Optimal Trade-Off for Merkle Tree Traversal

Piotr Berman* Marek Karpinski† Yakov Nekrich‡

Abstract. In this paper we describe optimal trade-offs between time and space complexity of Merkle tree traversals along with their associated authentication paths, improving on the previous results of Jakobsson, Leighton, Micali and Szydlo [JLMS03] and Szydlo [S03]. In particular we show that our algorithm requires $2\log n/\log^{(3)} n$ hash function evaluations and storage of less than $(\log n/\log^{(3)} n)\log\log n + 2\log n$ hash values, where n is the number of leaves in the Merkle tree. We also prove that these trade-offs are optimal, i.e. there is no algorithm that requires less than $\Theta(\log n/\log t)$ time and less than $\Theta(t\log n/\log t)$ space for any choice of parameter t.
Our algorithm could be of special use in the case when both time and storage are simultaneously limited.

Keywords: Merkle Trees, Public Key Signatures, Authentication Path, Fractal Tree Traversal, Trade-off, Amortization.

*Dept.of Computer Science and Engineering, The Pennsylvania State University. Research done in part while visiting Dept. of Computer Science , University of Bonn. Work partially supported by NSF grant CCR-9700053 and NIH grant 9R01HG02238-12. E-mail berman@cse.psu.edu

†Dept. of Computer Science, University of Bonn. Work partially supported by DFG grants , Max-Planck Research Prize, DIMACS and IST grant 14036 (RAND-APX). E-mail marek@cs.uni-bonn.de

‡Dept. of Computer Science, University of Bonn. Work partially supported by IST grant 14036 (RAND-APX). E-mail yasha@cs.uni-bonn.de

1 Introduction

Merkle trees have found wide applications in cryptography mainly due to their conceptual simplicity and applicability. Merkle trees were first described by Merkle [M79] in 1979 and studied intensively in a number of papers, cf., e.g., [JLMS03] and [S03]. In cryptographic applications, however, Merkle trees were not very useful for small computational devices, as the best known techniques for traversal required a relatively large amount of computation and storage. In this paper we address the issue of possible improvements of Merkle tree traversals.

Merkle tree is a complete binary tree such that values of internal node, are one-way functions of the values of their children. Every leaf value in Merkle tree can be identified with respect to a publicly known root and the *authentication path* of that leaf. An authentication path of a leaf consists of the siblings of all nodes on the path from this leaf to the root.

Merkle trees have had many cryptographic applications, such as certification refreshal [M97], broadcast authentication protocols [PC02], third party data publishing [DG01], zero-knowledge sets [MRK03] and micro-payments [RS96]. A frequent problem faced in such applications is, so called, *Merkle tree traversal* problem, the problem of outputting the authentication data for every leaf. In [M87] Merkle has proposed a technique for traversal of Merkle trees, which required $O(\log^2 n)$ space and $O(\log n)$ time per authentication path in the worst case. Recently two results, improving a technique of Merkle, have appeared. Jakobsson, Leighton, Micali and Szydlo [JLMS03] describe a Merkle tree traversal algorithm with $O(\log^2 n/\log\log n)$ space and $O(\log n/\log\log n)$ time per output. In [S03] Szydlo describes a method, requiring $O(\log n)$ space and $O(\log n)$ time and provides a proof that this bound is optimal, i.e. he proves, that there is no traversal algorithm, that would require both $o(\log n)$ space and $o(\log n)$ time.

In this paper we investigate further the trade-off between time and space requirements of Merkle tree traversals. [JLMS03] and [S03] were the starting points of these investigations.

First, we present an algorithm, that works in $O(\log n/h)$ time and $O((\log n/h)2^h)$ space per round for arbitrary parameter h. For $h = O(1)$ our result is equivalent to the result of Szydlo, however we consider all operations (not just computations of one-way functions) in our analysis. Our result is also an extension of that of Jakobsson, Leighton, Micali and Szydlo [JLMS03], in that we prove that it can be extended for arbitrary values of h. Besides that, we achieve better constants in the space bound.

Secondly, we show that the results of [S03] and [JLMS03] remain true, if we consider all operations and not just hash computations. (If h is higher than constant we ignore times, that we need to output the values in the last case).

In particular, we show that an algorithm with $2\log n/\log\log\log n$ hash functions evaluations and storage requirement of $(\log n/\log\log\log n+1)\log\log n+2\log n$ hash values per output can be constructed. This algorithm works with $O(\log n/\log\log\log n)$ operations per output.

At the end, we show that if a tree traversal algorithm works in time $O(\log n/h)$ than required space is $O((\log n/h)2^h)$. Thus we show that our trade-off is optimal.

2 Preliminaries and Notation

Below we denote by a *hash (unit)* a one-way function and hash computation will denote a computation of the value of a one-way function. In a Merkle tree leaf values are hash values of the *leaf pre-images.* Leaf pre-images can be, for instance, generated with a pseudo-random generator. We will denote by *leaf-calc* a function, that computes pre-images of the leaves. Let ϕ_1=*hash*$\circ$*leaf-calc* be the function that computes value of the i-th leaf. Let ϕ_2*(parent)=hash(left-child*$||$*right-child)* be the function, that computes the value of the parent node from the values of its children. We will presume, that we need one computation unit to compute ϕ_1 or ϕ_2.

We must generate n outputs, where n is the number of leaves. Every output consists of the leaf pre-image and its *authentication path.* An authentication path consists of the siblings of all nodes on the path to the root. Outputs for the leaves must be generated consecutively left-to-right. This makes our task easier, because outputs for consecutive leaves have many common node values.

In order to verify the leaf one consecutively computes the values of its ancestors. Verification succeeds only if the computed root value equals to the known root value.

In this paper the following notation will be used. H will denote the Merkle tree height. We will say, that a node is on level A, if its depth is $H-A$. The i-th node from the left on level A will be denoted by (A,i). A job, computing node (A,i) will also be denoted by (A,i). We will say, that A is the job level and i is the index of the job. Sometimes we will identify a subtree of the Merkle tree by its root node (A,i). We will use a *subtree height* h as a parameter in our algorithm and L will be equal to H/h.

We say, that a node N is *needed* if it is a part of an authentication path.

3 Main Idea

We describe here the main idea of our algorithm and give key observations on which the algorithm is based.

The following well-known evaluation algorithm is used to compute the value of the i-th node at level A and is an important part of all Merkle tree traversal algorithms.

```
**Eval** (A,i)
    **if**($A == 0$)
        **return** $\phi_1$(**i**);
    **else**
        $V = Eval(A-1, 2i)$;
        **return** $\phi_2(V, Eval(A-1, 2i+1))$
```

Figure 1: Algorithm **Eval**

This basic version of algorithm *Eval* requires 2^A computational units and A storage units. The last follows from the fact, that at most one node value V for every height $i = 0, 1, \ldots, A$ has to be stored at every stage of the algorithm. This stored values will be further called *tail values.*

Our algorithm uses procedure *Eval* to estimate the values of nodes, that will be needed in the future authentication path. Jobs, that compute the values of nodes (A, i) and called by our algorithm (and not by another job) will be called *root jobs.*

The key observation on which our algorithm is based, is that during the computation of node (A, i) its children $(A-1, 2i)$, $(A-1, 2i+1)$ as well as all other descendants will be computed. Therefore by storing intermediate results of evaluation some future computations can be saved. Actually for every computed node N on level ih all its descendants on levels $ih-1, \ldots, in-h$ (i.e. a complete subtree of height h rooted in N) will be retained to be used in the future authentication paths. Thus only nodes at height ih $i = 1, \ldots, L$ will be computed directly.

Another key observation, is that we can schedule the computations of the nodes, needed in the future in such a way, that at most H storage units are necessary to store all tail values.

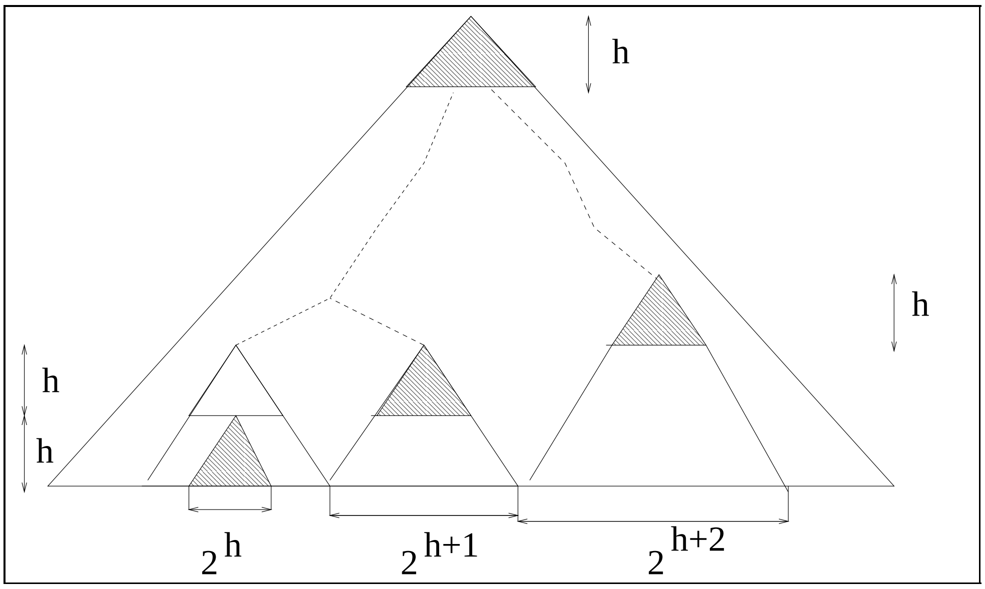

Figure 2: Subtrees computed at a round of the algorithm

4 Algorithm Description

Our algorithm consists of three phases: **root generation**, **output** and **verification**. During the first phase the root of the Merkle tree is generated. Additionally, the initial set of subtrees with roots at $(0, 2^{sh})$, $i = 1, \ldots, L$ is computed and stored.

The verification phase is identical to the traditional verification phase (see, for instance, [JLMS03]).

The output phase consists of 2^H rounds and during round j an image of the j-th leaf and its authentication path are output. In the rest of this section we will describe an algorithm for the output phase and prove its correctness.

For convenience we will measure time in rounds. During each round L computation units will be spent on computation of subtrees, needed in the future authentication paths. Thus our algorithm will start at time 0 and end at time $2^H - 1$ and i-th round will start at time i. In the first part of the algorithm description we will ignore the costs of all operations, except of the computations of hash functions. Later we will show, that the number of other operations, performed during a round, is $O(L)$.

During round j we store L already computed subtrees with roots at (sh, m_s) with $j \in [m_s 2^{sh}, (m_s{+}1)2^{sh})$, $s = 0, 1, \ldots, L$. During the same round we also spend $2L$ computation units in order to compute jobs $(sh, m_s + 1)$ and construct the corresponding subtrees. At round $(m_s + 1)2^{sh}$ the subtree

(sh, m_s) will be discarded, However the subtree $(sh, m_s + 1)$ will be retained for the next 2^{sh} rounds, while subtree $(sh, m_s + 2)$ is computed.

During each round there are at most L different jobs competing for $2L$ computation units. These jobs will be called *active*. Active jobs are scheduled according to the following rules:

1. A root job $(2^{ih}, k)$ $k = 1, \dots, H/2^{ih}$ becomes active at time $(k-1)2^{ih}$, i.e. during the $(k-1)2^{ih}$-th round.

2. All recursive calls for computation of nodes $(s, \cdot)$ performed by root jobs $(s', \cdot)$ with $s' > s$, that already started when job $(s, \cdot)$ becomes active, must be completed, before job $(s, \cdot)$ starts.

3. In all other cases the jobs with the lower level have priority over the jobs with the higher level.

Consider job (sh, i), that starts at time $2^{sh}i$. Rule 2 guarantees us, that, all jobs with levels $s' > s$ will finish their level s calls, before computation of job (sh, i) starts. Therefore, when job (sh, i) is computed only one tail node on each of the levels $(s-1)h, (s-1)h + 1, \dots, sh - 1$ will be stored. Now consider a job, with level $s' > s$, calling a level s job in ... . All jobs with levels $s'' > s'$ do not store any tail nodes at levels $0, 1, \dots s'$. All jobs with levels $\tilde{s} < s'$ do not store any nodes, according to ruled 2 and 3.

This scheduling guarantees us, that at any time only one tail value for a level $i = 1, 2, \dots, H$ will be stored by all jobs (sh, i). Since only $2L$ subtrees (one currently used and one currently computed for each level ih) must be stored at each round and subtrees require $(2H/h)2^h$ space. Hence the memory requirement of our algorithm is $(2H/h)2^h + H = O((H/h)2^h)$.

This considerations allow us to formulate the following trade-off between time and space complexity.

Theorem 1 *Merkle tree can be traversed in time $O(H/h)$ with $O((H/h)2^h)$ storage units.*

Corollary 1 *Merkle tree can be traversed in time $O(\log n / \log^{(3)} n)$ with $O(\log n \log\log n / \log^{(3)} n)$ storage units.*

In the next subsections, we will prove the algorithm correctness by showing, that all the values are computed on time and we prove the time bound, stated in the theorem by analysis of the operations, necessary for the job scheduling.

4.1 Correctness Proof

In the section we show, that job (s,k) will be completed at time $k2^s$.

Lemma 1 *Suppose, that at time $(k-1)2^{sh}$ for every level $i = h, 2h, \ldots, (s-1)h, (s+1)h, \ldots Lh$ there is at most one unfinished job on level i. Then the job (sh,k) will complete before $k2^{sh}$.*

Proof: Consider the time interval $[(2k-2)2^{sh}, (2k-1)2^{sh})$. Since there are at most $(L-s)$ jobs with unfinished recursive calls to $Eval(s,\cdot)$ the time to complete the recursive calls is limited by $(L-s)2^{sh+1}$. Besides that, there are also jobs with lower indices, that must be completed before (sh,k) can be completed. There are at most $2^{(s-s')h}$ such jobs with index $s' < s$. Hence the total number of computation units, needed for these jobs is $(s-1)2^{sh+1}$. Thus we have 2^{sh+1} computation units left to complete the job $(sh,2k)$.

Lemma 2 *At every moment of time there is only one running job on level sh, $s = 1, 2, \ldots, L$.*

Proof: At time 0 we start only one job on level sh. For every level sh and every index i we can easily prove by induction, using Lemma 1, that at time interval $[2^{sh}i, 2^{sh}(i+1))$ there is only one running job with index i on level sh.

Lemma 3 *Computation of job (sh,i) will be finished before time $k2^{sh}$*

Proof: Easily follows from Lemma 1 and Lemma 2.

In our computation only every h-th node on the computation path is computed directly. Below we will show which nodes should be retained during the computation of (sh,i).

All nodes $(ih-m, s2^m+j)$, where $m = 1, \ldots, h$ and $j = 0, \ldots, m-1$ must be retained. In other words, all descendants of (ih,s) at levels $ih-1, \ldots, (i-1)h$ must be retained.

Proposition 1 *Descendants of a node $(2^{ih}, m)$ are needed during rounds $[m2^{ih}, (m+1)2^{ih})$.*

Proof: Indeed, children of $(2^{ih}, m)$ are needed during rounds $[m2^{ih}+2^{h-1}, (m+1)2^{ih})$ and $[2^{ih}, 2^{ih}+2^{h-1})$. For descendants on other levels, this proposition is proved by the fact, that when a node is needed, the sibling of its parent is also needed.

Combining Lemma 3 with the above statement we see, that every node will be computed before it is needed the first time.

4.2 Time Analysis

We have shown above, that our algorithm performs L hash-function computations per round. Now we will show, that all other operations will take $O(L)$ time per round.

Lemma 4 *Job scheduling, according to rules 1.-3. can be implemented in $O(L)$ time per round.*

For every level $s = ih$ we store a list Q_i of level s jobs, that have to be performed. When a new job on level ih becomes active, or when a level s job is called by another job, it is added to Q_i. Lists Q_i are implemented as queues.

At round j our algorithm, checks all queues Q_i in ascending order. If a non-empty Q_i is found, we spend L hash computations on computing the last job l in Q_i. If the job l is finished after $k < L$ hash computations we remove l to the job, that called it. If l is not a root job, we return its result to the job, that called it. and traverse queues $Q_i, Q_{i+1}, \dots Q_l$ until another non-empty queue is found.

When a job (s, i) recursively calls job $(s', 2^{s-s'}i)$ we add this new job to list $Q_{s'}$. When a non-root job is completed it returns its value to the job, that called it.

We also have to modify the procedure *Eval* in order to limit the number of recursive calls. . In the modified version, the number of recursive calls per round does not exceed L, because a procedure on level s calls procedures on level $s - h$. In this procedure variable $Tail_i$ stores the value of the tail node on level i. Note that variables $Tail_i$, $i = 1, 2, \dots, H$ are common to all jobs. The value of node (s, k) is stored in $Tail_s$, if k is even. If k is odd we compute the value of parent of (s, k). (This is possible because value of the sibling of (s, k) is stored in $Tail_s$). The modified version of *Eval* is shown on Fig. 3.

5 The Lower Bound

In this section we prove the lower bound on space and time complexity of Merkle tree traversals and show that that the algorithm, described above is asymptotically optimal. We prove the following result:

Theorem 2 *Any Merkle tree traversal algorithm with average time per round $O(\log n/a)$ requires $\Omega((\log n/a)2^a)$ space.*

```
Eval(A,i)
   if(A== 0)
      return($\phi_1(i)$)
   else
      $ind := 2^h i$
      $lev := A - h$
      while($lev \neq A$)
         $V := Eval(lev, ind)$
         while( $ind$ mod $2 = 1$)
            $V := \phi_2(Tail_{lev}, V)$
            $lev := lev + 1$
            $ind := ind/2$
         $Tail_{lev} := V$
         $ind := (ind + 1)2^{lev+L-A}$
```

Figure 3: Modified procedure **Eval**

In order to prove this theorem we will consider only time required for the hash computations.

First we will make a difference between nodes with even and odd indices, further called even and odd nodes respectively. Even nodes are needed after their children. In case of odd non-leaf nodes the situation is opposite: they are needed before their children. Namely, $(s, 2i+1)$ is needed during the time interval $[2i2^s, (2i+1)2^s)$ and its children, $(s-1, 4i+3)$ and $(s-1, 4i+2)$, are needed during $[2^{s-1}(4i+2), 2^{s-1}(4i+3))$ and $[2^{s-1}(4i+3), 2^{s-1}(4i+4))$ respectively. We can generalize this observation: an odd node is needed before all its proper descendants. We have just proved it for children; to extend the proof by one more generation, observe that when a node is needed and it is not a child or the root, then the sibling of its parent is needed.

During the computation, when we execute

$$v = Eval(s,i) = \phi_2(Eval(s-1, 2i), Eval(s-1, 2i+1))$$

we can remove $v_0 = Eval(s-1, 2i)$ and $v_1 = Eval(s-1, 2i+1)$ or not. Suppose that we are not removing value v_j even though we will not keep v_j until it is needed (directly). Then we can normalize our algorithm by removing v_j and keeping v instead: computing v is the only use for v_j other than including it in a certificate. Clearly, this normalization increases neither memory nor time.

Computing $Eval(s, i)$ takes $2^{s+1} - 1$ steps (that evaluate ϕ_1 or ϕ_2) and in our lower bound reasoning we can estimate this as 2^s steps. By adding s's over all needed odd nodes we obtain the total number of *job units*. The number of job units for odd nodes on level s is $2^s 2^{H-s-1} = 2^{H-1} = n/2$. Therefore the total number of job units for odd nodes of the Merkle tree is $Hn/2$. We do not count the costs of computing needed values of even nodes in our lower bound proof.

When we decide to remember a value that is used to compute another, we do three things: (a) we account for a certain number of steps – steps used to compute this value that were not accounted for by other remembered values, (b) we account for a certain number of memory units (one memory unit allows to store one value through one round) and (c) we account for a certain number of job units – job units that correspond to the steps that could be executed each time that this value is computed.

We account for the remembered values in an order in which children precede the parents.

Suppose that we rememeber the value of node v_0 during the computation of node v, but do not remember the value of v_1, where v_1 is an ascendant of v_0. Then we can save more job units by remembering v_1 instead of v_0. Hence, if we remember the value of v_0 on level l_0 during computation of node v on level l, then values of all nodes on levels $l_0, l_0 + 1, \ldots, l$ are also remembered. Therefore when a node on level s is computed it is either computed "from scratch" with 2^s steps or it is computed with 1 step because its children were already computed and remebered.

Suppose that we remember the result $Eval(s, 2i+1)$ and we use this value a times for computation of node values (including node $(s, 2i + 1)$). The last use, when $Eval(s, 2i + 1)$ is needed, requires 2^s memory units. If we want to use this value twice, we have to compute it before the parent (or other odd ancestor is needed), and since the parent (ancestor) is needed for 2^{s+1} rounds (or more), we need at least 2^{s+1} memory units. By induction, if we want to use $Eval(s, 2i + 1)$ for a node values, we need to use at least $2^{a-1}2^s$ memory units.

Consider a node $(s, 2i + 1)$. Suppose that its value was used in a computations. As shown above, we need either 2^s steps or 1 step to compute it. If we need 1 step, than the total number of job values we accounted for is a and the total number of memory units is $2^{a-1}2^s$. Suppose, that we needed 2^s steps to compute $(s, 2i + 1)$. Then the total number of job units is $a2^s$ and the number of memory units is $2^{a-1}2^s$. Now we can distribute the steps and memory units between the job units that we have accounted for. Each of them receives a^{-1} steps and at least $2^{a-1}/a$ memory units.

If we use z to express the amount of steps a job unit obtains, then the minimal number of obtained memory units is $\frac{1}{2}z2^{1/z}$. Note that this is a convex function of z (the second derivative is positive for positive z). Thus if job units receive z steps on average then on average they receive at least $\frac{1}{2}z2^{1/z}$ memory units.

As a result, if the computation takes $Hn/2a$ steps then it uses at least $2^a/a \times Hn/4$ memory units, and given that we have n rounds, in average round we need to remember at least $H2^a/4a$ values.

6 A Constant Improvement

In this section we describe an improved version of the algorithm from the section 4. In our improved version we do not compute all nodes in the subtrees. Instead of this only the nodes with even indices are computed. This is possible because even nodes will be needed after their children are needed. Therefore, if we store both children of an even node until their parent is needed, we can compute its value with one hash computation.

Thus in a subtree (ih, k) we only compute nodes $(ih-1, 2k+1), (ih-2, 4k+1), (ih-3, 8k+1), \ldots$ and only the nodes $(ih-1, 2k-1), (ih-2, 4k+1), (ih-2, 4k+3), \ldots, (ih-h, k2^h+1), \ldots, (ih-h, k2^h+2^h-1)$ must be stored.

Computation of all odd descendants of (ih, k) will take time $2^{ih-1+1} - 1 + 2^{ih-2=1} - 1 + 2^{ih-h+1} - 1 = \sum_{k=0}^{h-1} 2^{ih-k} - h = 2^{ih+1} - 2^{(i-1)h+1} - h$. We will need h extra hash computations to compute the even nodes. Therefore the total number of computations for subtree (ih, k) is $2^{ih+1} - 2^{(i-1)h+1}$.

It is easy to see, that there is at most one "new" even node at every round. Therefore it takes at most one extra computation per round to deal with even nodes (if we compute an even node just as it is needed).

To compute the node (s, j) with one hash computation we have to store its odd child $(s-1, 2j+1)$ during rounds $[(2j+1)2^{s-1}, (2j+2)2^{s-1})$. Thus there are at most h odd nodes, that should be kept "extra time" and at most h nodes that are a part of an authentication path during each round. Therefore the total memory requirement is $(2^h - 1 + 2h)L$ per subtree. We need the first summand to store the odd nodes in the subtree and we need the second summand to store the even nodes from the current authentication path and odd nodes kept "extra time".

The nature of our trade-off depends on the subtree height h. For a subtree of height $h = 1$ this improvement results in almost 3/2-fold increase in space and speed-up of almost factor 2. This allows us to formulate the following result

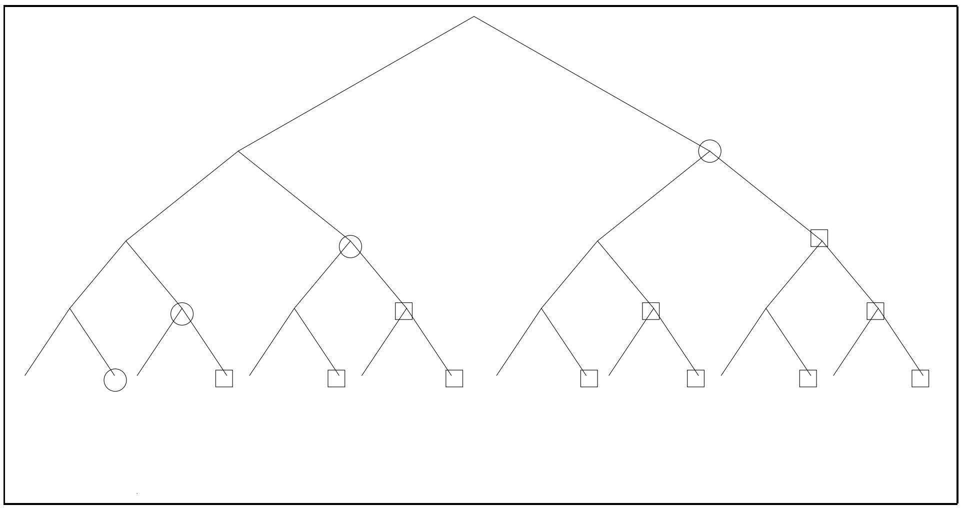

Figure 4: Example of a subtree. Computed nodes are marked by circles. Nodes, marked by circles or squares are stored.

Corollary 2 *A Merkle tree traversal algorithm can be implented with* $\log n$ *hash function evaluations,* $3\log n$ *memory locations for hash values and* $O(\log n)$ *time for other operations per round*

For larger values of h the time improvement becomes very small but we have an almost two-fold decrease of the space used by hash values. In the last case we can also schedule our computation in such way that the values in the next subtree are computed almost exactly at the time when the corresponding values in the current subtree "expire" and can be discarded. In the last case at most one extra value per subtree would have to be stored. In our modified algorithm computation of odd nodes of subtree (ih, k), $i = 2, 3, , \dots, L - 1$ will be divided into two stages. In the first stage level $((i-1)h$ descendants of (ih, k) ("leaves" of the subtree) will be computed. We will further call nodes $((i-1)h, 2^h k + j)$, $j \in [0, 2^h)$ *bottom level* nodes of subtree (ih, k). In the second stage the odd nodes will be computed from bottom level nodes. Observe that computation of the subtree (ih, k) takes place in the same time interval $[2^{ih}(k-1), 2^{ih}k)$ as in our first algorithm. The idea of our modification is that nodes $((i-1)h, 2^h k + j)$, $j \in [0, 2^h)$ i.e. bottom level nodes of (ih, k), will be computed slower than odd nodes of subtree $(ih, k-1)$ will be discarded. Computation of the odd nodes from the bottom tree nodes is performed during the last 2^h rounds of the interval $[2^{ih}(k-1), 2^{ih}k)$. We will further call the jobs, computing the bottom level nodes *secondary jobs* and the last job, computing the remaining odd nodes of the subtree will be called a *primary job*.

In order to reserve 2^h rounds for computation of odd nodes we allocate $2^{(i-1)h}-1$ rounds for computation of every secondary job. Below we reformulate the scheduling rules for job computation. Observe that these modified rules are applied for subtrees on levels $2h, 3h, \ldots, (L-1)h$. We also reserve two computational units per round for the subtrees on level h.

1. An m-th secondary job of subtree (ih, k) becomes active at time $2^{ih}(k-1)+2^{(i-1)h}m-m$. Primary job of $(2^{ih}, k)$ becomes active at time $2^{ih}k-2^h$
2. All recursive calls for computation of nodes $(s, \cdot)$ performed by root jobs $(s', \cdot)$ with $s' > s$, that already started when job $(s, \cdot)$ becomes active, must be completed, before job $(s, \cdot)$ starts.
3. In all other cases the jobs with the lower level have priority over the jobs with the higher level.

Now we prove the space bound of our modified algorithm. First we show, that a secondary job of a node on ih can be completed in $2^{(i-1)h}-1$ rounds.

Lemma 5 *Suppose, that at time $(k-1)2^{ih}+m2^{((i-1)h}-m$, $m = 0, 1, \ldots, 2^h-1$ for every level $l = h, 2h, \ldots, (i-1)h, (i+1)h, \ldots Lh$ there is at most one unfinished secondary job of a job on level l. Then the m-th secondary job of (ih, k) will complete before $(k-1)2^{ih}+(m+1)2^{((i-1)h}-m-1$.*

Proof: Consider the time interval $[(k-1)2^{ih}+m2^{((i-1)h}-m, (k-1)2^{ih}+(m+1)2^{((i-1)h}-m-1)$.

Since there are at most $(L-i)$ jobs with unfinished recursive calls to $Eval(s, \cdot)$ the time to complete the recursive calls is limited by $(L-i)(2^{sh+1}-1)$. Besides that, there are also jobs with lower indices, that must be completed before (sh, k) can be completed. There are at most $2^{(i-i')h}$ such jobs for every index $i' < i$. Hence the total number of computation units, needed for all such jobs is less than $(s-1)(2^{sh+1}-1)$. Another $(2^{h+1}-1)2^{(s-1)h}$ computation units are claimed by subtrees on level h. Thus we have more than 2^{sh+1} computation units left to complete the job $(sh, 2k)$.

Lemma 6 *Computation of the m-th secondary job of (ih, k), $i = 2, 3, \ldots, L-1$ will be finished before time $(k-1)2^{ih}+m+1)2^{((i-1)h}-m-1$*

Proof is analogous to the Proof of Lemma 3

It easily follows from Lemma 6 and Rule 1 that m-th bottom node of (sh, k) will be finished in interval $[2^{sh}(k-1)+t_{m-1}, 2^{sh}(k-1)+t_m)$.

It remains to compute how many odd nodes of $(ih, k-1)$ will be discarded before $2^{ih}(k-1)+t_{m-1}$.

Let $w = 2^h$. After $2^{h(i-1)}m$ rounds the number of remaining nodes can be estimated as $(w-m)/2+(w-m)/4+\ldots+(w-m)/w \leq (w-m)$. We did not count the nodes of the current authentication path in this estimation. Therefore the total number of stored nodes in (ih, k) and $(ih, k-1)$ in interval $[2^{sh}(k-1), 2^{ih}k-2^h)$ is limited by 2^h.

The primary job for (sh, k) can be computed in 2^h rounds. This job can be perormed in-place, because when a new node is computed its even child can be discarded.

In the modified algorithm we apply the job scheduling scheme only to subtrees on levels ih, $i = 2, \ldots, L-1$. Since there is only one subtree for $i = L$ it is not recomputed. Therefore the total number of tail nodes does not exceed $H - h$.

During each round we use two reserved computation units to compute the next level h subtree. By the same argument as above we can see that the number of remaining nodes in the current level h subtree after m rounds is limited by $2^h - m$. Therefore the total number of nodes in the current and future subtrees of level h is limited by 2^h. This computation would require up to h additional units for the tail values. Therefore the total number of tail values is $H - h + h = H$.

The above considerations allow us to formulate the following

Theorem 3 *A Merkle tree traversal can be implemented in $O(L)$ time with $2L$ hash operations. This algorithm requires $L2^h + 2H$ memory locations to store hash values.*

In the last Corollary we have ignored the time necessary to output the $\log n$ values per round. The result, described in the abstract follows, if we choose $h = \log^{(3)} n$.

7 Conclusion

In this paper we describe the first optimal trade-off between time and space complexity of Merkle tree traversals.

We believe it is possible to improve further the constants in the described trade-off by differentiating between various types of nodes in our procedure.

References

[CJ02] D. Coppersmith, M. Jakobsson, "Almost Optimal Hash Sequence Traversal", Financial Cryptography, 2002, 102-119

[DG01] P.Devanbu, M. Gertz, C. Martel, S.G. Stublebine "Authentic Third Party Data Publication" 14th IFIP Workshop on Database Security, 2000

[J02] M. Jakobsson, "Fractal Hash Sequence Representation and Traversal", ISIT, 2002, p. 437

[JLMS03] M. Jakobsson, T. Leighton, S. Micali and M. Szydlo, *Fractal Merkle Tree Representation and Traversal*, RSA Cryptographers Track, RSA Security Conference, 2003.

[L02] H. Lipmaa, "On Optimal Hash Tree Traversal for Optimal Time Stamping", Proc. Information Security Conference, 2002, LNCS 2433, 357-371.

[M79] R. Merkle, "Secrecy, Authentication and Public Key Systems", UMI Research Press, 1982

[M87] R. Merkle, *A Digital Signature Based on a Conventional Encryption Function*, Proc. Crypto 1987, 369-378.

[M97] S. Micali, "Efficient Certificate Revocation" , Technical Report TM-542b, MIT Laboratory for Computer Science, March 22, 1996

[MRK03] S. Micali, M. Rabin, J. Kilian "Zero-Knowledge Sets", Proc. 44th FOCS (2003), 80-91 .

[PC02] A. Perrig, R.Canetti, D. Tygar, D. Song, " The TESLA Broadcast Authentication Protocol" , Cryptobytes, vol 5, pp. 2-13,
Available at `http://citeseer.nj.nec.com/perrig02tesla.html`

[RS96] R. Rivest, A. Shamir, "PayWord and MicroMint - Two Simple Micropayment Schemes", CryptoBytes, vol. 1, pp. 7-11.
Available at `theory.lcs.mit.edu/ rivest/RivestShamir-mpay.ps`

[S03] M. Szydlo, *Merkle Tree Traversal in Log Space and Time*, to appear in Eurocrypt 2004 Available at `http://www.szydlo.com/logspacetime.ps.gz`