HAL
open science

New models for efficient authenticated dictionaries

Kevin Atighehchi, Alexis Bonnecaze, Gabriel Risterucci

To cite this version:

Kevin Atighehchi, Alexis Bonnecaze, Gabriel Risterucci. New models for efficient authenticated dictionaries. Computers & Security, 2015, 53, pp.203-214. 10.1016/j.cose.2015.04.010 . hal-01222953

HAL Id: hal-01222953

https://hal.science/hal-01222953v1

Submitted on 31 Oct 2015

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.

New Models for Efficient Authenticated Dictionaries

K. Atighehchi, A. Bonnecaze, G. Risterucci

Aix Marseille University, CNRS, Centrale Marseille, I2M, UMR 7373, 13453 Marseille, France

Abstract

We propose models for data authentication which take into account the behavior of the clients who perform queries. Our models reduce 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. Our models better reflect reality and the communication cost between clients and the server provider is reduced allowing the server to save bandwidth. The obtained gain on the average proof size compared to existing schemes depends on the parameter of Zipf law. The greater the parameter, the greater the gain. When the frequency distribution follows a perfect Zipf's law, we obtain a gain that can reach 26%. Experiments show the existence of applications for which Zipf parameter is greater than 1, leading to even higher gains.

Keywords: Authenticated dictionary, Data structure, Merkle tree, Huffman code, Zipf, Time series

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 [1, 2], Web mail search results [3], geographic information system querying, or third party data publication on the Internet [4, 5]. 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 [6, 7]: 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,

Email addresses: `kevin.atighehchi@univ-amu.fr` (K. Atighehchi), `alexis.bonnecaze@univ-amu.fr` (A. Bonnecaze), `gabriel.risterucci@gmail.com` (G. Risterucci)

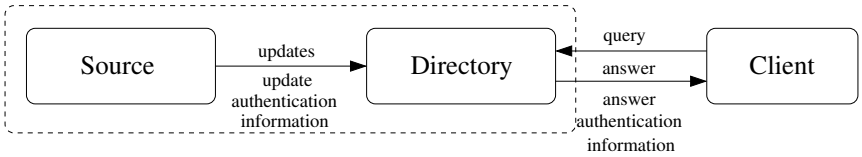


Figure 1: The three-party authentication model

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. Note that the source and the directory are not necessarily hosted on two distinct machines.

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 [6]. 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 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 behavioral criterion could be taken into account.

In this paper, we introduce authenticated dictionary schemes which take into account the frequency of data being accessed. As regards Web traffic, it is well known that its frequency distribution follows Zipf's law [8, 9]. 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. Zipf law depends on a parameter (also called exponent). When this parameter is equal to 1, Zipf law is so-called perfect and the greater the parameter, the greater the gain.

The paper is organized as follows. Section 2 contains background information regarding data structures and the dictionary problem. Section 3 briefly presents the main types of existing dictionaries. In Section 4, we introduce the notion of frequency and we show its importance to optimize the efficiency of the dictionary, in particular in terms of authentication proof size. Since the frequency distribution varies over time, the way of updating the content of the data structure represents a major issue. A possible solution is to regularly reconstruct the structure. Another solution is to make some updates without reconstructing the whole structure. In the case of a reconstruction, a (current) frequency distribution must be considered since the structure depends on it. This frequency distribution can be calculated based on the preceding frequencies, either directly or using a predicting function. Section 5 focuses on the data structures that have to be used in order to take into account the frequency parameter. In Section 6, we propose the construction of a dictionary based on append/disjoin-only data structures. In Section 7, we introduce Huffman trees to improve efficiency. When the frequency distribution follows a perfect Zipf's law, we obtain a gain that can reach 26% on the average proof size compared to existing schemes. The use of an adaptive Huffman tree is discussed in Section 8. The decision of completely rebuild or just update the

structure depends on the nature of the dictionary and its desired efficiency properties. In Section 9, we consider two use cases and we analyze the obtained gains. The first experiment gives non significant gains since Zipf law parameter is less than 1, whereas the second experiment exceeds the expected gains of a perfect Zipf law because the parameter is equal to 1.4.

2. Authenticated Dictionary Problem

2.1. Dictionary features and efficiency

The authenticated dictionary problem has already been defined in the literature, for example in [6, 10]. 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 the rest of the article, n represents the number of data elements ($n := \#S$) and H represents a cryptographic hash function. The unique identifier of an element is denoted Id_i for $i \in \{1, \ldots, n\}$, its hashed identifier is $u_j = H(Id_i)$ for $j \in \{1, \ldots, n\}$ such that the values u_j are ordered ascendingly. Its content is denoted C_j and its hashed value c_j.

2.2. 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 [11] are hash tables, trees like 2-3 Trees, B-Trees or red-black trees, or other random structures like non-deterministic skip-lists [12].

Authentication can be provided by multiple cryptographic primitives like hash functions, digital signature algorithms or arithmetic accumulator functions. The simplest, albeit not the most efficient approach consists in signing each element from the dictionary with a standardized signature algorithm like ECDSA. In fact, the performance of the system mainly depends on the type of cryptographic tools and the way to use them.

3. Existing Authenticated Dictionaries

In this section, we list the three main types of authenticated dictionaries. Many of their performances are analyzed in the state of the art of Crosby and Wallach [13].

3.1. Signature Based Authenticated Dictionary

The easiest way for creating an authenticated dictionary is to sign and timestamp each pair element (u_i, c_i) with a digital signature algorithm. However, in order to also provide proof of non-existence, it is preferable to sign and timestamp each triplet (u_i, u_{i+1}, c_i). This signature, authenticating the pair (u_i, c_i), certifies that there is no key in the interval $]u_i, u_{i+1}[$.

The first drawback of this system is the number of signatures to generate, which is in $\mathcal{O}(n)$. Moreover, signatures have to be regularly recomputed, either for datation or updating version (in the case of a "persistent" dictionary) reasons. Thus, this system is not to consider if the dictionary has a lot of elements and must be updated very often [13]. In fact, the only benefit of this system is the size of the authentication proof that has to be given to the user, since it is a single signature. In the context of persistent authenticated dictionaries, a speculation technique [14] was proposed to slightly improve the cost of an update, from $\mathcal{O}(n)$ to $\mathcal{O}(Cn^{1/C})$ signatures at the counterpart of longer proofs, where C is a chosen parameter. In this paper, we focus on "non-persistent" authenticated dictionaries.

In order to make efficient the access to a pair element, each element should be referenced in an efficient data structure. Such data structure can be an array or a search balanced tree which has a search operation cost in $\mathcal{O}(\log n)$. In case the dictionary is dynamic, the data structure is also dynamic and a red-black tree or a 2-3 tree can make efficient updates in $\mathcal{O}(\log n)$ (Throughout the paper, $\log_2$ is denoted $\log$).

3.2. Tree Based Authenticated Dictionary

In addition to their basic features, data structures can be used to construct authenticating mechanisms, while reducing the number of signatures to be generated by the authenticated dictionary. Data structures based on rooted graphs are well adapted to deal with such mechanisms [6, 10, 15, 16] 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 [17, 18] 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, 20]. 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.

3.3. Accumulator Based Authenticated Dictionary

While tree constructions allow authentication and updating to be done at best in logarithmic costs, some accumulators allow these operations to be done in constant time. Two types of accumulators can be distinguished: those which are based on an arithmetic problem like the RSA accumulator [21] or pairing based accumulators [22], and those which are based on other problems like the Nyberg accumulator [23] or the Bloom filter [24], based on probabilistic data structures. While similar, Bloom filter and Nyberg accumulator come from different domains. The first one comes from the domain of data structure whereas the later one as well as the RSA accumulator were introduced in a cryptographic context.

In order to implement an authenticated dictionary system, an accumulator alone is not enough. As with a hash tree, an accumulator is just an intermediary authentication system that has to be authenticated by another mechanism such as a digital signature or a more elaborated system.

Nyberg accumulator and Bloom filter are not competitive compared to hash tree in terms of efficiency. Similarly, Crosby and Wallach show in [13] the limits of the RSA accumulator in terms of calculations and communication. However, in 2008 Papamanthou et al. [25] propose a cryptographic construction based on RSA accumulators which exploits the efficiency of hash tables. In 2009, the same authors (in [26], published in 2015) proposed to combine the RSA accumulator with a pairing one in a nested way over a tree of constant depth. Asymptotically the client can optimally authenticate operations on hash tables with constant time and communication complexities. However, the cost for performing an update is much higher than the cost induced by using Merkle trees or their equivalent.

In summary, an authenticated dictionary based on a balanced hash tree still represents a good solution in terms of calculation cost. Its implementation and updates are faster. As stressed in the study of Crosby et Wallach, its drawback is the size of a proof, which is logarithmic in n, when compared with the first method based on signatures. Overall, if the number of requests between two updates is huge, the first solution based on a unique signature can be considered. Otherwise, the second solution using hash trees is more efficient. In this paper, we will always consider solutions based on tree structures.

4. Frequency Distribution Parameter

4.1. Frequency Notion

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. In order to take into account this frequency notion, our aim is to 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.

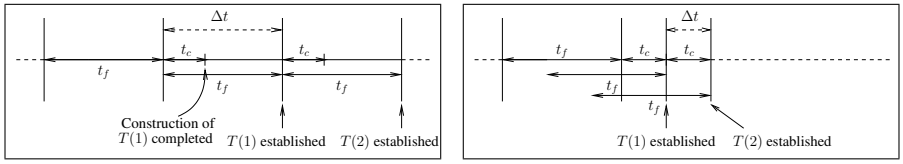
When the frequency distribution is uniform, it is preferable to use an almost balanced tree (or an equivalent data structure). However, when the frequency distribution is not uniform, there is no reason to use such a data structure. Rather, we should look for unbalanced tree structures in order to improve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

It is well known that the distribution of requests mainly follows Zipf's law. This 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.

4.2. Freshness of the Frequency Distribution

The tree structure needs to change according to the fluctuations of frequencies. Statistics are maintained during an interval which can either depend on a number of elements or be temporal. This interval is called a window and elements are key-value pairs. We can indeed estimate the frequencies on a fixed length window rather than upon the full sequence from scratch. An element which enters (respectively exits) the window has a frequency incremented (respectively decremented) by 1. Frequencies of elements entering the window are then increased, while frequencies of exiting elements are decreased. The window works as a first in first out (FIFO) queue. The length of the window is chosen in order to obtain the best estimation of the current distribution. When the length of the window is equal to a fixed number of elements, the frequency of a key-value pair is bounded by the window size and this window is moving at a variable speed, depending on the fluctuation (over time) of the number of accesses to the dictionary elements.

Galager [27] has proposed a method to age frequencies by multiplying them by a value $\alpha < 1$, that is, either every X accesses to the dictionary, or every t time units, according to the chosen convention. Frequencies are not reset to 0 but they are lowered before being incremented afresh, either after X accesses or t time units. This method



(a) Window

(b) Sliding window

Figure 2: Time-window

raises several problems including the need to entirely reconstruct the authentication tree each time we want to age past events. Therefore, it can not be used frequently if the tree construction is expensive (when the dictionary is huge), and so X or t should not be too large. Conversely the method should not be used too rarely otherwise it loses much of its relevance.

A better solution, from Cormack et al. [28], is to age past events relatively to current events, in the following way. We choose a value $s > 1$, close to 1, for instance $s = 1.01$. To increase a frequency, the next power of s is added to it. For example, the first element accessed has to be incremented by 1, the second has to be incremented by s, the third by s^2, and so on. When there is danger of arithmetic overflow for a large power of s, all frequencies are divided by this power of s. Then we continue the process from the beginning, incrementing by 1, then s, and so on. This method allows the weight (*i.e.* frequency) of recently accessed elements to be increased, moving them up near the root. This method was introduced by Cormack et al. [28] for real time compression of data. It is well adapted when accesses are consecutive. In the case of an authenticated dictionary, accesses can be simultaneous, and as a result, slightly changes have to be made (for example, simultaneous accessed elements can be incremented by the same power of s).

We propose another solution, much simpler, which consists to use a temporal window. This method allows the window to move at a constant speed and in this case, neither the frequency of an element nor the number of elements are bounded by the window size. We need to define some notations:

- t_c is the tree construction time. This time is a function of the number of elements to authenticate and the computer capabilities;
- t_f is the time interval in which we measure the frequencies to take into account for the next tree construction. Frequencies are then reinitialized when we start again to accumulate the number of accesses;
- Δt is the time interval between two consecutive (completed) constructions of T;
- $T(i)$ denotes the tree T constructed using the measures of the i-th time interval t_f.

The time unit has to be defined during practical tests. We necessarily have the constraint $\Delta t \geq \max(t_c, t_f)$. Note that $T(i)$ can be obtained either by completely reconstructing the tree or by updating it in order to reflect the changes between the $(i-1)$-th and the i-th measurement period. We can use either a window, as shown in Figure 2a, or a sliding window scheme, as shown in Figure 2b, which will slide the measurement period every t_c time units. In this case, after the first construction, we

have the constraint $\Delta t \geq t_c$ if $t_c \leq t_f$.

4.3. Frequency Predictions

The current frequency distribution is calculated from the preceding frequencies and can be determined in two ways. It can either use the preceding frequencies as they are or use them in a predictive model. When the fluctuation of frequencies is important between two time intervals (for example when considering a news website), the use of a predictive model can be of interest. This is the subject of this subsection.

There exist many techniques of prediction like time series forecasting [29] and more particularly the exponential smoothing [30] which makes use of the exponentially weighted moving average, a method often used in computer networking to estimate average round trip times, timeouts for TCP or queue lengths in routers.

More powerful models [29] are the ARIMA (Autoregressive Integrated Moving Average), SARIMA (Seasonal ARIMA to take into account seasonality) or VARIMA (Vector ARIMA for multivariate analysis) to name just a few. Notice that an approach could be to use an univariate model for each key-value pair. In such a case, the Box-Jenkins methodology [29] allows the model and its parameters to be selected correctly. These last methods represent powerful tools for accurate predictions and can be used to find the frequency distribution which will optimize the size of the authentication proofs.

5. Adequate Data Structures

Our aim is to construct an authenticated dictionary with all the required features, while reducing the size of the authentication proofs. In order to achieve this goal, we use three tree structures. Authentication proofs use a structure T which cannot be used for searching. In fact, T does not arrange data in order of key identifiers. Thus, two other (non authenticated) structures are needed, one ordering all the data according to key identifiers and the other one ordering data according to frequencies. More precisely, 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 a related information, for instance its corresponding content, frequency or a pointer toward a node of T. The second one, A_2, is used to find a particular frequency and to retrieve its corresponding pointers. The searching operations are done in $\mathcal{O}\big(\log(n)\big)$.
- Authentication proofs are constructed using the third data structure, T. As shown earlier, Zipf's 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. We divide the data into two sets according to their frequencies, and for example, according to the weighted median of the frequency distribution.

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.

In the next sections, we propose several possible structures for T_1 which take into account the frequency parameter. The first structure is presented in [31] and allows the efficiency to be improved compared to existing schemes. The second one allows efficiency to get closer to optimality and thus represents a better solution.

6. An Authenticated Dictionary Based on Append/Disjoin-only Data Structure

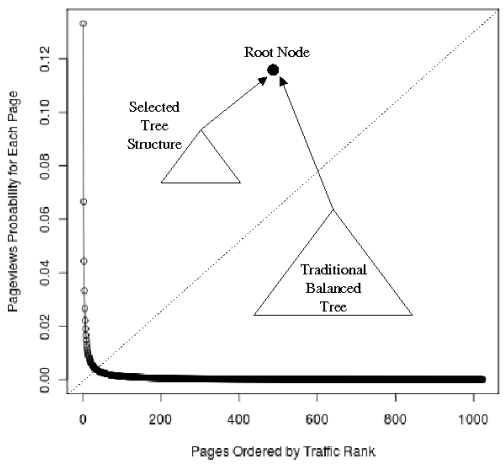
Intuitively, a good solution to our problem is to consider a tree T_1 having its leaves ever closer to the root as frequency increases, so as to reduce the size of proofs corresponding to high frequency data. In [31], we have proposed to use for T_1 an almost balanced[1] tree with internal leaves, as depicted in Figure 3b. In this paper, we simplify a little by proposing to use a Merkle-like tree for T_1 (thus T_1 and T_2 are both Merkle tree). Such a simplification will allow us to obtain a more efficient procedure to update frequencies with, as counterpart, a slightly reduced gain for the average proof size. We call this simpler solution Solution MM and the one using internal leaves Solution IM.

6.1. Authenticated data structure construction

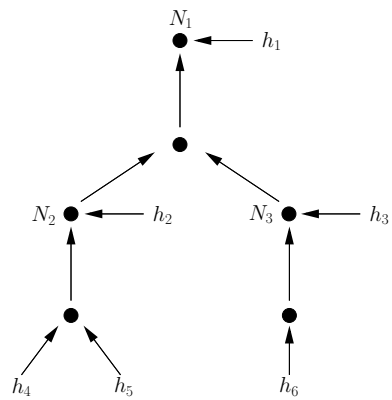
Notice that in order to construct its data structures the source needs information about frequency of access. It receives these information from the directory. As already introduced, $\{c_1, c_2, \ldots, c_n\}$ is the set of hashed data. Let $(f_1, f_2, \ldots, f_n)$ be the corresponding list of n frequencies. We denote by Π the permutation in $[1, \ldots, n+1]$ such that u_i has a frequency $f_{\Pi(i)}$.

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 weighted median of the frequency distribution (see Figure 3a). 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 trees T_1 and T_2 are Merkle trees 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. We set $f_{\Pi(n+1)} = f_{n+1} = 0$, $u_{\Pi(n+1)} = u_{n+1} = \infty$ and $u_0 = -\infty$.

[1]We say that a tree is almost balanced if the maximum difference in depth between two leaves is in $\mathcal{O}(\log n)$.



(a) Zipf law (linear scale) and the tree T



(b) Example of a tree T_1 with internal leaves for 6 elements. This diagram represents the flow of the computation of the nodes.

Figure 3: Append/Disjoin-only data structures for authentication

The source constructs the following ordered sets

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

and

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

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 then construct two Merkle trees, a hashed tree T_1 that authenticates the leaves h_1, ..., h_k and a hashed tree T_2 that authenticates the rest of the leaves. We recall that the leaves are listed in descending order of frequency, with the tree T_1 having the leaves of highest frequency. Finally the root node of T is obtained by computing the hash of the concatenation of the root nodes of T_1 and T_2.

When the tree T is calculated, the source transmits the list of elements $(Id_i, C_i)_{i=1\ldots n}$ together with the timestamped signature of the root node of T to the directory. Then, the directory is able to construct the data structures.

6.2. *Proof construction and verification algorithms*

The authentication proof for a data of identifier u is similar to the one used in a Merkle tree. The proof contains the list of hashed values which serve to calculate the value at the root of T starting from the leaf of identifier u. It also contains the time-stamped signature of the root. Let L be the list of leaves and nodes of T. The proof construction is done using the list L while building a list P which is initially empty. We add to P the hashed identifier directly preceding u in L^u. The process consists in finding in L the leaf corresponding to identifier u and adding the value of its sibling node in P. Finally, siblings of its ancestors have to be added until one of the two children has been added. Finally, sibling of each successive ancestor have to be added until the root has been reached.

At this point, P contains the hashed values representing the proof along with a binary code describing the path being taken in the tree.

The verification by the user of the authentication proof of an element of hashed identifier u_j consists of the following steps: (i) Compute c_j from the downloaded content; (ii) Retrieve u_{j-1} in P and compute $H(u_{j-1}, u_j, c_j)$; (iii) Iteratively apply this hash function on the rest of the data of P, in accordance with the path described in P, so as to compute the root of T. This step is performed in $\mathcal{O}(\log n)$ hash function evaluations; (iv) Check the signature on this root. If it is valid, it means that the calculated root is the correct one and the element of key u_j is considered as authentic.

In case of a proof of existence of u_j, the user does not download the associated content and as a result, the value c_j is added to the proof.

Lastly, if a hashed identifier u is not in the dictionary, a non-existence proof is needed. It consists in finding the smaller index k in the list of ranked hashed identifiers satisfying $u_k > u$ and constructing a proof of existence for u_k, the chaining (u_{k-1}, u_k) authenticated by the leaf $h_{\Pi(k)}$ certifying the non-existence of u.

6.3. *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_1 and T_2 are "append/disjoin-only" Merkle trees in which incremental insertions/deletions can be made efficiently on the right side of the trees. We will see that in fact deletion and insertion at any position remain efficient since, inside T_1 or T_2, the position of an element does not depend on any rank.

6.3.1. *Insertion of an element*

Insertion of a new *pair element* (Id, C) where $H(Id) \notin (u_i)_{i=1...n}$, 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: (i) We first determine the

largest index j such that $u_j < H(Id) < u_{j+1}$; (ii) The existing leaf $h_{\Pi(j+1)} = H(u_j, u_{j+1}, c_{j+1})$ is changed to $h_{\Pi(j+2)} = H(H(Id), u_{j+1}, c_{j+1})$; (iii) A new leaf $h_{\Pi(j+1)} = H(u_j, H(Id), H(C))$ is created on the right side of the tree; (iv) Internal nodes corresponding to paths from each of these two leaves to the root node of T are recomputed, leading to an overall computation time in $\mathcal{O}\big(\log n\big)$ hash evaluations.

6.3.2. Updating an element

Here, we focus on the operation which changes the content of an existing element (Id, C) to C'. Let us denote $c' = H(C')$ the new hashed content. We first find j such that $u_j = H(Id)$. Then, we set $h'_{\Pi(j)} = H(u_{j-1}, u_j, c')$. Finally, 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). The overall computation time to perform this update is in $\mathcal{O}\big(\log n\big)$ hash operations.

6.3.3. Content reordering

When the frequency of an element has changed, T must be updated. There are three possibilities: (i) The leaf belongs to T_1 and will stay in T_1; (ii) The leaf belongs to T_1 and will move to T_2; (iii) The leaf belongs to T_2 and will move to T_1.

In this paper, we focus on the third case, the two other cases are easier to deal with and are left to the reader. 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 $\mathcal{O}\big((m-i)\log(n)\big)$). We describe an updating algorithm to maintain the frequencies in a certain descending order. In fact, we allow the leaves to be in disorder in each of the two trees but each leaf of T_1 must have a frequency higher than each of the leaves of T_2. The leaf of frequency f_m is inserted at the position of the leaf having the lowest frequency in the tree T_1 and this last leaf is moved down to the former position of the leaf of frequency f_m. Finally, nodes which are on the path of these two leaves are recomputed back up the root of the tree. Since the height of T is in $\mathcal{O}\big(\log n\big)$, the overall computation time to perform this exchange stays in $\mathcal{O}\big(\log n\big)$ hash operations. 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.

This complexity has to be compared with the $\mathcal{O}\big(\log^2 n\big)$ hash operations required when using the tree with internal leaves introduced in [31] (and depicted in Figure 3b).

6.3.4. Deletion of an element

Deleting an element consists of moving the right most leaf of $T_i,\ i \in \{1, 2\}$ at the place of the leaf corresponding to the element to delete. An updating of the pairwise chaining is also required.

6.4. Proof size

We give here a summary about the authentication proof size. If we use a tree T_1 with internal leaves (as depicted in Figure 3b), then 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. If we use a Merkle-like tree for T_1, the authentication proof

is of length $\lceil \log(m) \rceil + 2$ in the best case and stays of length $\lceil \log(n-m) \rceil + 2$ in the worst case. If the queries are uniformly distributed in the set of elements, there is no reason to divide the set in two parts. In this case only a standalone Merkle-like tree is used in order to have an average size for the authentication proof tightly upperbounded by $\lceil \log(n) \rceil + 1$.

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

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.

Finally, we can see that the use of two subtrees instead of a single Merkle tree leads to significant lower size of proofs. Besides, the (relatively small) gain brought by the use of internal leaves in T_1 is thwarted by a cost of content reordering operation which is much more favorable with Solution MM.

Dictionary size	Merkle-like struct.	Sol. MM	Sol. IM	Improvement
10^3	9.97	8.26	8.05	$17.1\% - 19.5\%$
$5 \cdot 10^4$	15.61	12.49	12.25	$20\% - 21.5\%$
$5 \cdot 10^5$	18.93	14.99	14.73	$20.8\% - 22.1\%$
10^6	19.93	15.73	15.46	$21\% - 22.5\%$

Table 1: Average proof size and verification cost results. The first improvement corresponds to Solution MM while the second one corresponds to Solution IM.

7. Authenticated Dictionary Based on Static Huffman Coding

The preceding solutions are more efficient than existing systems and can be improved even further. In fact, reducing the authentication proof size is equivalent to reducing in average the size of the corresponding leaf-root path in the tree. This is therefore a lossless data compression problem and it is well known that Huffman trees can produce codes of minimal average size. In this section, we introduce an authenticated dictionary based on a static Huffman tree and we describe its properties. The tree T is regularly rebuilt every Δt to take into account frequency distribution. In order to obtain a frequency distribution which is closer to reality, we propose to use a prediction function. We therefore optimize the size of authentication proofs.

7.1. Authenticated data Structure Construction

In this solution, the difference between Solution IM and Solution MM is that T_1 is a Huffman tree. For a sequence of relative strictly positive frequencies, the height of a

Huffman tree is bounded by (see [32]) $\min\left\{n-1, \left\lfloor \log_\phi\left(\frac{\phi+1}{f_{n-1}\phi+f_n}\right)\right\rfloor\right\}$ where ϕ is the golden ratio. Considering the asymptotic estimation of the n-th harmonic number, this means for Zipf's law that the height of the tree is in $\mathcal{O}\big(\log n\big)$, which is comparable to the asymptotic height of a Merkle tree. As previously seen, the lower tail of the page access distribution of a Web site decreases faster than that of Zipf's Law. If a real data set deviates too much from this distribution, the height of the tree T must be monitored to ensure that it does not exceed a certain value. It is always possible to define a frequency threshold s such that frequency data that are greater than s are covered by a Huffman tree and the rest of the data by a Merkle tree. If the dictionary contains n elements, just choose $s = P(n)$ where $P(n)$ is the access probability of the n-th least consulted element for a perfect Zipf's law . Therefore, the root of T will have as child nodes these two trees and its height will stay in $\mathcal{O}\big(\log n\big)$ regardless of the data set distribution. In practice, the number of leaves of T_2, denoted N_r, is much larger than T_1, denoted N_l.

The construction of T uses the same objects than for Solution MM (namely the sentinel $\pm\infty$, L^u and L^f). The calculation of a leaf h_i is also identical to the one of Solution MM.

The static construction of the Huffman tree T_1 follows an iterative procedure and, for practical reasons, it is deterministic. It involves the construction of a list L of nodes (initially empty) having five fields: the parent identifier, the current node identifier, the frequency of the current node identifier, the left child node and the right one. We first consider a temporary list Lt containing leaf values ranked in ascending order of frequency. The two elements of lowest frequency in Lt are added to L along with their parent. The symbol $\perp$ represents either a field that is not yet calculated or that has a null value. The list L is now:

$$\begin{aligned} L = \quad & \{(H_{1.2} := H(h_1, h_2), h_1, f_1, \perp, \perp), (H_{1.2}, h_2, f_2, \perp, \perp), \\ & (\perp, H_{1.2}, f_1 + f_2, h_1, h_2)\} \end{aligned}$$

The first two elements are siblings and the third one is their parent node, which is incomplete since the grandparent is not yet known. The frequency of the parent node is the sum of the two children. Then, Lt is updated by deleting the two leaves already being processed and by adding the parent node and we again consider the two smaller frequencies of Lt (which is maintained ordered). We add to L (or complete) these two elements along with their parent and repeat this process until Lt is a singleton. The list L is then complete. When the tree T is calculated, the source transmits the list of key-value pair elements together with the timestamped signature of the root node of T to the directory. Then, the directory is able to construct the data structures.

The cost of an efficient algorithm to construct a Huffman tree is in $\mathcal{O}\big(n \log n\big)$ if frequencies are not sorted, and in $\mathcal{O}\big(n\big)$ otherwise (in fact $\mathcal{O}\big(n \log n\big)$ comparisons and swaps to sort the frequencies and $\mathcal{O}\big(n\big)$ creations of nodes). Then, the cost to make a Huffman tree "authenticated" is in $\mathcal{O}\big(n\big)$ hash operations, where the hash operations are relatively much more expensive in comparison to the basic operations used to construct a Huffman tree. The construction of the authentication proof for a data of identifier u is similar to the case of a Merkle tree. The tree being static, insertion and deletion of an element are not possible. However, it is still possible to modify the

content of an element for a cost of $\mathcal{O}(\log n)$ hash operations.

7.2. *Proof size*

Let us suppose that the n elements of S follow a perfect Zipf distribution (Zipf exponent being equal to 1) and let H_n be the entropy of this distribution. We are interested in the gain on the proof size when authenticating these elements by a hashed Huffman tree T instead of a balanced Hash tree. We denote $G = 1 - L/log_2(n)$ this proof size gain, where L is the average path length in such a Huffman tree. From the Noiseless Coding Theorem [33], we have the well known bound $H_n \leq L < H_n + 1$. Thus, a bound for the gain is $1 - \frac{H_n+1}{\log_2 n} < G \leq 1 - \frac{H_n}{\log_2 n}$. For a finite number of elements, for example 10^3, $5 \cdot 10^4$, $5 \cdot 10^5$ and 10^6 elements and a perfect Zipf distribution we can expect a reduction of the proof size of at least 14.8%, 20.7%, 24.7%, 26.9% and 27.7% respectively. This kind of gain is to be compared to the 20% obtained using our first solution based on append/disjoin-only tree structures. When n is fixed, the gain is an increasing function of the Zipf exponent parameter.

Notice that the average proof length when using a Huffman tree for T_1 and a Merkle tree for T_2 is given in Section 9, the threshold being defined as the weighted median.

8. Authenticated Dictionary Based on Dynamic Huffman Coding

When a dictionary has a lot of elements but not many accesses during Δt, we may, in some cases, opt for an update instead of a full reconstruction of T. The changes in the data structures come from changes in the dictionary (adding or deleting elements) or modification of frequencies. More precisely, adding an element in the dictionary corresponds to inserting an element in T_2 (at the right side) which supports append/disjoin-only operations, for a cost of $\mathcal{O}(\log n)$ hash operations. Removing an element from the dictionary corresponds to delete an element from either T_1 or T_2. Changing the frequency of an element corresponds to moving an element inside T.

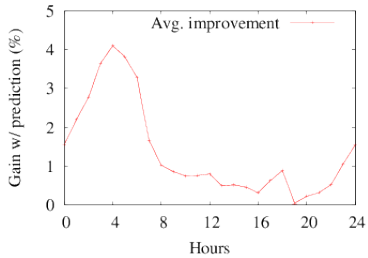
Notice that a splay tree [34] or a treap [35] could be used to efficiently provide insertion, deletion or modification. However, we prefer to make use of compression techniques to minimize authentication proof size and to perform search operations with an adjacent dynamic data structure like a red-black tree. In order to maintain an optimal tree (with respect to size of proofs), updates can be done with an adaptive Huffman coding algorithm. Recall that updates consist of changing the frequency of an element. For an increment or a decrement by unity, the FGK Algorithm from Faller, Gallager and Knuth [36] updates the Huffman tree in $\mathcal{O}(k)$ steps where k is the length of the path from the corresponding leaf to the root node. Here, a step consists in a swap of nodes, subtrees or both. For an increment or a decrement by any positive value, the algorithm from Cormack et al. [28] updates the tree in $\mathcal{O}(k)$ expected time, although this one has a particularly bad worst case. If we use this kind of algorithm to update authenticated Huffman trees, we can count the expected number of hash operations in the following way: In each step of the algorithm we need to update ancestor nodes that have been exchanged. Consequently, the expected number of hash operations is in $\mathcal{O}(k^2)$. Notice that in the case of a Zipf distribution, the length of a leaf-root path is in $\mathcal{O}(\log n)$. Thus, the number of hash operations is in $\mathcal{O}(\log^2 n)$.

9. Experiments

Evaluations on Wikipedia. Our first application was to authenticate HTTP responses. We made experimentations on Wikipedia France using a static dictionary of $3.6 \cdot 10^5$ elements. The statistics were collected on an hourly basis ($t_f = \Delta t = 1$ hour), the finest granularity available on Wikipedia. We evaluated solutions MM, IM, and two solution based on Huffman trees denoted H and HM. The solution H used an Huffman tree whereas HM used a tree T composed of a Huffman tree T_1 and a Merkle tree T_2, the threshold being the weighted median. Compared to a basic Merkle tree, we did not obtain a significant average gain on the length of the proof size. When we simulated a perfect prediction, we had an average gain of 5.6% with Solution H. These small gains are due to the high entropy of the Wikipedia samples, which is far from the one of a perfect Zipf's law.

We also determined, over a 24-hour period, the average proof size improvement depending on whether or not a perfect prediction function is used (see Figure 4a). The peak of the curve between 0 am and 8 am shows that users behavior is more chaotic at that time.

During our tests [2], multiple trees were constructed using the aforementioned methods. The largest Huffman tree had around 1.2M leaves. The tree construction took an average of 16 seconds. Notice that the construction time dropped to less than 2 seconds when T was composed of left Huffman subtree and right Merkle subtree. These construction times are compatible with dynamic sources even though the tree itself is static. However, the use of an update method like FGK is not adequate given the significant variation in the number of queries per hour.



(a) Improvement provided by prediction (Wikipedia experiment)

$t_f = 5min$	H	34.8%	IM	25.5%
	HM	26.9%	MM	25.4%
$t_f = 30min$	H	39.2%	IM	27.5%
	HM	29.8%	MM	27.5%
Perfect	H	54.7%	IM	38.5%
Prediction	HM	39.1%	MM	38.3%

(b) gain on the length of the proof size, compared to a Merkle tree, for the financial stock values.

Figure 4

Evaluations on Stock Exchange. The second application was to authenticate, for a given stock, the trade confirmations of the transactions that have been carried out during

[2]The computer used to run our tests use an Intel Core i7-2600K CPU running at 3,4GHz and providing eight logical cores. It has 8GB of DDR3 RAM and operates with a GNU/Linux OS based on the Linux kernel version 3.16. The software used to perform our tests and comparisons was built with GCC 4.9.1 (Debian version 4.9.1-19).

the previous minute. In particular, we authenticated the price, the volume and the time of completion of the orders. We considered the number of transactions of the 163 financial values of the Paris stock exchange. In order to obtain the average day gain on the length of the proof size compared with a basic Merkle tree, we considered the IM, MM, H and HM solutions (see Figure 4b): T is either a 163 leaves Huffman tree (Solution H) or composed of T_1 and a Merkle tree T_2, the threshold being the weighted median (an other choice of threshold could have been considered, giving different results). We used a window with $t_f = 5min$ then $t_f = 30min$, and finally we used a perfect prediction. For all cases, $\Delta t = 1min$.

Interpretation of Results. When there is no prediction, the tree is constructed using the distribution calculated during the previous t_f interval and the average proof length is calculated using the weighting given by the current distribution (established during the current Δt). Thus, when Solution H is considered, the expected gain is increased when these two distributions are similar. This explains the large gain obtained in the second experiment when using a perfect prediction. Notice than it may not be the case when we consider the other methods (MM, IM, HM) since the cut at the weighted median is not an optimal method. We also have to mention that the results greatly depend on the curve distribution used to construct the tree. For the second experiment, the distribution of the number of transactions may be treated as a Zipf distribution with an exponent parameter greater than 1 (the parameter has been estimated using the software R at 1.4 compared to 0.8 for Wiki experiment). Therefore, we obtain better results than for a perfect (exponent equal to 1) Zipf law. Solutions IM and MM cannot be significantly differentiated because of the low number of leaves in T_1. Nevertheless, this use case highlights the benefits of our models.

Notice that the use of a prediction model makes more sense for stock exchange experiment than for Wiki experiment since stock market behavior is more predictable.

10. Caching strategy and the proximity of pages

Between two updates of the dictionary, if a user receives several authentication proofs, they are likely to share a common set of hash values. In particular, these proofs contain with a greater likelihood values of nodes that are close to the root node. Thus, it would be interesting to cache the content of the proofs in order to be partially reused in responses to future queries. This simple optimization (a caching method) can be applied to any tree structure but it seems particularly well suited to Huffman trees since nodes located near the root are of higher frequency. We made an experimentation on a Huffman tree of approximately $1.2 \cdot 10^6$ leaves. For three consecutive random requests following Zipf's law, we obtained an average gain of 33.3% on the overall number of hash values of the proofs.

Web browsing may be likened to a graph traversal. In addition to access frequencies, an other parameter called *access conditional frequencies* could be taken into account. In this case, the tree would have the additional following property: leaves that are closer to the leaf x are those for which the probability is higher to be accessed from x. However, we believe that the bandwidth gain would remain marginal compared to the use of a Huffman tree with the simple optimization described above.

11. Conclusion

We have proposed models for authenticated dictionaries which take 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. Our models can be used when the entropy of the distribution is smaller than the one of a uniform law. In practice, this corresponds to a Zipf parameter greater than 0.8. When comparing the models using a window of fixed length, a sequence of tree updates is of interest if the overall cost is reduced by a factor $\log n$ compared to the cost of a tree reconstruction. We have to distinguish between the number of frequency changes between two consecutive measurement periods and the total sum of the differences. We can state the following:

- If the number of changes is in $\mathcal{O}\big(n/\log^3 n\big)$, then Solution IM or a solution based on the method of Cormack and Horspool [28] may be considered.
- If the sum of differences is in $\mathcal{O}\big(n/\log^3 n\big)$, then an adaptive (dynamic) Huffman based solution using the algorithm FGK [36] may be relevant.
- If the number of changes is in $\mathcal{O}\big(n/\log^2 n\big)$, then Solution MM may be considered if optimality of the proof length is not required.
- For all other cases, the use of a static Huffman tree provides the most adequate solution.

Significant average gains can be expected when the distribution curve behaves like a Zipf law with an exponent greater than 1. This is the case of the second experiment which provides a gain of more than 39% with a window of 30 minutes. We also showed by experimentation that caching techniques can further reduce bandwidth when using Huffman trees.

References

[1] M. Naor, K. Nissim, Certificate revocation and certificate update., IEEE Journal on Selected Areas in Communications 18 (4) (2000) 561–570.

[2] I. Gassko, P. Gemmell, P. D. MacKenzie, Efficient and fresh certification, in: Int. Workshop on Practice and Theory in Public Key Cryptography, PKC'00, Springer-Verlag, 2000, pp. 342–353.

[3] O. Ohrimenko, H. Reynolds, R. Tamassia, Authenticating email search results, in: Int. Workshop on Security and Trust Management, STM'12, Springer-Verlag, 2012, pp. 225–240.

[4] P. T. Devanbu, M. Gertz, C. U. Martel, S. G. Stubblebine, Authentic third-party data publication, in: IFIP 11.3 Work. Conf. in Database Security, Kluwer, B.V., 2001, pp. 101–112.

[5] R. J. Bayardo, J. Sorensen, Merkle tree authentication of http responses, in: Int. Conf. on World Wide Web, WWW '05, ACM, 2005, pp. 1182–1183.

[6] R. Tamassia, N. Triandopoulos, On the cost of authenticated data structures, in: Europ. Symp. on Algorithms, ESA'03, Springer, 2003, pp. 2–5.

[7] M. T. Goodrich, R. Tamassia, A. Schwerin, Implementation of an authenticated dictionary with skip lists and commutative hashing, in: DARPA Information Survivability Conf. amp, Vol. 2, 2001, pp. 68–82.

[8] L. A. Adamic, B. A. Huberman, Zipf's law and the internet, Glottometrics 3 (2002) 143–150.

[9] A. Mahanti, N. Carlsson, A. Mahanti, M. Arlitt, C. Williamson, A tale of the tails: Power-laws in internet measurements, IEEE Network 27 (1) (2013) 59–64.

[10] M. T. Goodrich, R. Tamassia, Efficient authenticated dictionaries with skip lists and commutative hashing, Tech. rep., Johns Hopkins Information Security Institute (2001).

[11] D. P. Mehta, S. Sahni (Eds.), Handbook of data structures and applications, Chapman & Hall/CRC, 2005.

[12] W. Pugh, Skip lists: A probabilistic alternative to balanced trees, Communications of the ACM 33 (6) (1990) 668–676.

[13] S. A. Crosby, D. S. Wallach, Authenticated dictionaries: Real-world costs and trade-offs, ACM Trans. on Information and System Security 14 (2) (2011) 17:1–17:30.

[14] S. A. Crosby, D. S. Wallach, Super-efficient aggregating history-independent persistent authenticated dictionaries, in: Europ. Conf. on Research in Computer Security, ESORICS'09, Springer-Verlag, 2009, pp. 671–688.

[15] C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong, S. G. Stubblebine, A general model for authenticated data structures, Algorithmica 39 (1) (2004) 21–41.

[16] M. T. Goodrich, R. Tamassia, N. Triandopoulos, Efficient authenticated data structures for graph connectivity and geometric search problems, Algorithmica 60 (3) (2011) 505–552.

[17] R. C. Merkle, Protocols for public key cryptosystems, in: Symposium on Security and Privacy, S&P'80, IEEE, 1980, pp. 122–122.

[18] R. Merkle, A certified digital signature, in: Annual International Cryptology Conference, CRYPTO '89, Springer-Verlag, 1990, pp. 218–238.

[19] B. Preneel, B. V. Rompay, J. J.Quisquater, H. Massiasand, J. S. Avila, Digital timestamping and the evaluation of security primitives, Tech. rep., TIMESEC project (1999).

[20] K. Blibech, A. Gabillon, Chronos: An authenticated dictionary based on skip lists for timestamping systems, in: Workshop on Secure Web Services, SWS '05, ACM, 2005, pp. 84–90.

[21] J. Benaloh, M. de Mare, One-way accumulators: A decentralized alternative to digital signatures, in: Annual International Cryptology Conference, EUROCRYPT '93, Springer-Verlag, 1994, pp. 274–285.

[22] L. Nguyen, Accumulators from bilinear pairings and applications, in: Int. Conf. on Topics in Cryptology, CT-RSA'05, Springer-Verlag, 2005, pp. 275–292.

[23] K. Nyberg, Fast accumulated hashing, in: Int. Workshop on Fast Software Encryption, FSE'96, Springer, 1996, pp. 83–87.

[24] B. H. Bloom, Space/time trade-offs in hash coding with allowable errors, Communications of the ACM 13 (7) (1970) 422–426.

[25] C. Papamanthou, R. Tamassia, N. Triandopoulos, Authenticated hash tables, in: Conf. on Computer and Communications Security, CCS '08, ACM, 2008, pp. 437–448.

[26] C. Papamanthou, R. Tamassia, N. Triandopoulos, Authenticated hash tables based on cryptographic accumulators, Algorithmica (2015) 1–49.

[27] R. Gallager, Variations on a theme by huffman, IEEE Trans. on Information Theory 24 (6) (1978) 668–674.

[28] G. V. Cormack, R. Horspool, Algorithms for adaptive huffman codes, Information Processing Letters 18 (3) (1984) 159–165.

[29] G. E. P. Box, G. Jenkins, Time Series Analysis, Forecasting and Control, Holden-Day, Incorporated, 1990.

[30] R. Brown, Exponential Smoothing for Predicting Demand, Little, 1956.

[31] K. Atighehchi, A. Bonnecaze, T. Muntean, Authenticated dictionary based on frequency, in: ICT Systems Security and Privacy Protection, IFIP SEC'14, Springer, 2014, pp. 293–306.

[32] M. Buro, On the maximum length of huffman codes, Information Processing Letters 45 (1993) 219–223.

[33] C. Shannon, A mathematical theory of communication, Bell System Technical Journal 27 (1948) 379–423.

[34] D. D. Sleator, R. E. Tarjan, Self-adjusting binary search trees, Journal of the ACM 32 (3) (1985) 652–686.

[35] R. Seidel, C. Aragon, Randomized search trees, Algorithmica 16 (4-5) (1996) 464–497.

[36] D. E. Knuth, Dynamic huffman coding, Journal of Algorithms 6 (2) (1985) 163–180.