Merkle Tree Traversal in Log Space and Time (2003 Preprint version)

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[1]. For a tree with N nodes, our algorithm computes sequential tree leaves and authentication path data in time $Log_2(N)$ and space less than $3Log_2(N)$, 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. Relative to this algorithm, we show our bounds to be necessary and sufficient. 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, and can also enhance other traversal algorithms which relax space constraints to gain speed.

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

[1] The algorithm in this (unpublished) preprint predates, and differs somewhat from the simplified version published in the Eurocrypt 2004 proceedings (see szydlo.com).

data for successive leaves. 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 $Log_2(N)$ elementary operations per round and requires storage of only $3\,Log_2(N)-2$ node values. In this analysis, a hash function computation, and a leaf value computation are each counted as a single elementary operation. 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, Merkle proposed a straightforward technique [7] for the required tree traversal. This method requires maximum space of $1/2\,Log^2(N)$ units, and maximum computation of $2\,Log(N)-2$ hash evaluations per round. This complexity had been conjectured to be optimal.

In [6], an algorithm is presented which allows various time-space tradeoffs. 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. Other related work includes involves 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 indi-

vidually for authentication purposes. For example, see TESLA [10]. Other applications include certificate refreshal [9], and micro-payments [3, 11].

Outline. We begin by presenting the background and standard algorithms of Merkle trees. In Section 3 we introduce some notation and describe our approach to Merkle tree traversal. The precise algorithm is presented in Section 4, and we prove the associated time and space bounds in Section 5. In the appendix we prove the theorem which states that our complexity result is optimal.

2 Merkle Trees and Background

The definitions and algorithms in this section are well known, but 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 [12].

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 was 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 *LEAF-CALC*, which will produces $\Phi(l)$ at the cost of single computational unit.[2]

Desired Output. 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 $\{Auth_i \mid 0 \leq i < H\}$.

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 $Auth_1$, 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.

Computing Nodes: TREEHASH. By construction, each interior node value $\Phi(n)$ 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 $\Phi_l = LEAF\text{-}CALC(leaf)$.
   - 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

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

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, it outputs the desired value $\Phi(n)$, and destroys the stack. We call the intermediate values stored in the stack *tail node* values.

3 Intuition and Notation

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 give a brief description of the classic $O(n^2)$-space traversal algorithm.

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. For simplicity, in this discussion we will only consider the total computational effort spent on right nodes. There are 2^{H-h} right nodes at height h, and if computed independently, each costs $2^{h+1} - 1$ operations. Rounding up, this is $2^H = N$ operations, or one per round. Adding together the costs for each height h ($0 \leq h < H$), we expect, on average, $H = log(N)$ operations per round to be required.

The Classic Traversal Algorithm. Merkle's tree traversal algorithm runs one instance of $TREEHASH$ for each height h to compute the upcoming right node value. At each round the $TREEHASH$ state is updated with a single unit of computation. Once every 2^{h+1} rounds the node value computation will be completed, and a new instance of $TREEHASH$ begins for the next upcoming right node value.

This simplified description of the classic Merkle tree traversal ignores the left nodes, but it suffices to describe our idea for an improvement.

Intuition for an Improvement. We see that with the classic algorithm described, up to H instances of $TREEHASH$ may be concurrently active: one for each height less than H. Because the stack employed by $TREEHASH$ may contain up to $h+1$ node values, we are only guaranteed a space bound of $1 + 2 + \cdots + N$. We now see that the possibility of so many tail nodes is source of the $O(n^2)$ space complexity in the classic algorithm.

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 favors computation of $Need_h$ for lower

h, but delays beginning of a new $TREEHASH$ instance until all stacks $\{Stack_i\}$ are partially completed, containing no tail nodes of height less than h. This delay, which we informally call "zipping up the tails", results in smaller stacks. It is also true, but not obvious, that the upcoming node values will be ready on time!

Notation. We denote the stored node values according the different roles they play in algorithm. There four kinds of nodes are *authentication*, *history*, *needed* and *tail* nodes.

First, for each height, $0 \leq i < H$, there is an authentication node, $Auth_i$, which is part of the authentication data to be output. Secondly, expired authentication nodes may be retained as a history node, $Keep_i$, for the purpose of speeding up a left node computation. Next, a completed upcoming right node is denoted $Need_i$. Lastly, there may be some intermediate tail node value stored in $Stack_i$, the stack associated to an incomplete *TREEHASH* computation.

Our algorithm consists of $N = 2^H$ rounds, one for each leaf, and we use the index $leaf$ to denote the round number. Because multiple nodes are being calculated concurrently, we use the index *active* to focus on a particular height i.

4 Main Algorithm Description

Three Components. As with a digital signature scheme, our tree-traversal algorithm consists of three components: *key generation*, *output*, and *verification*. During key generation, the root of the tree and first authentication path 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. More intricately, 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.

4.1 Key Generation

The main task of key generation is to compute and publish the root value. This is a direct application of $TREEHASH$. During this computation, however, it is also important to record the initial values $\{Auth_i\}$. If we denote the first right node at height h by $n_{i,1}$, we have $Auth_i = \Phi(n_{i,1})$.

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. **Public Key** Calculate and publish tree root, $\Phi(root)$.

4.2 Output and Update Phase

Each round of the execution phase consists of several stages: *generating an output*, *left node computation*, *releasing space*, *stack creation* and *building needed future nodes*.

Generating an Output. At any round $leaf$, the calculations needed to obtain the authentication path will have been completed. We prove this later in section 5. If the index $leaf$ is even we need to compute $\Phi(leaf)$, otherwise it is already available.

Left Node Computation. For each round exactly one left node L is computed, to supplant one authentication node. If $leaf$ is even, it had just been computed in the previous step. Otherwise, let L be the first left node above the leaf. Letting h be the height of L, it can still be calculated in a single step. This can be done since its left child was contained in the previous output ($Auth_{h-1}$), and its right child had been specifically stored in $Keep_{h-1}$ for this purpose.

The computed value of L is stored in $Auth_h$. Also as part of this step, for each $i < h$ the $Auth_i$ are set to be $Need_i$, as these nodes become the new authentication nodes. We see in Section 5 that the values of $Need_i$ are always computed on time.

Releasing Space. Once an authentication path has been output, certain of the node values $Auth_i$ will never again be needed for a subsequent path, and can be discarded. When L is an interior node, these expired nodes are precisely the nodes which are the descendants of L. Namely, these are $Auth_i$ for $i < h$, and $Keep_{i-1}$. The right sibling of L, will never again be output as part of an authentication path, but may be saved as $Keep_i$, if it is needed to help compute a future left node $Auth_{i+1}$.

Stack Creation. Now that L has replaced $Auth_h$, as the new authentication node, there remain only 2^h further rounds before it will be discarded in favor of a new right node value. Because $Need_h$ has become $Auth_h$, we must mark the next right node at that height (provided one exists) as the new $Need_h$ and begin its calculation. A new $Stack_h$ is created for

its computation with *TREEHASH*. To set up the stack, we need to specify the height h and the beginning $leaf - index$. To set this beginning $leaf - index$, one calculates that the first leaf below $Need_h$ has index $leaf + 1 + 2^{h+1}$.

Building Needed Future Nodes. In this step, certain *TREEHASH* instances are updated to work toward the upcoming right node, $Need_i$. Out of a budget of H computational units, one has been spent on a left node, so we must choose how to apply the rest. The winning index, denoted *active* is chosen to be the index i for which $Stack_i$ has the lowest node. In case of a tie, the lower stack index is chosen, and we count an empty $Stack_h$ as having height h. The fact that a new computation of $Need_h$ is deferred unit all other stacks are "zipped up" to height h is used below in Section 5 to bound the total number of tail nodes.

Algorithm 3: Logarithmic Merkle Tree Traversal

1. Set $leaf = 0$;
2. **Output:** If $leaf$ is even, compute $\Phi(leaf)$ with $LEAF - CALC(leaf)$. Output $\Phi(leaf)$, and authentication data $\{Auth_i\}$, $i = 0, 1 \ldots H-1$.
3. **Release Nodes:** Let L be the current leaf if $leaf$ is even, or its first ancestor which is a left node. Let h be the height of L (equal to the largest h with $2^h \mid (leaf+1)$). Remove certain expired node values below L [a]:
   - Remove all node values $Auth_i$ for $i < (h-1)$.
   - If $h = 0$, record $\Phi(leaf+1) = Auth_0$.
   - if L's parent is a right node, remove L's sibling, $Auth_h$.
   - if L's parent is a left node, set $Keep_h = Auth_h$.
4. **Add Left Node**
   - if $h = 0$, set $Auth_0 = \Phi(leaf)$.
   - if $h > 0$, compute $Auth_h = hash(Auth_{h-1} || Keep_{h-1})$.
   - Remove node values $Auth_{h-1}$ and $Keep_{h-1}$.
   - Copy new lower right nodes: set $Auth_i = Need_i$ for $i < h$.
5. **Add Stack** Create $Stack_h$ at height h, with starting value of *leaf-index*$= leaf + 1 + 2^{h+1}$.
6. **Building Needed Nodes** Do these substeps $H-1$ times:
   - Set *active* to be the stack index with the lowest node. (Choose the lowest such index in case of a tie.)
   - If there no such active stack, break and go to Step 7.
   - Spend 1 unit building $Stack_{active}$, as in *TREEHASH*.
   - If $Stack_{active}$ is complete, put result in $Need_{active}$, and destroy $Stack_{active}$.
7. **Loop to Next Round** Set $leaf = leaf + 1$.
   - If $leaf < 2^H$ go to Step 2.

[a] Note the release of two nodes is be delayed until step 4.

5 Correctness and Analysis

In this section we show that our computational budget of H is sufficient to complete every $Need_h$ computation before it is required as an authentication node. We also show that the space required for hash values does not exceed $3H - 2$. Lastly we show that these bounds are tight for this algorithm.

5.1 Nodes are Computed on Time.

As presented above, our algorithm allocates exactly a budget of H computational units per round: one for a left node computation, and the rest for building node values $Need_i$ with the stacks, $Stack_i$. Here, a computational unit is defined to be either a call to *LEAF-CALC*, or the computation of a hash value. We do not model any extra expense due to complex leaf calculations. Since the left node is always completed with one computational unit each round, the important question to ask is if there is sufficient budget for the right nodes $Need_i$ to be completed on time. To show this is true, 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 $Need_h$ is required to be completed. We show $Need_h$ is completed on time by showing that the total budget over this period exceeds the cost of *all* nodes computed within this period.

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 $Stack_i$ 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 $H2^h$ computational units available in this period. These nodes include the future needed right nodes $Need_i\, i < h$, for lower levels (2), the left node of step 4 (3). Finally there may be a partial contributions to a node $Need_i\, i > h$, (4) so that its stack contain no low nodes.

It is easy to count the number of such needed nodes in the interval, and we know the cost of each such node. As for the contributions to higher stacks, we at least know that 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
1	$Need_h$	1	$2^{h+1}-1$
2	$Need_{h-1}$	1	2^h-1
	$\cdots$	$\cdots$	$\cdots$
	$Need_k$	2^{h-1-k}	$2^{k+1}-1$
	$\cdots$	$\cdots$	$\cdots$
	$Need_0$	2^{h-1}	1
3	L	2^h	1
4	Tail	1	$\leq 2^{h+1}-2$

We proceed to tally up the total cost incurred during the interval. We start with the nodes of type 2 - the many lower needed nodes, $Need_k$. The total is

$$\sum_{k=0}^{h-1}(2^{h-1-k}) \times (2^{k+1}-1) = h\,2^h - \sum_{k=0}^{h-1} 2^{h-1-k} = (h-1)2^h + 1. \qquad (2)$$

Next we add in nodes of type 1, $Need_h$ itself, and the left nodes. The total cost for node types 1, 2, and 3 is

$$(2^{h+1}-1) + (2^{h-1}+1) + (2^h) = (h+2)\,2^h. \qquad (3)$$

Because the maximum computation to ensure any existing tail has no nodes below height h is $2^{h+1}-2$, the maximum possible cost including those of type 4 is

$$MaxCost = (h+4)\,2^h - 2. \qquad (4)$$

For heights $h \leq H-4$, this quantity (4) is clearly less than $H\,2^h$, the total computational budget for the period. So we have proven our assertion that $Need_h$ is completed on time for most cases. We consider the other cases separately in the appendix, where a routine argument essentially shows that there is no expense for tail nodes arising when $h > H-3$.

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

5.2 Space is Bounded by 3H-2.

Our motivation leading to this relatively complex scheduling is to use as little space as possible, so let's prove it! We simply add up the quantities of each kind of node: (1) H nodes $Auth_i$, (2) less than $H-1$ nodes $Keep_i$,

(3) up to $H-1$ nodes $Need_i$, and (4) the tail nodes contained in some $Stack_i$. We begin with the H nodes $Auth_i$.

$$\#Auth_i = N. \tag{5}$$

Next, suppose that there is a saved node value $Keep_{h-1}$. This implies that the left node value at height h (replacing the right node $Auth_h$) has not been computed yet, and so $Stack_h$ so has not even been created yet. This limits the number of nodes $Keep_i$ plus the number of nodes $Need_i$ plus the number of empty stacks as follows.

$$\#Keep_i + \#Need_i \leq H - \#\{Stack_i \neq \emptyset\}. \tag{6}$$

For the tail nodes, 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, and no heights for which a stack has three or more. 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 stack. Thus

$$\#Tail \leq H - 2 + \#\{Stack_i \neq \emptyset\}. \tag{7}$$

Adding Inequalities (5), (6), and (7), we see that

$$\#Auth_i + \#Keep_i + \#Need_i + \#Tail \leq 3H - 2. \tag{8}$$

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

Tightness of Bounds. To check that these bounds are tight for this algorithm, the state at time round $2^{H-1}-2$ is checked. We omit this verification, which simply counts the number of stored nodes, and also shows that with a smaller budget, $Need_{H-2}$ would not be completed on time.

Of course, these bounds only apply to the exact algorithm we present. Indeed, there are many possible logarithmic space and time variations. For example, the technique in [6] may be used to form logarithmic space and time algorithms with a different space-time trade off.

6 Optimality

A more 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 linear amount, then computational complexity must also be at least $O(Log(N))$. 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 just 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 an argument using a certain sequence of subtrees to formulate the contradiction.

7 Future Work

This algorithm can be constructively combined with the work in [6], and applications can be sought for which efficient Merkle tree traversal is useful. An interesting question might be to search for optimal constants for the bounds we have presented. More effective and tight complexity bounds in the spirit of (6) may also be sought. Finally, it is important to take into consideration the size of the code used to implement any traversal algorithm.

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.

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," Proceedings of RSA '97, and U.S. Patent No. 5,666,416.
10. 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/`
11. R. Rivest and A. Shamir, "PayWord and MicroMint–Two Simple Micropayment Schemes," CryptoBytes, volume 2, number 1 (RSA Laboratories, Spring 1996), pp. 7–11. Available at `www.rsasecurity.com/rsalabs/cryptobytes/`
12. FIPS PUB 180-1, "Secure Hash Standard, SHA-1". Available at `www.itl.nist.gov/fipspubs/fip180-1.htm`

APPENDIX

A Time Analysis - Special Cases

In this section, we complete the proof that $Need_h$ is completed on time in the remaining cases, i.e, when $h \geq H - 3$.

- First, there are no needed nodes at height $h \geq H - 1$.
- For height $h = H - 2$ there is a single node appearing as $Need_{H-2}$, and no tails for needed nodes above it, so by equation (3), the cost still less than $H\,2^h$.
- Finally for height $h = H - 3$, there are three nodes that appear as $Need_{H-3}$, required at rounds 2^{H-2}, $2 * 2^{H-2}$, and $3 * 2^{H-2}$, respectively. However, the only possible expense for tail nodes can be for those of $Stack_{H-2}$, which appears at round $2*2^{H-3}$ and is gone by round $4*2^{H-3}$. Thus the first $Need_{H-3}$ is completed before $Stack_{H-2}$ even appears. The second is guaranteed to be completed on time since it is needed at the same time $(2 * 2^{H-2})$ as $Need_{H-2}$, which we know is completed after the second, yet also on time. Lastly the third possible node for $Need_{H-3}$ is computed with a $Stack_{H-3}$ which does not appear until $Need_{H-2}$ is computed. • Thus for $h = H - 3$, the cost again does not include any expense for tail nodes and by equation (3) also less than $H\,2^h$.

B 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 $k, 2 * k, 3 * k \ldots H$. 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.

$$a_i + b_i = N/2^{(i+1)k}. \tag{9}$$

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.

$$Cost > \Sigma_0^{L-1}(a_i 2^{ik}). \tag{10}$$

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 at time 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 us 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 $space \times rounds$.

$$Space * Rounds \geq \Sigma_0^{L-1}(b_i - 1)(2^k - 1)2^{(i+1)k}. \tag{11}$$

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 (9), (10), and (11) to yield combined bounds. First the cost is related to the b_i, which is then related to a space bound.

$$2^k Cost > \Sigma_0^{L-1} a_i 2^{(i+1)k} = \Sigma_0^{L-1} N - 2^{(i+1)k} b_i. \tag{12}$$

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

$$2^k Cost + \Sigma_0^{L-1} 2^{(i+1)k} b_i > NL. \tag{13}$$

$$2^k Cost + \Sigma_0^{L-1} 2^{(i+1)k}/(2^{k-1}) + Space * Rounds./(2^{k-1}) > NL \tag{14}$$

$$2^k Cost + 2N + Space * Rounds/(2^{k-1}) > NL. \tag{15}$$

$$2^k Average\, Cost + Average\, Space/(2^{k-1}) > (L-2) \geq L/2. \tag{16}$$

$$(k\, 2^{k+1}) Average\, Cost + (k/2^{k-2}) Average\, Space > L/2 * 2k = H. \tag{17}$$

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

Proof by Contradiction Let 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.

$$(k\, 2^{k+1}) Average\, Cost < H/2. \tag{18}$$

$$(k/2^{k-2}) Average\, Space < H/2. \tag{19}$$

By adding these two last equations, we contradict Equation (17).
QED.