Authenticated Dictionary Based on Frequency

Kévin Atighehchi, Alexis Bonnecaze, Traian Muntean

▶ **To cite this version:**

Kévin Atighehchi, Alexis Bonnecaze, Traian Muntean. Authenticated Dictionary Based on Frequency. 29th IFIP International Information Security Conference (SEC), Jun 2014, Marrakech, Morocco. pp.293-306, 10.1007/978-3-642-55415-5_24 . hal-01370376

HAL Id: hal-01370376

https://inria.hal.science/hal-01370376v1

Submitted on 22 Sep 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Authenticated Dictionary Based on Frequency

Kévin Atighehchi, Alexis Bonnecaze, and Traian Muntean

Aix Marseille University, CNRS, Centrale Marseille, ERISCS, I2M, UMR 7373,
13453 Marseille, France
firstname.lastname@univ-amu.fr

Abstract. We propose a model for data authentication which takes into account the behavior of the clients who perform queries. Our model reduces the size of the authenticated proof when the frequency of the query corresponding to a given data is higher. Existing models implicitly assume the frequency distribution of queries to be uniform, but in reality, this distribution generally follows Zipf's law. Therefore, our model better reflects reality and the communication cost between clients and the server provider is reduced allowing the server to save bandwith. When the frequency distribution follows Zipf's law, we obtain a gain of at least 20% on the average proof size compared to existing schemes.

Keywords: Authenticated dictionary, Data structure, Merkle tree, Zipf

1 Introduction

Authenticated dictionaries are used to organize and manage a collection of data in order to answer queries on these data and to certify the answers. They have been heavily studied recently and have many applications including certificate revocation in public key infrastructure [4, 7, 10, 16], geographic information system querying, or third party data publication on the Internet [5, 2]. This last application is of great interest with the advent of cloud computing and Web services. For example, it is important that a user who consults a Web page can be confident of the authenticity of that page (or some of its contents).

Classical schemes involve three actors [22, 8, 9]: a trusted *source* which is generally the owner of the data, an untrusted provider also called *directory* and a set of *users* (also called clients). The directory receives a set of data from the source together with authentication information. These contents are stored by both the source and the directory but only the latter communicates with users. Therefore, as shown in Figure 1, users communicate directly with the directory to query the authentication information on a given data. This information contains a cryptographic proof and allows the users to authenticate the data.

Most of authenticated dictionaries use Merkle trees, red-black trees or skip-lists as data structures. These structures are closely equivalent in terms of cost of storage, communication and time [22]. They are well adapted as long as no distinction is made between data. However, in some situations, it may be useful to manage data as a function of some parameters. In the case of publications on the Internet, some pages are

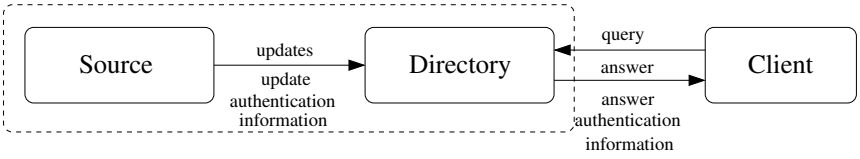

Fig. 1: The three-party authentication model

accessed more frequently, depending on user behavior. Some pages have a better reputation than others, and it may prove useful to order them following this criterion. in fact, any behavioural criterion could be taken into account.

In this paper, we introduce an authenticated dictionary scheme which takes into account the frequency of data being accessed. As regards Web traffic, it is well known that its frequency distribution follows Zipf's law [1, 18, 17, 12, 6, 21]. More precisely, most traffic follows this law except for the traffic residue corresponding to very low frequencies. In fact, there is a drooping tail, which means that for these frequencies, the distribution decreases much faster than Zipf's law.

The paper is organized as follows. Section 2 contains background information regarding data structures and the dictionary problem. Section 3 introduces our scheme. We present the underlying data structures and the updating, searching and certification operations provided by the dictionary. In Section 4 we discuss the efficiency of our method, and show that, compared to existing schemes and when the frequency distribution follows Zipf's law, the reduction of proof size is better than 20%.

2 Background

2.1 Data structures and authentication

Data structures represent a way of storing and organizing data so that searching, adding or deleting operations can be done efficiently. A static structure has a size that cannot be changed and therefore it is not possible to delete or add any data a posteriori. However, the size of dynamic data structures can change allowing insertion and deletion operations. In this paper, the term *dynamic data structure* refers to any data structure which accepts insertion and deletion of data at any position. The term *append/disjoin-only data structure* refers to any data structure which accepts insertion and deletion at the end of the structure. Examples of dynamic structures [13] are hash tables, trees like 2-3 Trees, B-Trees or red-black trees, or other random structures like non-deterministic skip-lists [20].

In addition to these basic features, data structures can be used to construct authenticating mechanisms. Data structures based on rooted graphs are well adapted to deal with such mechanisms [22, 8] since authenticating all the data covered by the graph just requires one single signature and some hash computations. An example of authenticated data structure is the static Merkle tree [14, 15] of which the number of leaves is a power of 2. There exist variants accepting any number of leaves. Although these variants can

still be considered as static, they can also be considered as append/disjoin-only data structures since structural changes can be done at the right side of the tree. This type of structure is suitable for time stamping [19, 3]. In the following, we briefly detail one of these variants [19]. It is an almost balanced tree in which values of the internal nodes are calculated in the following way. Let $(e_1, e_2, \ldots, e_n)$ be the values of the leaves at the base of the tree. Values of nodes at the previous level are $(h(e_{2i+1}, e_{2i+2})_{i=0\ldots(n-2)/2})$ if n is even, and $(h(e_{2i+1}, e_{2i+2})_{i=0\ldots(n-3)/2}, e_n)$ otherwise. This process is repeated until a single value is obtained (this is the root node value). Adding an element e^* after e_n is a very simple operation. The value v of the root of the smallest (perfectly) balanced subtree to where e_n belongs is changed to $v' = h(v, e^*)$. Then, values of the internal nodes on the path from this root to the root of the tree are updated. The disjoin operation is just the inverse operation. Note that this structure is equivalent to a deterministic skip-list. Finally, one might add that static structures should always be preferred for their better complexity when there is no need for complex operations.

2.2 The dictionary problem

The authenticated dictionary problem has already been defined in the literature, for example in [22, 8]. In this section, we summarize the main features of an authenticated dictionary. The source has a set S of elements which evolves over time through insertion and deletion of items. The directory maintains a copy of this set and its role is to answer queries from the users. A user may request a given element or may perform a membership query on S in order to know whether an item belongs or not to S. The user must be able to verify the attached cryptographic proof (in particular, public information about the source must be available).

Efficiency makes the difference between a good dictionary and a bad one. This efficiency can be measured in terms of computation cost, which is the time taken by the computation together with the cost of the hardware (memory space and bandwidth) used by the entities. The size of the proofs is perhaps the most important parameter since it plays a significant role on the interface bandwidth of the directory. Moreover, it may reduce the time for a user to verify the answer to a query. The time spent by the directory to answer a query is also an important parameter when the number of users is very large. Space used by the data structure as well as source to directory communication should be optimized. Finally, the time to perform an update should also be optimized.

In this paper, our objective is to reduce the average size of the proof. This improvement is done at the expense of a slightly greater need for memory and computation of both the source and directory.

3 A new authenticated dictionary based on frequency

So far, authentication schemes have relied on data structures like Merkle trees or skip-lists. These data structures allow us to obtain small sizes of proof. In this sense, they seem to be optimal whenever each data has the same probability to be queried. However, in real life, users can make more queries on a given data than another. This means that the frequency of queries may be far from uniform. In the case of publication on

the Internet, some Web pages are consulted more frequently than others. Taking into account this parameter, we introduce a scheme in which the size of authentication proof answering a query is smaller when the frequency of this query is higher. We obtain the following benefits:

- for the directory, we minimize on average the LAN/WAN interface bandwidth usage. This interface bandwidth represents a critical aspect because the number of simultaneous queries may be high.
- If the directory caches proofs which are frequently queried, the number of proofs being cached will be higher, improving at the same time efficiency.
- On average, for a given user, the LAN/WAN interface bandwidth and the number of calculations to verify a proof is reduced.

When the frequency distribution is uniform, it is preferable to use an almost balanced tree (or an equivalent data structure). However, when the frequency distribution is not uniform, there is no reason to use such a data structure. Rather, we should look for unbalanced tree structures in order to improve efficiency, in particular on the size and construction of proofs.

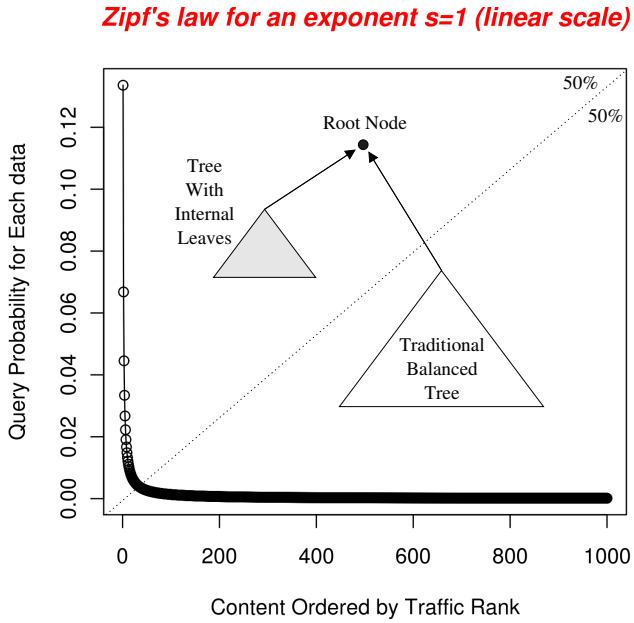

Fig. 2: Zipf's law and the tree T

We consider here a distribution which follows Zipf's law. Figure 2 shows that the distribution curve is close to the vertical axis for high frequency events whereas it is close

to the horizontal axis for the many very low frequency events. The latter part of the curve (which corresponds to the lower tail) behaves like a uniform distribution. Therefore, if we had to construct an authenticated dictionary corresponding to the lower tail, we would certainly use a balanced tree or any equivalent data structure (denoted T_2). However, for the rest of the distribution, we should use an unbalanced tree (denoted T_1), having its leaves ever closer to the root as frequency increases. Finally, in order to take into account the whole distribution, we propose to use a tree T whose root has T_1 as left child and T_2 as right child.

Since T does not arrange data in order of key identifiers but in descending order of frequencies, we need to use two other (non authenticated) structures, one ordering all the data according to frequencies and the other one ordering data according to key identifiers. Our scheme relies on the following data structures.

- We assume the use of two efficient dynamic binary trees which serve to organize and manage data. The first one, denoted A_1, ranks the $(u_i)_{i=1\ldots n}$ in ascending order and allows us to search a given u_i and to retrieve its corresponding frequency. The searching operation only uses A_1 and is done in $\mathrm{O}\big(\log(n)\big)$. The second one, denoted A_2, is used to arrange frequencies in decreasing order and allows the rank of a given frequency to be retrieved.
- Authentication proofs are constructed using the third data structure, T. Its right child T_2, is a Merkle-like tree which processes data having very low frequencies. The left child, T_1 is a height-balanced tree with special properties: each node has three children, two of them being either parent nodes or leaves and the third one being exclusively a leaf. The structure T_1 is designed to reduce the size of proofs corresponding to high frequency data. The place of each leaf depends on the frequency of the data. The higher the frequency, the closer the leaf is to the root.

Even though the system uses more data structures than existing authenticated dictionaries, the global memory space taken by these structures is not significantly increased. In fact, adding structures is mainly equivalent to adding pointers which do not have a high memory cost.

Remark 1. For the sake of simplicity we assume that all the frequencies are distinct. We note that this is in fact the case if we consider the exact Zipf distribution. Furthermore, for the construction of the structure, we just consider absolute frequencies (an absolute frequency being the number of data access requests).

The next subsection presents some details about this authenticated data structure and assumes the use of A_1 and A_2.

3.1 Authenticated data structure construction

Let n be the number of data. Considering the use of a cryptographic hash function H, let $\{u_1, u_2, \ldots, u_n\}$ be the set of hashed identifiers, let $\{c_1, c_2, \ldots, c_n\}$ be the set of hashed data and let $(f_1, f_2, \ldots, f_n)$ be the corresponding list of n frequencies. We denote by Π the permutation in $[1, \ldots, n+1]$ such that u_i has a frequency $f_{\Pi(i)}$.

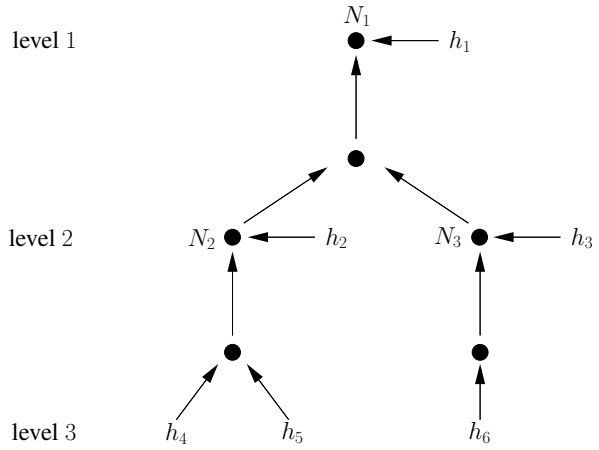

Fig. 3: Example of a tree T_1 for 6 elements. This diagram represents the flow of the computation of the nodes. Note that here the pairwise chaining for the computation of leaves is not depicted and that arrows denote the flow of information, not pointer values in the data structure.

In order to construct our tree T and its two children T_1 and T_2, we divide the data into two sets according to their frequencies, or more precisely in our case, according to the median of the frequency distribution. Each data corresponds to a leaf. Leaves of T_1 correspond to data having the highest frequencies $f_1, f_2, \ldots, f_k$ (ranked in descending order, where k is the smallest integer such that $\sum_{i=1}^{k} f_i \geq (\sum_{i=1}^{n} f_i)/2$. Leaves of T_2 correspond to the rest of the data. In practice, the number of leaves of T_2, denoted N_r, is much larger than that of T_1, denoted N_l. The tree T_2 is a Merkle tree for standard authenticated dictionaries. We also consider two special data, $\pm\infty$ both of frequency equal to zero which are used as sentinels in order to chain data according to their identity. The sentinel $+\infty$ is the last element of both lists and $f_{\Pi(n+1)} = f_{n+1} = 0$.

The source constructs the ordered sets

$$L^u = \{(u_1, f_{\Pi(1)}), \ldots, (u_n, f_{\Pi(n)}), (+\infty, 0)\}$$

and

$$L^f = \{(u_{\Pi^{-1}(1)}, f_1), \ldots, (u_{\Pi^{-1}(n)}, f_n), (u_{\Pi^{-1}(n)}, f_n), (+\infty, 0)\}.$$

In the first list the values u_i are ordered from the smallest to the largest, whereas the second list is ranked according to frequency. From these lists, the source calculates the tree T. Calculation of a leaf h_i is done as follows:

- $h_{\Pi(1)} = H(-\infty, u_1, c_1)$,
- $h_{\Pi(i)} = H(u_{i-1}, u_i, c_i)$ where $i \in [2, \ldots, n]$,
- $h_{\Pi(+\infty)} = H(u_n, +\infty, 0)$,

where 0 denotes empty content. Note that a pairwise chaining between the u_i (and $-\infty$, $+\infty$) is used when calculating the leaves, this device serves for constructing proofs of non-existence.

We determine i such that $2^i \leq N_l < 2^{i+1} - 1$. The calculation of nodes of T_1 is done as follows:

- $N_k = H(h_k, H(h_{2k}, h_{2k+1}))$ for $k \in [\![2^{i-1}, \lfloor \frac{N_l}{2} \rfloor [\![$;
- $N_{\lfloor \frac{N_l}{2} \rfloor} = H(h_{\lfloor \frac{N_l}{2} \rfloor}, H(h_{2\lfloor \frac{N_l}{2} \rfloor}, h_{2\lfloor \frac{N_l}{2} \rfloor + 1}))$ if N_l is odd,
  $N_{\lfloor \frac{N_l}{2} \rfloor} = H(h_{\lfloor \frac{N_l}{2} \rfloor}, H(h_{2\lfloor \frac{N_l}{2} \rfloor}))$ otherwise;
- $N_k = H(h_k, H(N_{2k}, N_{2k+1}))$ for $k \in [\![2^{i-2}, \lfloor \frac{N_l}{4} \rfloor [\![$;
- $N_{\lfloor \frac{N_l}{4} \rfloor} = H(h_{\lfloor \frac{N_l}{4} \rfloor}, H(N_{2\lfloor \frac{N_l}{4} \rfloor}, h_{2\lfloor \frac{N_l}{4} \rfloor + 1}))$ if $\lfloor \frac{N_l}{2} \rfloor$ is even,
  $H(h_{\lfloor \frac{N_l}{4} \rfloor}, H(h_{2\lfloor \frac{N_l}{4} \rfloor}, h_{2\lfloor \frac{N_l}{4} \rfloor + 1}))$ otherwise;
- $N_k = H(h_k, H(h_{2k}, h_{2k+1}))$ for $k \in]\!] \lfloor \frac{N_l}{4} \rfloor, 2^{i-1} - 1]\!]$;
- $N_k = H(h_k, H(N_{2k}, N_{2k+1}))$ for $k \in [\![2^j, 2^{j+1} - 1]\!]$ and $j \in [\![0, i-3]\!]$;

Leaves are listed in descending order of frequency, from the root to the base level and in a given level from left to right. Figure 3 shows the structure of T_1 for 6 data.

The calculation of nodes of T_2 is not detailed since T_2 is a Merkle-like tree. When the tree T is calculated, the source transmits the list of elements $(Id_i, c_i)_{i=1...n}$ together with the timestamped signature of the root node of T to the directory. Then, the directory is able to construct the data structures.

3.2 Proof construction and verification algorithms

In order to construct a proof of existence or non-existence for a data of (hashed) identifier u, we first use A_1 to determine the frequency $f_{\pi(j)}$ which corresponds to the smallest u_j such that $u_j \geq u$. Then, we use A_2 to obtain the place $\pi(j)$ of this frequency. Suppose that $h_{\pi(j)}$ is a leaf of T_1. The binary representation of $\pi(j)$ is used to obtain the correct path in T_1 leading to the authentication proof. We consider a list P initially empty, which will contain the hashed values representing the proof. When we know the path, the construction of the proof is similar to the one used in a Merkle tree:

- The value of the root node of T_2 is added to P.
- The most significant bit of $\pi(j)$ is not considered but we consider the following one. If this bit is equal to 0, we add to P both the value of the right child node and of the internal leaf and we move to the left node. If this bit is equal to 1, we add to P both the value of the left child node and of the internal leaf and we move to the right node.
- The process is repeated for the next bit and so forth until the last bit of $\pi(j)$. Note that, at the end, if the data corresponds to an internal leaf, we add to P the hash of the concatenation of the two children (or the hash of the child, if there is just one child).
- In the case of a proof of existence, we add to P the value u_{j-1}. In the case of a proof of non existence, we add to P the values u_{j-1}, u_j and c_j.

All the other cases (and in particular the one where $h_{\pi(j)}$ belongs to T_2) can be easily handled and are left to the reader. The verification of the proof is done by the user and involves recalculating the root node of T from the value of the leaf corresponding to the

data and hashes of the values of the proof. Note that the use of a commutative hash [8] for node calculation facilitates the calculation of the verification and slightly reduces the size of the proof.

3.3 Updating algorithms

The source maintains its own copy of the authenticated dictionary and provides the directory with the necessary information for updating. Such information contains the type of operation to be made, the element $(Id, C(Id))$, and a signed timestamp of the new value of the root node of T. When updating the dictionary, dynamic data structures A_1, A_2 and T must be partially modified while maintaining the overall consistency of the system.

Updating T consists of updating either T_1 or T_2 or both T_1 and T_2 and recomputing the root node of T. We suppose that T_2 is an "append/disjoin-only" Merkle tree, that is to say a Merkle tree in which incremental insertions/deletions are made on the right side of the tree.

Remark 2. The use of a static structure for T_2 allows us to obtain a proof (at least) as short as would be the case with a dynamic structure. Moreover, deletion and insertion remain efficient since the position of the elements does not depend on any rank.

Insertion of an element Insertion of a new *pair element* $(Id,\ c)$ where $H(Id) \notin (u_i)_{i=1\ldots n}$, is done in T_2 since we consider the data to have zero frequency (it has never been queried before). The following operations must be done on T.

- We first determine the largest index j such that $u_j < H(Id) < u_{j+1}$.
- The existing leaf $h_{\Pi(j+1)} = H(u_j, u_{j+1}, c_{j+1})$ is changed to $h_{\Pi(j+2)} = H(H(Id), u_{j+1}, c_{j+1})$.
- A new leaf $h_{\Pi(j+1)} = H(u_j, H(Id), c)$ is created on the right side of the tree.
- Internal nodes corresponding to paths from each of these two leaves to the root node of T are recomputed.

Updating an element Here, we focus on the operation which changes the content of an existing element (Id, c) to c'.

- We find j such that $u_j = H(Id)$.
- We set $h'_{\Pi(j)} = H(u_{j-1}, u_j, c')$.
- We recompute the nodes of the path from the updated leaf $h'_{\Pi(j)}$ to the root node (these nodes may belong to either T_1 or T_2).

Content reordering When the frequency of an element has changed, T must be updated. There are three possibilities:

- The leaf belongs to T_1 and will stay in T_1.
- The leaf belongs to T_1 and will move to T_2.
- The leaf belongs to T_2 and will move to T_1.

In this paper, we focus on the first case, the two other cases are easier to deal with and are left to the reader. We describe an updating algorithm to maintain the frequencies in a certain descending order. For the sake of simplicity, we suppose that we just have to move one element of frequency f_m ($m > i$) between f_i et f_{i+1}. We avoid the use of cyclic permutations since it would lead to update too many nodes (the cost would be in $\mathrm{O}\big((m-i)\log(n)\big)$). We prefer to use an algorithm that we call *min-max* which limits to one the number of changes at each level of the tree T_1. Let h be the depth of the tree. We denote by S_i ($i = 1 \dots h+1$) the set of elements $(Id, C(Id))$ whose leaves belong to level i of the tree. Let $f(.)$ be the map which associates its frequency to a key identifier. At the same level, frequencies are not ordered but we must have:

$$\forall\, i = 1 \dots h-1,\ \forall\, Id_x \in S_i,\ Id_y \in S_{i+1}\ f(Id_x) \geq f(Id_y). \tag{1}$$

Suppose that the leaf authenticating an element $e = (Id, C(Id))$ belonging to level i must move up to level j ($i > j$). This leaf is inserted at level j at the position of the leaf having the lowest frequency in this level. This last element is moved down to level $j+1$ at the position of the leaf having the lowest frequency, and so on. The leaf having the lowest frequency at level $i-1$ is moved up to the former position of e at level i. Finally, nodes which are on the path of the leaves that have been moved are recomputed back up the root of the tree. The algorithm is similar when $i < j$ except that the lowest frequency must be replaced by the highest frequency. If $i = j$, no change has to be done. If more than one frequency has changed, this algorithm can be applied for each change, albeit optimizations are possible but out of the scope of this paper.

Deletion of an element Here, we just outline the main steps of the deleting operation. Deleting an element leads to similar operations to that of reordering a leaf in T, with an updating of the pairwise chaining. Suppose that an element e is deleted at level i, the highest frequency element at level $i+1$ is moved up to the position of e and its position will be taken by the element having the highest frequency at the next level, and so on until level h. At level h, the element at the right side is moved to the position left empty. Suppose that the frequency of e is f_i, then an updating of the pairwise chaining must be done. The new value of $h_{\pi(\pi^{-1}(i)+1)}$ is $H(u_{\pi^{-1}(i)-1}, u_{\pi^{-1}(i)+1}, c_{\pi^{-1}(i)+1})$. Furthermore, values of the ancestor nodes of leaves that have been moved have to be recomputed to restore the consistency of information.

4 Complexity analysis

In this section, we analyze the complexity of the authentication part of our dictionary. We first concentrate on the size of a proof (of existence), then on the complexity of the proof construction and on the verification. Finally, we analyze the complexity of the updating operations. For the sake of simplicity, we express the operation cost in terms of the number of hash operations. We can then deduce the overall number of blocks

processed by the hash function, which is approximately a multiple[1] of the number of hash evaluations.

Authentication proof size and verification run time. In the following, we give the existence proof size in terms of the number of hash values needed to recompute the root node of T. The proof of non-existence for a hashed *identifier*, denoted u^*, is not detailed since this can be considered as a proof of existence for a particular hashed *identifier* u_i such that $u_{i-1} < u^* < u_i$ for a given $i \in [\![1, n+1]\!]$. In this case the server has to provide to the client, in addition, the values of u_i and c_i. We detail different cases, the best case, the worst case and the average case. From the average proof size standpoint, we discuss for which probability distribution it is preferable to use only T_1 or T_2, or the combination of both (T). Note that we express the verification complexity in terms of number of hash operations. In our construction, this number is close to the number of hash values contained in the proof.

Theorem 1. *Considering a number of elements $n \geq 1$, the authentication proof is of length 3 in the best case, and of length $\lceil \log(n-m) \rceil + 2$ in the worst case, where m is the Zipf distribution median.*

Proof. In the best case, the requested content is the most frequently viewed. The leaf corresponding to the requested *identifier* is then located at the root node of T_1. Assuming that the requested identifier corresponds to a hashed value u_i, the user needs the preceding hashed identifier u_{i-1}, plus the hash of the concatenation of the sibling nodes $H(N_2, N_3)$ (Both u_i and c_i are determined locally, once the content is downloaded), and finally the root node value of T_2. In the worst case, the requested content is located in the tree T_2 which is an almost balanced binary tree, the user needs the preceding hashed identifier u_{i-1} plus the siblings of the nodes along the path from the leaf to the root node, plus the root node value of T_1, for a total of at most $\lceil \log(N_r) \rceil + 2$ values.

Average case. If the queries are uniformly distributed in the set of elements, there is no reason to use the tree T_1, due to the overhead of the internal nodes. In this case we should only use the almost balanced binary tree T_2 in order to have an average size for the authentication proof tightly upperbounded by $\lceil \log(n) \rceil + 1$. In order to show that the tree T_1 is useless in this case, let us determine the average proof size when we only use this tree. Consider n such that $n = 2^m - 1$. The proof size for an internal leaf belonging to a level i, for $i \in [\![1, m-1]\!]$, is $2(i-1)+2$ hash values whereas it is $2(m-2)+2$ for a base level leaf (level m). Take the derivative of the geometric series $\sum_{i=0}^{m-1} x^{i+1}$ and simplify the following average proof size of $\frac{1}{(2^m-1)}(\sum_{i=0}^{m-2}(2i+2)2^i + 2^{m-1}(2(m-2)+2))$. We then deduce that, when using only T_1, the average proof size is close to $2\lfloor \log(n) \rfloor$ hash values.

By contrast, when the queries are distributed according to a geometric distribution of parameter $p = 1/2$ (which is close to a discrete equivalent of an exponential law), it

[1] In our system, the number of blocks processed in one evaluation of the hash function can vary: this is the tree arity of T_1 plus one (that is, 4) for a node evaluation of T_1, the tree arity of T_2 (that is, 2) for a node evaluation of T_2 and 3 for a leaf evaluation.

is best to use only T_1. Indeed, by evaluating a geometric series, one can deduce that in such a case the average proof size is asymptotically 4 hash values.

As regards the Zipf distribution, it is preferable to use an authenticated data structure like T because frequencies do not decrease as fast as an exponential law. Zipf is based on harmonic series and therefore it is difficult to provide a bound of complexity that closely reflects reality. Consequently, we give in Table 1 numerical results by varying the dictionary size, along with the percentage gain compared to what is obtained with the use of a standalone Merkle-like data structure.

Dictionary size	Merkle-like structure	Our system	Improvement
10^3	9.97	8.05	19.5%
$5 \cdot 10^4$	15.61	12.25	22.5%
$5 \cdot 10^5$	18.93	14.73	22.5%
10^6	19.93	15.46	22.5%

Table 1: Average proof size and verification cost results

Proof construction. If, for each node of the tree, hashes of the concatenation of left and right children are stored in memory, the construction cost is equal to the cost of a searching operation (expressed as the number of comparisons) which is in $\mathrm{O}\big(\log(n)\big)$. However, if these hashes have to be recalculated, the global cost is upper bounded by the number of hash calculations to be done on the path, which is itself upper bounded by the depth of the tree.

Update complexity. We focus here on the updating of one element, in terms of hash computations.

Theorem 2. *When modifying the content of an element, the number of hash evaluations to update T is upper bounded by $\lceil \log(n) \rceil + 2$ where n is the overall number of elements in the dictionary.*

Proof. We update a leaf for a cost of one hash evaluation. Ancestor nodes of this leaf until the root node of T_1 (or T_2) need to be updated for a cost bounded by $\lceil \log(n) \rceil$. Finally the root node of T needs to be updated for a cost of one hash evaluation.

Theorem 3. *When inserting a new element of frequency $f = 0$, the number of hash evaluations is upper bounded by $2\lceil \log(n) \rceil + 3$ where n is the overall number of elements after insertion.*

Proof. One pairwise link (one leaf) is changed in two pairwise links (2 leafs) for a cost of two hash function evaluations. The ancestor nodes of these two leaves, which can be located in T_2 or in T_1 and T_2, need to be computed (or re-computed) for an overall cost of at most $2\lceil \log(n) \rceil + 2$ hash evaluations. Finally a last hash computation is needed to recompute the root node of T.

Theorem 4. *Assume that the absolute frequency of one element has changed and that this element which belongs to T_1 stays in T_1, the number of hash computations needed to meet the order property (1) is at most* $\frac{\lfloor \log(n) \rfloor (\lfloor \log(n) \rfloor + 1)}{2}$.

Proof. Let us suppose that the frequencies of the leaves from the root to the base level and from left to right are denoted f_1, f_2, ..., f_n. We consider the worst case which appears when a leaf of the base level needs to be moved at the root level, for instance if the frequency f_n is changed in f_n^* and $f_n^* > f_1$. Let h be the depth of the tree. In this scenario, by using the "min-max" choice criteria, we move the leaf h_n to the root level, while h_1 is moved down to the next level at the position of the leaf of lowest frequency. This last leaf is itself moved to the next level at the position of the leaf of lowest frequency, and so on, until the base level is reached. The lowest frequency leaf at level h is moved to the former position of h_n at level $h+1$ (the base level). Overall, one leaf has been replaced at each level of the tree. Nodes along the path from changed nodes to the root node are updated. This non-optimal strategy leads to the following upper bound on the number of hash computations: $\sum_{i=1}^{h} i = \frac{\lfloor \log(n) \rfloor (\lfloor \log(n) \rfloor + 1)}{2}$

Theorem 5. *When deleting an element in T_1, by using the "min-max" choice criteria, the number of hash computations is in* $\mathrm{O}\big(\log(n)^2\big)$.

Proof. Operations are similar to that of reordering an element, except that one leaf and its ancestor nodes need to be recomputed.

Search complexity. The cost of a search operation is given in terms of comparisons.
Search of a content: Since non authentifying structures use well known mechanisms, we do not describe the search algorithm. The content and frequency associated to a given u_i is obtained using A_1 and is done in $\mathrm{O}\big(\log(n)\big)$ comparisons. The rank of the frequency is obtained using A_2 and is also done in $\mathrm{O}\big(\log(n)\big)$ comparisons. Globally, the search of a leaf and the construction of the proof is done in $\mathrm{O}\big(\log(n)\big)$ comparisons.
Search of an element: The cost is done in $\mathrm{O}\big(\log(n)\big)$ comparisons for insertion and modification of an element. The position of a leaf having a minimal (or maximal) frequency in a given level of the tree is done in $\mathrm{O}\big(\log(n)\big)$ comparisons.

Remark 3. From the previous analysis, we can deduce that reordering or deleting operations is done in $\mathrm{O}\big(\log(n)^2\big)$ comparisons.

5 The choice of the structure

Our objective is to optimize the average proof size, avoiding the expensive worst cases. With our structure, the maximal proof length is bounded by $\lceil \log(n) \rceil + 2$. A dynamic Huffman tree gives slightly better results for the average case but the proof size in the worst case is in $\mathrm{O}\big(n\big)$ for outlier (discrepant) samples. This worst case occurs for example when the distribution is exponential. An alternative is to use a dynamic Huffman tree for a small subset of data. This solution limits the expensive cost of the worst case. When considering a sample distant from Zipf's law, we may obtain a degenerate tree. In that case, the use of a length-limited Huffman tree [11] may be considered.

Note that our structure has the advantage of simplicity (in particular, the construction of our tree is done in $\mathrm{O}(n)$). Moreover, it provides all needed operations while keeping append/disjoin-only structures.

6 A framework to authenticate http responses

The aforementioned authenticated dictionary can be used for authentication of HTTP responses of a Web server [2] when the request distribution follows Zipf's law. The server returns either the requested page together with a 200 success response or a 404 error message and a proof of authenticity of the content of that page (or possibly a proof of non existence). In this context, u_i represents the hash of a url (Uniform Resource Locator) and c_i is the content of the corresponding page. However, it is important to note that in a dynamic site, a page has many contents which vary over time. Hence, it makes no sense to consider the hash of a page. When creating a page, one should define a scheme allowing authentication to be performed on the static fields which are of interest to the user.

7 Conclusion

We have proposed a model for authenticated dictionaries which takes into account the frequency of queries with the aim of obtaining a smaller proof size. This contrasts with the assumption made by existing dictionaries that the frequency distribution is uniform. Based on a frequency distribution following Zipf's law, we introduced a data structure with two components, each of which being nearly optimal for a portion of the distribution. We obtained an average gain of more than 20% on proof size while response time remains similar. In our complexity analysis, comparisons were made with a Merkle tree which is less costly than the dynamic structures used in [8]. However, since our system provides operations like insertion and deletion, it could also be compared to these dynamic structures, with the expectation that even greater gains would be realized.

In this paper, we have not discussed possible optimizations, including the use of length-limited Huffman trees, which will be developed in future papers.

References

1. L. A. Adamic and B. A. Huberman. Zipf's law and the internet. *Glottometrics*, 3:143–150, 2002.
2. R. J. Bayardo. Merkle tree authentication of http responses. In *In Proc. 14th WWW*, pages 1182–1183, 2005.
3. K. Blibech and A. Gabillon. Chronos: an authenticated dictionary based on skip lists for timestamping systems. In *SWS*, pages 84–90, 2005.
4. A. Buldas, P. Laud, and H. Lipmaa. Accountable certificate management using undeniable attestations. *IACR Cryptology ePrint Archive*, 2000:27, 2000.
5. P. Devanbu, M. Gertz, C. Martel, and S. G. Stubblebine. Authentic Third-party Data Publication. In *14th IFIP 11.3 Working Conference in Database Security*, 2000.

6. D. Easley and J. Kleinberg. Power laws and rich-get-richer phenomena. In *Networks, Crowds, and Markets: Reasoning about a Highly Connected World*. Cambridge University Press, 2010.
7. I. Gassko, P. Gemmell, and P. D. MacKenzie. Efficient and fresh cerification. In *Public Key Cryptography*, pages 342–353, 2000.
8. M. T. Goodrich and R. Tamassia. Efficient authenticated dictionaries with skip lists and commutative hashing. Technical report, tech. rep., Johns Hopkins Information Security Institute, 2001.
9. M. T. Goodrich, R. Tamassia, and A. Schwerin. Implementation of an authenticated dictionary with skip lists and commutative hashing. 2:68–82, 2001.
10. C. Kaufman, R. J. Perlman, and M. Speciner. *Network security - private communication in a public world.* Prentice Hall series in computer networking and distributed systems. Prentice Hall, 1995.
11. L. L. Larmore and D. S. Hirschberg. A fast algorithm for optimal length-limited huffman codes. *J. ACM*, 37(3):464–473, July 1990.
12. A. Mahanti, N. Carlsson, A. Mahanti, M. Arlitt, and C. Williamson. A tale of the tails: Power-laws in internet measurements. *Network, IEEE*, 27(1):59–64, 2013.
13. D. P. Mehta and S. Sahni, editors. *Handbook of data structures and applications*. Chapman & Hall/CRC, 2005.
14. R. Merkle. Protocols for public key cryptosystems. In *SIMMONS: Secure Communications and Asymmetric Cryptosystems*, 1982.
15. R. Merkle. A certified digital signature. In *Proceedings on Advances in Cryptology*, volume 435 of *CRYPTO '89*, pages 218–238. Gilles Brassard, 1990.
16. M. Naor and K. Nissim. Certificate revocation and certificate update. *IEEE Journal on Selected Areas in Communications*, 18(4):561–570, 2000.
17. J. Nielsen. Do websites have increasing returns? `http://www.nngroup.com/articles/do-websites-have-increasing-returns/`, 1997.
18. J. Nielsen. A note about page popularity. `http://www.nngroup.com/articles/traffic-log-patterns/`, 2006.
19. B. Preneel, B. Van Rompay, J. J.Quisquater, H. Massiasand, and J. Serret Avila. Design of a timestamping system, 1999.
20. W. Pugh. Skip lists: A probabilistic alternative to balanced trees, 1990.
21. A. I. Saichev, Y. Malevergne, and D. Sornette. *Theory of Zipf's Law and Beyond*. Lecture notes in economics and mathematical systems. Springer, 2009.
22. R. Tamassia and N. Triandopoulos. On the cost of authenticated data structures. In *In Proc. European Symp. on Algorithms, volume 2832 of LNCS*, volume 2832, pages 2–5. Springer, 2003.