Merkle Tree Traversal in Log Space and Time

Michael Szydlo

RSA Laboratories, Bedford, MA 01730. `mszydlo@rsasecurity.com`

Abstract. We present a technique for Merkle tree traversal which requires only logarithmic space and time. For a tree with N leaves, our algorithm computes sequential tree leaves and authentication path data in time $2\log_2(N)$ and space less than $3\log_2(N)$, where the units of computation are hash function evaluations or leaf value computations, and the units of space are the number of node values stored. This result is an asymptotic improvement over all other previous results (for example, measuring $cost = space * time$). We also prove that the complexity of our algorithm is optimal: There can exist no Merkle tree traversal algorithm which consumes both less than $O(\log_2(N))$ space and less than $O(\log_2(N))$ time. Our algorithm is especially of practical interest when space efficiency is required.

Keywords: amortization, authentication path, Merkle tree, tail zipping, binary tree, fractal traversal, pebbling

1 Introduction

Twenty years ago, Merkle suggested the use of complete binary trees for producing multiple *one-time signatures* [4] associated to a single public key. Since this introduction, a *Merkle tree* [8] has been defined to be a complete binary tree with a k bit value associated to each node such that each interior node value is a one-way function of the node values of its children.

Merkle trees have found many uses in theoretical cryptographic constructions, having been specifically designed so that a leaf value can be verified with respect to a publicly known root value and the *authentication data* of the leaf. This authentication data consists of one node value at each height, where these nodes are the siblings of the nodes on the path connecting the leaf to the root. The *Merkle tree traversal problem* is the task of finding an efficient algorithm to output this authentication data for successive leaves. The trivial solution of storing every node value in memory requires too much space. On the other hand, the approach of computing the authentication nodes on the round they are required will be very expensive for some nodes. The challenge is to conserve both space and computation by amortizing the cost of computing such expensive nodes. Thus, this goal is different from other, more well known, tree traversal problems found in the literature.

In practice, Merkle trees have not been appealing due to the large amount of computation or storage required. However, with more efficient traversal techniques, Merkle trees may once again become more compelling, especially given the advantage that cryptographic constructions based on Merkle trees do not require any number theoretic assumptions.

Our Contribution. We present a Merkle tree-traversal algorithm which has a better space and time complexity than the previously known algorithms. Specifically, to traverse a tree with N leaves, our algorithm requires computation of at most $2\log_2(N)$ elementary operations per round and requires storage of less than $3\ \log_2(N)$ node values. In this analysis, a hash function computation, and a leaf value computation are each counted as a single elementary operation[1]. The improvement over previous traversal algorithms is achieved as a result of a new approach to scheduling the node computations. We also prove that this complexity is optimal in the sense that there can be no Merkle Tree traversal algorithm which requires both less than $O(log(N))$ space and less than $O(log(N))$ space.

History and Related Work. In his original presentation [7], Merkle proposed a straightforward technique to amortize the costs associated with tree traversal. His method requires storage of up to $\log^2(N)/2$ hash values, and computation of about $2\ \log(N)$ hash evaluations per round. This complexity had been conjectured to be optimal.

In [6], an algorithm is presented which allows various time-space trade-offs. A parameter choice which minimizes space requires a maximum storage of about $1.5\ \log^2(N)/\log(\log(N))$ hash values, and requires $2\ \log(N)/\log(\log(N))$ hash evaluations per round. The basic logarithmic space and time algorithm of our paper does not provide for any time-space trade-offs, but our scheduling techniques can be used to enhance the methods of [6].

Other work on tree traversal in the cryptographic literature (e.g. [5]) considers a different type of traversal problem. Related work includes efficient hash chain traversal (e.g [1, 2]). Finally, we remark that because the verifier is indifferent to the technique used to produce the authentication path data, these new traversal techniques apply to many existing constructions.

Applications. The standard application of Merkle trees is to digital signatures [4, 8]. The leaves of such a binary tree may also be used individually for authentication purposes. For example, see TESLA [11]. Other applications include certificate refreshal [9], and micro-payments [3, 12]. Because this algorithm just deals with traversing a binary tree, it's applications need not be restricted to cryptography.

Outline. We begin by presenting the background and standard algorithms of Merkle trees (Section 2). We then introduce some notation and review the classic Merkle traversal algorithm (Section 3). After providing some intuition (Section 4), we present the new algorithm (Section 5). We prove the time and space

---

1 This differs from the measurement of *total* computational cost, which includes, e.g., the scheduling algorithm itself.

bounds in (Section 6), and discuss the optimal asymptotic nature of this result in (Section 7). We conclude with some comments on efficiency enhancements and future work. (Section 8). In the appendix we sketch the proof of the theorem stating that our complexity result is asymptotically optimal.

2 Merkle Trees and Background

The definitions and algorithms in this section are well known, but are useful to precisely explain our traversal algorithm.

Binary Trees. A complete binary tree T is said to have *height* H if it has 2^H leaves, and $2^H - 1$ interior nodes. By labeling each left child node with a "0" and each right child node with a "1", the digits along the path from the root identify each node. Interpreting the string as a binary number, the leaves are naturally indexed by the integers in the range $\{0, 1, \ldots 2^H - 1\}$. The higher the leaf index, the further to the right that leaf is. Leaves are said to have *height* 0, while the *height* of an interior node is the length of the path to a leaf below it. Thus, the root has height H, and below each node at height h, there are 2^h leaves.

Merkle Trees. A Merkle tree is a complete binary tree equipped with a function $hash$ and an assignment, Φ, which maps the set of nodes to the set of k-length strings: $n \mapsto \Phi(n) \in \{0,1\}^k$. For the two child nodes, n_{left} and n_{right}, of any interior node, n_{parent}, the assignment Φ is required to satisfy

$$\Phi(n_{parent}) = hash(\Phi(n_{left})||\Phi(n_{right})). \tag{1}$$

The function $hash$ is a candidate one-way function such as SHA-1 [13].

For each leaf l, the value $\Phi(l)$ may be chosen arbitrarily, and then equation (1) determines the values of all the interior nodes. While choosing arbitrary leaf values $\Phi(l)$ might be feasible for a small tree, a better way is to generate them with a keyed pseudo-random number generator. When the leaf value is the hash of the random number, this number is called a *leaf-preimage*. An application might calculate the leaf values in a more complex way, but we focus on the traversal itself and model a leaf calculation with an oracle $LEAFCALC$, which will produces $\Phi(l)$ at the cost of single computational unit[2].

Authentication Paths. The goal of Merkle tree traversal is the sequential output of the leaf values, with the associated authentication data. For each height $h < H$, we define $Auth_h$ to be the value of the sibling of the height h node on the path from the leaf to the root. The authentication data is then the set $\{Auth_i \,|\, 0 \leq i < H\}$.

The correctness of a leaf value may be verified as follows: It is first hashed together with its sibling $Auth_0$, which, in turn, is hashed together with $Auth_1$,

[2] It is straightforward to adapt the analysis to more expensive leaf value calculations.

etc., all the way up to the root. If the calculated root value is equal to the published root value, then the leaf value is accepted as authentic. Fortunately, when the leaves are naturally ordered from left to right, consecutive leaves typically share a large portion of the authentication data.

Efficiently Computing Nodes. By construction, each interior node value $\Phi(n)$ (also abbreviated Φ_n) is determined from the leaf values below it. The following well known algorithm, which we call *TREEHASH*, conserves space. During the required $2^{h+1}-1$ steps, it stores a maximum of $h+1$ hash values at once. The *TREEHASH* algorithm consolidates node values at the same height before calculating a new leaf, and it is commonly implemented with a stack.

Algorithm 1: TREEHASH (start, maxheight)

1. Set $leaf = start$ and create empty stack.
2. **Consolidate** If top 2 nodes on the stack are equal height:
   - Pop node value Φ_{right} from stack.
   - Pop node value Φ_{left} from stack.
   - Compute $\Phi_{parent} = hash(\Phi_{left}||\Phi_{right})$.
   - If height of Φ_{parent} = maxheight, output Φ_{parent} and stop.
   - Push Φ_{parent} onto the stack.
3. **New Leaf** Otherwise:
   - Compute $\Phi_l = LEAFCALC(leaf)$.
   - Push Φ_l onto stack.
   - Increment $leaf$.
4. Loop to step 2.

Often, multiple instances of *TREEHASH* are integrated into a larger algorithm. To do this, one might define an object with two methods, *initialize*, and *update*. The initialization step simply sets the starting leaf index, and height of the desired output. The update method executes either step 2 or step 3, and modifies the contents of the stack. When it is done, the sole remaining value on the stack is $\Phi(n)$. We call the intermediate values stored in the stack *tail node* values.

3 The Classic Traversal

The challenge of Merkle tree traversal is to ensure that all node values are ready when needed, but are computed in a manner which conserves space and time. To motivate our own algorithm, we first discuss what the average per-round computation is expected to be, and review the classic Merkle tree traversal.

Average Costs. Each node in the tree is eventually part of an authentication path, so one useful measure is the total cost of computing each node value exactly once. There are 2^{H-h} right (respectively, left) nodes at height h, and if computed independently, each costs $2^{h+1}-1$ operations. Rounding up, this is $2^{H+1} = 2N$ operations, or two per round. Adding together the costs for each height h $(0 \leq h < H)$, we expect, on average, $2H = 2\log(N)$ operations per round to be required.

Three Components. As with a digital signature scheme, the tree-traversal algorithms consists of three components: *key generation*, *output*, and *verification*. During key generation, the root of the tree, the first authentication path, and some upcoming authentication node values are computed. The root node value plays the role of a public key, and the leaf values play the role of one-time private keys.

The *output* phase consists of N rounds, one for each leaf. During round $leaf$, the leaf's value, $\Phi(leaf)$ (or leaf pre-image) is output. The authentication path, $\{Auth_i\}$, is also output. Additionally, the algorithm's state is modified in order to prepare for future outputs.

As mentioned above, the *verification* phase is identical to the traditional verification phase for Merkle trees.

Notation. In addition to denoting the current authentication nodes $Auth_h$, we need some notation to describe the stacks used to compute upcoming needed nodes. Define $Stack_h$ to be an object which contains a stack of node values as in the description of $TREEHASH$ above. $Stack_h.initialize$ and $Stack_h.update$ will be methods to setup and incrementally compute $TREEHASH$. Additionally, define $Stack_h.low$ to be the height of the lowest node in $Stack_h$, except in two cases: if the stack is empty $Stack_h.low$ is defined to be h, and if the $TREEHASH$ algorithm has completed $Stack_h.low$ is defined to be ∞.

3.1 Key Generation and Setup

The main task of key generation is to compute and publish the root value. This is a direct application of $TREEHASH$. In the process of this computation, every node value is computed, and, it is important to record the initial values $\{Auth_i\}$, as well as the upcoming values for each of the $\{Auth_i\}$.

If we denote the j'th node at height h by $n_{h,j}$, we have $Auth_h = \Phi(n_{h,1})$ (these are right nodes). The "upcoming" authentication node at height h is $\Phi(n_{h,0})$ (these are left nodes). These node values are used to initialize $Stack_h$ to be in the state of having completed $TREEHASH$.

Algorithm 2: Key-Gen and Setup

1. **Initial Authentication Nodes** For each $i \in \{0, 1, \ldots H-1\}$: Calculate $Auth_i = \Phi(n_{i,1})$.
2. **Initial Next Nodes** For each $i \in \{0, 1, \ldots H-1\}$: Setup $Stack_k$ with the single node value $Auth_i = \Phi(n_{i,1})$.
3. **Public Key** Calculate and publish tree root, $\Phi(root)$.

3.2 Output and Update (Classic)

Merkle's tree traversal algorithm runs one instance of $TREEHASH$ for each height h to compute the next authentication node value for that level. Every 2^h rounds, the authentication path will shift to the right at level h, thus requiring a new node (its sibling) as the height h authentication node.

At each round the $TREEHASH$ state is updated with two units of computation. After 2^h rounds this node value computation will be completed, and a new instance of $TREEHASH$ begins for the next authentication node at that level.

To specify how to refresh the $Auth$ nodes, we observe how to easily determine which heights need updating: height h needs updating if and only if 2^h divides $leaf + 1$ evenly. Furthermore, we note that at round $leaf + 1 + 2^h$, the authentication *path* will pass though the $(leaf + 1 + 2^h)/2^h$'th node at height h. Thus, its *sibling's* value, (the new required upcoming $Auth_h$) is determined from the 2^h leaf values starting from leaf number $(leaf + 1 + 2^h) \oplus 2^h$, where $\oplus$ denotes bitwise XOR.

In this language, we summarize Merkle's classic traversal algorithm.

Algorithm 3: Classic Merkle Tree Traversal

1. Set $leaf = 0$.
2. **Output:**
   - Compute and output $\Phi(leaf)$ with $LEAFCALC(leaf)$.
   - For each $h \in [0, H-1]$ output $\{Auth_h\}$.
3. **Refresh Auth Nodes:**
   For all h such that 2^h divides $leaf + 1$:
   - Set $Auth_h$ be the sole node value in $Stack_h$.
   - Set startnode $= (leaf + 1 + 2^h) \oplus 2^h$.
   - $Stack_k.initialize(startnode, h)$.
4. **Build Stacks:**
   For all $h \in [0, H-1]$:
   - $Stack_h.update(2)$.
5. **Loop:**
   - Set $leaf = leaf + 1$.
   - If $leaf < 2^H$ go to Step 2.

4 Intuition for an Improvement.

Let us make some observations about the classic traversal algorithm. We see that with the classic algorithm above, up to H instances of $TREEHASH$ may be concurrently active, one for each height less than H. One can conceptualize them as H processes running in parallel, each requiring also a certain amount of space for the "tail nodes" of the $TREEHASH$ algorithm, and receiving a budget of two hash value computations per round, clearly enough to complete the $2^{h+1} - 1$ hash computations required over the 2^h available rounds.

Because the stack employed by $TREEHASH$ may contain up to $h+1$ node values, we are only guaranteed a space bound of $1+2+\cdots+N$. The possibility of so many tail nodes is the source of the $\Omega(N^2/2)$ space complexity in the classic algorithm.

Considering that for the larger h, the $TREEHASH$ calculations have many rounds to complete, it appears that it might be wasteful to save so many intermediate nodes at once. Our idea is to schedule the concurrent $TREEHASH$

calculations differently, so that at any given round, the associated stacks are mostly empty. We chose a schedule which generally favors computation of upcoming $Auth_h$ for lower h, (because they are required sooner), but delays beginning of a new $TREEHASH$ instance slightly, waiting until all stacks $\{Stack_i\}$ are partially completed, containing no tail nodes of height less than h.

This delay, was motivated by the observation that in general, if the computation of two nodes at the same height in different $TREEHASH$ stacks are computed serially, rather than in parallel, less space will be used. Informally, we call the delay in starting new stack computations "zipping up the tails". We will need to prove the fact, which is no longer obvious, that the upcoming needed nodes will always be ready in time.

5 The New Traversal Algorithm

In this section we describe the new scheduling algorithm. Comparing to the classic traversal algorithm, the only difference will be in how the budget of $2H$ hash function evaluations will be allocated among the potentially H concurrent $TREEHASH$ processes.

Using the idea of zipping up the tails, there is more than one way to invent a scheduling algorithm which will take advantage of this savings. The one we present here is not optimal, but it is simple to describe. For example, an earlier version of this work presented a more efficient, but more difficult algorithm.

Algorithm 4: Logarithmic Merkle Tree Traversal

1. Set $leaf = 0$.
2. **Output:**
   - Compute and output $\Phi(leaf)$ with $LEAFCALC(leaf)$.
   - For each $h \in [0, H-1]$ output $\{Auth_h\}$.
3. **Refresh Auth Nodes:**
   For all h such that 2^h divides $leaf + 1$:
   - Set $Auth_h$ be the sole node value in $Stack_h$.
   - Set startnode $= (leaf + 1 + 2^h) \oplus 2^h$.
   - $Stack_k.initialize(startnode, h)$.
4. **Build Stacks:**
   Repeat the following $2H - 1$ times:
   - Let l_{min} be the minimum of $\{Stack_h.low\}$.
   - Let $focus$ be the least h so $Stack_h.low = l_{min}$.
   - $Stack_{focus}.update(1)$.
5. **Loop:**
   - Set $leaf = leaf + 1$.
   - If $leaf < 2^H$ go to Step 2.

This version can be concisely described as follows. The upcoming needed authentication nodes are computed as in the classic traversal, but the various stacks do not all receive equal attention. Each $TREEHASH$ instance can be characterized as being either not started, partially completed, or completed. Our schedule to prefers to complete $Stack_h$ for the lowest h values first, *unless another stack has a lower tail node*. We express this preference by defining l_{min} be the minimum of the h values $\{Stack_h.low\}$, then choosing to focus our attention on the smallest level h attaining this minimum. (setting $Stack_h.low = \infty$ for completed stacks effectively skips them over).

In other words, all stacks must be completed to a stage where there are no tail nodes at height h or less before we start a new $Stack_h$ $TREEHASH$ computation. The final algorithm is summarized in the box above.

6 Correctness and Analysis

In this section we show that our computational budget of $2H$ is indeed sufficient to complete every $Stack_h$ computation before it is required as an authentication node. We also show that the space required for hash values is less than $3H$.

6.1 Nodes are Computed on Time.

As presented above, our algorithm allocates exactly a budget of $2H$ computational units per round to spend updating the h stacks. Here, a computational unit is defined to be either a call to *LEAFCALC*, or the computation of a hash value. We do not model any extra expense due to complex leaf calculations.

To prove this, we focus on a given height h, and consider the period starting from the time $Stack_h$ is created and ending at the time when the upcoming authentication node (denoted $Need_h$ here) is required to be completed. This is not immediately clear, due to the complicated scheduling algorithm. Our approach to prove that $Need_h$ is completed on time is to showing that the total budget over this period exceeds the cost of *all* nodes computed within this period which can be computed before $Need_h$.

Node Costs. The node $Need_h$ itself costs only $2^{h+1} - 1$ units, a tractable amount given that there are 2^h rounds between the time $Stack_h$ is created, and the time by which $Need_h$ must be completed. However, a non trivial calculation is required, since in addition to the resources required by $Need_h$, many other nodes compete for the total budget of $2H2^h$ computational units available in this period. These nodes include all the future needed nodes $Need_i$, $(i < h)$, for lower levels, and the 2^N output nodes of Algorithm 4, Step 2. Finally there may be a partial contribution to a node $Need_i$ $i > h$, so that its stack contains no low nodes by the time $Need_h$ is computed.

It is easy to count the number of such needed nodes in the interval, and we know the cost of each one. As for the contributions to higher stacks, we at least know that the cost to raise any low node to height h must be less than $2^{h+1} - 1$ (the total cost of a height h node). We summarize these quantities and costs in the following figure.

Nodes built during 2^h rounds for $Need_h$.

Node Type	Quantity	Cost Each
$Need_h$	1	$2^{h+1}-1$
$Need_{h-1}$	2	2^h-1
...	...	...
$Need_k$	2^{h-k}	$2^{k+1}-1$
...	...	...
$Need_0$	2^h	1
$Output$	2^h	1
$Tail$	1	$\leq 2^{h+1}-2$

We proceed to tally up the total cost incurred during the interval. Notice that the rows beginning $Need_0$ and $Output$ require a total of 2^{h+1} computational units. For ever other row in the node chart, the number of nodes of a given type multiplied by the cost per node is less than 2^{h+1}. There are $h+1$ such rows, so the total cost of all nodes represented in the chart is

$$TotalCost_h < (h+2)2^h.$$

For heights $h \leq H-2$, it is clear that this total cost is less than $2H2^H$. It is also true for the remaining case of $h = H-1$, because there are no tail nodes in this case.

We conclude that, as claimed, the budget of $2H$ units per round is indeed always sufficient to prepare $Need_h$ on time, for any $0 \leq h < H$.

6.2 Space is Bounded by 3H.

Our motivation leading to this relatively complex scheduling is to use as little space as possible. To prove this, we simply add up the quantities of each kind of node. We know there are always H nodes $Auth_h$. Let $C < H$ be the number completed nodes $Next_h$.

$$\#Auth_i + \#Next_i = H + C. \tag{2}$$

We must finally consider the number of tail nodes in the $\{Stack_h\}$. As for these, we observe that since a $Stack_h$ never becomes active until all nodes in "higher" stacks are of height at least h, there can never be two distinct stacks, each containing a node of the same height. Furthermore, recalling algorithm *TREEHASH*, we know there is at most one height for which a stack has two node values. In all, there is at most one tail node at each height ($0 \leq h \leq H-3$), plus up to one additional tail node per non-completed stack. Thus

$$\#Tail \leq H - 2 + (H - C). \tag{3}$$

Adding all types of nodes we obtain:

$$\#Auth_i + \#Next_i + \#Tail \leq 3H - 2. \tag{4}$$

This proves the assertion. There are at most $3H-2$ stored nodes.

7 Asymptotic Optimality Result

An interesting optimality result states that a traversal algorithm can never beat both *time* $= O(\log(N))$ and *space* $= O(\log(N))$. It is clear that at least $H-2$ nodes are required for the TREEHASH algorithm, so our task is essentially to show that if space is limited by any constant multiple of $\log(N)$, then the computational complexity must be $\Omega(\log(N))$. Let us be clear that this theorem does not quantify the constants. Clearly, with greater space, computation time can be reduced.

Theorem 1. *Suppose that there is a Merkle tree traversal algorithm for which the space is bounded by $\alpha\, log(N)$. Then there exists some constant β so that the time required is at least $\beta\, log(N)$.*

The theorem simply states that it is not possible to reduce space complexity below logarithmic without increasing the time complexity beyond logarithmic!

The proof of this technical statement is found in the appendix, but we will briefly describe the approach here. We consider only right nodes for the proof. We divide all right nodes into two groups: those which must be computed (at a cost of $2^{h+1}-1$), and those which have been saved from some earlier calculation. The proof assumes a sub-logarithmic time complexity and derives a contradiction.

The more nodes in the second category, the faster the traversal can go. However, such a large quantity of nodes would be required to be saved in order to reduce the time complexity to sub-logarithmic, that the average number of saved node values would have to exceed a linear amount! The rather technical proof in the appendix uses a certain sequence of subtrees to formulate the contradiction.

8 Efficiency Improvements and Future Work

Halving the required time. The scheduling algorithm we presented above is not optimally efficient. In an earlier version of this paper, we described a technique to half the number of required hash computations per round. The trick was to notice that all of the *left* nodes in the tree could be calculated nearly for free. Unfortunately, this resulted in a more complicated algorithm which is less appealing for a transparent exposition.

Other Variants. A space-time trade-off is the subject of [6]. For our algorithm, clearly a few extra node values stored near the top of the tree will reduce total computation, but there are also other strategies to exploit extra space and save time. For Merkle tree traversal all such approaches are based on the idea that during a node computation (such as that of $Need_i$) saving some wisely chosen set of intermediate node values will avoid their duplicate future recomputation, and thus save time.

Future work. It might be interesting to explicitly combine the idea in this paper with the work in [6]. One might ask the question, for any size tree, what is the least number of hash computations per round which will suffice, if only S hash nodes may be stored at one time.

Perhaps a more interesting direction will be to look for applications for which an efficient Merkle tree traversal would be useful. Because the traversal algorithms are a relatively general construction, applications outside of cryptography might be discovered.

Within cryptography, there is some interest in understanding which constructions would still be possible if no public-key functionality turned out to exist. (For example due to quantum computers). Then the efficiency of a signature scheme based on Merkle tree's would be of practical interest. Finally, in any practical implementation designed to conserve space, it is important to take into consideration the size of the algorithm's code itself.

9 Acknowledgments

The author wishes to thank Markus Jakobsson, and Silvio Micali, and the anonymous reviewers for encouraging discussions and useful comments. The author especially appreciates the reviewer's advice to simplify the algorithm, even for a small cost in efficiency, as it may have helped to improve the presentation of this result, making the paper easier to read.

References

1. D. Coppersmith and M. Jakobsson, "Almost Optimal Hash Sequence Traversal," Financial Crypto '02. Available at `www.markus-jakobsson.com`.
2. M. Jakobsson, "Fractal Hash Sequence Representation and Traversal," ISIT '02, p. 437. Available at `www.markus-jakobsson.com`.
3. C. Jutla and M. Yung, "PayTree: Amortized-Signature for Flexible Micropayments," 2nd USENIX Workshop on Electronic Commerce, pp. 213–221, 1996.
4. L. Lamport, "Constructing Digital Signatures from a One Way Function," SRI International Technical Report CSL-98 (October 1979).
5. H. Lipmaa, "On Optimal Hash Tree Traversal for Interval Time-Stamping," In Proceedings of Information Security Conference 2002, volume 2433 of Lecture Notes in Computer Science, pp. 357–371. Available at `www.tcs.hut.fi/~helger/papers/lip02a/`.
6. M. Jakobsson, T. Leighton, S. Micali, M. Szydlo, "Fractal Merkle Tree Representation and Traversal," In RSA Cryptographers Track, RSA Security Conference 2003.
7. R. Merkle, "Secrecy, Authentication, and Public Key Systems," UMI Research Press, 1982. Also appears as a Stanford Ph.D. thesis in 1979.
8. R. Merkle, "A Digital Signature Based on a Conventional Encryption Function," Proceedings of Crypto '87, pp. 369–378.
9. S. Micali, "Efficient Certificate Revocation," In RSA Cryptographers Track, RSA Security Conference 1997, and U.S. Patent No. 5,666,416.

10. T. Malkin, D. Micciancio, and S. Miner, "Efficient Generic Forward-Secure Signatures With An Unbounded Number Of Time Periods" Proceedings of Eurocrypt '02, pp. 400-417.
11. A. Perrig, R. Canetti, D. Tygar, and D. Song, "The TESLA Broadcast Authentication Protocol," Cryptobytes, Volume 5, No. 2 (RSA Laboratories, Summer/Fall 2002), pp. 2–13. Available at `www.rsasecurity.com/rsalabs/cryptobytes/`.
12. R. Rivest and A. Shamir, "PayWord and MicroMint–Two Simple Micropayment Schemes," CryptoBytes, Volume 2, No. 1 (RSA Laboratories, Spring 1996), pp. 7–11. Available at `www.rsasecurity.com/rsalabs/cryptobytes/`.
13. FIPS PUB 180-1, "Secure Hash Standard, SHA-1". Available at `www.itl.nist.gov/fipspubs/fip180-1.htm`.

A Complexity Proof

We now begin the technical proof of Theorem 1. This will be a proof by contradiction. We assume that the time complexity is sub logarithmic, and show that this is incompatible with the assumption that the space complexity is $O(log(N))$.

Our strategy to produce a contradiction is to find a bound on some linear combination of the average time and the average amount of space consumed.

Notation The theorem is an asymptotic statement, so we will be considering trees of height $H = log(N)$, for large H. We need to consider L *levels* of subtrees of height k, where $kL = H$. Within the main tree, the roots of these subtrees will be at heights $k, 2*k, 3*k \dots H$. We say that the subtree is at *level* i if its root is at height $(i+1)k$. This subtree notation is similar to that used in [6].

Note that we will only need to consider right nodes to complete our argument. Recall that during a complete tree traversal every single right node is eventually output as part of the authentication data. This prompts us to categorize the right nodes in three classes.

1. Those already present after the key generation: *free nodes*.
2. Those explicitly calculated (e.g. with $TREEHASH$): *computed nodes*.
3. Those retained from another node's calculation (e.g from another node's $TREEHASH$): *saved nodes*.

Notice how type 2 nodes require computational effort, whereas type 1 and type 3 nodes require some period of storage. We need further notation to conveniently reason about these nodes. Let a_i denote the number of level i subtrees which contain *at least 1* non-root computed (right) node. Similarly, let b_i denote the number of level i subtrees which contain *zero* computed nodes. Just by counting the total number of level i subtrees we have the relation.

$$a_i + b_i = N/2^{(i+1)k}. \qquad (5)$$

Computational costs Let us tally the cost of some of the computed nodes. There are a_i subtrees containing a node of type 2, which must be of height at least ik. Each such node will cost at least $2^{ik+1} - 1$ operations to compute. Rounding down, we find a simple lower bound for the cost of the nodes at level i.

$$Cost > \Sigma_0^{L-1}(a_i 2^{ik}). \tag{6}$$

Storage costs Let us tally the lifespans of some of the retained nodes. Measuring units of *space* $\times$ *rounds* is natural when considering average space consumed. In general, a saved node, S, results from a calculation of some computed node C, say, located at height h. We know that S has been produced before C is even needed, and S will never become an authentication node before C is discarded. We conclude that such a node S must be therefore be stored in memory for at least 2^h rounds.

Even (most of) the free nodes at height h remain in memory for at least 2^{h+1} rounds. In fact, there can be at most one exception: the first right node at level h.

Now consider one of the b_i subtrees at level i containing only free or stored nodes. Except for the leftmost subtree at each level, which may contain a free node waiting in memory less than $2^{(i+1)k}$ rounds, every other node in this subtree takes up space for at least $2^{(i+1)k}$ rounds. There are $2^k - 1$ nodes in a subtree and thus we find a simple lower bound on the *space* $\times$ *rounds*.

$$Space * Rounds \geq \Sigma_0^{L-1}(b_i - 1)(2^k - 1)2^{(i+1)k}. \tag{7}$$

Note that the $(b_i - 1)$ term reflects the possible omission of the leftmost level i subtree.

Mixed Bounds We can now use simple algebra with Equations (5), (6), and (7) to yield combined bounds. First the cost is related to the b_i, which is then related to a space bound.

$$2^k Cost > \Sigma_0^{L-1} a_i 2^{(i+1)k} = \Sigma_0^{L-1} N - 2^{(i+1)k} b_i. \tag{8}$$

As series of similar algebraic manipulations finally yield (somewhat weaker) very useful bounds.

$$2^k Cost + \Sigma_0^{L-1} 2^{(i+1)k} b_i > NL. \tag{9}$$

$$2^k Cost + \Sigma_0^{L-1} 2^{(i+1)k}/(2^{k-1}) + Space * Rounds./(2^{k-1}) > NL \tag{10}$$

$$2^k Cost + 2N + Space * Rounds/(2^{k-1}) > NL. \tag{11}$$

$$2^k Average\, Cost + Average\, Space/(2^{k-1}) > (L-2) \geq L/2. \tag{12}$$

$$(k\, 2^{k+1}) Average\, Cost + (k/2^{k-2}) Average\, Space > L/2 * 2k = H. \tag{13}$$

This last bound on the sum of average cost and space requirements will allow us to find a contradiction.

Proof by Contradiction Let us assume the opposite of the statement of Theorem 1. Then there is some α such that the space is bounded above by $\alpha\, log(N)$. Secondly, the time complexity is supposed to be sub-logarithmic, so for every small β the time required is less than $\beta\, log(N)$ for sufficiently large N.

With these assumptions we are now able to choose a useful value of k. We pick k to be large enough so that $\alpha > 1/k2^{k+3}$. We also choose β to be less than $1/k2^{k+2}$. With these choices we obtain two relations.

$$(k\, 2^{k+1}) Average\, Cost < H/2. \tag{14}$$

$$(k/2^{k-2}) Average\, Space < H/2. \tag{15}$$

By adding these two last equations, we contradict Equation (13).
QED.