Fractal Merkle Tree Representation and Traversal

Markus Jakobsson[1], Tom Leighton[2,3], Silvio Micali[3], and Michael Szydlo[1]

[1] RSA Laboratories, Bedford, MA 01730. {mjakobsson, mszydlo}@rsasecurity.com
[2] MIT Laboratory for Computer Science, Cambridge, MA 02139
[3] Akamai Technologies, Cambridge, MA 02142

Abstract. We introduce a technique for traversal of Merkle trees, and propose an efficient algorithm that generates a sequence of leaves along with their associated authentication paths. For one choice of parameters, and a total of N leaves, our technique requires a worst-case computational effort of $2\, log\, N/loglog\, N$ hash function evaluations per output, and a total storage capacity of less than $1.5\, log^2\, N/loglog\, N$ hash values. This is a simultaneous improvement both in space and time complexity over any previously published algorithm.

Keywords: amortization, authentication, fractal, Merkle tree

1 Introduction

A Merkle tree [8] is a tree where the value associated with a node is a one-way function of the values of the node's children. Merkle trees find a wide range of applications within cryptography, due to their simplicity and versatility. For many applications, one wishes to output a sequence of consecutive leaves (or leaf pre-images), along with their "authentication paths" – the latter consists of the interior nodes that constitute the siblings on the path from the leaf to the root. Given an authentication path and a leaf, one can verify the correctness of the latter with respect to the publicly known root value.

However, as elegant as Merkle trees are, they are used less than one might expect. One reason is that known techniques for traversal of trees require a relatively large amount of computation, storage, or both. Such constraints make all but the smallest trees impractical, and in particular not very useful for small and powerless devices [11].

Our Contribution. We propose a technique for traversal of Merkle trees which is structurally very simple and allows for various tradeoffs between storage and computation. For one choice of parameters, the total space required is bounded by $1.5\,log^2 N/loglog\,N$ hash values, and the worst-case computational effort is $2\,log\,N/loglog\,N$ hash function evaluations per output.

It should be noted that the use of our techniques is "transparent" to a verifier, who will not need to know how a set of outputs were generated, but only that they are correct. Therefore, our technique can be employed in any construction for which the generation and output of consecutive leaf pre-images and corresponding authentication paths is required.

Related Work and Applications. Our technique relates to and improves on a previous result by Merkle [7], who proposed a technique for Merkle tree traversal requiring a maximum of $O(log^2 N)$ space and $O(log\,N)$ computation per output, where N is the number of leaves of the tree, and one unit of computation corresponds to one hash function evaluation. (An alternative – but less efficient – method was independently proposed by Vaudenay [15] some ten years later, where *average* instead of *worst-case* costs were considered.) Our improvement is achieved by means of a careful choice of what nodes to compute, retain, and discard at each stage.

Our result also relates to recent methods for fractal traversal of hash chains [3, 1, 14]. The most notable similarities involve the storage and computation requirements and trade-offs, and the fractal scheduling pattern. On the other hand, there are large intrinsic differences between what needs to be computed. For a Merkle tree, the desired outputs are the consecutive authentication paths, while for a hash chain, the only output is a single element. Moreover, while all elements of a hash chain are determined by a *single* starting-value, the leaves of a Merkle tree may be selected independently (via a keyed pseudo-random number generator).

The leaves of the tree may either be used one by one, or many at the same time. The former type of use is well suited for applications such as TESLA [10], certification refreshal [9], wireless security [2], and micro-payments [4, 12], while the latter type finds direct use for Merkle signatures [8, 5]. This partition of applications also corresponds to the birth of the techniques we describe; while the second and third author were motivated by the case relating to Merkle signatures, the first and fourth author focused on the general case.

Outline. We begin by reviewing the goals and standard algorithms of Merkle trees (section 2). We then introduce notation for our subtrees and describe the intuition and tools for their use in our solution (section 3). After that, we describe our technique on a more detailed level (section 4), followed by a correctness and complexity analysis (section 5). A small but technical improvement (section 6) yields our final result, followed by conclusions and ideas for further work (section 7).

2 Merkle Trees and Background

Binary Trees. We first fix notation to describe binary trees. We say that a complete binary tree T has *height* H if it has 2^H leaves, and $2^H - 1$ interior nodes. Each interior node has two children labeled "0" (left), and "1" (right). With this naming convention the leaves are naturally ordered, indexed according to the binary representation of the path from the root to the leaf. Visually, the higher this *leaf index* in $\{0, 1, \dots 2^H - 1\}$ is, the further to the right that leaf is. We define the *altitude* of any node n to be the height of the maximal subtree of T for which it is the root. The node heights range from 0 (leaves) to H (the root). As with the leaves, interior nodes of a given height h_0 may be assigned an index in $\{0, 1, \dots 2^{h_0} - 1\}$.

Merkle trees. A Merkle tree is a binary tree with an assignment of a string to each node: $n \mapsto P(n) \in \{0,1\}^k$, such that the parent's node values are one-way functions of the children's node values.

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

In the above and onwards, *hash* denotes the one-way function; a possible choice of such a function is SHA-1 [13].

The value of a leaf, in turn, is a one-way function of some *leaf pre-image*. For small trees these pre-images may be simply stored; alternatively for larger trees, the leaves may be calculated with a keyed pseudo-random generator. Either way, in this paper we model a leaf calculation with an oracle *LEAF-CALC*, which is assumed to require computation equal in quantity to that of *hash*. The value of the root is considered public, while (to begin with) all the values associated with leaf pre-images are known by the "tree owner" alone.

Desired output. We wish to generate a sequence of outputs, one for each leaf. Each output has two components; (1) a leaf pre-image; and (2) the *authentication path* of the leaf, i.e., the values of all nodes that are siblings of nodes on the path between the leaf in question and the root. This is illustrated in Figure 1. Visiting the leaves according to the natural indexing, (from left to right), has the advantage that usually, $leaf_i$ and $leaf_{i+1}$ share a large portion of their authentication paths.

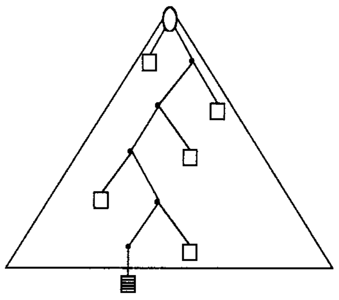

Fig. 1. The circle corresponds to the publicly known root value; the grey square to the current leaf pre-image; and the white squares to the current path siblings. The set of white squares make up the authentication path of the grey square.

In order to *verify* the value of a leaf pre-image, one computes (with Equation 1) the potential values of its ancestors by iterated hashing. A leaf pre-image is accepted as correct if and only if the computed root value is equal to the already known root value.

Digital Signatures. Merkle trees were originally presented as a method to convert a *one-time signature scheme* into a digital signature scheme [8] by using a block of $2k$ leaf pre-images as a one time secret key. The resulting scheme needs only the key to the pseudo random number generator as a secret key, and the root node value as the public key.

Computing nodes: TREEHASH. A well-known technique used with Merkle trees is the use of an algorithm which computes the value $P(n)$ of a height H node, while only storing only up to $H + 1$ hash values. We use several variants of this *TREEHASH* algorithm, and recall this algorithm now to simplify the exposition of our traversal technique. Algorithm *TREEHASH* computes the value of a node n, assuming access to an oracle, *LEAF-CALC*, which returns the value of the leaf node with index $leaf \in \{0, \ldots 2^H - 1\}$. The idea is to compute the leaves sequentially, computing interior nodes whenever possible, and discarding nodes which are no longer needed. The algorithm essentially just stores the node values in a stack[1], and repeatedly applies Equation 1.

[1] The use of a stack to simplify the algorithm description was influenced by recent work on time-stamping [6], which also relates to hash trees.

Algorithm 1: TREEHASH (maxheight)

```
1. Set $leaf = 0$ and create empty stack.
2. Consolidate If top 2 nodes on the stack are at the same height:
  • Pop node value $P_{left}$ from stack.
  • Pop node value $P_{right}$ from stack.
  • Compute $P_{parent} = hash(P_{left} || P_{right})$.
  • If height of $P_{parent}$ = maxheight, output $P_{parent}$ and stop.
  • Push $P_{parent}$ onto the stack.
3. New Leaf Otherwise:
  • Compute $P_{leaf} = LEAF\text{-}CALC(leaf)$.
  • Increment $leaf$ .
4. Loop to step 2.
```

Algorithm *TREEHASH* requires a total of $2^{maxheight} - 1$ computational units for a tree of height *maxheight*, assuming we count *hash* computations and *LEAF-CALC* computations equally. Fortunately, due to the fact that nodes are discarded when no longer needed, *TREEHASH* only requires storage of $maxheight + 1$ hash values at any stage. This is because at most one height may have two pebbles; the rest have at most one each. This bound is important in situations where a larger algorithm computes *TREEHASH* incrementally, applying some number of computational units to the iteration of steps 2 to 4 above, and modifying the state of the algorithm's stack. Such intermediate *pebbles* in the stack are said to comprise the *tail* of the node calculation. For a node n at height h_0, we express this bound as

$$Space\ Tail(n) \leq h_0 + 1. \qquad (2)$$

We describe our uses and variants of *TREEHASH* as needed.

3 Subtree Notation and Intuition

The crux of our algorithm is the selection of which node values to compute and retain at each step of the output algorithm. We describe this selection by using a collection of subtrees of fixed height h. We begin with some notation and then provide the intuition for the algorithm.

3.1 Notation

Starting with a Merkle tree T of height H, we introduce further notation to deal with subtrees. First we choose a *subtree height* $h < H$. We let the altitude of a node n in T be the length of the path from n to a leaf of T (where, therefore, the altitude of a leaf of T is zero). Consider a node n with altitude at least h. We define the h-subtree at n to be the unique subtree in T which has n as its root and which has height h. For simplicity in the suite, we assume h is a divisor of H, and let the ratio, $L = H/h$, be the number of *levels* of subtrees. We say that an *h-subtree at n* is "at level i" when it has altitude ih for some $i \in \{1, 2, \ldots H\}$. For each i, there are 2^{H-ih} such h-subtrees at level i.

We say that a series of h-subtrees $\{Tree_i\}$ $(i = 1 \ldots L)$ is a *stacked series of h-subtrees*, if for all $i < L$ the root of $Tree_i$ is a leaf of $Tree_{i+1}$. We illustrate our subtree notation and provide a visualization of a *stacked series of h-subtrees* in Figure 2.

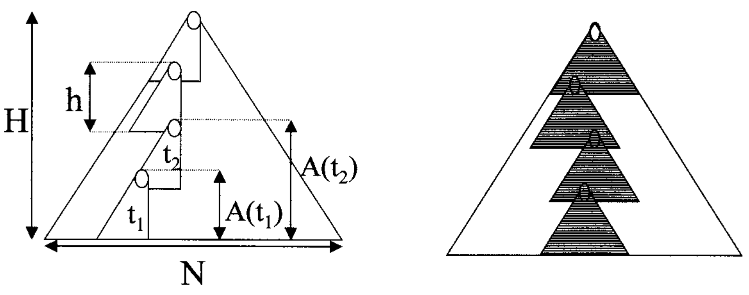

Fig. 2. *(Left)* The height of the Merkle tree is H, and thus, the number of leaves is $N = 2^H$. The height of each subtree is h. The altitude $A(t_1)$ and $A(t_2)$ of the subtrees t_1 and t_2 is marked. *(Right)* Instead of storing all tree nodes, we store a smaller set - those within the stacked subtrees. The leaf whose pre-image will be output next is contained in the lowest-most subtree; the entire authentication path is contained in the stacked set of subtrees.

3.2 Existing and Desired Subtrees

Pebbles. We say that we place a *pebble* on a node n of the tree T when we store the value $P(n)$ associated with this node.

Static view. As previously mentioned, we store some portion of the node values, and update what values are stored over time. Specifically, during any point of the output phase, there will exist a series of stacked *existing* subtrees, as in Figure 2. There are always L such subtrees $Exist_i$ for each $i \in \{1, \ldots L\}$, with pebbles on each of their nodes (except their roots). By design, for any leaf in $Exist_1$, the corresponding authentication path is completely contained in the stacked set of existing subtrees.

Dynamic view. Apart from the above set of *existing* subtrees, which contain the next required authentication path, we will have a set of *desired* subtrees. If the root of the tree $Exist_i$ has index a, according to the ordering of the height-ih nodes, then $Desire_i$ is defined to be the h-subtree with index $a+1$ (provided that $a < 2^{H-i*h} - 1$). In case $a = 2^{H-i*h} - 1$, then $Exist_i$ is the last subtree at this level, and there is no corresponding desired subtree. In particular, there is never a desired subtree at level L. The left part of Figure 3 depicts the adjacent existing and desired subtrees.

As the name suggests, we need to compute the pebbles in the desired subtrees. This is accomplished by adapting an application of Algorithm 2 to the root of $Desire_i$. For these purposes, the algorithm *TREEHASH* is altered to save the pebbles needed for $Desire_i$, rather than discarding them, and secondly to terminate one round early, never actually computing the root. Using this variant of *TREEHASH*, we see that each desired subtree being computed has a *tail* of saved intermediate pebbles as described in Section 2. We depict this dynamic computation in the right part of Figure 3, which shows partially completed subtrees and their associated tails.

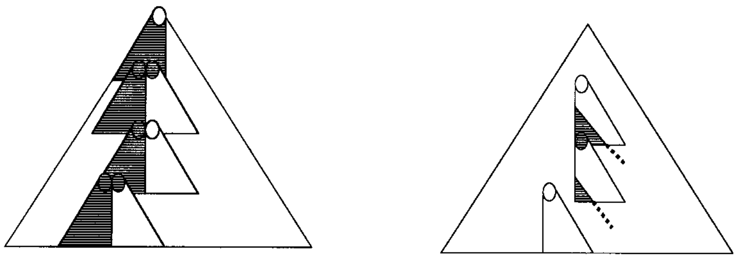

Fig. 3. *(Left)* The grey subtrees correspond to the *existing* subtrees (as in figure 3.1) while the white subtrees correspond to the *desired* subtrees. As the existing subtrees are used up, the desired subtrees are gradually constructed. *(Right)* The figure shows the set of desired subtrees from the previous figure, but with grey portions corresponding to nodes that have been computed and dotted lines corresponding to pebbles in the tail.

3.3 Algorithm Intuition

We now can present intuition for our main algorithm, and explain why the existing subtrees $Exist_i$ will always be available.

Overview. The goal of the traversal is to output the leaf pre-images and authentication paths, sequentially. By design, the existing subtrees should always contain the next authentication path to be output, while the desired subtrees contain more and more completed pebbles with each round, until the existing subtree expires.

When $Exist_i$ is used in an output for the last time, we say that it *dies*. At that time, the adjacent subtree, $Desire_i$ will need to have been completed, i.e., have values assigned to all its nodes but its root (since the latter node is already part of the parent tree.) The tree $Exist_i$ is then *reincarnated* as $Desire_i$: First all the old pebbles of $Exist_i$ are discarded; then the pebbles of $Desire_i$ (and their associated values) taken by $Exist_i$. (Once this occurs, the computation of the new and adjacent subtree $Desire_i$ will be initiated.) This way, if one can ensure that the pebbles on trees $Desire_i$ are always computed on time, one can see that there will always be completed existing subtrees $Exist_i$.

Modifying *TREEHASH*. As mentioned above, our tool used to compute the desired tree is a modified version of the classic *TREEHASH* in Section 2 applied to the root of $Desire_i$. This version differs in that (1) it stops the algorithm one round earlier (thereby skipping the root calculation), and (2) every pebble of height greater than ih is saved into the tree $Desire_i$. For purposes of counting, we won't consider such saved pebbles as part of the tail "proper".

Amortizing the computations. For a particular level i, we recall that the computational cost for tree $Desire_i$ is $2 * 2^{ih} - 2$, as we omit the calculation of the root. At the same time, we know that $Exist_i$ will serve for 2^{ih} output rounds. We amortize the computation of $Desire_i$ over this period, by simply computing two iterations of *TREEHASH* each round. In fact, $Desire_i$ will be ready before it is needed, exactly 1 round in advance!

Thus, for each level, allocating 2 computational units ensures that the desired trees are completed on time. The total computation per round is thus $2(L - 1)$.

4 Solution and Algorithm Presentation

Three phases. We now describe more precisely the main algorithm. There are three phases, the *key generation* phase; the *output* phase; and the *verification* phase. During the key generation phase (which may be performed offline by a relatively powerful computer), the root of the tree is computed and output, taking the role of a public key. Additionally, the iterative output phase needs some setup, namely the computation of pebbles on the initial *existing* subtrees. These are stored on the computer performing the output phase.

The *output* phase consists of a number of rounds. During round j, the (previously unpublished) pre-image the j'th leaf is output, along with its authentication path. In addition, some number of pebbles are discarded and some number of pebbles are computed, in order to prepare for future outputs.

The *verification* phase is identical to the traditional verification phase for Merkle trees and has been described above. We remark again that the outputs our algorithm generates will be indistinguishable from the outputs generated by a traditional algorithm. Therefore, we do not detail the verification phase, but merely the key generation phase and output phase.

4.1 Key Generation

First, the pebbles of the left-most set of stacked *existing* subtrees are computed and stored. Each associated pebble has a *value*, a *position*, and a *height*. In addition, a list of *desired* subtrees is created, one for each level $i < L$, each initialized with an empty stack for use in the modified *TREEHASH* algorithm.

Recalling the indexing of the leaves, indexed by $leaf \in \{0, 1, \ldots N - 1\}$, we initialize a counter $Desire_i.position$ to be 2^{ih}, indicating which Merkle tree leaf is to be computed next

Algorithm 2: Key-Gen and Setup

1. **Initial Subtrees** For each $i \in \{1, 2, \ldots L\}$:
   - Calculate all (non-root) pebbles in existing subtree at level i.
   - Create new empty desired subtree at each level i (except for $i = L$), with leaf *position* initialized to 2^{ih}.
2. **Public Key** Calculate and publish tree root.

4.2 Output and Update Phase

Each round of the execution phase consists of the following portions: *generating an output*, *death and reincarnation of existing subtrees*, and *growing desired subtrees*.

Generating an output. At round j, the output consists of the j'th leaf pre-image, and the authentication path associated to this leaf. The pebbles for this authentication path will be contained in the existing subtrees, and only the pre-image needs to be computed during this round.

Death and reincarnation of existing subtrees. When the last authentication path requiring pebbles from a given existing subtree has been output, then the subtree is no longer useful, and we say that it "*dies*." By then, the corresponding desired subtree has been completed, and the recently died existing subtree "*reincarnates*" as this completed desired subtree. Notice that a new subtree at level i is needed once every 2^{ih} rounds, and so once per 2^{ih} rounds the pebbles in the existing tree are discarded. More technically, at round j, $j = 0 \pmod{2^{ih}}$ the pebbles in the old tree $Exist_i$ are discarded; the completed tree $Desire_i$ becomes the tree new $Exist_i$; and a new, empty desired subtree is created.

Growing desired subtrees. In this step we grow each desired subtree that is not yet completed a little bit. More specifically, we apply two computational units to the new or already started invocations of the *TREEHASH* algorithm. Recall that the counter *position* corresponds to the next leaf to be computed within the *TREEHASH* algorithm, (which is presented starting with index *leaf* starting from 0). We concisely present this algorithm as follows:

Algorithm 3: Stratified Merkle Tree Traversal

```
1. Set $leaf = 0$.
2. **Output** Authentication Path for leaf number $leaf$.
3. **Next Subtree** For each $i$ for which $Exist_i$ is no longer
   needed, i.e, $i \in \{1, 2, \dots L\} \,|\, leaf = 1 (mod\, 2^{hi})$:
 • Remove Pebbles in $Exist_i$.
 • Rename tree $Desire_i$ as tree $Exist_i$.
 • Create new, empty tree $Desire_i$ (if $leaf + 2^{hi} < 2^H$).
4. **Grow Subtrees** For each $i \in \{1, 2, \dots h\}$: Grow tree
   $Desire_i$ by applying 2 units to modified *TREEHASH* (un-
   less $Desire_i$ is completed).
5. Increment $leaf$ and loop back to step 2 (while $leaf < 2^H$).
```

5 Time and Space Analysis

Time. As presented above, our algorithm allocates 2 computational units to each desired subtree. Here, a computational unit is defined to be either a call to *LEAF-CALC*, or the computation of a hash value. Since there are at most $L-1$ desired subtrees, the total computational cost per round is

$$T_{max} = 2(L-1) < 2H/h. \tag{3}$$

Space. The total amount of space required by our algorithm, or equivalently, the number of available pebbles required, may be bounded by simply counting the contributions from (1) the existing subtrees, (2) the desired subtrees, and (3) the tails.

First, there are L existing subtrees and up to $L-1$ desired subtrees, and each of these contains up to $2^{h+1}-2$ pebbles, since we do not store the roots. Additionally, by equation 2, the tail associated to a desired subtree at level $i > 1$ contains at most $h * i + 1$ pebbles. If we count only the pebbles in the tail which do not belong to the desired subtree, then this "proper" tail contains at most $h(i-1)+1$ pebbles. Adding these contributions, we obtain the sum $(2L-1)(2^{h+1}-2) + h\,\Sigma_{i=1}^{L-2}\; i+1$, and thus the bound:

$$Space_{max} \leq (2L-1)(2^{h+1}-2) + L - 2 + h(L-2)(L-1)/2. \tag{4}$$

A marginally worse bound is simpler to write:

$$Space_{max} < 2\,L\,2^{h+1} + H\,L\,/2. \tag{5}$$

Trade-offs. The solution just analyzed presents us with a trade-off between time and space. In general, the larger the subtrees are, the faster the algorithm will run, but the larger the space requirement will be. The parameter affecting the space and time in this trade-off is h; in terms of h the computational cost is below $2H/h$, the space required is bounded above by $2\,L\,2^{h+1} + H\,L/2$. Alternatively, and in terms of h, the space is bounded above by $2\,H\,2^{h+1}/h + H^2/2\,h$.

Low Space Solution. If one is interested in parameters requiring little space, there is an optimal h, due to the fact that for very small h, the number of tail pebbles increases significantly (when $H^2/2h$ becomes large). An approximation of this value is $h = logH$. One could find the exact value by differentiating the expression for the space: $2\,H\,2^{h+1}/h+H^2/2\,h$. For this choice of h = log H = loglog N, we obtain

$$T_{max} = 2\,log\,N/loglog\,N. \tag{6}$$

$$Space_{max} \leq 5/2\,log^2\,N/loglog\,N. \tag{7}$$

These results are interesting because they asymptotically improve Merkle's result with respect to both space and time. Merkle's approach required $T_{max} = 2\,log\,N$ and $Space_{max} approximately 1/2\,log^2\,N$.

We now return to our main algorithm, and explain how a small technical modification will improve the constants in the space bound, ultimately yielding the result presented in the introduction.

6 Additional Savings

Although this modification does not affect the complexity *class* of either the space or time costs, it is of practical interest as it nearly halves the space bound in certain cases. It is presented after the main exposition in order to retain the original simplicity, as this analysis is slightly more technical. The modification is based on two observations: (1) There may be pebbles in existing subtrees which are no longer useful, and (2) The desired subtrees are always in a state of partial completion. In fact, we have found that pebbles in the an existing subtree may be discarded *nearly* as fast as pebbles are entered into the corresponding desired subtree. The modifications are as follows:

1. Discard pebbles in the trees $Exist_i$ as soon as they will never again be required.
2. Omit the *first* application of 2 units to the modified *TREEHASH* algorithm.

We note that with the second modification, the desired subtrees still complete, just in time. With these small changes, for all levels $i < L$, the number of pebbles contained in *both* $Exist_i$, and $Desire_i$ can be bounded by the following expression.

$$Space_{Exist(i)} + Space_{Desire(i)} \leq 2^{ih+1} - 2 + (h-2). \tag{8}$$

This is nearly half of the previous bound of $2*(2^{ih+1}-2)$. We relegate the technical proof of this bound to the appendix, but do remark here that the quantity $h-2$ measures the maximum number of pebbles contained in $Desire_i$ *exceeding* the number of pebbles contained in $Exist_i$ which have been discarded. Using the estimate (8), we revise the space bound computed in the previous section to be

$$Space_{max} <= (L)(2^{h+1}-2)+(L-1)(h-2)+L-2+h(L-2)(L-1)/2. \quad (9)$$

We again round this up to obtain a simpler bound.

$$Space_{max} < L\, 2^{h+1} H\, L\, /2. \quad (10)$$

Specializing to the choice $h = loglog\, N$, we improve the above result to

$$Space_{max} \leq 3/2\, log^2\, N/loglog\, N. \quad (11)$$

by reducing the constant from 5/2 to 3/2.

7 Future Work

While general hash functions allow for arbitrary input sizes, Merkle trees use very particular input sizes. In particular, the one-way function we need for interior nodes is two-to-one; and that used to compute leaf values from their respective pre-images is one-to-one. If special-purpose functions are developed for these purposes, this may improve the efficiency of the resulting algorithm, most notably by avoiding the overhead associated with the initial padding of short inputs, as is performed by standard hash functions. Moreover, if these special-purpose functions are derived from particular primitives, such as AES, it would be possible to prove the security of the resulting Merkle-based scheme on the assumed hardness of the primitive.

Acknowledgments

We wish to thank Ari Juels for providing helpful feedback towards 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. Y.-C. Hu, A. Perrig, and D.B. Johnson, "Packet Leashes: A Defense against Wormhole Attacks in Wireless Ad Hoc Networks," Proceedings of the Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2003), IEEE, San Francisco, CA, April 2003, to appear

3. M. Jakobsson, “Fractal Hash Sequence Representation and Traversal,” ISIT ’02, p. 437. Available at `www.markus-jakobsson.com`.
4. C. Jutla and M. Yung, “PayTree: amortized-signature for flexible micropayments,” 2nd USENIX Workshop on Electronic Commerce, pp. 213–221, 1996.
5. L. Lamport, “Constructing Digital Signatures from a One Way Function,” SRI International Technical Report CSL-98 (October 1979).
6. 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/`
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. K. S. J. Pister, J. M. Kahn and B. E. Boser, “Smart Dust: Wireless Networks of Millimeter-Scale Sensor Nodes. Highlight Article in 1999 Electronics Research Laboratory Research Summary.”, 1999. Available at `robotics.eecs.berkeley.edu/˜pister/SmartDust/`
12. 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/`
13. FIPS PUB 180-1, “Secure Hash Standard, SHA-1”. Available at `www.itl.nist.gov/fipspubs/fip180-1.htm`
14. Y. Sella, “On the Computation-Storage Trade-offs of Hash Chain Traversal,” To appear in Financial Crypto ’03.
15. S. Vaudenay, “One-time identification with low memory,” EUROCODE’92, CISM Course and Lecture 339, pp. 217-228, Springer-Verlag 1993

A Proof of Space Bound

Here we prove assertion (6) which states for any level i the number of pebbles in the $Exist_i$ plus the number of pebbles in the $Desire_i$ is less than $2*2^h-2+(h-2)$. This basic observation reflects the fact that that desired subtree can grow only slightly faster than the existing subtree shrinks. Without loss of generality, in order to simplify the exposition, we do not specify the subtree indices, and restrict our attention to the first existing-desired subtree pair at a given level i.

Returned Pebbles. The first modification ensures that pebbles are returned more continuously than previously, so we quantify this. Subtree $Exist_i$, has 2^h leaves, and as each leaf is no longer required, neither may be some interior nodes above it. These leaves are finished at rounds $2^{(i-1)h}a-1$ for $a \in \{1, \ldots 2^h\}$. We may determine the number of pebbles returned at these times by observing that a leaf is returned every single round, a pebble at height $i\,h+1$ every two rounds, one at height $i\,h+2$ every four rounds, etc. We are interested in the number returned at all times *up to* the time $2^{(i-1)}ha-1$; this is the sum of the greatest integer functions:

$$A + [A/2] + [A/4] + [A/8] + \ldots + [A/2^h]$$

Writing a in binary notation $a = a_0 + 2^1 a_1 + 2^2 a_2 + \ldots 2^h a_h$, this sum is also

$$a_0(2^1-1) + a_1 * (2^2-1) + a_2 * (2^3-1) + \ldots a_h(2^{h+1}-1).$$

New Pebbles. The cost to calculate the corresponding pebbles in $Desire_i$ may also be calculated with a similar expression. Using the fact that a height h_0 node needs $2^{h_0+1}-1$ units to compute, we see that the desired subtree requires

$$a_0(2^{(i-1)h+1}-1) + a_1(2 * 2^{(i-1)h+2}-1) + \ldots a_h(2 * 2^{ih+1}-1)$$

computational units to place those same pebbles. This cost is equal to $2 * 2^{i-1h} * a - z$, where z denotes the number of nonzero digits in the binary expansion of a.

Difference. At time $2^{(i-1)\,h}a-1$, a total of $2 * 2^{(i-1)\,h}a-2$ units of computation has been applied to $Desire_i$, (factoring in our 1 round delay). Noting that $2^{(i-1)\,h}-1$ more rounds may pass before $Exist_i$ loses any more pebbles, we see that the maximal number of pebbles during this interval must be realized at the very end of this interval. At this point in time, the desired subtree has computed exactly the pebbles that have been removed from the existing tree, plus whatever additional pebbles it can compute with its remaining $2*2^{ih}-2+z-2$ computational units. The next pebble, (a leaf) costs $2*2^{ih}-1$ which leaves $z-3$ computational units. Even if all of these units result in new pebbles, the total extra is still less than or equal to $1+z-3$. Since $z \leq h$, this number of extra pebbles is bounded by $h-2$, as claimed, and Equation 8 is proved.