CHRONOS* : An Authenticated Dictionary Based on Skip Lists for Timestamping Systems†

Kaouthar Blibech
LIUPPA/CSYSEC
Université de Pau
IUT de Mont de Marsan
France
k.blibech@etud.univ-pau.fr

Alban Gabillon
LIUPPA/CSYSEC
Université de Pau
IUT de Mont de Marsan
France
alban.gabillon@univ-pau.fr

ABSTRACT

Skip Lists were first used as data structures for their simple implementation and optimal update and search time. Goodrich [7][8] was the first to propose an authenticated dictionary based on skip lists. More recently, Maniatis and Baker [10][11][12] have proposed an authenticated append only dictionary based on perfect skip lists. In this paper, we propose an authenticated append only dictionary which offers better performances than the previous proposals. Moreover this dictionary allows comparing the relative order of elements. Such a dictionary could be used for timestamping purposes.

Categories and Subject Descriptors

H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing—*Dictionaries*; E.1 [**Data Structures**]: Lists, stacks, and queues; E.2 [**Data Storage Representations**]: Linked representations; J.m [**Computer Applications**]: Miscellaneous

General Terms

Algorithms, Security

Keywords

Authenticated dictionaries, skip lists, timestamping systems, order of insertion

1. INTRODUCTION

Dictionaries are data structures that support both update queries, such as insertion, removal, and membership queries. A membership query is of the form "does element e belong to S?". The answer to the query is a Boolean (Y/N).

Authenticated dictionaries are data structures that support both update queries, and *tamper-evident* membership queries. A tamper-evident membership query is of the form "does element e belong to S?". The answer to such a query is a Boolean response and a tamper-evident proof for this response. The proof consists in showing that element e participated to the construction of the *root value* of data set S. The root value is a one way digest of data set S.

Merkle's tree [13][14] was the first authenticated dictionary. It is a binary tree where leaf nodes are labelled by the hashed values of the elements of S and internal nodes are labelled by the hashed values of concatenated labels of their children. The root value is then the label of the root node, and the proof that element e belongs to S consists of the labels of all sibling nodes on the path from the leaf node representing e to the root node. If we assume that the used hash function is collision resistant then the root value constitutes a tamper-evident digest for S and proofs are tamper-evident.

Several implementations of Merkle's hash trees were proposed as a solution to the certificate revocation problem, for example by Micali [15], Kocher [9], Naor and Nissim [16], and Buldas and al [3]. Later, Merkle's hash trees were also used for timestamping purposes [2] and third party publication on the Internet [6].

Different tree like data structures were also proposed as authenticated dictionaries, like binary linking schemes [4] and threaded trees [5].

Recently, Goodrich et al. [7] [8] have proposed an authenticated dictionary based on skip lists and commutative hashing. Several variations of such a dictionary have been proposed, like the persistent authenticated skip list [1] and the authenticated append only skip list [10][11][12].

In this paper, our aim is to build an authenticated dictionary based on skip lists, specifically designed for timestamping. The motivation for this work comes from our involvment in the CHRONOS[1] project which aims at building a secure timestamping system. Our authenticated dictionary returns tamper evident membership proofs and is also able to return a tamper evident indication of the relative order of any two given elements in a given skip list. Our dictionary also provides minimal space use (storage capacity), construction time, verification time and proof size.

*CHRONOgraphie Sécurisée (Secure TimeStamping).

†This work was supported by the Conseil Général des Landes and the French ministry for research under *ACI Sécurité Informatique 2003-2006. Projet CHRONOS*

SWS'05, November 11, 2005, Fairfax, Virginia, USA.

[1]Project home page: http://extranet.iaai.fr/enseignement-recherche/recherche/chronos

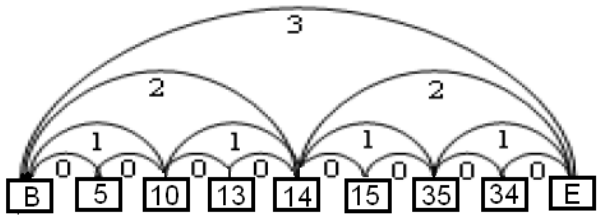

Figure 1: a skip list is a linked list with several pointers per element. Boxes the elements of the data set, edges are the pointers and numbers within [0..3] are the levels.

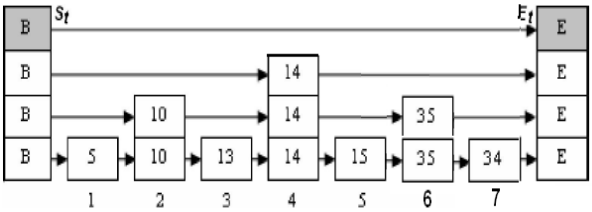

Figure 2: A skip list is a set of linked lists. Numbers within [1..7] are the indexes of the elements.

We are not concerned with the update time since we only need append operations (insertion at the end of the list) and deletions are prohibited.

Our dictionary could be used to build a secure timestamping web service.

In section 2 we review related works. In section 3 we present our dictionary. In section 4 we evaluate the performances of our dictionary. In section 5 we show the advantage of using our dictionary for timestamping purposes and in section 6 we conclude this paper.

2. RELATED WORKS

Merkle's tree was the first authenticated dictionary. It uses linear space and has $O(logn)$ proof size, query time and verification time. But the use of Merkle's tree as an authenticated dictionary has some drawbacks. First, the number of elements in the data set has to be a power of 2. If not, we need to add a number of insignificant elements to make it a power of 2. Moreover we need to add to every hash value contained in a membership proof some space indications to specify the order of the hashing operations when verifying the proofs. Finally, it is not possible to determine the order in which elements were inserted.

Skip lists are easy to implement, and present the same optimal performances in terms of query time, verification time, and proof size, while having fewer drawbacks. W. Pugh introduced skip lists as an alternative data structure to search trees [17]. The main idea is to add pointers to a simple linked list in order to skip a large part of the list when searching for a particular element. While each element in a simple linked list points only to its immediate successor, elements in a skip list can point to several successors. Thus skip lists are linked lists with more than one pointer per element (Figure 1). They can also be seen as a set of linked lists, one list per level (Figure 2).

All the elements are stored at the first level 0. A selection of elements of level k is included in the list at level $k+1$. In *probabilistic* skip lists, if element e belongs to level k then it belongs to level $k+1$ with probability p. In *deterministic* skip lists, if element e belongs to level k and respects a given constraint, then it belongs to level $k+1$. For example, in *perfect* skip lists (Figure 2), which are the most known deterministic skip lists, element e belongs to level i if its index is a multiple of 2^i. Consequently, element at index 5 belongs only to the first level; while element at index 4 belongs to the three first levels. In Figure 2, B and E nodes are stored at all levels and called *sentinel* elements. The highest B node is called *starting node* S_t. The highest E node is called *ending node* E_t.

2.1 Goodrich dictionary

Goodrich et al. were the first to propose an authenticated dictionary based on skip lists [7][8]. Elements of the dictionary are sorted and hashed over the skip list and the value associated to the starting node S_t is the root value: Goodrich et al. suggest using a commutative hashing function which supports a commutative collision resistance property defined as follows: for a given couple (a, b) it is difficult to find (c, d) such as $f(a, b) = f(c, d)$ if $(a, b) \neq (c, d)$ and $(b, a) \neq (c, d)$, where f is the commutative hash function. The commutative property is used while verifying proofs in order to avoid additional indications of hashing order as it is the case with Merkle's tree scheme.

However, the fact that commutative operations do not allow preserving and verifying the relative order of elements represents the greatest limit of this dictionary.

2.2 Maniatis Dictionary

Recently, Maniatis and Baker have proposed an append-only authenticated dictionary based on perfect skip lists [10][11][12]. Elements are added at the end of the list, and belong to one or several levels according to their index.

Such a scheme corresponds to our needs. Indeed, perfect skip lists use the index of each element to determine its height. Thus, the hashing scheme depends on the order in which elements were inserted. This characteristic can be useful to provide relative order property. However, when compared to Goodrich scheme, this dictionary has worse construction time and proof size.

3. CHRONOS DICTIONARY

Our dictionary is inspired from both Goodrich dictionary and Maniatis dictionary.

We use a perfect skip list structure on which we implement a hashing scheme inspired from the one proposed by Goodrich. In fact, we modify the hashing scheme in order to use simple (non commutative) hash functions and to limit the number of hash operations. The verification process is defined according to our hashing scheme and allows us to prove the relative order of elements in the skip list.

3.1 Hashing scheme

Each element inserted in the skip list has two characteristics: its *index* and its *value*. Since we are dealing with perfect skip lists, we associate one or several nodes to each element according to the element index. Nodes associated

to the same element have the same value and index. For example, consider node a and node p in Figure 3. They have the same index (20) and value (h20).

In addition to the index and the value inherited from the element to which it is associated, each node has two other properties: its *level* and its *label*. Hence, node a and node p have different levels (respectively 0 and 2) and different labels computed as explained below.

For each node n, we denote *value(n)* the value of node n and *label(n)* the label of node n. Moreover, we define the *left* and *down* functions as follows: *left(n)* returns the left node of n, and *down(n)* returns the bottom node of n.

We also define the *plateau node* of an element as the highest node associated to that element.

Our hashing scheme is defined in Algorithm 1. h is a simple hash function and || is a concatenation operation.

ALGORITHM 1. *Hashing scheme*

```
1: If left(n) = null, then label (n) := 0.
2: Else if down(n) = null, (n is on the first level):
3:       If left(n) is not a plateau node
4:           then label(n) := value(n).
5:       If left(n) is a plateau node
6:           then label(n) := h (value(n) || label(left(n))).
7: Else if down(n) != null, (n is not on the first level):
8:       If left(n) is not a plateau node
9:           then label (n) := label(down(n)).
10:      If left(n) is a plateau node
11:          then label(n) := h(label(down(n))||label(left(n))).
```

Algorithm 1 starts with the B element (see Figure 3) and assigns to its nodes an initialisation value (0) (line 1 in Algorithm 1). Then, it iteratively computes for each element the label of each of its nodes starting from the node at level 0 to the plateau node. The label of each node n is computed from its level and its immediate left node as follows:

- If the immediate left node of node n is a plateau node:
  - if node n is at the first level, then the label of node n is equal to the hash of its value concatenated with the label of the left node (case 1 : lines 2, 5 and 6 in Algorithm 1),
  - if node n is at higher levels, then the computation remains the same but it uses the label of the node under node n in place of its value (case 2 : lines 7, 10 and 11 in Algorithm 1).
- If the immediate left node of node n is not a plateau node,
  - if node n is at the first level, then the label of node n is equal to its value (case 3: lines 2, 3 and 4 in Algorithm 1),
  - if node n is at higher levels, then the label of node n is equal to the label of the node under it (case 4 : lines 7, 8 and 9 in Algorithm 1).

For example, consider node r in Figure 3 (index 24, value h24 and level 0) and node q (index 23, value 23 and level 0). The label of node r is equal to the hash of the value of node r (h24) concatenated with the label of node q (case 1).

Now, consider node s (index 24, value h24 and level 1) and node d (index 22, value h22 and level 1). The label of node s is equal to the hash of the label of node r concatenated with the label of node d (case 2).

Consider also node b (index 21, value h21 and level 0) and node p (index 20, value h20 and level 0). Node p is not a plateau node, so the label of node b is equal to its value h21 (case 3).

Finally, consider node d (index 22, value h22 and level 1) and node c (index 22, value h22 and level 0). The label of node d is equal to the label of node c (case 4).

3.2 Membership proof

The membership proof for element e consists of the information needed to compute the label of the ending node E_t from the value of element e. To do so, several labels have to be recomputed. We call a *traversal chain* for an element e the smallest sequence of labels that have to be computed in order to determine the label of the ending node E_t from the value of element e. An example of such a chain is given by the labels of the thick nodes in Figure 3. They represents the traversal chain associated to element of index 21 and value h21.

More precisely, we construct a traversal chain as follows: we start with the lowest node of element e (at level 0), and use two actions: *rise* and *skip*. The label of every node traversed using those actions is added to the traversal chain.

A rise action is repeated until reaching a plateau node. A skip action is performed after reaching a plateau node.

The rise action reaches the node on top of the current node.

The skip action reaches the node which is immediately at the right of the current node. Both nodes have the same level l. Index of the reached node is $i + 2^l$ (with i being the index of the current node).

The membership proof for an element e contains all the information necessary to recompute its traversal chain. It includes three components:

- the index of element e in the skip list,
- a *head proof*,
- a *tail proof*.

The index i and the head proof for an element can be returned immediately after the element has been inserted in the skip list. There is no need to wait for the termination of Algorithm 1. Computation of the head proof is done by Algorithm 2.

The head proof for element e contains preliminary information needed to start the computation of its traversal chain. Consider the nodes having an index strictly lower than i. Among these nodes, for each level l, consider the node which has the highest index. If it is a plateau node then it belongs to the head proof.

For example, for element of index 21 in Figure 3, the head proof consists of the label of node a (that will be used later to compute the label of node t) and the label of node o (that will be used later to compute the label of the ending node).

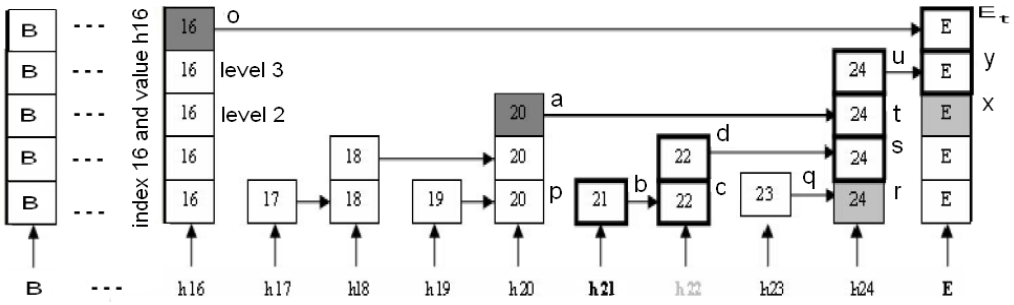

Figure 3: Hash flow computation. The arrows denote the flow of information. Labels of dark grey nodes constitute the head proof for element of index 21. Labels of clear grey nodes and value h22 constitute the tail proof for element of index 21. Labels of thick nodes are computed during the verification process. The label of the ending node is the root node.

In order to compute the head proof for element e, we need to define function *height(S)* that returns the number of levels in the skip list and function *last(l)* that returns the last node inserted at level l before the insertion of e.

ALGORITHM 2. *Head proof computation*

1: HP := {} (the heaf proof)
2: For i from 0 to height(S)
3: If last(i) is a plateau node,
4: then append label(last(i)) to HP.

The tail proof for an element e is returned immediately after Algorithm 1 terminates. Computation of the tail proof is done by Algorithm 3.

The tail proof is a list including the labels or values of some nodes having higher indexes than element e. These nodes are needed for computing the traversal chain as follows:

Consider the traversal chain of element e of index i. Let j be an index strictly greater than i. Let C_j be the set of nodes of index j whose labels belong to the traversal chain. Let n_j be the node of C_j with the smallest level (if C_j is not empty). Let l be the level of that node.

- If $l = 0$, then the value of node n_j shall belong to the tail proof (lines 4, 5 in Algorithm 3) .
- If $l \neq 0$, then the label of the node immediately under node n_j (index j and level $l-1$)shall belong to the tail proof (line 6 in Algorithm 3).

For example, for element of index 21 in Figure 3, the tail proof consists of the value h22, the label of node r and the label of node x.

ALGORITHM 3. *Tail proof computation*

1: TP := {} (the tail proof).
2: While right(n) != null, (right(n) is not null) :
3: n := right(n).
4: if down(n) = null,
5: then append value(n) to TP.
6: Else append label(down(n)) to TP.
7: While top(n) != null,
8: n = top(n).

In order to compute the tail proof, we need to define function *right(n)* that returns the right node of n and function *top(n)* that returns the top node of n. For example, if n is the level 1 node of element 22, $right(n)$ is the level 1 node of element 24, and $top(n)$ is null.

As we are going to see in the next section, the verification process strongly depends on the index included in the membership proof. In particular, we could show that if the maintainer of the data structure returns a wrong index, then the verification process would fail.

3.3 Verification process

The *verification process* for element e consists in recomputing the traversal chain using only the membership proof and the value v of element e. Only the membership proof, the value of element e and the label of the ending node are known by the verification process.

The verification process succeeds if the last value of the computed traversal chain is equal to the label of the ending node returned by the hashing scheme. If not, the verification fails.

Assuming that the one-way hash function used in the hashing scheme is collision resistant, the membership proof constitutes a tamper evident proof of inclusion of element e in the skip list (identified by the label of its ending node E_t).

In order to compute the label of the ending node from the membership proof, Algorithm 4 requires the following as parameters:

- the membership proof for element e which consists of the head proof for element e, the tail proof for element e and the index i_e of element e,
- the value v of element e,
- the label of the ending node.

Objective of Algorithm 4 is to reconstruct the traversal chain from e to the ending node E_t. Algorithm 4 starts by considering node of index $i = i_e$ and level $l = 0$. This node is the current node and is denoted by n.

- If the left[2] node m of n is a plateau[3] node,
  - if index $i - 2^l$ of node m is strictly lower than i_e, then the label of node n is equal to the hash of the label of the node under n (or the value of node n if $l = 0$) concatenated with the label of node m which can be extracted from the head proof (case 1 : lines 7, 8 and 9 in Algorithm 4),
  - if index $i - 2^l$ is greater (or equal) than i_e, then the label of node n is equal to the hash of the label of the node under n (or the value of node n if $l = 0$) which can be extracted from the tail proof, concatenated to the label of node m (case 2 : lines 7, 10 and 11 in Algorithm 4).
- If the left node m of n is not a plateau node, then the label of node n is equal to the label of the node under it (or is equal to the value of node n if $l = 0$) (case 3).

After computing the label of the current node, Algorithm 4 aims at processing the next node in the traversal chain.

- If the tail proof is not empty,
  - if node n is not a plateau node, then the next node has index i and level $l + 1$ (lines 5 and 6 in Algorithm 4),
  - if node n is a plateau node, then the next node has index $i + 2^l$ and level l (lines 5, 12 and 13 in Algorithm 4).
- If the tail proof is empty, then Algorithm 4 concludes that the current node n is associated to the ending element.
  - If the head proof is not empty, then the next node has index i and level $l + 1$ (lines 14 and 15 in Algorithm 4). Its label is processed by Algorithm 4 as in *case 1*.
  - If the head proof is empty, then Algorithm 4 concludes that node n is the ending node.

Consider example of Figure 3. Value of the inserted element is h21. Index of the inserted element is 21 (i_e=21). Initial node of level 0 and index 21 is node b. Index of left node of node b is $21 - 2^0$ (=20). Left node of node b is not a plateau node (case 3). Therefore, label of node b is equal to its value (h21).

Node b is a plateau node. Therefore, the next node processed by Algorithm 4 is node c of index $21 + 2^0$ (=22) and of level 0. Index of left node of node c is $22 - 2^0$ (=21). Left node of node c is a plateau node. Therefore, label of node c is equal to the hash of the first value extracted from the tail proof (the value of node c) concatenated to the label of node b (h(h22, h21)) (case 2).

Node c is not a plateau node. Therefore, the next node processed by Algorithm 4 is node d of the same index 22 and of level 0+1 (=1). Left node of node d is not a plateau node (case 3). Therefore, label of node d is equal to label of node c (h(h22, h21)).

Node d is a plateau node. Therefore, the next node processed by Algorithm 4 is node s of index $22 + 2^1$ (=24) and of level 1. Index of left node of node s is $24 - 2^1$ (=22). Left node of node s is a plateau node. Therefore, label of node s is equal to the hash of the second value extracted from the tail proof (label of node r) concatenated to the label of node d (case 2).

Node s is not a plateau node. Therefore, the next node processed by Algorithm 4 is node t of index 24 and of level 2. Index of left node of node t is $24 - 2^2$ (=20). Left node of node t is a plateau node. Therefore, label of node t is equal to the hash of the label of node s concatenated to the first value extracted from the head proof (label of node a) (case 1).

Node t is not a plateau node. Therefore, the next node processed by Algorithm 4 is node u of index 24 and of level 3. Left node of node u is not a plateau node. Therefore, label of node u is equal to label of node t (case 3).

Node u is a plateau node. Therefore, the next node processed by Algorithm 4 is the node of index $24 + 2^3$ (=32) and level 3. Note that in Figure 3, there is no node of index 32. In fact, everything works as if 32 was the index of the ending element. Consequently, the next node is node y. Left node of node y is node u which is a plateau node. Index of left node is $32 - 2^3$ (=24). Therefore, the label of node y is equal to the hash of the third (and last) value extracted from the tail proof (label of node x) concatenated to the label of node u (case 2).

Finally, since node y is not a plateau node, the next node processed by Algorithm 4 is the node of index 32 and level 4 i.e. the node on top of node y (it is in fact the ending node, but Algorithm 4 does not know it). Index of left node of that node is $32 - 2^4$ (=16). Left node is a plateau node. Therefore, label of the current node is equal to the hash of the label of node y concatenated to the last value extracted from the head proof (label of node o) (case 1).

Since there is no more values to extract, neither from the tail proof nor from the head proof, Algorithm 4 concludes that the last computed label was the label of the ending node. If that computed label is equal to the label of the ending node given as parameter, then the verification process succeeds. If not, the verification process fails.

ALGORITHM 4. *Verification process*

```
1 : e=(value, index) (the element to be verified).
2 : value := e.value.
3 : index := e.index.
4 : level := 0.
5 : While TP ≠ {}
6 :       For i from level to height(index) :
7 :           If hasPlateauOnLeft(index, i)
8 :               If leftIndex(index, i) ≺ e.index
9 :                   then value=h(value||getPrec()).
10:               If leftIndex(index, i) ≥ e.index
11:                   then value=h(getNext()||value).
12:       level := i.
13:       index := getNextIndex(index).
14 : While HP ≠ {}
15 :       value=h(value||getPrec()).
16: if value = label(Et) then return TRUE
17: Else return FALSE
```

[2]Consider node n of index i and level l. Since we are dealing with perfect skip lists, index of the left node of n is $i - 2^l$.

[3]Consider node n of index i and level l. Consider k such that $i - 2^l = k * 2^l$. Since we are dealing with perfect skip lists, if $HCF(2, k) = 1$ then the left node of n is a plateau node.

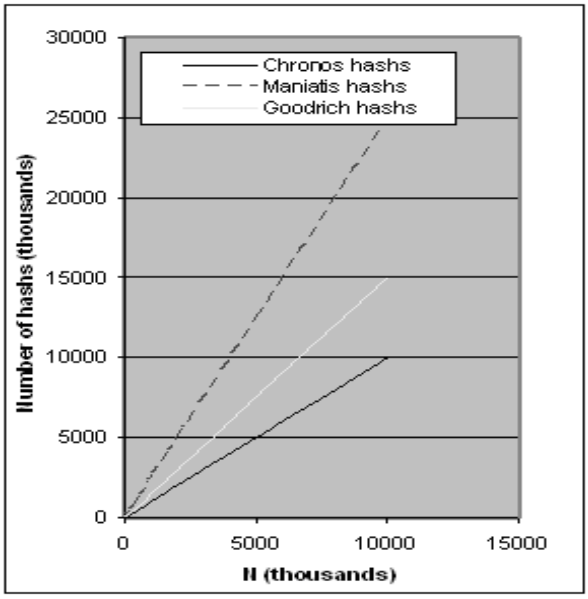

Figure 4: Hashing cost.

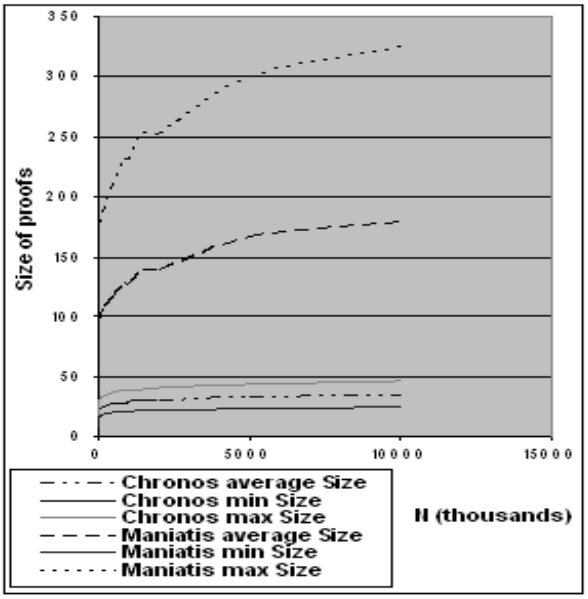

Figure 5: Size of proofs.

Regarding Algorithm 4, we need to define the following functions:

- *height(index)*[4] that returns the height of an element at position *index*,
- *leftIndex(index, level)* that returns the index of the left node[5],
- *hasPlateauOnLeft(index, level)* that indicates if the node of index *index* and level *level* has a plateau node on its left[6].
- *getNext()* that extracts the next label from the tail proof,
- *getPrec()* that extracts the next label from the head proof,
- *getNextIndex(index)*[7] that returns the index of the node whose label was returned by getNext().

In future works, we plan to publish the formal proof showing that if the maintainer would lie on the index of the considered element, then the verification process would fail.

4. PERFORMANCES

In this section, we compare the performances of our dictionary in terms of construction time and proof size.

[4]The height h of any element can be computed from its index $i : i = 2^h * k$ where $HCF(2, k) = 1$.
[5]The left node of a node of index i and level l has an index $j = i - 2^l$
[6]Consider node n of index i and level l. Consider k such that $i - 2^l = k * 2^l$. If $HCF(2, k) = 1$, then the left node of n is a plateau node.
[7]The next index j can be computed from the current index i: $j = i+2^h$, where h is the height of the element at position i.

We choose the number of hash operations as a unit for Figure 4, and the number of digests as a unit for Figure 5.

In Figure 4, we can see that our dictionary requires less hashing operations than both Maniatis and Baker dictionary and Goodrich dictionary.

Experiments showed that the average (respectively minimal and maximal) size of membership proofs in our scheme is the same as the average (respectively minimal and maximal) size of the proofs in the Goodrich's scheme. However, compared to Maniatis and Baker dictionary, our proofs are smaller (see Figure 5).

5. TIMESTAMPING SYSTEMS

Timestamping systems can use authenticated dictionaries to timestamp documents received within a time interval which is called *round*. The root value of the constructed dictionary is then published in a widely distributed media like a newspaper for example, while the membership proof for each document, which is called the *timestamp*, is returned to the user who submitted the document [4] [5].

In traditional timestamping systems, trusting the authority is not necessary when comparing the timestamps of documents belonging to different rounds, as the precedence relationship between rounds can easily be determined from the published values. On the other hand, the precedence of two timestamps belonging to the same round cannot be determined without the help of the timestamping authority, which, in that case has to be trusted.

With our dictionary, the authority will no longer be able to lie on the order of the timestamps even for documents belonging to the same round. In our system, the precedence relationship between every pair of timestamps can be proved. Therefore, there is no need to trust the timestamping authority. All the information needed to compare elements are included in the timestamps.

6. CONCLUSION

We have presented an efficient and practical scheme for an authenticated dictionary based on skip lists that preserves the order of insertion of elements into the structure.

While our proofs are smaller and require less hashing operations than the proofs in Goodrich scheme and Maniatis and Baker scheme, we can prove that an element belongs to the dictionary and vouch for its index.

In future works, we are planning to publish the proof showing that the maintainer of the dictionary cannot lie on the index of an element. We are also planning to use our scheme for building a protoype of our secure timestamping system, CHRONOS.

7. REFERENCES

[1] A. Anagnostopoulos, M. T. Goodrich, and R. Tamassia. Persistent authenticated dictionaries and their applications. *Lecture Notes in Computer Science*, 2001.

[2] D. Bayer, S. Haber, and W. Stornetta. Improving the efficiency and reliability of digital time-stamping. *In Sequences'91: Methods in Communication, Security, and Computer Science*, pages 329–334, 1992.

[3] A. Buldas, P. Laud, and H. Lipmaa. Accountable certificate management with undeniable attestations. *7th ACM Conference on Computer and Communications Security*, pages 9–18, 2000.

[4] A. Buldas, P. Laud, H. Lipmaa, and J. Villemson. Time-stamping with Binary Linking Schemes. In H. Krawczyk, editor, *Advances on Cryptology — CRYPTO '98*, volume 1462 of *Lecture Notes in Computer Science*, pages 486–501, Santa Barbara, USA, Aug. 1998. Springer-Verlag.

[5] A. Buldas, H. Lipmaa, and B. Schoenmakers. Optimally efficient accountable time-stamping. In *Public Key Cryptography*, pages 293–305, 2000.

[6] P. Devanbu, M. Gertz, C. Martel, and S. Stubblebine. Authentic third-party data publication, 1999.

[7] M. Goodrich and R. Tamassia. Efficient authenticated dictionaries with skip lists and commutative hashing. Technical report, Johns Hopkins Information Security Institute, 2000.

[8] M. Goodrich, R. Tamassia, and A. Schwerin. Implementation of an authenticated dictionary with skip lists and commutative hashing. *DISCEX II*, 2001.

[9] P. C. Kocher. On certificate revocation and validation. *Financial Cryptography, Second International Conference*, pages 172–177, 1998.

[10] P. Maniatis. *Historic Integrity in Distributed Systems*. PhD thesis, Computer Science Department, Stanford University, Stanford, CA, USA, 2003.

[11] P. Maniatis and M. Baker. Secure history preservation through timeline entanglement. Technical Report arXiv:cs.DC/0202005, Computer Science Department, Stanford University, Stanford, CA, USA, Feb. 2002. Available at http://www.arxiv.org/abs/cs.DC/0202005.

[12] P. Maniatis and M. Baker. Authenticated append-only skip lists, 2003.

[13] R. C. Merkle. Protocols for public key cryptosystems. *IEEE Symposium on Security and Privacy*, pages 122–134, 1980.

[14] R. C. Merkle. A certified digital signature. *Advances in Cryptology*, 1990.

[15] S. Micali. Efficient certificate revocation. Technical report, MIT Laboratory for Computer Science, 1996.

[16] M. Naor and K. Nissim. Certificate revocation and certificate update. *In Proceedings 7th USENIX Security Symposium*, 1998.

[17] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, pages 668–676, 1990.