authentication paths is very slow while other authentication paths can be computed very quickly.

Here we propose an authentication path algorithm which is significantly faster in the worst case than the best algorithm known so far. This is Szydlo's algorithm from [13] which provides the optimal time-memory trade-off. In fact, the worst case runtime of our algorithm is very close to its average case runtime which, in turn, equals the average case runtime of the best known algorithm proposed in [13]. The idea of our algorithm is to balance the number of leaves

that are computed in each authentication path computation, since leaves are by far the most expensive nodes in the Merkle tree. All other known approaches balance the number of nodes. This does not balance the running time since computing an inner node only requires one hash function evaluation, while computing a leaf takes several hundred hash function evaluations[1]. This problem is pointed out in [1, 11] but no solution has been provided so far. Our algorithm balances the number of leaves that are computed in each round. Inner nodes are computed as required and since their cost is negligible compared to leaves, the worst case time required by our algorithm is extremely close to the average case time. To be more precise, for each authentication path our algorithm computes $H/2$ leaves and $3/2(H-3)+1$ inner nodes in the worst case and $(H-1)/2$ leaves and $(H-3)/2$ inner nodes on average, where H is the height of the Merkle tree. Our algorithm needs memory to store $3.5H-4$ nodes.

Previous work. There are two different approaches to compute authentication paths. In [10] Merkle proposes to compute each authentication node separately. This idea is adopted by Szydlo [14], where he implements a better scheduling of the node calculations and achieves the optimal trade-off, that is $O(H)$ time and $O(H)$ space. In [13], Szydlo further improves the constants. For each authentication path his algorithm computes H nodes of the Merkle tree and requires storage for $3H-2$ nodes.

The second approach is called fractal Merkle tree traversal [7]. This approach splits the Merkle tree into smaller subtrees and stores a stacked series of subtrees that contain authentication paths for several succeeding leaves. Varying the height h of the subtrees allows a trade-off between time and space needed for the tree traversal. Using the low space solution ($h = \log H$) requires $O(H/\log H)$ time and $O(H^2/\log H)$ space. In [1], the authors improve the constants of this algorithm and prove the optimality of the fractal time-memory trade-off.

Organisation. Section 2 describes a simplified version of our algorithm for Merkle trees of even height. The general algorithm is presented in Appendix A. Section 3 compares the new algorithm with that of Szydlo [13]. Section 4 states our conclusion. Appendix B considers the computation of leaves using a PRNG.

2 Authentication path computation

In this section we describe our new algorithm to compute authentication paths. It is based on Szydlo's algorithm from [13]. We describe the algorithm in detail, prove its correctness, and estimate the worst case and average case runtime as well as the required space.

Definitions and notations. In the following, $H \geq 2$ denotes the height of the Merkle tree. The index of the current leaf is denoted by $\varphi \in \{0, \dots, 2^H - 1\}$. The nodes in the Merkle tree are denoted by $y_h[j]$, where $h = 0, \dots, H$ denotes the height of the node in the tree (leaves have height 0 and the root has height H) and $j = 0, \dots, 2^{H-h} - 1$ denotes the position of this node in the tree counting from left to right. Further, let $f : \{0,1\}^* \rightarrow \{0,1\}^n$ be a cryptographic hash function. Using this notation, inner nodes of a Merkle tree are computed as

$$y_h[j] = f\big(y_{h-1}[2j] \parallel y_{h-1}[2j+1]\big), \tag{1}$$

for $h = 1, \dots, H$ and $j = 0, \dots, 2^{H-h} - 1$.

[1] This is because leaves are essentially one-time verification keys and thus the cost for computing a leaf is determined by the key pair generation cost of the respective one-time signature scheme.

Next, we define the value τ. In round $\varphi \in \{0, \ldots, 2^H - 1\}$, we define τ as the height of the first parent of leaf φ which is a left node. If leaf φ is a left node itself, then $\tau = 0$. Otherwise τ is given as $\tau = \max\{h : 2^h | (\varphi + 1)\}$. Figure 1 shows an example.

Fig. 1. The height of the first parent of leaf φ that is a left node is $\tau = 2$. The dashed nodes denote the authentication path for leaf φ. The arrows indicate the path from leaf φ to the root.

The value τ tells us on which heights the authentication path for leaf $\varphi + 1$ requires new nodes. It requires new right nodes on heights $h = 0, \ldots, \tau - 1$ and a single new left node on height τ.

Computing inner nodes. A basic tool to compute inner nodes of a Merkle tree is the treehash algorithm shown in Algorithm 1. This algorithm uses a stack STACK with the usual push and pop operations and the LEAFCALC(φ) routine which computes the φth leaf[2]. To compute a node on height h, Algorithm 1 must be executed 2^h times and requires the leaf indices to be input successively from left to right, i.e. $\varphi = 0, \ldots, 2^h - 1$. In total, the computation of 2^h leaves and $2^h - 1$ hashes (inner nodes) is required. After the last call the stack contains one node, the desired inner node on height h. The treehash algorithm stores at most h nodes, so-called tail nodes, on the stack.

Algorithm 1 Treehash

Input: Leaf index φ, stack STACK
Output: Updated stack STACK

1. LEAF $\leftarrow$ LEAFCALC(φ)
2. **while** LEAF has the same height in the tree as the top node on STACK **do**
   (a) TOP $\leftarrow$ STACK.pop()
   (b) LEAF $\leftarrow f$(TOP $\|$ LEAF)
3. STACK.push(LEAF)
4. **Return** STACK

2.1 Our authentication path algorithm

We now describe our Merkle tree traversal algorithm in detail. On input a leaf index $\varphi \in \{0, \ldots, 2^H - 2\}$ and the previous algorithm state our algorithm computes and outputs the authentication path for the next leaf $\varphi + 1$.

Like Szydlo's algorithm [13], we deploy two different strategies to compute authentication nodes, depending on whether the node is a left child (left authentication node, left node) or a right one. The difference to Szydlo's algorithm is, that we use a different approach for scheduling the computation of right nodes.

Data structures. Our algorithm uses the following data structures:

- AUTH$_h$, $h = 0, \ldots, H - 1$. An array of nodes that stores the current authentication path.
- RETAIN. The single right authentication node on height $H - 2$.

[2] That is, it computes the φth one-time key pair and obtains the leaf from the one-time verification key.

- STACK. A stack of nodes with the usual push and pop operations.
- $\text{TREEHASH}_h, h = 0, \ldots, H-3$. These are instances of the treehash algorithm. All these treehash instances share the stack STACK. Further, each instance has the following entries and methods.
  - TREEHASH_h.node. This entry stores a single tail node. This is the first node Algorithm 1 pushes on the stack. The remaining tail nodes are pushed on the stack STACK.
  - TREEHASH_h.initialize(φ). This method initializes this instance with the index φ of the leaf to begin with.
  - TREEHASH_h.update(). This method executes Algorithm 1 once, meaning that it computes the next leaf (Line 1) and performs the necessary hash function evaluations to compute this leaf's parents (Line 2b), if tail nodes are stored on the stack.
  - TREEHASH_h.height. This entry stores the height of the lowest tail node stored by this treehash instance, either on the stack STACK or in the entry TREEHASH_h.node. If TREEHASH_h does not store any tail nodes $\text{TREEHASH}_h.\text{height} = h$ holds. If TREEHASH_h is finished or not initialized $\text{TREEHASH}_h.\text{height} = \infty$ holds.
- $\text{KEEP}_h, h = 0, \ldots, H-2$. An array of nodes that stores certain nodes for the efficient computation of left authentication nodes.

Initialization. The initialization of our algorithm is done during the MSS key pair generation. We store the authentication path for the first leaf ($\varphi = 0$): $\text{AUTH}_h = y_h[1], h = 0, \ldots, H-1$. We also store the next right authentication node in the treehash instances: $\text{TREEHASH}_h.\text{node} = y_h[3]$, for $h = 0, \ldots, H-3$. Finally we store the single next right authentication node on height $H-2$: $\text{RETAIN} = y_{H-2}[3]$. Figure 2 shows which nodes are stored during the initialization

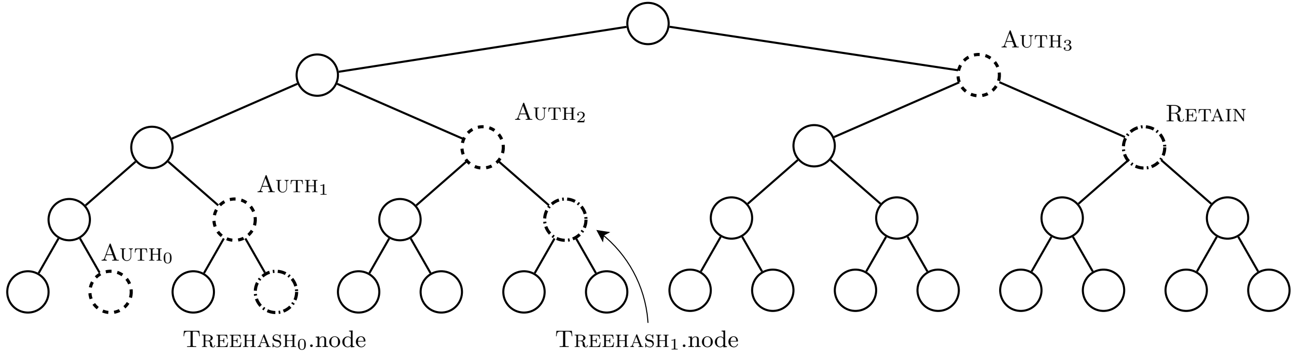

Fig. 2. Initialization of our algorithm. Dashed nodes denote the authentication path for leaf ($\varphi = 0$). Dash-dotted nodes denote the nodes stored in the treehash instances and the single node RETAIN.

Update and output phase. In the following we describe the update and output phase of our algorithm. Algorithm 2 shows a pseudo-code description. Input is the index of the current leaf $\varphi \in \{0, \ldots, 2^H - 2\}$, the height of the Merkle tree $H \geq 2$, where H must be even, and the algorithm state AUTH, KEEP, RETAIN, and TREEHASH prepared in previous rounds or during the initialization. Output is the authentication path for the next leaf $\varphi + 1$.
Computing left authentication nodes. We review the computation of left nodes due to [13]. The basic idea is to store certain right nodes in an array KEEP_h, $h = 0, \ldots, H-2$ and use them later to compute left authentication nodes using only one evaluation of the hash function.

If in round $\varphi \in \{0, \ldots, 2^H - 2\}$, the parent of leaf φ on height $\tau + 1$ is a left node (this can be verified by checking if $\lfloor \varphi / 2^{\tau+1} \rfloor$ is even), then AUTH_τ is a right node and we store it in KEEP_τ (Line 2). In round $\varphi' = \varphi + 2^\tau$ the authentication path for leaf $\varphi' + 1$ requires a new left authentication node on height $\tau' = \tau + 1$. The left child of this authentication node is the authentication node on height $\tau' - 1$ of leaf φ'. The right child of this node was stored in $\text{KEEP}_{\tau'-1}$ in round φ. The new left authentication node on height τ' is then computed as $\text{AUTH}_{\tau'} = f\big(\text{AUTH}_{\tau'-1} \parallel \text{KEEP}_{\tau'-1}\big)$ (Line 4a). For those rounds φ where $\tau = 0$ holds, the

single new left node required for the authentication path of leaf $\varphi+1$ is the current leaf φ. We compute it using the algorithm LEAFCALC, i.e. we set $\text{AUTH}_0 = \text{LEAFCALC}(\varphi)$ (Line 3).
Computing right authentication nodes. Unlike left authentication nodes, right authentication nodes must be computed from scratch, i.e. starting from the leaves. This is because none of their child nodes were used in previous authentication paths. We use one TREEHASH instance for each height where right authentication nodes must be computed, i.e. for heights $h = 0, \ldots, H-3$.

In round $\varphi \in \{0, \ldots, 2^H - 2\}$, the authentication path for leaf $\varphi + 1$ requires new right authentication nodes on heights $h = 0, \ldots, \tau - 1$. Our algorithm is constructed such that for $h \leq H - 3$ these nodes are already computed and stored in TREEHASH_h.node. If a new authentication node is required on height $h = H - 2$ we copy it from the node RETAIN. Note that there is only one new right node required on this height during the whole runtime of Algorithm 2. The authentication path for leaf $\varphi + 1$ is obtained by copying the nodes from TREEHASH_h.node and RETAIN to AUTH_h for $h = 0, \ldots, \tau - 1$ (Line 4b).

After copying the right nodes, all treehash instances on height $h = 0, \ldots, \tau-1$ are initialized for the computation of the next right authentication node. The index of the leaf to begin with is $\varphi + 1 + 3 \cdot 2^h$. If $\varphi + 1 + 3 \cdot 2^h \geq 2^H$ holds, then no new right node will be required on this height and the treehash instance is not initialized anymore (Line 4c).

The last step of the algorithm is to update the treehash instances using the TREEHASH_h.update() method (Line 5). We perform $H/2 - 1$ updates in each round. We use the strategy from [13] to decide which of the $H - 2$ treehash instances receives an update. The treehash instance that receives an update is the instance where TREEHASH_h.height contains the smallest value. If there is more than one such instance, we choose the one with the lowest index (Line 5a).

2.2 Correctness

In this section we show the correctness of Algorithm 2. First we show that the budget of $H/2-1$ updates per round is sufficient for the treehash instances to compute the required authentication nodes on time. Then we will show that it is possible for all treehash instances to share a single stack.

Nodes are computed on time. If TREEHASH_h is initialized in round φ, the authentication node on height h computed by this instance is required in round $\varphi + 2^{h+1}$. During these 2^{h+1} rounds there are $(H-2)2^h$ updates available and TREEHASH_h requires 2^h updates to complete.

On height $i = 0, \ldots, h-1$, a new instance is initialized every 2^{i+1} rounds, each requiring 2^i updates to complete. During the 2^{h+1} available rounds, $2^{h+1}/2^{i+1}$ treehash instances are initialized on heights $i = 0, \ldots, h-1$. All these lower instances are completed before TREEHASH_h (Line 5a).

In addition, active treehash instances on heights $i = h+1, \ldots, H-3$ might receive updates until their lowest tail node has height h. Once they have a tail node on height h they don't receive further updates while the instance on height h is active (Line 5a). Computing a tail node on height h requires at most 2^h updates.

The number of updates required to complete TREEHASH_h on time is at most

$$\sum_{i=0}^{h-1} \frac{2^{h+1}}{2^{i+1}} \cdot 2^i + 2^h + \sum_{i=h+1}^{H-3} 2^h = (H-2)2^h \tag{2}$$

This shows that the budget of $H/2 - 1$ leaves per round suffices. For $h = H - 3$ this bound is tight.

Sharing a single stack works. To show that it is possible for all treehash instances to share a single stack, we have to show that if TREEHASH_h receives an update and has previously stored tail nodes on the stack, all these tail nodes are on top of the stack.

Algorithm 2 Authentication path computation, simplified version

Input: $\varphi \in \{0, \ldots, 2^H - 2\}$, $H \geq 2$ even, and the algorithm state.
Output: Authentication path for leaf $\varphi + 1$

1. Let $\tau = 0$ if leaf φ is a left node or let τ be the height of the first parent of leaf φ which is a left node:
   $\tau \leftarrow \max\{h : 2^h | (\varphi + 1)\}$
2. If the parent of leaf φ on height $\tau + 1$ is a left node, store the current authentication node on height τ in KEEP_τ:
   if $\lfloor \varphi / 2^{\tau+1} \rfloor$ is even **and** $\tau < H - 1$ **then** $\text{KEEP}_\tau \leftarrow \text{AUTH}_\tau$
3. If leaf φ is a left node, it is required for the authentication path of leaf $\varphi + 1$:
   if $\tau = 0$ **then** $\text{AUTH}_0 \leftarrow \text{LEAFCALC}(\varphi)$
4. Otherwise, if leaf φ is a right node, the authentication path for leaf $\varphi + 1$ changes on heights $0, \ldots, \tau$:
   if $\tau > 0$ **then**
   (a) The authentication path for leaf $\varphi + 1$ requires a new left node on height τ. It is computed using the current authentication node on height $\tau - 1$ and the node on height $\tau - 1$ previously stored in $\text{KEEP}_{\tau-1}$. The node stored in $\text{KEEP}_{\tau-1}$ can then be removed:
   $\text{AUTH}_\tau \leftarrow f(\text{AUTH}_{\tau-1} || \text{KEEP}_{\tau-1})$, remove $\text{KEEP}_{\tau-1}$
   (b) The authentication path for leaf $\varphi + 1$ requires new right nodes on heights $h = 0, \ldots, \tau - 1$. For $h \leq H - 3$ these nodes are stored in TREEHASH_h and for $h = H - 2$ in RETAIN:
   for $h = 0$ **to** $\tau - 1$ **do**
      if $h \leq H - 3$ **then** $\text{AUTH}_h \leftarrow \text{TREEHASH}_h.\text{node}$
      if $h = H - 2$ **then** $\text{AUTH}_h \leftarrow \text{RETAIN}$
   (c) For heights $0, \ldots, \tau - 1$ the treehash instances must be initialized anew. The treehash instance on height h is initialized with the start index $\varphi + 1 + 3 \cdot 2^h$ if this index is smaller than 2^H:
   for $h = 0$ **to** $\tau - 1$ **do**
      if $\varphi + 1 + 3 \cdot 2^h < 2^H$ **then** $\text{TREEHASH}_h.\text{initialize}(\varphi + 1 + 3 \cdot 2^h)$
5. Next we spend the budget of $H/2 - 1$ updates on the treehash instances to prepare upcoming authentication nodes:
   repeat $H/2 - 1$ **times**
   (a) We consider only stacks which are initialized and not finished. Let s be the index of the treehash instance whose lowest tail node has the lowest height. In case there is more than one such instance we choose the instance with the lowest index:
   $s \leftarrow \min \left\{ h : \text{TREEHASH}_h.\text{height} = \min_{j=0,\ldots,H-3} \{\text{TREEHASH}_j.\text{height}\} \right\}$
   (b) The treehash instance with index s receives one update:
   $\text{TREEHASH}_s.\text{update}()$
6. The last step is to output the authentication path for leaf $\varphi + 1$:
   return $\text{AUTH}_0, \ldots, \text{AUTH}_{H-1}$.

When $\textsc{Treehash}_h$ receives its first update, the height of the lowest tail node of $\textsc{Treehash}_i$, $i \in \{h+1, \dots, H-3\}$ is at least h. Otherwise, one of the instances on height i would receive an update (Line 5a). This means that $\textsc{Treehash}_h$ is completed before $\textsc{Treehash}_i$ receives another update and thus tail nodes of higher treehash instances do not interfere with tail nodes of $\textsc{Treehash}_h$.

While $\textsc{Treehash}_h$ is active and stores tail nodes on the stack, it is possible that treehash instances on lower heights $i \in \{0, \dots, h-1\}$ receive updates and store nodes on the stack. If $\textsc{Treehash}_i$ receives an update, the height of the lowest tail node of $\textsc{Treehash}_h$ has height $\geq i$. This implies that $\textsc{Treehash}_i$ is completed before $\textsc{Treehash}_h$ receives another update and therefore doesn't store tail nodes on the stack anymore.

2.3 Time and space bounds

This section considers the time and space requirements of Algorithm 2. We will show that

i) On average, our algorithm computes $(H-1)/2$ leaves and $(H-3)/2$ inner nodes.
ii) The number of tail nodes stored on the stack is bounded by $H-4$.
iii) The number of inner nodes computed by all treehash instances per round is bounded by $3/2(H-3)$.
iv) The number of nodes stored in $\textsc{Keep}$ is bounded by $H/2+1$.

For the space, we have to add the H nodes stored in $\textsc{Auth}$, the $H-2$ nodes $\textsc{Treehash}$.node and the single node stored in $\textsc{Retain}$. For the worst case time, we have to add the $H/2-1$ leaves to compute right nodes and one leaf and one inner node to compute left nodes (Lines 3, 4a in Algorithm 2). All together we get the following theorem:

Theorem 1. *Let $H \geq 2$ be even. Algorithm 2 needs to store at most $3.5H-4$ nodes and needs to compute at most $H/2$ leaves and $3/2(H-3)+1$ inner nodes per step to successively compute authentication paths. On average, Algorithm 2 computes $(H-1)/2$ leaves and $(H-3)/2$ inner nodes per step.*

Average costs. We now estimate the average cost of our algorithm in terms of leaves (L) and inner nodes (I) to compute. We begin with the right nodes. On height $h=0$ there are 2^{H-1} right leaves to compute. On heights $h=1,\dots,H-3$, there are 2^{H-h-1} right nodes to compute. Each of these nodes requires the computation of 2^h leaves and 2^h-1 inner nodes. For the left nodes, we must compute one leaf and one inner node every second step, alternating. This makes a total of 2^{H-1} leaves and inner nodes. Summing up yields

$$\left(\sum_{h=0}^{H-3} 2^{H-h-1} \cdot 2^h + 2^{H-1}\right) L + \left(\sum_{h=1}^{H-3} 2^{H-h-1} \cdot (2^h - 1) + 2^{H-1}\right) I \tag{3}$$

$$= \left(\frac{H-1}{2} \cdot 2^H\right) L + \left(\frac{H-3}{2} \cdot 2^H + 4\right) I \tag{4}$$

as total number of leaves and inner nodes that must be computed. To obtain the average cost per step we divide by 2^H.

Space required by the stack. We will show that the stack stores at most one tail node on each height $h = 0, \dots, H-5$ at a time.

$\textsc{Treehash}_h$, $h \in \{0, \dots, H-3\}$ stores up to h tail nodes on different heights to compute the authentication node on height h. The tail node on height $h-1$ is stored in $\textsc{Treehash}_h$.node and the remaining tail nodes on heights $0, \dots, h-2$ are stored on the stack. When $\textsc{Treehash}_h$ receives its first update, the following two conditions hold:

1. All treehash instances on heights $< h$ are either empty or completed and store no tail nodes on the stack.
2. All treehash instances on heights $> h$ are either empty or completed or have tail nodes of height at least h.

Both conditions follow directly from Line 5a in Algorithm 2. These conditions imply that while TREEHASH_h is active, all tail nodes on the stack that have height at most $h-2$ are on different heights.

If a treehash instance on height $i = h+1, \dots, H-3$ stores a tail node on the stack, then all treehash instances on heights $i+1, \dots, H-3$ have tail nodes of height at least i, otherwise the treehash instance on height i wouldn't have received any updates in the first place (recall that $\text{TREEHASH}_i.\text{height} = i$ holds if TREEHASH_i was just initialized). This implies that all tail nodes on the stack that have height at least h and at most $H-5$ are on different heights.

If a treehash instance on height $j < h$ is initialized while TREEHASH_h is active, the same arguments can be applied to the instance on height j.

In total, there is at most one tail node on each height $h = 0, \dots, H-5$ which bounds the number of nodes stored on the stack by $H-4$. This bound is tight for round $\varphi = 2^{H-1} - 2$, before the update that completes the treehash instance on height $H-3$.

Inner nodes computed by treehash. For now we assume that the maximum number of inner nodes is computed in the following case: TREEHASH_{H-3} receives all $u = H/2 - 1$ updates and is completed in this round. On input an index φ, the number of inner nodes computed by treehash in the worst case equals the height of the first parent of leaf φ which is a left node, if the corresponding tail nodes are stored on the stack. On height h, a left node occurs every 2^h leaves, which means that every 2^h updates at most h inner nodes are computed by treehash. This implies that during the u available updates, at most one inner node on height h is computed every $\lceil u/2^h \rceil$ updates for $h = 1, \dots, \lceil \log_2 u \rceil$. The last update requires the computation of $H-3 = 2u-1$ inner nodes to obtain the desired node on height $H-3$, i.e. completing this treehash instance. So far only $\lceil \log_2 u \rceil$ inner nodes were counted, so *additional* $2u - 1 - \lceil \log_2 u \rceil$ inner nodes must be added. In total, we get the following upper bound for the number of inner nodes computed per round.

$$B = \sum_{h=1}^{\lceil \log_2 u \rceil} \left\lceil \frac{u}{2^h} \right\rceil + 2u - 1 - \lceil \log_2 u \rceil \tag{5}$$

In round $\varphi = 2^{H-1} - 2$ this bound is tight. This is the last round before the treehash instance on height $H-3$ must be completed and as we saw in Section 2.2, all available updates are required in this case. The desired upper bound is estimated as follows:

$$\begin{aligned} B &\leq \sum_{h=1}^{\lceil \log_2 u \rceil} \left(\frac{u}{2^h} + 1 \right) + 2u - 1 - \lceil \log_2 u \rceil \\ &= u \sum_{h=1}^{\lceil \log_2 u \rceil} \frac{1}{2^h} + 2u - 1 = u \left(1 - \frac{1}{2^{\lceil \log_2 u \rceil}} \right) + 2u - 1 \\ &\leq u \left(1 - \frac{1}{2u} \right) + 2u - 1 = 3u - \frac{3}{2} = \frac{3}{2}(H-3) \end{aligned}$$

The next step is to show that the above mentioned case is indeed the worst case. If a treehash instance on height $< H-3$ receives all updates and is completed in this round, less than B hashes are required. The same holds if the treehash instance receives all updates but is not completed in this round. The last case to consider is the one where the u available updates are

spend on treehash instances on different heights. If the active treehash instance TREEHASH_h stores a tail node ν on height j, it will receive updates until it has a tail node on height $j+1$. This requires 2^j updates and the computation of 2^j inner nodes. *Additional* $t \in \{1 \dots H-j-4\}$ inner nodes are computed to obtain ν's parent on height $j+t+1$, if TREEHASH_h stores tail nodes on heights $j+1 \dots j+t$ on the stack and in TREEHASH_h.node. The next treehash instance that receives the remaining updates has a tail node on height $\geq j$. This instance computes *additional* inner nodes only, if there are enough updates left to compute an inner node on height $\geq j+t$, the height of the next tail node possibly stored on the stack. But this is the same scenario that appears in the above mentioned worst case, i.e. if a node on height $j+1$ is computed, the tail nodes on the stack are used to compute its parent on height $j+t+1$ and the same instance receives the next update.

Space required to compute left nodes. First we remark that because of Steps 2 and 4a in Algorithm 2, the node stored in KEEP_{h-1} is removed whenever an authentication node is stored in KEEP_h in the same round, $h = 1, \dots, H-2$. Next we show that if a node gets stored in KEEP_h, $h = 0, \dots, H-3$, then KEEP_{h+1} is empty. To see this we have to consider in which rounds a node is stored in KEEP_{h+1}. This happens in rounds $\varphi \in S_a = \{2^{h+1} - 1 + a \cdot 2^{h+3}, \dots, 2^{h+2} - 1 + a \cdot 2^{h+3}\}, a \in \mathbb{N}_0$. In rounds $\varphi' = 2^h - 1 + b \cdot 2^{h+2}, b \in \mathbb{N}_0$, a node gets stored in KEEP_h. It is straight forward to compute that $\varphi' \in S_a$ implies that $2a + 1/4 \leq b \leq 2a + 3/4$ which is a contradiction to $b \in \mathbb{N}_0$.

As a result, at most $H/2$ nodes are stored in KEEP at a time and two consecutive nodes can share one entry. One additional entry is required to temporarily store the authentication node on height h (Step 2) until node on height $h-1$ is removed (Step 4a).

3 Comparison

We now compare our algorithm with Szydlo's algorithm from [13]. We compare the number of leaves, inner nodes, and total hash function evaluations computed per step in the worst and average case.

The computation of an inner node costs one hash function evaluation. This follows directly from the construction rule for Merkle trees of Equation (1). The cost to compute one leaf, in terms of hash function evaluations, depends on the one-time signature scheme used for the MSS. The Lamport–Diffie one-time signature scheme [8] requires $2n$ evaluations of the hash function, where n is the output length of the hash function. The Winternitz one-time signature scheme [5] roughly requires $2^w \cdot n/w$ evaluations of the hash function, where w is the Winternitz parameter. For our comparison, we use a cost of 100 hash function evaluations for each leaf calculation.

Table 1 shows the number of leaves, inner nodes, and total hash function evaluations computed per step in the worst case. These values were obtained experimental. The number of leaves and inner nodes our algorithm requires in the worst case according to Theorem 1 are given in parentheses.

	Our Algorithm			Szydlo's Algorithm		
H	leaves	inner nodes	hashes	leaves	inner nodes	hashes
4	2 (2)	1 (2.5)	201	4	0	400
10	5 (5)	8 (11.5)	508	7	4	704
14	7 (7)	14 (17.5)	714	10	4	1000
20	10 (10)	24 (26.5)	1024	12	8	1208

Table 1. Comparison of the worst case runtime of our algorithm and Szydlos algorithm from [13]. The values according to Theorem 1 are given in parentheses.

This table shows, that the cost for the inner nodes is negligible compared to the cost for the leaf calculations. Our algorithm reduces the total number of hash function evaluations required

in the worst case by more than $49\%, 27\%, 28\%, 15\%$ for $H = 4, 10, 14, 20$, respectively, even when using the comparatively low ratio of 100 hash function evaluations per leaf. When using larger ratios, as they occur in practice, the advantage of our algorithm is more distinct. We state the comparison only for Merkle trees up to a height of $H = 20$, since for larger heights the MSS key pair generation becomes too inefficient so that Merkle trees of height $H > 20$ cannot be used in practice [2].

For $H = 4, 10, 14, 20$, our algorithm needs to store $10, 31, 45, 66$ nodes and Szydlo's algorithm needs to store $10, 28, 40, 58$ nodes, respectively. Although Szydlo's algorithm requires slightly less storage, additional implementing effort and possibly overhead must be taken into account when using Szydlo's algorithm on platforms without dynamic memory allocation. This is because Szydlo's algorithm uses separate stacks for each of the H treehash instances, where, roughly speaking, each stack can store up to $O(H)$ nodes but all stacks together never store more than $O(H)$ nodes at a time. The simple approach of reserving the maximal required memory for each stack yields memory usage quadratic in H.

Table 2 shows the number of leaves, inner nodes, and total hash function evaluations computed per step in the average case. These values were also obtained experimental. The number of leaves and inner nodes our algorithm requires in the average case according to Theorem 1 are given in parentheses.

	Our Algorithm			Szydlo's Algorithm		
H	leaves	inner nodes	hashes	leaves	inner nodes	hashes
4	1.2 (1.5)	0.6 (0.5)	120.6	1.5	0.6	150.6
10	4.0 (4.5)	3.0 (3.5)	403.0	4.5	3.5	453.5
14	6.0 (6.5)	5.0 (5.5)	605.0	6.5	5.5	655.5
20	9.0 (9.5)	8.0 (8.5)	908.0	9.5	8.5	958.5

Table 2. Comparison of the average case runtime of our algorithm and Szydlos algorithm from [13]. The values according to Theorem 1 are given in parentheses.

This table shows, our algorithm on average performs slightly better than Szydlo's algorithm. This is a result of the slightly increased memory usage of our algorithm. More importantly, comparing Tables 1 and 2 shows that the worst case runtime of our algorithm is extremely close to its average case runtime. This certifies that our algorithm provides balanced timings for the authentication path generation and thus the MSS signature generation.

4 Conclusion

We proposed a new algorithm for the computation of authentication paths in a Merkle tree. In the worst case, our algorithm is significantly faster than the best algorithm known so far, namely Szydlo's algorithm from [13]. In fact, the worst case runtime of our algorithm is very close to its average case runtime which, in turn, equals the average case runtime of Szydlo's algorithm. The main idea of our algorithm is to distinguish between leaves and inner nodes of the Merkle tree and balance the number of leaves computed in each step. In detail, our algorithm computes $H/2$ leaves and $3/2(H-3)+1$ inner nodes in the worst case and $(H-1)/2$ leaves and $(H-3)/2$ inner nodes on average. For example, we reduce the worst case cost for computing authentication paths in a Merkle tree of height $H = 20$ by more than 15% compared to Szydlo's algorithm. When implementing our algorithm, the space bound of $3.5H - 4$ nodes can be achieved without additional effort, even on platforms that do not offer dynamic memory allocation.

References

1. Piotr Berman, Marek Karpinski, and Yakov Nekrich. Optimal trade-off for Merkle tree traversal. *Theoretical Computer Science*, 372(1):26–36, 2007.
2. Johannes Buchmann, Carlos Coronado, Erik Dahmen, Martin Döring, and Elena Klintsevich. CMSS — an improved Merkle signature scheme. In *Progress in Cryptology - INDOCRYPT 2006*, volume 4329 of *Lecture Notes in Computer Science*, pages 349–363. Springer-Verlag, 2006.
3. Johannes Buchmann, Erik Dahmen, Elena Klintsevich, Katsuyuki Okeya, and Camille Vuillaume. Merkle signatures with virtually unlimited signature capacity. In *Applied Cryptography and Network Security - ACNS 2007*, volume 4521 of *Lecture Notes in Computer Science*, pages 31–45. Springer-Verlag, 2007.
4. Carlos Coronado. On the security and the efficiency of the Merkle signature scheme. Cryptology ePrint Archive, Report 2005/192, 2005. `http://eprint.iacr.org/`.
5. Chris Dods, Nigel Smart, and Martijn Stam. Hash based digital signature schemes. In *Cryptography and Coding*, volume 3796 of *Lecture Notes in Computer Science*, pages 96–115. Springer-Verlag, 2005.
6. Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual Symposium on the Theory of Computing*, pages 212–219, New York, 1996. ACM Press.
7. Markus Jakobsson, Tom Leighton, Silvio Micali, and Michael Szydlo. Fractal Merkle tree representation and traversal. In *Cryptographer's Track at RSA Conference - CT-RSA'03*, volume 2612 of *Lecture Notes in Computer Science*, pages 314–326, 2003.
8. Leslie Lamport. Constructing digital signatures from a one way function. Technical Report SRI-CSL-98, SRI International Computer Science Laboratory, 1979.
9. Arjen K. Lenstra and Eric R. Verheul. Selecting cryptographic key sizes. *Journal of Cryptology*, 14(4):255–293, 2001. Updated version from 2004 available at `http://plan9.bell-labs.com/who/akl/index.html`.
10. Ralph C. Merkle. A certified digital signature. In *CRYPTO '89: Proceedings on Advances in cryptology*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238. Springer-Verlag, 1989.
11. Dalit Naor, Amir Shenhav, and Avishai Wool. One-time signatures revisited: Practical fast signatures using fractal merkle tree traversal. *IEEE – 24th Convention of Electrical and Electronics Engineers in Israel*, pages 255–259, 2006.
12. Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proc. 35th Annual Symposium on Foundations of Computer Science*, pages 124–134. IEEE Comput. Soc. Press, 1994.
13. Michael Szydlo. Merkle tree traversal in log space and time. 2003. Preprint, available at `http://www.szydlo.com/`.
14. Michael Szydlo. Merkle tree traversal in log space and time. In *Advances in Cryptology - EUROCRYPT 2004*, volume 3027 of *Lecture Notes in Computer Science*, pages 541–554, 2004.

A General version of the authentication path algorithm

We now describe the general version of our algorithm that is also able to handle Merkle trees of odd height. The basic idea is to store more right authentication nodes (like the single node RETAIN) to obtain an even number of treehash instances that need updating. The parameter K denotes the number of upper levels where all right nodes are stored permanently. We must choose $K \geq 2$, such that $H - K$ is even. Instead of a single node RETAIN, we now use stacks RETAIN_h, $h = H-K, \ldots, H-2$, to store all right authentication nodes during the initialization: $\text{RETAIN}_h.\text{push}(y_h[2j+3])$, for $h = H-K, \ldots, H-2$ and $j = 2^{H-h-1}-2, \ldots, 0$. The pseudo-code of the general version is shown in Algorithm 3. Including the parameter K in Theorem 1 yields Theorem 2.

Theorem 2. *Let $H \geq 2$ and $K \geq 2$ such that $H - K$ is even. Algorithm 3 needs to store at most $3H + \lfloor H/2 \rfloor - 3K - 2 + 2^K$ nodes and needs to compute at most $(H - K)/2 + 1$ leaves and $3(H - K - 1)/2 + 1$ inner nodes per step to successively compute authentication paths. On average, Algorithm 3 computes $(H - K + 1)/2$ leaves and $(H - K - 1)/2$ inner nodes per step.*

The simplified version described in Section 2 corresponds to the choice $K = 2$. The proofs for the correctness and the time and space bounds of the general version can be obtained by substituting $H - K + 2$ for H in the proofs for the simplified version. The general version provides an additional trade-off between the computation time and the storage needed.

B Computing leaves using a PRNG

In [2], the authors propose to use a forward secure PRNG to successively compute the one time signature keys. The benefit is that the storage cost for the private key is reduced drastically, since only one seed must be stored instead of 2^H one-time signature keys. Let SEED_φ denote the seed required to compute the one-time key pair corresponding to leaf φ.

During the authentication path computation, leaves which are up to $3 \cdot 2^{H-K-1}$ steps away from the current leaf must be computed. Calling the PRNG that many times to obtain the seed required to compute this leaf is too inefficient. Instead we propose the following scheduling strategy that requires $H - K$ calls to the PRNG in each round to compute the seeds. We have to store two seeds for each height $h = 0, \ldots, H-K-1$. The first (SEEDACTIVE) is used to successively compute the leaves for the authentication node currently constructed by TREEHASH_h and the second (SEEDNEXT) is used for upcoming right nodes on this height. SEEDNEXT is updated using the PRNG in each round. During the initialization, we set $\text{SEEDNEXT}_h = \text{SEED}_{3 \cdot 2^h}$ for $h = 0, \ldots, H - K - 1$. In each round, at first all seeds SEEDNEXT_h are updated using the PRNG. If in round φ a new treehash instance is initialized on height h, we copy SEEDNEXT_h to SEEDACTIVE_h. In that case $\text{SEEDNEXT}_h = \text{SEED}_{\varphi+1+3 \cdot 2^h}$ holds and thus is the correct seed to begin computing the next authentication node on height h.

Algorithm 3 Authentication path computation, general version

Input: $\varphi \in \{0, \ldots, 2^H - 2\}$, H, K and the algorithm state.
Output: Auththentication path for leaf $\varphi + 1$

1. Let $\tau = 0$ if leaf φ is a left node or let τ be the height of the first parent of leaf φ which is a left node:
   $\tau \leftarrow \max\{h : 2^h | (\varphi + 1)\}$
2. If the parent of leaf φ on height $\tau + 1$ is a left node, store the current authentication node on height τ in $\textsc{Keep}_\tau$:
   if $\lfloor \varphi / 2^{\tau+1} \rfloor$ is even **and** $\tau < H - 1$ **then** $\textsc{Keep}_\tau \leftarrow \textsc{Auth}_\tau$
3. If leaf φ is a left node, it is required for the authentication path of leaf $\varphi + 1$:
   if $\tau = 0$ **then** $\textsc{Auth}_0 \leftarrow \textsc{Leafcalc}(\varphi)$
4. Otherwise, if leaf φ is a right node, the authentication path for leaf $\varphi + 1$ changes on heights $0, \ldots, \tau$:
   if $\tau > 0$ **then**
   (a) The authentication path for leaf $\varphi + 1$ requires a new left node on height τ. It is computed using the current authentication node on height $\tau - 1$ and the node on height $\tau - 1$ previously stored in $\textsc{Keep}_{\tau-1}$. The node stored in $\textsc{Keep}_{\tau-1}$ can then be removed:
   $\textsc{Auth}_\tau \leftarrow f(\textsc{Auth}_{\tau-1} || \textsc{Keep}_{\tau-1})$, remove $\textsc{Keep}_{\tau-1}$
   (b) The authentication path for leaf $\varphi + 1$ requires new right nodes on heights $h = 0, \ldots, \tau - 1$. For $h \leq H - K - 1$ these nodes are stored in $\textsc{Treehash}_h$ and for $h \geq H - K$ in $\textsc{Retain}_h$:
   for $h = 0$ **to** $\tau - 1$ **do**
       if $h \leq H - K - 1$ **then** $\textsc{Auth}_h \leftarrow \textsc{Treehash}_h$.node
       if $h > H - K - 1$ **then** $\textsc{Auth}_h \leftarrow \textsc{Retain}_h$.pop()
   (c) For heights $0, \ldots, \min\{\tau - 1, H - K - 1\}$ the treehash instances must be initialized anew. The treehash instance on height h is initialized with the start index $\varphi + 1 + 3 \cdot 2^h$ if this index is smaller than 2^H:
   for $h = 0$ **to** $\min\{\tau - 1, H - K - 1\}$ **do**
       if $\varphi + 1 + 3 \cdot 2^h < 2^H$ **then** $\textsc{Treehash}_h$.initialize($\varphi + 1 + 3 \cdot 2^h$)
5. Next we spend the budget of $(H - K)/2$ updates on the treehash instances to prepare upcoming authentication nodes:
   repeat $(H - K)/2$ **times**
   (a) We consider only stacks which are initialized and not finished. Let s be the index of the treehash instance whose lowest tail node has the lowest height. In case there is more than one such instance we choose the instance with the lowest index:
   $s \leftarrow \min \left\{ h : \textsc{Treehash}_h.\text{height}() = \min_{j=0,\ldots,H-K-1} \{\textsc{Treehash}_j.\text{height}()\} \right\}$
   (b) The treehash instance with index s receives one update:
   $\textsc{Treehash}_s$.update()
6. The last step is to output the authentication path for leaf $\varphi + 1$:
   return $\textsc{Auth}_0, \ldots, \textsc{Auth}_{H-1}$.