A New Timestamping Scheme Based on Skip Lists[1]

Kaouthar Blibech, Alban Gabillon

LIUPPA/CSySEC,
Université de Pau – IUT de Mont de Marsan
France
k.blibech@etud.univ-pau.fr alban.gabillon@univ-pau.fr

Abstract. Time stamping is a cryptographic technique providing us with a proof-of-existence of a message/document at a given time. Several timestamping schemes have already been proposed [1-10]. In this paper, we first define a new timestamping scheme which is based on skip lists [11]. Then, we show that our scheme offers nice properties and optimal performances.

1 Introduction

Timestamping is a technique for providing proof-of-existence of a message/document at a given time. Timestamping is mandatory in many domains like patent submissions, electronic votes or electronic commerce. Timestamping can ensure non-repudiation. Indeed, a digital signature is only legally binding if it was made when the user's certificate was still valid, and a timestamp on a signature can prove this. Parties of a timestamping system are the followings:

Client: forms the *timestamping request* which is the *digest* of the document to be timestamped. The client computes this digest by using a well known one-way[2] collision-free[3] hashing function. Submitting the digest of the document instead of the document itself preserves the confidentiality of the document.

TimeStamping Authority (TSA): receives the timestamping request at time *t* and issues the *timestamp*. The timestamp is a proof that the digest was received at time *t*. The TSA produces the timestamp according to a *timestamping scheme*.

Verifier: verifies the correctness of the timestamp by using the *verification scheme* corresponding to the timestamping scheme which was used to produce the timestamp.

Most of the existing timestamping schemes are *linking* schemes. Linking schemes were introduced by Haber and Stornetta [7]. Such schemes significantly reduce the scope of operations the TSA has to be trusted for. Basically, they work as follows:

During a time interval which is called a *round*, the TSA,

---

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

[2] one-way means that no portion of the original document can be reconstructed from the digest

[3] collision-free means that it is infeasible to find x and x' satisfying $h(x) = h(x')$

- receives a set of timestamping requests,
- aggregates the requests in order to produce a *round token*,
- returns the timestamps to the clients. Each timestamp consists of the round token, the digest and the authentication path proving that the round token depends on the digest.

Each round token is one-way dependent on the round tokens issued before. Round tokens are regularly published in a widely distributed media (a newspaper). After the publication it becomes impossible to forge timestamps (either to issue fake ones afterwards, or modify already issued ones), even for the TSA.

In the case of *partially ordered linking schemes* [1][2][3], only timestamps from different rounds are comparable whereas in the case of *totally ordered linking schemes* [5][6][7], the temporal order of any two timestamps can be verified even if these two timestamps belong to the same round. Partially ordered schemes are generally simpler than totally ordered schemes. However, as mentioned by Arne et al. [12], since totally ordered linking schemes allow us to compare two timestamps of the same round, longer rounds can be used. Using longer rounds enables reducing the amount of data to be published and the amount of data to be verified.

The purpose of this paper is to define a new totally ordered scheme which is simpler than the existing ones and which shows optimal performances. Our scheme uses a skip list. A skip list is a data structure which was defined by Pugh [11].

This paper is organized as follows: section 2 reviews related works. Section 3 presents our scheme. Section 4 deals with performance issues. Finally section 5 concludes this paper.

2 Related Works

Our scheme can be compared to the following existing schemes:

- Partially ordered timestamping schemes [1] [2][3]
- Totally ordered timestamping schemes [5][6][7]

Most of the existing partially ordered timestamping schemes are either based on Merkle trees (binary trees) [1][2] or on cryptographic accumulators [3]. With these schemes, only timestamps from different rounds are comparable. Moreover, schemes based on Merkle trees require the number of requests per round to be a power of 2 whereas schemes based on accumulators generally introduce a cryptographic trapdoor due to the use of the RSA modulus.

Existing totally ordered timestamping schemes are the simply linking scheme [7], the binary linking scheme [5] and the threaded authentication tree scheme [6]. The verification procedure for the simply linking scheme is costly (O(n), where n is the number of received requests) and requires that the TSA saves the entire chronological chain of timestamps.

The binary linking scheme uses a simply connected authentication graph. In addition to its complexity, this scheme is less efficient in terms of time complexity than the Merkle tree scheme for both timestamping and verification due to additional concatenation operations.

The threaded tree scheme can be seen as an improvement of the Merkle tree scheme. It is easier to implement than the binary linking scheme and it issues smaller timestamps. However, when compared to other schemes based on Merkle trees, it still has larger time complexity for both timestamping and verification due to the additional concatenation operations.

3 A New Timestamping Scheme

3.1 Skip lists

W. Pugh introduced skip lists as an alternative data structure to search trees [11]. 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.

Skip lists can be seen as a set of linked lists, one list per level (see figure 1). 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 some given constraints, then it belongs to level k+1. For example, in *perfect* skip lists (see figure 1), 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 figure1, B and E nodes are stored at all levels and called *sentinel* elements. The highest B node is called *starting node St*. The highest E node is called *ending node Et*.

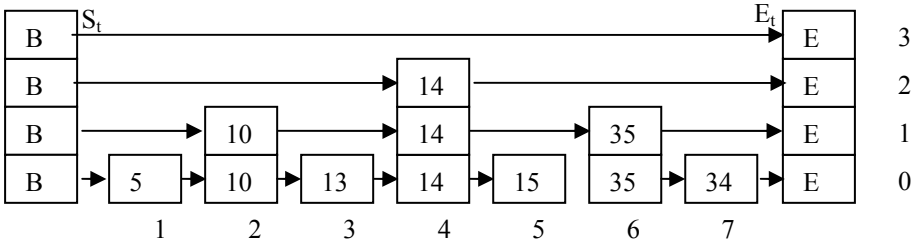

Fig. 1. Nodes contain the elements of the set {5,10,13,14,15,35,34}. Edges are the pointers. Numbers [0..3] are the levels. Numbers [1..7] are the indexes

3.2 Timestamping scheme

In [13], we defined an *authenticated dictionary* based on skip lists. An authenticated dictionary is a data structure that supports both update queries and *tamper-evident* membership queries. A tamper-evident membership query is of the form "does

element *e* belong to set *S*?". If *e* belongs to S then the answer to such a query is a proof that *e* belongs to *S*.

The purpose of this paper is to define a new totally linking timestamping scheme based on the dictionary we defined in [13]. Our scheme uses one *append-only* perfect skip list per round. Elements of the skip lists are the timestamping requests. Each new request is appended to the skip list. Since we are dealing with perfect skip lists, each element of the skip list is associated to one or several nodes according to the index of the request. Each node has the following four properties:

- its *value*, which is a timestamping request (digest)
- its *level*, ranging from 0 to the highest level of the skip list
- its *index*, which is its position in the skip list
- its *label*, which is a hash value one way dependent on the labels of the previous nodes.

Nodes associated to the same element have the same value and index. For example, let us consider nodes *a* and *p* in figure 2. They have the same index (20) and value (h_{20}). Level of node *a* is 2 whereas level of node *p* is 0. Labels of nodes *a* and *p* are not shown but they are different from each other.

The label of the starting node is the round token of the previous round whereas its value is the last request which was received during the previous round. Basically, our scheme works as follows:

- Alice sends a timestamping request which is the digest *h* of a document.
- The TSA appends *h* to the skip list.
- The TSA immediately returns to Alice a signed *acknowledgment* containing the index of *h* in the skip list and the proof that *h* is inserted after the elements which are already in the skip list. We call this proof the *head proof* (see algorithm 1).
- The TSA computes the label of each node associated to element *h* (see algorithm 2).
- At the end of the round, the TSA inserts the last request which becomes the ending sentinel element. The label of the ending node is the round token.
- The TSA publishes the round token and sends to Alice (and other clients) some additional information allowing her to prove that her request belongs to the round whose token has been published. We call this information the *tail proof* (see algorithm 3). The final timestamp consists of the digest *h*, the index of *h*, the head proof, the tail proof and the round token.
- If a verifier, Bob, needs to check the validity of the timestamp then he has to verify that he can compute the round token from *h*, the index of *h*, the head proof and the tail proof. Bob does the verification by processing algorithm 4.

Figure 2 shows the insertion of h_{21} at index 21. h_{16} to h_{21} are requests (digests of documents). Numbers [16..21] are indexes. Labels are not shown. The arrows denote the flow of information for computing the labels (see algorithm 2). The head proof for h_{21} consists of the labels of the dark grey nodes (nodes *q, o* and *a*) (see algorithm 1).

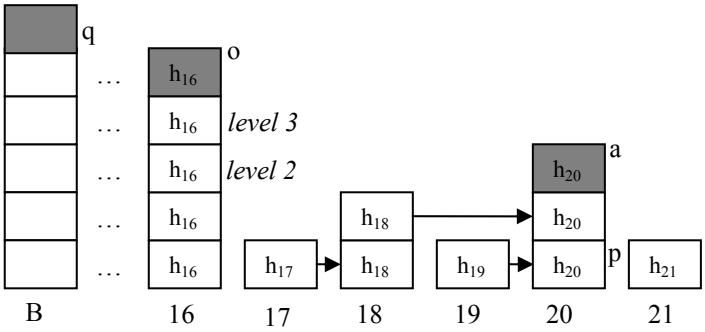

Fig. 2. Insertion of h_{21}

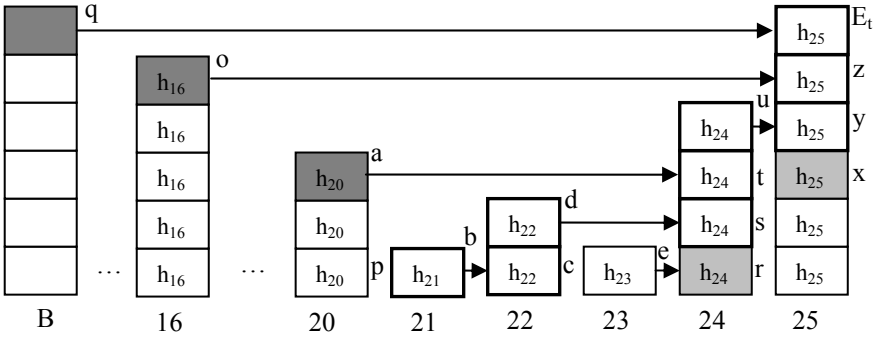

Fig. 3. Insertion of the ending element

Figure 3 shows the insertion of the ending node (last request of the round). The arrows denote the flow of information for computing the labels (see algorithm 2). The label of the ending node is the round token. The tail proof for h_{21} consists of the value h_{22} and the labels of the light grey nodes (nodes *r* and *x*) (see algorithm 3). Note that the last request of the round is h_{25}. Since it is the ending element, it belongs to all levels although 25 is not a multiple of 2^5. Figure 3 shows also the verification process for h_{21}. Labels of thick nodes are computed during the verification process (see the next section 2.3 about verification).

Algorithm 1 is used to compute the head proof (*hp*) of the newly inserted element *h*. *S* denotes the skip list. Function *height(S)* returns the highest level of the skip list *S*. Function *last(i)* returns the last node which was inserted at level *i* before the insertion of *h*. Function *label(n)* returns the label of node n. We define the *plateau* node of an element as the highest node associated to that element. Let us consider the nodes having an index strictly lower than the index of *h*. Among these nodes, for each level *l*, let us consider the node which has the greatest index. If it is a plateau node then its label belongs to the head proof.

Algorithm 1. Head Proof Computation

```
1: hp := {}
2: For i ranging from 0 to height(S) :
3:         If last(i) is a plateau node then
4:                 append label(last(i)) to hp
```

Figure 3 shows that the head proof of index 21 consists of the label of node *a* (that will be used during the verification to compute the label of node *t*), the label of node *o* (that will be used during the verification to compute the label of node *z*) and the label of node *q* (that will be used during the verification to compute the label of the ending node i.e. the round token).

Algorithm 2 is used to compute the labels of the nodes associated to the newly inserted element *h*. Function *value(n)* returns the value of node *n*. Function *left(n)* returns the left node of node *n*. For example, the left node of node *t* is node *a* (see figure 3). Function *down(n)* returns the bottom node of node *n*. For example, the bottom node of node *d* is node *c* (see figure 3). *hash* is a one-way collision-free hashing function and || is the concatenation operation. Algorithm 2 applies to each node associated to the newly inserted element starting from the node at level 0 until the plateau node.

Algorithm 2. Hashing Scheme

```
1: If down(n) = null, {n is at level 0}:
2:         If left(n) is not a plateau node then
3:                 label(n) := value(n).                          {case 1}
4:          Else
5:                 label(n) := hash (value(n) || label(left(n)))  {case 2}
6: Else :
7:         If left(n) is not a plateau node then
8:                 label (n) := label(down(n))                    {case 3}
9:         Else
10:                label(n) := hash(label(down(n))||label(left(n)))  {case 4}
```

Let us consider node *r* in figure 3 (index 24, value h_{24} and level 0) and node *e* (index 23, value h_{23} and level 0). The label of node *r* is equal to the hash of the value of node *r* (h_{24}) concatenated to the label of node *e* (case 2). Now, let us consider node *s* (index 24, value h_{24} and level 1) and node *d* (index 22, value h_{22} and level 1). The label of node *s* is equal to the hash of the label of node *r* concatenated to the label of node *d* (case 4). Let us consider also node *b* (index 21, value h_{21} and level 0) and node *p* (index 20, value h_{20} and level 0). Node *p* is not a plateau node, so the label of node *b* is equal to its value h_{21} (case 1). Finally, let us consider node *d* (index 22, value h_{22} and level 1) and node *c* (index 22, value h_{22} and level 0). The label of node *d* is equal to the label of node *c* (case 3).

Algorithm 3 is used to compute the tail proof (*tp*) of elements which were inserted during the round. Function *right(n)* returns the right node of node *n*. For example, the right node of node *d* is node *s* (see figure 3). Function *top(n)* returns the node on top of node *n*. For example, the top node of node *s* is node *t* (see figure 3). Computation

of the tail proof of element *h* starts from the plateau node associated to element *h* (in algorithm 3, *n* is initialized to the plateau node of element *h*).

Algorithm 3. Tail proof computation

```
{n is initialized to the plateau node of the element}
1: tp := {}
2: While right(n) != null :
3:         n := right(n)
4:         if down(n) = null then
5:                   append value(n) to TP
6:         Else
7:                  append label(down(n)) to TP
8:         While top(n) != null :
9:                   n: = top(n)
```

Figure 3 shows that the tail proof of element h_{21} consists of h_{22} (that will be used during the verification to compute the label of node *c*), the label of node *r* (that will be used during the verification to compute the label of node *s*) and the label of node *x* (that will be used during the verification to compute the label of node *y*).

3.3 Verification scheme

We call the *traversal chain* of element *h* the smallest sequence of labels that have to be computed from *h* in order to determine the round token (label of the ending node *Et*). An example of such a chain is given by the labels of the thick nodes in Figure 3. They represent the traversal chain of element h_{21}. The final timestamp consists of the digest *h*, the index of *h*, the head proof of *h*, the tail proof of *h* and the round token. It contains all the necessary information to compute the traversal chain of *h*. The verification process succeeds if the last label of the computed traversal chain is equal to the round token. If not, the verification fails.

Algorithm 4 describes the verification process. Regarding that algorithm, we need to define the following functions:

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

---

4 Since we are dealing with perfect skip lists, the height *h* of any element can be computed from its index *i* : $i = 2^h * k$ where $HCF(2, k) = 1$.

5 Since we are dealing with perfect skip lists, 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 $i - 2^l = k * 2^l$. Since we are dealing with perfect skip lists, if $HCF(2,k) = 1$ then the left node of *n* is a plateau node.

- *getNextIndex(index)*[7] that returns the index of the node whose label is the next label to be extracted by *getNext()*. That index can be computed from *index*.

In algorithm 4, h denotes the request and i_h the index of h (included in the timestamp). *token* denotes the round token included in the timestamp. Variable *label* denotes the label of the *current* node in the traversal chain. It is initialized to h.

As we can see, the index of the request in the skip list is a parameter of algorithm 4. If the TSA would lie on the index of the request, then the verification would fail since it would not be possible to compute the labels of the nodes belonging to the traversal chain. Since the head proof is returned as a signed acknowledgement immediately after the request was received, the TSA cannot reorder elements in the skip list even before publishing the round token.

Algorithm 4. Verification process

```
1 : {h is the request, i_h the index of h}
2 : label := h
3 : index := i_h
4 : level := 0
5 : While TP != {}
6 :       For i from level to height(index) :
7 :             If hasPlateauOnLeft(index, i) then
8 :                   If leftIndex(index, i) < i_h then
9 :                         label := hash(label||getPrec())
10:                   If leftIndex(index, i) ≥ i_h then
11:                         label := hash(getNext()||label)
12:       level := i.
13:       index := getNextIndex(index).
14 : While HP != {} :
15 :      label := hash(label||getPrec()).
16: If label = token then return TRUE
17:Else return FALSE
```

Figure 3 shows the verification process for h_{21} ($i_{h21} = 21$). Labels of thick nodes are computed during the verification process. Variable *label* is initialized to h_{21}. Initial node of level 0 and index 21 is node *b*. Index of left node of node *b* is $21\text{-}2^0$ (=20). Left node of node *b* is not a plateau node. Therefore, label of node *b* is equal to the value h_{21} contained in variable *label*. 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 $22\text{-}2^0$ (=21). Left node of node *c* is a plateau node. Therefore, label of node *c* is equal to the hash of the first label extracted from the tail proof (value of node *c*) concatenated to the label of node *b* ($hash(h_{22}||h_{21})$). 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. Therefore, label of node *d* is equal to label of node *c* ($hash(h_{22}||h_{21})$). Node *d* is a plateau node. Therefore, the next node processed by algorithm 4 is node *s* of index

---

[7] Since we are dealing with perfect skip lists, 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.

$22+2^1$ (=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 label extracted from the tail proof (label of node *r*) concatenated to the label of node *d*. 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 label extracted from the head proof (label of node *a*). 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*. 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) label extracted from the tail proof (label of node *x*) concatenated to the label of node *u*. 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. node *z*. Index of left node of node *z* is $32-2^4$ (=16). Left node of node *z* is a plateau node. Therefore, label of node *z* is equal to the hash of the label of node *y* concatenated to the second label extracted from the head proof (label of node *o*). Since node *z* is not a plateau node, the next node processed by algorithm 4 is the node of index 32 and level 5 i.e. the node on top of node *z* i.e. the ending node. Index of left node of the ending node is $32-2^5$ (=0). Left node of the ending node is a plateau node. Therefore, label of the ending node is equal to the hash of the label of node *z* concatenated to the last label extracted from the head proof (label of node *q*). Since there is no more labels to extract, neither from the tail proof nor from the head proof, Algorithm 4 compares the last computed label to the round token included in the timestamp. If the two labels are equal then the verification succeeds. If not, the verification fails.

4 Performances

We have implemented a prototype of our time-stamping scheme. We present the performances of our prototype in terms of space complexity. We focus on the number of hashing operations which are necessary to timestamp *n* documents (figure 4), and on the size of the timestamps (figure 5). In both figure 4 and figure 5, the X-axes stands for the number of requests. In figure 4, the Y-axis denotes the number of hashing operations made by the timestamping system whereas in figure 5, it denotes the number of digests included in the timestamps. From these two figures, we can see that the number of hashing operations is O(n) and the number of digests included in the timestamps is O(lg(n)). In fact, our scheme has the same space complexity than the most efficient schemes used for timestamping (Merkle trees, threaded tree scheme, binary linking scheme…). However, compared to the binary linking scheme and to the threaded authentication tree scheme, our timestamping scheme has a smaller time complexity both for timestamping and verification. Indeed, our scheme

needs as many concatenation operations as hashing operations, whereas binary linking scheme and threaded tree scheme need at least twice as many concatenation operations as hashing operations. Moreover, our scheme avoids the drawbacks of binary structures and accumulator systems.

Finally, let us mention that we could also compare our scheme to existing authenticated dictionary based on skip lists [10][14][15][16][17]. The reader can refer to [13] for such a comparison.

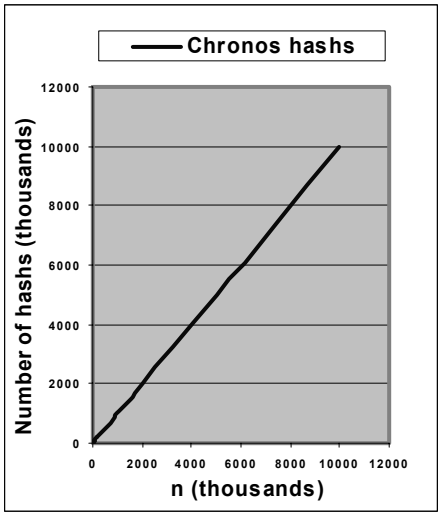

Fig. 4. Hashing cost

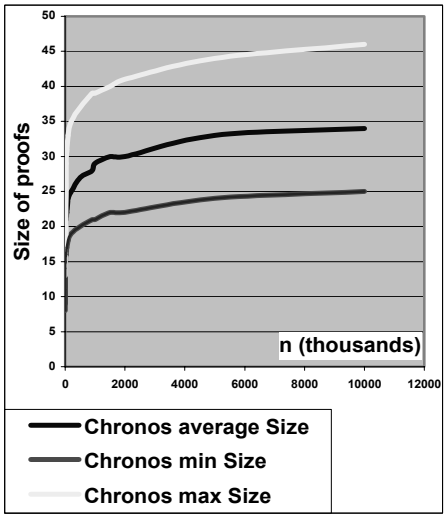

Fig. 5. Size of proofs

5 Conclusion

In this paper, we define a new totally ordered linking scheme based on skip lists. Our scheme offers better performances than existing totally ordered timestamping schemes. Moreover, it is easy to implement.

Our scheme is for a single server TSA. The main drawback of single server TSAs is that they are vulnerable to denials of service. In [18], we suggest some directions to implement a multi-server timestamping system. The main idea used in [18] is to randomly choose k servers among n. In a future work, we plan to develop a distributed version of our scheme based on skip lists, which would use this concept of k among n.

References

1. Bayer, D., Haber, S., Stornetta, W.: Improving the efficiency and reliability of digital time-stamping. In Sequences'91: Methods in Communication, Security and Computer Science, (1992) 329–334.
2. Benaloh, J., De Mare, M.: Efficient Broadcast time-stamping. Technical report 1, Clarkson University Department of Mathematics and Computer Science (1991).
3. Benaloh, J., De Mare, M.: One-way accumulators: A decentralized alternative to digital signatures. Advances in Cryptology (1993).
4. Buldas, A., Laud, P.: New Linking Schemes for Digital Time-Stamping. First International Conference on Information Security and Cryptology (1998).
5. Buldas, A., Laud, P., Lipmaa, A., Villemson J.: Time-stamping with Binary Linking Schemes. Lecture Notes in Computer Science, Vol. 1462. Springer-Verlag, Santa Barbara, USA (1998) 486–501.
6. Buldas, A., Lipmaa, A., Schoenmakers, B.: Optimally efficient accountable time-stamping. Public Key Cryptography (2000) 293–305.
7. Haber, S., Stornetta, W. S.: How to Time-stamp a Digital Document. Journal of Cryptology: the Journal of the International Association for Cryptologic Research 3(2) (1991).
8. Massias, H., Quisquater, J.J., Serret, X.: Timestamps : Main issues on their use and implementation. Proc. of IEEE 8th International workshop on enabling technologies: Infrastucture for collaborative enterprises (1999).
9. Massias, H., Quisquater, J.J., Serret, X.: Proc. of the 20th symposium on Information Theory in the Benelux (1999).
10. Maniatis, P., Giuli, T. J., Baker, M.: Enabling the long-term archival of signed documents through Time Stamping. Technical Report, Computer Science Department, Stanford University, California, USA, 2001.
11. Pugh, W.: Skip lists: a probabilistic alternative to balanced trees. Communications of the ACM (1990) 668–676.
12. Ansper, A., Buldas, A., Willemson, J.: General linking schemes for digital time-stamping. Technical Report (1999).
13. Blibech, K., Gabillon, A.: Authenticated dictionary based on skip lists for timestamping systems. Proc. of the 12th ACM Conference on Computer Security, Secure Web Services Workshop (2005).
14. Maniatis, P., Baker, M.: Secure history preservation through timeline entanglement. Technical Report arXiv:cs.DC/0202005, Computer Science department, Stanford University, Stanford, CA, USA (2002).
15. Maniatis, P.: Historic Integrity in Distributed Systems. PhD thesis, Computer Science Department, Stanford University, Stanford, CA, USA (2003).
16. Goodrich, M., Tamassia, R.: Efficient authenticated dictionaries with skip lists and commutative hashing. Technical report, Johns Hopkins Information Security Institute (2000).
17. Goodrich, M., Tamassia, R., Schwerin, A.: Implementation of an authenticated dictionary with skip lists and commutative hashing (2001).
18. Bonnecaze, A., Liardet, P., Gabillon, A., Blibech, K.: A Distributed time stamping scheme. Proc. of the IEEE conference on Signal Image Technology and Internet based Systems (SITIS '05), Cameroon (2005).