On Optimal Hash Tree Traversal for Interval Time-Stamping

Helger Lipmaa

Laboratory for Theoretical Computer Science
Department of Computer Science and Engineering
Helsinki University of Technology
P.O.Box 5400, FI-02015 HUT, Espoo, Finland
{helger}@tcs.hut.fi

Abstract. Skewed trees constitute a two-parameter family of recursively constructed trees. Recently, Willemson proved that suitably picked skewed trees are space-optimal for interval time-stamping. At the same time, Willemson proposed a practical but suboptimal algorithm for nonrecursive traversal of skewed trees. We describe an alternative, extremely efficient traversal algorithm for skewed trees. The new algorithm is surprisingly simple and arguably close to optimal in every imaginable sense. We provide a detailed analysis of the average-case storage (and communication) complexity of our algorithm, by using the Laplace's method for estimating the asymptotic behavior of integrals. Since the skewed trees can be seen as a natural generalization of Fibonacci trees, our results might also be interesting in other fields of computer science.

Keywords: analysis of algorithms, implementation complexity, interval time-stamping, Laplace's method for integrals, tree traversal.

1 Introduction

Hash trees were originally introduced by Merkle in [Mer80]. Since then, hash trees together with their generalization to arbitrary graphs have been used in many different application areas of cryptography. The hash tree paradigm is very flexible since one has a large freedom of choosing the trees that are the "best" for a particular area.

Cryptographic time-stamping is a prime example area where a large amount of work has been done to devise graph families that are "optimal" in some well-defined sense [BLLV98,BL98,BLS00]. In particular, Buldas, Lipmaa and Schoenmakers proved in [BLS00] that the family of complete binary trees is optimal in the "usual" time-stamping scenario.

The kind of time-stamping where one must show that a certain document was stamped during some interval of time is called *interval time-stamping*. It has been argued informally for a long time that interval time-stamping is an important flavor of time-stamping, necessary to establish whether a certain document was signed during the period of validity of the signing key. (See, for example, [BLLV98].)

Recently, Buldas and Willemson [Wil02b] have described a two-parameter family of trees $\mathfrak{S}(d, w)$ that we call the *skewed (hash) trees*. This family is constructed recursively, so that $\mathfrak{S}(d, w)$ is constructed from the trees $\mathfrak{S}(d-1, w-1)$ and $\mathfrak{S}(d-1, w)$. Willemson [Wil02b] gave an elegant combinatorial proof that a suitable chosen subfamily of $\mathfrak{S}(d, w)$ results in about 1.4 times shorter interval time-stamps than the family of complete binary trees $\mathfrak{T}(d)$ that is optimal in the non-interval scenario [BLS00]. Willemson also proved that this subfamily is optimal in this sense.

But (there is always a but), to be practically applicable, a family of hash trees must have an efficient graph traversal algorithm. The family of complete binary trees has an almost trivial traversal algorithm. Existence of such an algorithm, however, is caused by the extremely simple constitution of complete binary trees. A priori it is not clear at all that, given some tree family, there is a similar efficient algorithm that takes the tree parameters as inputs and thereafter performs the functionality of hash traversal.

While the skewed trees promise short time stamps, their recursive construction does not seem to make them well-suited for nonrecursive graph traversal that is actually used in applications. Willemson proposed in [Wil02b] a very interesting alternative combinatorial interpretation of skewed hash trees and described a traversal algorithm, based on this. However, the resulting algorithm is quite complicated. Moreover, its complexity forced Willemson actually to suggest that one would use a certain suboptimal subfamily $\{\mathfrak{S}(d, w)\}$ of skewed trees. Even in the latter case, during Willemson's algorithm one must store $\ell = \min(d+1, w+1)$ counters and ℓ hash values that amounts in the overall storage requirement of $(1 + o(1))\ell \cdot (\log_2 d + k)$ bits, where k is the output length of the employed hash function.

We propose a more efficient algorithm (Algorithm 1 on page 7) for traversing the whole family $\mathfrak{S}(d, w)$. We feel that such a generality is very important since this makes it very easy to switch to an arbitrary skewed tree very quickly, if needed. In particular, Algorithm 1 works for the subfamily of optimal skewed trees. During Algorithm 1 it is only necessary to store up to d hash values at every time moment and it is only necessary to do up to d hash computations every time when a new document is stamped. Both values are clearly the best that can be achieved when one is concerned about the worst-case.

Our algorithm is surprisingly simple. Its construction depends on somewhat subtle properties of well-known functions like addition, doubling and Hamming weight (population count). However, possession of an elegant algorithm is not a specific property of the skewed hash trees: As we will show later, one can derive as elegant, though a differently-looking, implementation for mirrored skewed trees. Both algorithms, when reduced to work on family $\{\mathfrak{T}(d)\}_d$ give rise to the well-known algorithm for traversing the complete binary trees. Based on this (may be slightly surprising) fact we conjecture that our approach works for many different families of recursively constructed tree families.

Our pseudocode implementation of the algorithm is very clean (the pseudocode for the *update* operation consists of five lines), and therefore potentially interesting by itself. Existence of a clean pseudocode implementation is a very desirable property of algorithms since it potentially reduces the work needed to debug real-life implementations. We hope that due to the cleanliness, our algorithm is almost immune to implementation errors. More precisely, between the invocations, our example implementation maintains a small stack, accessed only by the **push** and **pop** operations, and an additional d-bit counter. The stack consists solely of the hash values, necessary for incremental computing of the hash root; the counter encodes information about the current location in tree.

We provide a complexity analysis of our algorithm and argue that our storage complexity is almost optimal. In fact, the computational work done while traversing an arbitrary skewed hash tree is only slightly more complex than the computational work that must be done in the special case of complete binary trees. Thus we show that this very important flavor of time-stamping, the interval time-stamping, is almost as practical as the usual time-stamping were one must only prove that a document was stamped before a certain event. We think that the double-simplicity of skewed trees (the simplicity of its recursive construction and the simplicity of the nonrecursive traversal algorithm) is yet another witness of the immeasurable beauty and effectiveness of mathematics.

The most elaborated part of the analysis deals with the average-case storage complexity. (Which, by virtue of the algorithm, is practically equal to the average-case communication complexity.) By using the Laplace's method [dB82] for estimating the asymptotic behavior of integrals, we develop a surprisingly precise asymptotic approximation to the average-case storage complexity of our algorithm. For example, we find that the average-case storage is maximized when $w \approx \sqrt{d}$, then being $\approx (d - \sqrt{d})k + d$, where k is the output value of the employed hash function.

We hope that Lagrange's method could also be useful for analysing other cryptographic algorithms. We also hope that the skewed trees, being a simple generalization of Fibonacci trees, are of independent interest in the theory of data structures and other areas of computer science.

Road-map. In Section 2 we will introduce the kind reader to preliminaries. In particular, in Section 2.3 we will give a short description of the graph family $\mathfrak{S}(d, w)$ together with an explanation of its use in time-stamping. In Section 3 we describe our algorithm and give a proof of its correctness. In Section 4 we analyze the efficiency of our algorithm.

2 Preliminaries

2.1 Tree Traversal

By (hash) tree traversal, we mean the next problem: Given a tree with leaves $x_1, \ldots, x_n$ that has a prespecified shape, one must incrementally compute the hash of the root of the tree, as the leaf values $x_0, x_1, \ldots, x_{n-1}$ arrive, in this order, from an external source. A canonical example application area is time-stamping [HS91,BLLV98,Lip99], where the time-stamping authority outputs the root hash (the "round stamp") of a predefined authentication graph $\mathfrak{G}$ after the documents $x_0, \ldots, x_{n-1}$ have arrived sequentially from several clients. The documents x_i are positioned at the leaves of $\mathfrak{G}$ from left to right. Moreover, after the ith document has been inserted to the tree, the time-stamping authority TSA must immediately return to the client a so called *freshness token* (often called the *head* of a stamp) that consists of the minimal amount of information that proves that the ith stamp is dependent of all previous ones. Immediate return is necessary to prevent the authority from cheating.

If the latter requirement of immediate return could be omitted, one could always use a trivial traversal algorithm where the authority constructs the graph G post factum, after arrival of the last document, and only then returns the freshness tokens to every client. However, as already mentioned, this would not be an acceptable solution due to the security reasons.

There is also an efficiency reason. Since documents can arrive to the time-stamping authority from very different sources, it is well possible that at some point they arrive in large bursts. To guarantee that the TSA continues to work under such extreme situations, one must take care that the *maximal* workload of the server would be minimized. Here, the workload includes, in particular, the cost of updating the contents of memory, so as to make it possible to output the root hash without any extra delay after the last document arrives.

Other practical requirements for hash tree traversal include minimal amount of the memory usage at every time moment. Since the potential number of documents stamped during one round is huge (say, in order of 2^{48}), it is clearly impractical to store n elements, or to do n hashing, at any given time. Instead, one would like to upper bound both the number of hashings done when a document arrives and the maximum amount of memory that is used to store the hash values at any given time moment, by sequentially computing the hash values G after arrival of every document.

Last but not least, it is desirable for an algorithm to have minimal implementation complexity: That is, it must be short in description (to fit in low-memory devices) and easily implementable (to minimize implementation errors).

2.2 Interval Time-Stamping

The main goal of interval time-stamping is to enable one to prove that a document was stamped in a finite interval: That is, later than some other document but before than yet another document. One of the original motivations for interval time-stamping was probably first written down in [BLS00], where the next argument was given.

Namely, in many slightly different schemes it is assumed that the time-stamping authority (TSA) links submitted time-stamps by using some linkage-based authentication graph [HS91,BdM91,BHS92,BLLV98,Lip99,BLS00]. However, this approach is often criticized for the following reason. Let $H_1, \ldots, H_n$ be the stamping requests during a round. Before “officially” closing the round, the TSA may issue additional time stamps for H_n, H_{n-1}, $\ldots$, H_1 in reverse order. After that the TSA is able, for any pair of stamps of H_i and H_j, to present proofs for the statements “H_i was stamped before H_j” and “H_j was stamped before H_i”. Therefore, a critical reader might think that using linking schemes and signed receipts does not give any advantage compared to simple hash-and-sign scheme.

Interval time-stamping allows one to alleviate the situation as follows. First, a client requests a stamp L_n of a nonce from the TSA. Subsequently, she stamps her signature σ on the actual document X and L_n. The TSA may then issue additional stamps for X and σ. However, no one is able to back-date new documents relative to σ without

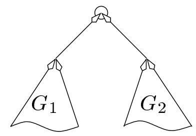

Fig. 1. The composed tree $\mathfrak{G}_1 \otimes \mathfrak{G}_2$

cooperating with the signer. This example shows that interval time-stamping is fundamentally useful in certificate management since without it, one cannot really avoid the reordering attacks. We refer to [Wil02b] for more examples.

The given example can be seen as an intuition, that an interval time-stamp consists of two separate parts, or more precisely, that an interval time-stamp is the union of two stamps: of the stamp of the nonce (*the freshness token*), and of the stamp on (X, σ) (*the existence token*). As shown in [Wil02b], to optimize the size of an interval time stamp, one must choose a tree G where the maximal sum of the lengths of freshness token and existence tokens is minimal. In graph-theoretic terms, the ith freshness token is defined as a (nonunique) set S of nodes with minimal cardinality, such that every leaf $j < i$ of G has some set in S as its parent. The ith existence token is defined as a set S of nodes with minimal cardinality such that the root hash can be "computed" from $S \cup \{i\}$ but not from S alone. In the next subsection we describe the concrete tree family that has been proven to provide minimal time stamp lengths.

2.3 Definition of the Skewed Trees

We will assume throughout this paper that d and w are nonnegative integers, used to parametrize different graph families. Define the family $\mathfrak{S}(d, w)$ (that we call *skewed trees*) of directed rooted trees recursively as follows [Wil02b]:

1. Set $\mathfrak{S}(0, x) = \mathfrak{S}(x, 0) = I$ for an arbitrary integer $x \geq 0$, where I is the singleton graph.
2. If $d, w \geq 1$ then set $\mathfrak{S}(d, w) = \mathfrak{S}(d-1, w-1) \otimes \mathfrak{S}(d-1, w)$. Here, the result of the tree composition operation $\mathfrak{G}_1 \otimes \mathfrak{G}_2$ is defined as binary directed tree with a root that has two subtrees: $\mathfrak{G}_1$ being the left subtree, and $\mathfrak{G}_2$ being the right subtree. (See Fig. 1.)

This composition operator is a standard tool of constructing the tree families. For example, the family $\{\mathfrak{T}(d)\}$ of (directed) complete binary trees is constructed by $\mathfrak{T}(d+1) = \mathfrak{T}(d) \otimes \mathfrak{T}(d)$ and $\mathfrak{T}(0) = I$. (As a simple consequence, $\mathfrak{T}(d)$ has 2^d leaves, and $\mathfrak{T}(d) = \mathfrak{S}(d, d)$.) The family $\{\mathfrak{F}(d)\}$ of Fibonacci trees is constructed by letting $\mathfrak{F}(d) = \mathfrak{F}(d-1) \otimes \mathfrak{F}(d-2)$ and $\mathfrak{F}(0) = \mathfrak{F}(1) = I$. For Fibonacci trees, $d \approx c_f \log(n)$, where n is the number of leaves in $\mathfrak{F}(d)$ and $c_f = 1/\log_2((1 + \sqrt{5})/2) \approx 1.44$.

If the complete binary tree is used for interval time-stamping then the worst-case length of the interval time-stamp is $c \log_2 n$, where $c = 2$ [Wil02b]. This corresponds to the case of family $\{\mathfrak{S}(d, w)\}_d$ with $\alpha := w/d = 1$. Setting $d = 2w + 1$ (that is, $\alpha = 2 + o(1)$) improves the value of the constant to $c = 3/2 + o(1)$. However, the

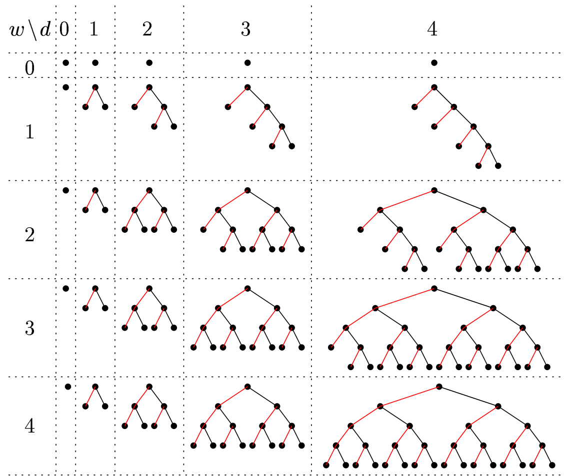

Fig. 2. The trees $\mathfrak{S}(d, w)$ for small d, w

asymptotically optimal (and already familiar!) constant $c = c_f \approx 1.44$ is obtained by setting $\alpha := w/d \approx (3 - \sqrt{5})/2$ [Wil02b].

Therefore, if short time-stamps are desired in interval time-stamping, one has to traverse trees from the family $\mathfrak{S}(d, w)$ with particular values of d and w for which $\mathfrak{S}(d, w)$ is quite different from the complete binary trees $\mathfrak{T}(d) = \mathfrak{S}(d, d)$. One would like to have a general algorithm for traversing trees from this family that would satisfy the desiderata of Section 2.1. This is the task solved in the next section.

Finally, $\mathfrak{S}(d, w) = \mathfrak{S}(d, d)$ whenever $d > w$. Some skewed graphs are depicted by Fig. 2. It is interesting to observe some similarities and differences between the Fibonacci and the skewed trees. First, the recurrent rule for constructing the families is deceivingly similar: $\mathfrak{F}(d)$ is constructed by composing two previous Fibonacci trees while $\mathfrak{S}(d, w)$ is constructed by composing two last skewed trees from "column" $d - 1$. The Fibonacci trees are obtained as a solution to a minimum problem (they are minimum-node AVL trees for given height [Knu98]), while the optimal skewed trees are a solution to a maximum problem (they are maximum-node trees for given sum of width and height) [Bul02]. The reason why the constant c_f pops out in both cases is, while clear mathematically, still an intellectual mystery.

3 Algorithm for Traversing the Tree Family

A highly optimized algorithm for traversing the family $(\mathfrak{S}(d, w))$ of skewed trees is depicted by Algorithm 1. It consists of two procedures. The first, *initialization*, inputs the parameters d and w. It is only invoked once, during the set-up. The second, *update*, gets as input the ith data item x_i (that is, the label of the ith leaf). We will first describe the used notation and after that explain the main ideas behind this algorithm. This includes a detailed correctness proof.

Algorithm 1 An algorithm for traversing $\mathfrak{S}(d, w)$. After *update*, contents of the stack is returned as the freshness token

funct *initialization*(d, w) $\equiv$
  state $:= 0$;
  if $w > d$ **then** $w = d$ **fi**;
  Store w, d;
  Create an empty stack of maximum possible size w.
funct *update*$($*value*$)$ $\equiv$
  if *state* $\geq 2^d$ **then** **return**(*Error*!); **fi**
  push(*value*);
  for $i := 1$ **to** $\mathtt{ntz}(\mathit{state} + 1)$ **do**
    push(H(**pop**(), **pop**())); **end**
  state $:=$ *state* $+ (1 \lll (\overline{w}_h(\mathit{state}) \dot{-} w))$.

3.1 Notation

We work in the RAM model where the word length is not smaller than $N = \max(w, d)$. In practice it could mean using, say, 64-bit integers. Our algorithms the Boolean bitwise AND "$x \wedge y$" and the left shift $x \lll y = x \cdot 2^y$. Common arithmetics is implicitly done modulo 2^N but it can be sometimes done modulo d or modulo w. We denote the proper subtraction, that is well-known from the theory of primitive recursive functions, by $a \dot{-} b$, that is, $a \dot{-} b = a - b$ when $a \geq b$, and $a \dot{-} b = 0$ when $b > a$. Let w_h be the Hamming weight function; that is, $w_h(x)$ is equal to the number of one-bits in the binary presentation of x. (For example, $w_h(17) = 2$ and $w_h(0011) = 2$.) Let $\overline{w}_h(e) := |e| - w_h(e)$ be the number of zeros in the binary encoding of e; for example, $\overline{w}_h(01101) = 2$. Let $\mathtt{ntz}(x)$ be the number of trailing zeros of x; that is, $\mathtt{ntz}(x) = k$ iff $2^k \mid x$ but $2^{k+1} \nmid x$. For example, $\mathtt{ntz}(48) = 4$ and $\mathtt{ntz}(0000) = 4$. Both functions w_h (and thus, the function $\overline{w}_h$) and ntz can be computed in time $\log_2 w$.

Several newer processors, including the Pentium MMX, have a special instruction for proper subtraction. An efficient $O(\log_2 N)$-time algorithm for w_h has been described, say, in [LM01]. The function ntz can then be computed in time $O(\log_2 N)$ as $\mathtt{ntz}(x) := w_h(x - (x \wedge (x - 1)) - 1)$. Many common processors have special instructions for both w_h (see [LM01] for a recent overview) and ntz (the instruction bsf on the Pentium, for example). This basically means that all nontrivial instructions of Algorithm 1 (namely, w_h, ntz and $\dot{-}$) can be seen as primitive in modern microprocessors.

We let $\alpha \mathbin{+\!\!+} \beta$ to denote the concatenation of binary strings α and β. We use a collision-resistant hash function H [Dam87]. We assume implicitly that the left argument of H is evaluated (that is, popped from the stack) earlier than the right argument of H.

3.2 Variables

Algorithm 1 uses a stack that has maximum size w, and is initially empty. The top element of the stack can be removed by using function **pop**(), and an element x can be inserted to the stack by using function **push**(x). There are no other means to access the contents of stack. Observe that like always, a stack implementation can be replaced with a possibly more efficient but less modular array implementation.

Intuitively, after an *update*, the stack contains the hash values of left children of root path that starts from the leaf that corresponds to the lastly arrived document. This set of hash values coincides with the ith freshness token [Wil02b] and therefore the whole contents of the stack must be returned to the ith client after the *update* function.

Except from the stack and parameters w and d, Algorithm 1 maintains only one internal value, *state*. Intuitively, *state* represents binary "Huffman" encoding of the location of the next leaf, labeled by say x_n, in the binary tree $\mathfrak{S}(d, w)$. Namely, by following the path from this leaf to the root, in every step one starts either from the left or from the right child and follows an arc to its parent node. Let X^i denote a run of i X-s, where i can be 0. If we encode left by $L = 0$ and right by $R = 1$, the path that starts from the root and ends with an arbitrary vertex v can be encoded as $\mathsf{enc}(v) := L^{a_d} \mathbin{+\!\!+} R^{b_d} \mathbin{+\!\!+} L^{a_{d-1}} \mathbin{+\!\!+} R^{b_{d-1}} \mathbin{+\!\!+} \cdots \mathbin{+\!\!+} L^{a_1} \mathbin{+\!\!+} R^{b_1}$ for some integers $(a_d, b_d, \dots, a_1, b_1)$. This holds since no leaf has depth greater than d. For the jth leftmost leaf v we denote $\mathsf{enc}(v)$ often by $\mathsf{enc}(j)$. For example, in Fig. 2, the three leaves of $\mathfrak{S}(2, 1)$ are encoded as $\mathsf{enc}(0) = L = 0$, $\mathsf{enc}(1) = RL = 10$ and $\mathsf{enc}(2) = RR = 11$, respectively. (Recall that the leftmost leaf is 0.) Note that for no two leafs $i \neq j$, $\mathsf{enc}(i)$ will be a prefix of $\mathsf{enc}(j)$. Finally, $\overline{w}_h(\mathit{state}) = d - w_h(\mathit{state})$ since we interpret *state* always as a bitstring of length d.

3.3 Correctness Proof

Theorem 1. *Algorithm 1 is a correct $\mathfrak{S}(d, w)$-traversal algorithm.*

Proof: A simple case arises when $d = w$, since $\mathfrak{S}(d, d)$ is equal to the complete binary tree $\mathfrak{T}(d)$. Its all leaves are at the same depth d, and have different d-bit encodings $\mathsf{enc}(0) := 0\dots000$, $\mathsf{enc}(1) := 0\dots001$, …, $\mathsf{enc}(2^d - 1) := 1\dots111$. Clearly, all encodings are different and therefore $\mathit{state} = \mathsf{enc}(j)$ is just equal to the binary d-bit representation of j.

We have a slightly more difficult case when $w \neq d$. As in the $\mathfrak{T}(d)$-case, $\mathsf{enc}(0)$ is always an all-zero string, $\mathsf{enc}(n-1)$ is always an all-one string and that for all j, $\mathsf{enc}(j) > \mathsf{enc}(j-1)$ always as a binary string. On the other hand, some of the nodes of $\mathfrak{S}(d, w)$ have depth smaller than w. Therefore, one must first determine during every invocation of *update* the next two things: First, what is the depth of the next node j, and second, how to compute $\mathsf{enc}(j)$ from $\mathsf{enc}(j-1)$.

Now, let $\mathsf{depth}_{d,w}(j)$ denote the depth of the jth leaf of $\mathfrak{S}(d, w)$. Equivalently, $\mathsf{depth}_{d,w}(j)$ is the length of binary string $\mathsf{enc}(j)$, for fixed d and w. Denote $\mathsf{enc}(j) = h \mathbin{+\!\!+} t$, where $h \in \{0, 1\}$. To compute $\mathsf{depth}_{d,w}(j)$, first note that since $\mathfrak{S}(d, w) = \mathfrak{S}(d-1, w-1) \otimes \mathfrak{S}(d-1, w)$ then

$$\mathsf{depth}_{d,w}(j) = \begin{cases} 1 + \mathsf{depth}_{d-1,w-1}(j') \ , & h = 0 \ , \\ 1 + \mathsf{depth}_{d-1,w}(j') \ , & h = 1 \end{cases} ,$$

since $h = 0$ exactly when j is in the left subtree, $\mathfrak{S}(d-1, w-1)$, of $\mathfrak{S}(d, w)$. (In both cases j' is a leaf of corresponding subtree with encoding $\mathsf{enc}(j') = t$.) Therefore, an arbitrary vertex $v \in \mathfrak{S}(d,w)$ is a root of the subtree $\mathfrak{S}(d - |\mathsf{enc}(j)|, w - \overline{w}_h(\mathsf{enc}(j)))$. Therefore, we have shown that

Lemma 1. *Let $\mathfrak{S}(d,w)$ be a fixed skewed tree. A binary string e is*

1. *Encoding of a leaf of $\mathfrak{S}(d,w)$ iff one of the two values $d - |e|$ and $w - \overline{w}_h(e)$ is 0 and both are nonnegative.*
2. *Encoding of an internal node of $\mathfrak{S}(d,w)$ iff both those values are positive;*
3. *An invalid encoding (not an encoding of any vertex of $\mathfrak{S}(d,w)$) iff either of those two values is negative;*

Now, let us look in more detail at the sequential generation process of encodings $\mathsf{enc}(0), \mathsf{enc}(1), \ldots, \mathsf{enc}(n-1)$ in the case of a fixed skewed tree $\mathfrak{S}(d,w)$. For the simplicity of description we also define an alternative padded encoding function $\mathsf{penc}(j) := \mathsf{enc}(j) + 0^{d - \mathsf{depth}_{d,w}(j)}$. That is, $\mathsf{penc}(j)$ is equal to $\mathsf{enc}(j)$, padded on right with zeros to the length d.

Algorithm 1 stores the encoding of next leaf $\mathsf{penc}(j)$ as *state*. The encoding of the leftmost leaf 0, $\mathsf{enc}(0)$, consists of all zeros, and therefore $\mathsf{penc}(0) = 0^d$. Thus, the initialization $\mathit{state} := 0$ is correct. Assume that we have computed the value $\mathsf{enc}(j-1)$ for some j. The next node, j, has encoding $\mathsf{penc}(j) > \mathsf{penc}(j-1)$.

To compute the encoding $\mathsf{penc}(j)$, given $\mathit{state} = \mathsf{penc}(j-1)$, note that if $j-1$ is a leaf with encoding $\mathsf{enc}(j-1) = e$ then

1. If $\overline{w}_h(e) < w$ then by Lemma 1, $\mathsf{depth}_{d,w}(e) = d$. But then $e = \mathit{state}$ and hence also $\overline{w}_h(\mathit{state}) < w$. In this case the next leaf j has encoding $\mathsf{penc}(j) = \mathit{state} + 1$. (If the least significant bit of *state* is 0 then $w_h(\mathit{state}+1) = w_h(\mathit{state}) + 1$, and then $\mathit{state}+1$ encodes the jth leaf with $\mathsf{depth}_{d,w}(j) = d$. Otherwise, $\mathsf{enc}(j)$ will be the longest prefix of $\mathit{state}+1$ that contains no more than w zeros; then $\mathsf{penc}(j) = \mathit{state}+1$.)
2. Otherwise, by Lemma 1, $\overline{w}_h(e) = w$. (Then clearly $\overline{w}_h(\mathit{state}) \geq \overline{w}_h(e) = w$.) That is, e is equal to the longest prefix of *state* that contains exactly w zeros, and $\mathsf{depth}_{d,w}(j-1) = d - (\overline{w}_h(\mathit{state}) - w) = w + w_h(\mathit{state})$. In this case, no leaf can have encoding of form $e + 0^k$ for $k \geq 1$. However, $e+1$ is a valid encoding for a vertex in $\mathfrak{S}(d,w)$. Therefore, $\mathsf{penc}(j) = \mathit{state} + 2^{\overline{w}_h(\mathit{state}) - w}$.

Now, we have shown that Algorithm 1 updates the *state* correctly: It starts with the encoding $\mathit{state} = \mathsf{penc}(0) = 0^d$ of the first node. At every step it updates *state* from $\mathsf{penc}(j-1)$ to $\mathsf{penc}(j)$: Depending on whether $w_h(\mathit{state}) = w$, it follows one of the two described updating rules that can be jointly described as $\mathsf{penc}(j) = \mathsf{penc}(j) + (1 \ll (\overline{w}_h(\mathsf{penc}(j-1)) \dot{-} w))$.

Apart from updating the state, Algorithm 1 must also update the stack, that is, it must remove from the the node labels (either original data items, corresponding to leafs, or internal hash representations) that are not anymore needed and replace them with the new ones. The intuition behind this simple procedure (starting from line 10 in Algorithm 1) is very simple.

To compute the hash chain from leaf j with encoding $\mathsf{enc}(j)$ to the root, one must have available all the hash values, corresponding to the left and right children of this path together with the endpoint of the path. Denote the set of left children of the root

Algorithm 2 An algorithm for traversing $\overline{\mathfrak{S}}(d, w)$

funct $init(d, w) \;\equiv$
  state $:= 0$;
  if $w > d$ **then** $w = d$ **fi**;
  Store w, d;
  Create an empty stack of maximum possible size w.
funct *update*(*value*) $\equiv$
  if *state* $\geq 2^d$ **then** **return**(*Error*!); **fi**
  if $w_h(\mathit{state}) < w$
    then $n := \mathtt{ntz}(\mathit{state} + 1)$; *increment* $:= 1$;
    else $n := \mathtt{ntz}((\mathit{state} \gg \mathtt{ntz}(\mathit{state})) + 1)$; *increment* $:= 1 \lll (\overline{w}_h(\mathit{state}) - w)$; **fi**
  push(*value*);
  for $i := 1$ **to** n **do**
    push(H(**pop**(), **pop**())); **end**
  state $:=$ *state* $+$ *increment*.

path that starts from leaf j by $X(j)$. (E.g., to compute the value of the root, it suffices to have available the hash values that correspond to the set $X(n-1) \cup \{n-1\}$, where $n-1$ is the rightmost leaf like always.)

Now, one can compute the set $X(j)$ from $X(j-1)$ as follows. The set $X(j-1) \setminus X(j)$ consists of hash values of nodes with encoding $p \mathbin{+\!\!+} 0 \mathbin{+\!\!+} 1^i \mathbin{+\!\!+} 0$, where $\mathsf{enc}(j-1) = p \mathbin{+\!\!+} 0 \mathbin{+\!\!+} 1^k$ and $i \in [0, k-1]$. The only element in $X(j)$ that is not in $X(j-1)$ is the node with encoding $p \mathbin{+\!\!+} 0$. The latter node is the root of the subtree that has nodes from $(X(j-1) \setminus X(j)) \cup \{j-1\}$. Therefore, before the **for** -loop in Algorithm 1, the stack clearly consists of the set of hash values of nodes $X(j)$, and the **for** -loop updates the stack to the state where it contains the hash values, corresponding to the nodes from set $X(j+1)$. ■

Finally, Algorithm 2 and Algorithm 3 work on families $\overline{\mathfrak{S}}(d, w)$ and $\mathfrak{T}(d)$ respectively, where $\overline{\mathfrak{S}}(d, w)$ is a mirror family of $\mathfrak{S}(d, w)$ (that is, one applies the recursive rule $\overline{\mathfrak{S}}(d, w) = \overline{\mathfrak{S}}(d, w-1) \otimes \overline{\mathfrak{S}}(d-1, w-1)$ instead of the rule $\mathfrak{S}(d, w) = \mathfrak{S}(d-1, w-1) \otimes \mathfrak{S}(d, w-1)$). In principal, Algorithm 2 algorithm is the same as Algorithm 1, except that the meanings of bits 0 and 1 in the encodings are switched. What is interesting is that Algorithm 2 is at least conceptually slightly more difficult than Algorithm 1. Algorithm 3 is mostly described for comparison purposes.

4 Complexity Analysis

We next analyse the complexity of *update* in Algorithm 1. The error verification and **for** -loop notwithstanding, during one run of *update*, the computer must execute two ordinary additions, one proper subtraction, one data-dependent shift left, two assignments, and once the functions $\overline{w}_h$ and $\mathtt{ntz}$. There are special instructions for all these functions in many new processors and therefore the total time for this part is negligible. Instead, the *update* time depends primarily on the **for** -loop, that consists of $2 \cdot \mathtt{ntz}(\mathit{state} + 1)$ **pop**() operations (easy) and $\mathtt{ntz}(\mathit{state} + 1)$ hash-function invocations (hard). Therefore,

Algorithm 3 An algorithm for traversing $\mathfrak{T}(d)$, given for comparison purposes

funct *init*(d) $\equiv$
  state := 0;
  Store d;
  Create an empty stack of maximum possible size d.
funct *update*(*value*) $\equiv$
  if *state* $\geq 2^d$ **then** **return**(*Error*!); **fi**
  push(*value*);
  for $i := 1$ **to** $\mathtt{ntz}(\mathit{state}+1)$ **do**
    push(H(**pop**(), **pop**())); **end**
  state := *state* + 1.

in the worst case, this algorithm can do up to d hash-function invocations. Moreover, the worst-case memory use is $d(k+1)$, where k is the output size of the employed hash function (usually in the range of $160\ldots512$ bits): Really, one stores at any time moment a freshness token (that takes up to $d \cdot k$) bits and a d-bit *state*.

Hence, one could argue that the time complexity of our algorithm is close to optimal. Really, the number of hashings must be the same for all algorithms that accomplish the same task. Moreover, $d \cdot k$ bits are necessary to store the greatest freshness token. Apart from that, our algorithms uses a small number of additional instructions that can all be executed very quickly on current processors.

Next, we will estimate the average storage complexity of Algorithm 1. For this, we first define the partial binomial sum $L(d,w) := \sum_{k=0}^{w} \binom{d}{k}$.

Lemma 2. *(1) $\mathfrak{S}(d,w)$ has $L(d,w)$ leaves. (2) Let $\ell(d,w,t)$ be the number of leaves of $\mathfrak{S}(d,w)$ that are at depth t. Then*

$$\ell(d,w,t) = \begin{cases} 0 \ , & t < w \ , \\ \binom{t-1}{w-1} \ , & t \in [w, d-1] \ , \\ 2 \cdot \sum_{k=0}^{w-1} \binom{d-1}{k} = 2L(d-1, w-1) \ , & t = d \ , \\ 0 \ , & t > d \ . \end{cases}$$

Proof: (1) Proof by induction [Wil02b].

(2) Proof by induction on (d,w). If $d = w = 1$ then $\ell(1,1,t) = 1$. Otherwise, $\ell(d,w,t) = \ell(d-1,w-1,t-1) + \ell(d-1,w,t-1)$. When $t < w$ then $\ell(d,w,t) = 0$. When $t = w$ then $\ell(d,w,t) = \binom{t-2}{w-2} + 0 = 1 = \binom{t-1}{w-1}$. When $t \in [w+1, d-1]$ then $\ell(d,w,t) = \binom{t-2}{w-2} + \binom{t-2}{w-1} = \binom{t-1}{w-1}$. When $t = d$ then $\ell(d,w,t) = 2 \cdot \left(\sum_{k=0}^{w-2} \binom{d-2}{k} + \sum_{k=0}^{w-1} \binom{d-2}{k}\right) = 2 \cdot \left(\sum_{k=0}^{w-1} \binom{d-2}{k-1} + \sum_{k=0}^{w-1} \binom{d-2}{k}\right) = 2 \cdot \sum_{k=0}^{w-1} \binom{d-1}{k}$. When $t > d$ then $t = 0$. ■

Lemma 3. *The number of hashings done on the jth step of Algorithm 1 is $\mathtt{ntz}(\mathit{state}+1)$. The size of the stack before the jth step is $w_h(\mathtt{enc}(j))$.*

Proof: The proof of the first claim is straightforward. That the second claim holds is also clear: $X(j)$ has $w_h(\mathtt{enc}(j))$ different elements, with $\mathtt{enc}(x)$ being equal to the maximum prefix of $\mathsf{enc}(j)$ that has exactly i, $0 \le i < w_h(\mathsf{enc}(j))$, ones that are appended by a 0. ■

Theorem 2. *(1) In average, the stack contains*

$$\mathsf{ft}(d,w) = \frac{d}{2} + \frac{w-1}{2^{d+1}(w+1)\int_0^{0.5} y^{d-w-1}(1-y)^w \, dy} \tag{1}$$

hash elements. (2) Asymptotically,

$$\mathsf{ft}(d,w) \sim \begin{cases} \frac{d}{2} + \frac{(w-1)(d-2w)}{2(w+1)} \ , & w \le d/2 \ , \\ \frac{d}{2} + \frac{d^d(w-1)}{2^{d+1}(w+1)\sqrt{2\pi/d}\cdot(d-w)^{d-w-1/2}w^{w+1/2}} \ , & w \ge d/2 \ . \end{cases}$$

The proof of this theorem will use the next two lemmas.

Lemma 4. $L(d,w) = 2^d(d-w)\binom{d}{w}\int_0^{0.5} y^{d-w-1}(1-y)^w \, dy$.

Proof: Multiple integration by parts gives $\int y^a(1-y)^b \, dy = \frac{y^{a+1}(1-y)^b}{a+1} + \frac{b}{a+1}\int y^{a+1}(1-y)^{b-1} \, dy$. Thus $B_1(d,w,p) := \sum_{k=0}^{w}\binom{d}{k}p^{d-k}(1-p)^k = (d-w)\binom{d}{w}\int_0^p y^{d-w-1}(1-y)^w \, dy$. Therefore, since $L(d,w) = \sum_{k=0}^{w}\binom{d}{k}$, $L(d,w) = 2^d B_1(d,w,1/2) = 2^d(d-w)\binom{d}{w}\int_0^{0.5} y^{d-w-1}(1-y)^w \, dy$. ■

Lemma 5. $L(d-1,w-1) = \frac{L(d,w)}{2} - \frac{1}{2}\cdot\binom{d-1}{w}$.

Proof: Define $J(a,b) := \binom{d}{w}\int_0^{0.5} y^a(1-y)^b \, dy$. Thus, $L(d,w) = 2^d(d-w)\binom{d}{w}J(d-w-1,w)$. Now, $\int y^a(1-y)^b \, dy = \int y^a(1-y)^{b-1} \, dy - \int y^{a+1}(1-y)^{b-1} \, dy$, and therefore $J(a,b) = J(a,b-1) - J(a+1,b-1)$. Most importantly, $J(d-w-1,w) = J((d-1)-(w-1)-1,w-1) - J(d-(w-1)-1,w-1)$, which gives us $L(d-1,w-1) = \frac{L(d,w)w}{2d} + \frac{L(d,w-1)(d-w)}{2d}$. Given that $L(d,w-1) = L(d,w) - \binom{d}{w}$, we get that $L(d-1,w-1) = \frac{L(d,w)w}{2d} + \frac{(L(d,w)-\binom{d}{w})(d-w)}{2d} = \frac{L(d,w)}{2} - \frac{1}{2}\cdot\binom{d-1}{w}$. ■

Proof: [of Thm. 2.] (1) According to Lemma 3, in average the stack contains $s(d,w)/L(d,w)$ hash elements, where $s(d,w) = \sum_{j=0}^{n-1} w_h(\mathsf{enc}(j))$. By Lemma 2, $s(d,w) = \sum_{t=w}^{d-1}\binom{t-1}{w-1}\cdot(t-w) + \sum_{k=0}^{w-1}\binom{d-1}{k}\cdot(2\cdot(d-k-1)+1) = \sum_{t=0}^{d-w-1}\binom{t+w-1}{w-1}\cdot t + (2d-1)\cdot\sum_{k=0}^{w-1}\binom{d-1}{k} - 2\cdot\sum_{k=0}^{w-1}\binom{d-1}{k}\cdot k = w\cdot\binom{d-1}{w-1} + (2d-1)\cdot L(d-1,w-1) - 2\cdot Q(d-1,w-1)$, where $Q(d,w) := \sum_{k=0}^{w}\binom{d}{k}\cdot k$.

Define $B_2(d,w,p) := \sum_{k=0}^{w}\binom{d}{k}kp^{d-k}(1-p)^k$. Thus, $B_2(d,w,p) = d(1-p)\sum_{k=0}^{w-1}\binom{d-1}{k}p^{d-1-k}(1-p)^k = d(1-p)B_1(d-1,w-1,p)$. In particular, $Q(d,w) = 2^d B_2(d,w,1/2) = \frac{d2^d}{2}B_1(d-1,w-1,1/2) = dL(d-1,w-1)$. Hence, $\mathsf{ft}(d,w) = \frac{w\cdot\binom{d-1}{w+1} + (2d-1)L(d-1,w-1) - 2(d-1)(L(d-2,w-2))}{L(d,w)}$. Now, according to Lemma 5, $\mathsf{ft}(d,w) = \frac{w\cdot\binom{d-1}{w+1} + dL(d-1,w-1) + (d-1)\binom{d-2}{w-1}}{L(d,w)} = \frac{2w\cdot\binom{d-1}{w+1} + dL(d,w) - d\binom{d-1}{w} + 2(d-1)\binom{d-2}{w-1}}{2L(d,w)} = \frac{d}{2} + \binom{d}{w}\frac{(w-1)(d-w)}{(w+1)2L(d,w)}$. According to Lemma 4, $\mathsf{ft}(d,w) = \frac{d}{2} + \frac{w-1}{2^{d+1}(w+1)\int_0^{0.5} y^{d-w-1}(1-y)^w \, dy}$.

(2) We will use the standard Laplace's method [dB82] for estimating the asymptotic behavior of integrals of type $I(d) = \int_a^b e^{d\cdot h(y)} g(y)\ dy$. Since this method is standard in analysis but might probably be not well-known to a potential reader, we will give a somewhat longish proof without any precise explanations. A curious reader is referred to [dB82]. (For completeness, we will give a shorter alternative proof directly after this one.)

Assume that $w = \alpha d$. Rewrite the integral from (1) as a Laplace integral, $I(d) = \int_0^{0.5} y^{d-\alpha d-1}(1-y)^{\alpha d} dy = \int_0^{0.5} e^{d((1-\alpha)\ln y+\alpha\ln(1-y))}\frac{1}{y}dy = \int_0^{0.5} e^{d\cdot h(y)}g(y)dy$, where $h(y) = (1-\alpha)\ln y + \alpha \ln(1-y)$ and $g(y) = \frac{1}{y}$. Now, $h(y)$ has a unique maximum $y_0 = \min(1-\alpha, 0.5)$ in the interval $[0, 0.5]$. Due to this reason, we divide the analysis into two parts: in the first part we analyse the case $\alpha \leq 0.5$ and in the second part the case $\alpha \geq 0.5$.

First case, $\alpha \leq 0.5$. For applying the Laplace's method for integrals, we first rescale the integration variable by setting $s := (y - y_0)d = (y - 0.5)d$. Then $I(d) \sim \int_{0.5-\varepsilon}^{0.5} e^{dh(y)}g(y)\ dy = \frac{1}{d}\int_{-\varepsilon d}^{0} e^{dh(1/2+s/d)}g\left(1/2+s/d\right)\ ds$, where $1/x \ll \varepsilon \ll 1/\sqrt{x}$. But $h(1/2 + s/d) = -\ln 2 - \frac{2(2\alpha-1)}{d}\cdot s - \frac{2}{d^2}\cdot s^2 + O(s^3)$ and $g(1/2 + s/d) = 2 - 4/s\cdot d + 8/s^2d^2 + O(s^3)$. Thus we can simplify $I(d)$, approximating $g(1/2 + s/d) \sim 2$. Then, $I(d) \sim \frac{2}{d}\int_{-\varepsilon d}^{0} e^{dh(1/2+s/d)}\ ds$. Furthermore, take $e^{d\cdot h(1/2+s/d)} = e^{-d\ln 2-2(2\alpha-1)s}\cdot e^{-2/(s^2d)+O(s^3)} \sim e^{-d\ln 2-2(2\alpha-1)s}\cdot (1-\frac{2}{s^2d}+O(s^3)) \sim 2^{-d}\cdot e^{-2(2\alpha-1)s}$. Thus, $I(d) \sim \frac{2^{1-d}}{d}\int_{-\varepsilon d}^{0} e^{-2(2\alpha-1)s}\ ds \sim \frac{2^{1-d}}{d}\int_{-\infty}^{0} e^{-2(2\alpha-1)s}\ ds = \frac{2^{1-d}}{d}\cdot\left(-\frac{1}{2(2\alpha-1)}\right) = -\frac{1}{d2^d(2\alpha-1)}$. Thus, in this case, $\mathsf{ft}(d,w) \sim \frac{d}{2} - \frac{(w-1)d2^d(2\alpha-1)}{2^{d+1}(w+1)} = \frac{d}{2} + \frac{(w-1)(d-2w)}{2(w+1)}$.

Second case, $\alpha \geq 0.5$. In this case, the function $h(y)$ has a maximum at the point $\beta := 1-\alpha \leq 0.5$. Split the integral into a local and nonlocal part: $I(d) = \int_0^{\beta-\varepsilon} e^{d\cdot h(y)}g(y)\ dy + \int_{\beta-\varepsilon}^{\beta+\varepsilon} e^{d\cdot h(y)}g(y)\ dy + \int_{\beta+\varepsilon}^{0.5} e^{d\cdot h(y)}g(y)\ dy$, where $1/x^{1/2} \ll \varepsilon \ll 1/x^{1/3}$ and the nonlocal part is exponentially small. Therefore, $I(d) \sim \int_{\beta-\varepsilon}^{\beta+\varepsilon} e^{d\cdot h(y)}g(y)\ dy$. We can expand $g(y)$ and $h(y)$ around $y = \beta$, which gives us $I(d) \sim \int_{\beta-\varepsilon}^{\beta+\varepsilon} e^{d\cdot(\alpha\ln\alpha+\beta\ln\beta-\frac{(y-\beta)^2}{2\alpha\beta}+O((y-\beta)^3))}(\frac{1}{\beta}+O(y-\beta))\ dy$. Change the variables, $s = \sqrt{d}(y-\beta)$. Thus, $I(d) \sim \frac{1}{\sqrt{d}}\int_{-\sqrt{d}\cdot\varepsilon}^{\sqrt{d}\cdot\varepsilon} e^{d\cdot(\alpha\ln\alpha+\beta\ln\beta-\frac{s^2}{2\alpha\beta d}+O(s^3))}(\frac{1}{\beta}+O(s))\ \ ds$. Next, discard all but the leading-order term of g, and approximate $e^{O(s^3)} \sim 1 + O(s^3)$. Thus, $I(d) \sim \frac{1}{\sqrt{d}\cdot\beta}\int_{-\sqrt{d}\cdot\varepsilon}^{\sqrt{d}\cdot\varepsilon} e^{d\cdot(\alpha\ln\alpha+\beta\ln\beta-\frac{s^2}{2\alpha\beta d})}\ \ ds = \frac{e^{d\cdot\alpha\ln\alpha+d\cdot\beta\ln\beta}}{\sqrt{d}\cdot\beta}\int_{-\sqrt{d}\cdot\varepsilon}^{\sqrt{d}\cdot\varepsilon} e^{-\frac{s^2}{2\alpha\beta}}\ \ ds = \frac{\alpha^{d\cdot\alpha}\beta^{d\cdot\beta}}{\sqrt{d}\cdot\beta}\int_{-\sqrt{d}\cdot\varepsilon}^{\sqrt{d}\cdot\varepsilon} e^{-\frac{s^2}{2\alpha\beta}}\ \ ds$. Expanding the integral gives $I(d) \sim \frac{\alpha^{d\cdot\alpha}\beta^{d\cdot\beta}}{\sqrt{d}\cdot\beta}\int_{-\infty}^{\infty} e^{-\frac{s^2}{2\alpha\beta}}\ \ ds = \frac{\alpha^{d\cdot\alpha}\beta^{d\cdot\beta}}{\sqrt{d}\cdot\beta}\cdot\sqrt{2\pi\beta\alpha} = \sqrt{2\pi/d}\alpha^{d\cdot\alpha+1/2}\beta^{d\cdot\beta-1/2}$. Therefore, in this case, $\mathsf{ft}(d,w) \sim \frac{d}{2} + \frac{d^{d+1/2}(w-1)}{2^{d+3/2}\sqrt{\pi}(w+1)\cdot(d-w)^{d-w-1/2}\cdot w^{w+1/2}} \sim \frac{d}{2}$. ■

Proof: [Shorter proof of Thm. 2, (2).] As before, rewrite the integral as $\int_a^b e^{d\cdot h(y)}g(y)\ dy$ with $a = 0$ and $b = 0.5$, and analyse separately the same two cases, $\alpha \geq 0.5$ and $\alpha \leq 0.5$. In the first case, one uses the well-known analytic result [dB82]

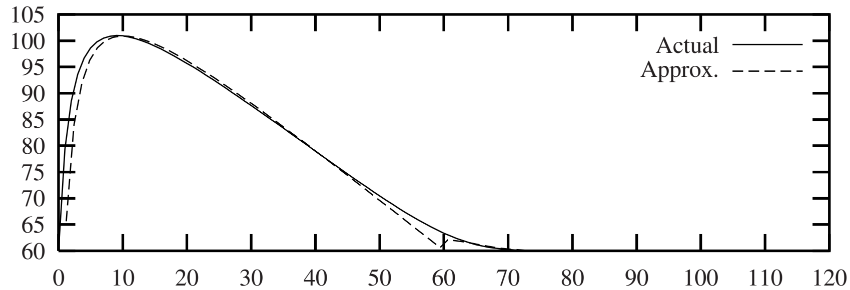

Fig. 3. Function $\mathsf{ft}(120, \cdot)$: its actual value together with approximation

that $I(d) \sim \frac{g(b)e^{dh(b)}}{d \cdot h'(b)} = -\frac{1}{d2^d(2\alpha-1)}$, and thus $\mathsf{ft}(d, w) \sim \frac{d}{2} + \frac{(w-1)(d-2w)}{2(w+1)}$. In the second case, one has that $I(d) \sim \frac{\sqrt{2\pi} \cdot e^{d \cdot h(\beta)} g(\beta)}{\sqrt{-d \cdot h''(\beta)}} = \sqrt{2\pi/d} \cdot \alpha^{d \cdot \alpha + 1/2} \beta^{d \cdot \beta - 1/2}$. ■

As noted by Willemson [Wil02a], the result $\mathsf{ft}(d, w) = \frac{d}{2} + \binom{d}{w} \frac{(w-1)(d-w)}{(w+1)2L(d,w)}$ can be derived in an easier way by using the alternative representation of the skewed trees from [Wil02b, Chapter 5].

The "actual" function $\mathsf{ft}(120, \cdot)$ together with approximation from the previous theorem are depicted by Fig. 3. As it can be seen, Thm. 2 results in very precise approximation even for small d-s.

Corollary 1. *(1) For fixed d, $\mathsf{ft}(d, w)$ obtains the maximum value when $w \approx \sqrt{d+2} - 1$. Then $\mathsf{ft}(d, w) \sim d - \sqrt{d+2} + 3$. Thus, in the maximum average storage complexity of Algorithm 1 is $\approx (d - \sqrt{d})k + d$. (2) Assume $\alpha = w/d = \frac{3-\sqrt{5}}{2}$. Then $\mathsf{ft}(d, w) \approx \frac{d(3-\sqrt{5})(d+2+\sqrt{5})}{2d+3+\sqrt{5}} \approx \frac{3-\sqrt{5}}{2} d$ and the total storage complexity of Algorithm 1 is $\approx \frac{3-\sqrt{5}}{2} d \cdot k + d$.*

(The double occurance of constant $\frac{3-\sqrt{5}}{2}$ in (2) is rather curious.) Finally, one can safely assume that when the protocols are designed reasonably then the communication between clients and the TSA will be dominated by the size of freshness tokens. Therefore, the average-case communication complexity of interval time-stamping is $\mathsf{ft}(d, w) \cdot (1 + o(1))$ that can be computed by using Thm. 2.

Acknowledgments. This work was partially supported by the Finnish Defence Forces Research Institute of Technology. Algorithm 1 was originally presented in the rump session of Estonian Winter School of Computer Science, March 4, 2002. We would like to thank Ahto Buldas, Markus Jakobsson, Jan Willemson and anonymous referees for useful comments.

References

[BdM91] Josh Benaloh and Michael de Mare. Efficient Broadcast Time-stamping. Technical Report 1, Clarkson University Department of Mathematics and Computer Science, August 1991.

[BHS92] Dave Bayer, Stuart A. Haber, and Wakefield Scott Stornetta. Improving the Efficiency And Reliability of Digital Time-stamping. In *Sequences'91: Methods in Communication, Security, and Computer Science*, pages 329–334. Springer-Verlag, 1992.

[BL98] Ahto Buldas and Peeter Laud. New Linking Schemes for Digital Time-stamping. In *The 1st International Conference on Information Security and Cryptology*, pages 3–14, Seoul, Korea, 18–19 December 1998. Korea Institute of Information Security and Cryptology.

[BLLV98] Ahto Buldas, Peeter Laud, Helger Lipmaa, and Jan Villemson. Time-stamping with Binary Linking Schemes. In Hugo Krawczyk, editor, *Advances in Cryptology — CRYPTO '98*, volume 1462 of *Lecture Notes in Computer Science*, pages 486–501, Santa Barbara, USA, 23–27 August 1998. International Association for Cryptologic Research, Springer-Verlag.

[BLS00] Ahto Buldas, Helger Lipmaa, and Berry Schoenmakers. Optimally Efficient Accountable Time-stamping. In Hideki Imai and Yuliang Zheng, editors, *Public Key Cryptography '2000*, volume 1751 of *Lecture Notes in Computer Science*, pages 293–305, Melbourne, Victoria, Australia, 18–20 January 2000. Springer-Verlag.

[Bul02] Ahto Buldas. Personal communication. June 2002.

[Dam87] Ivan Bjerre Damgård. Collision free hash functions and public key signature schemes. In David Chaum and Wyn L. Price, editors, *Advances in Cryptology — EUROCRYPT '87*, volume 304 of *Lecture Notes in Computer Science*, pages 203–216, Amsterdam, The Netherlands, 13–15 April 1987. Springer-Verlag, 1988.

[dB82] Nicholas Govert de Bruijn. *Asymptotic Methods in Analysis*. Dover, January 1982.

[HS91] Stuart A. Haber and Wakefield Scott Stornetta. How to Time-stamp a Digital Document. *Journal of Cryptology*, 3(2):99–111, 1991.

[Knu98] Donald E. Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching*. Addison-Wesley, 2 edition, 1998.

[Lip99] Helger Lipmaa. *Secure and Efficient Time-stamping Systems*. PhD thesis, University of Tartu, June 1999.

[LM01] Helger Lipmaa and Shiho Moriai. Efficient Algorithms for Computing Differential Properties of Addition. In Mitsuru Matsui, editor, *Fast Software Encryption '2001*, volume 2355 of *Lecture Notes in Computer Science*, pages 336–350, Yokohama, Japan, 2–4 April 2001. Springer-Verlag, 2002.

[Mer80] Ralph Charles Merkle. Protocols for Public Key Cryptosystems. In *Proceedings of the 1980 Symposium on Security and Privacy*, Oakland, California, USA, 14–16 April 1980. IEEE Computer Society Press.

[Wil02a] Jan Willemson. Personal communication. July 2002.

[Wil02b] Jan Willemson. *Size-Efficient Interval Time Stamps*. PhD thesis, University of Tartu, June 2002. Available from http://home.cyber.ee/jan/publ.html, May 2002.