

A New Digital Notary System¹

Kaouthar Blibech, Alban Gabillon

LIUPPA/CSySEC,
Université de Pau – IUT de Mont de Marsan
371 rue du ruisseau, 40000 Mont de marsan, France
k.blibech@etud.univ-pau.fr alban.gabillon@univ-pau.fr

Abstract. Timestamping 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. In this paper, we shortly review existing schemes and then fully define a new timestamping system based on skip lists. We show that our scheme offers good performances.

Keywords: Timestamping, totally ordered, partially ordered, authenticated dictionaries, skip lists, security analysis, 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 (also called digital notary 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 way² collision-free³ 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.

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 $h(x) = h(x')$.

Kaouthar Blibech, Alban Gabillon

Most of the existing timestamping schemes are *linking* schemes. Linking schemes were introduced by Haber and Stornetta [14]. 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,

- 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* [2][3][4], only timestamps from different rounds are comparable whereas in the case of *totally ordered linking schemes* [11][10][14], 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. [1], 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 [21]. 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 Survey of timestamping systems

2.1 Partially ordered schemes

In those schemes, the authority aggregates a set of timestamping requests in order to produce a round value. Round values are regularly published. Requests of the same round can not be ordered while the temporal order of any two requests of different rounds can be verified.

Merkle Tree scheme Most of the timestamping protocols use a binary tree structure also called *Merkle Tree* [19][20]. However, this method is not always accurate. This is trivially the case when the number of timestamped documents is very small while the frequency of publication is very low. In that case, the accuracy of the timestamp may not satisfy the client. Notice also that this method is not practical when the number of documents is not close to a power of 2. In fact, if it is the case, then we need to add a number of padding elements to make it a power of 2.

Accumulator scheme Accumulators were introduced by Benaloh and de Mare in 1993 [4]. By accumulators they designate a family of function having some properties⁴. Modular exponentiation is a typical accumulator. A timestamping authority can use the modular exponentiation to compute a round token by accumulating the received requests. The main drawback of this scheme is that it depends on an RSA modulus n [4]. Indeed, if the authority would know the factorization of the RSA modulus then she would be able to compute fake timestamps. Therefore, this method introduces a practical problem: who should be the trusted entity computing the RSA modulus? Notice also that the modular exponentiation is slower than simple hashing operations.

2.2 Totally ordered schemes

In those schemes, every timestamping request is linked to the other requests in such a way that requests of the same round can be ordered. The authority produces round values that are published.

Linear Linking schemes The first totally ordered scheme which was defined in [14] links the timestamping requests in a linear chronological chain. By this mean, the authority creates a linear cryptographic dependency between successive requests. Some values of the resulting chain are then published and provide us with an absolute time. The verification process consists in recomputing the entire chronological chain between the publications that bracket the timestamping request. Verification is then costly. It is the main drawback of this scheme.

Binary linking schemes In order to provide relative ordering, Buldas and al. use a simply connected authentication graph for their timestamping scheme [9]. Buldas and al. proved that there is a verification chain between each pair of elements so that the relative order of timestamped requests can be proved. However, their scheme is less efficient than Merkle tree scheme. This is due to the additional concatenation operations both for construction and verification. In addition, time-stamping tokens are greater than the tokens produced by Merle Tree Scheme. This scheme is also more difficult to understand and to implement than the previous schemes. Finally, it has the same constraint on the number of inserted elements than the Merkle tree scheme.

Threaded tree schemes In order to improve the efficiency of the previous scheme, Buldas and al. developed a new scheme [10] based on Merkle tree structure. This scheme is easier to implement than the binary tree scheme and provides smaller time-stamping tokens. But, when compared to the Merkle tree scheme, this scheme has larger time complexity, both for construction and verification, due to the additional concatenation operations. Moreover, this method has the same constraint on the number of inserted elements than the binary linking scheme or the Merkle tree scheme.

⁴ Describing these properties is out of the scope of this paper.

3 A New Timestamping Scheme

3.1 Skip lists

W. Pugh introduced skip lists as an alternative data structure to search trees [21]. 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 *perfect* skip lists (see figure 1), which are the most known 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 1, 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 . In the example of figure 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.

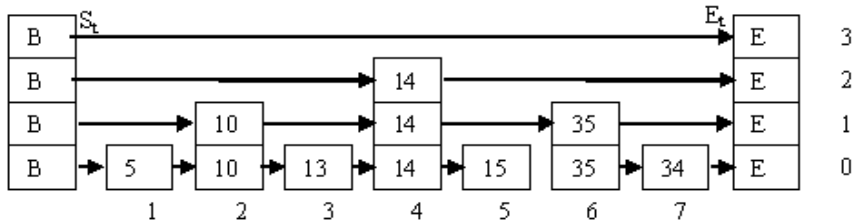


Fig. 1. An example of skip list

3.2 Timestamping scheme

In [5], 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 . In [6], we sketched a new totally linking timestamping system based on the authenticated dictionary we defined in [5]. The purpose of this paper is to fully define this new timestamping system and to study its performances.

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 are not shown but are different. 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 and 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).

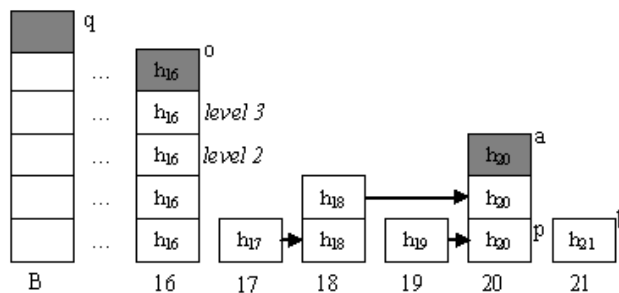


Fig. 2. Insertion of h_{21}

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).

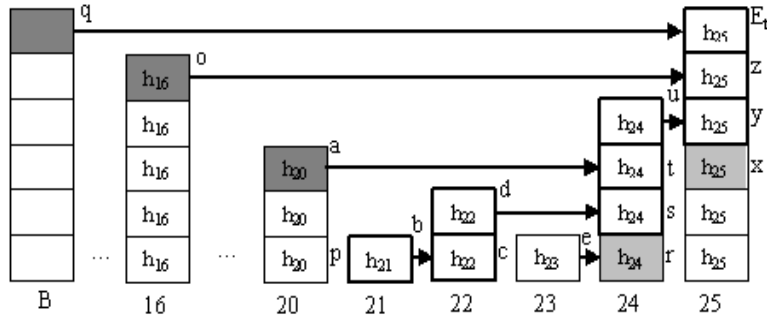


Fig. 3. Insertion of the ending element

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

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: $label(n) := value(n)$
- 4: Else
- 5: $label(n) := hash(value(n) || label(left(n)))$
- 6: Else
- 7: If $left(n)$ is not a plateau node then
- 8: $label(n) := label(down(n))$
- 9: Else
- 10: $label(n) := hash(label(down(n)) || label(left(n)))$

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. 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 figure3). $hash$ 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. 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 (line 5). 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 (line 10). 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} (line 3). 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 (line 8).

Algorithm 3. Tail proof computation
 $\{n$ is initialized to the plateau node of the element}
1: $tp := \{\}$
2: While $right(n) \neq null$
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 $top(n) \neq null$:
9: $n := top(n)$

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 figure3). 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). 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 sequence of nodes whose labels 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 thick nodes in Figure 3. They

Kaouthar Blibech, Alban Gabillon

represent the traversal chain of element h_{2^l} . 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 labels of the nodes that are in the traversal chain of h . The verification process succeeds if the last label of the computed sequence of labels is equal to the round token. If not, the verification fails.

Algorithm 4. Verification

```

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:         value := hash(label || getPrec())
10:      Else
11:        value := 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

```

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

- $height(index)$ ⁵ that returns the level of the plateau node at position $index$
- $leftIndex(index, level)$ ⁶ that returns the index of the left node of node of index $index$ and level $level$
- $hasPlateauOnLeft(index, level)$ ⁷ 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,
- $getNextIndex(index)$ ⁸ 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

⁵ 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 $HCF(2, k) = 1$.

⁶ Since we are dealing with perfect skip lists, the left node of a node of index i and level l has an index $j = i - 2^l$.

⁷ 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.

⁸ 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 .

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. Figure 3 shows the verification process for h_{21} ($i_{h_{21}} = 21$). 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 $21-2^0 (=20)$. 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 $22-2^0 (=21)$. 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 $22+2^1 (=24)$ 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 (<http://chronos.univ-pau.fr>). We present the performances of our prototype. Table 1 compares the performances of our scheme and the other timestamping schemes. For each scheme, it shows the number of hashing operations which are necessary to process one request and the number of hashing operations which are necessary to process all the requests belonging to one round (n is the number of processed requests). We can see that our scheme offers the same performances than the best timestamping schemes. In our scheme, each hashing operation comes after only one concatenation operation. BLS and TTS, need a greater number of concatenations before each hashing operation. Thus, time complexity in our scheme is much smaller than time complexity in those schemes. Furthermore, let us mention that the values presented for MTS, BLS and TTS stand for n being a power of 2. If it is not the case, then we need to add padding requests in MTS, BLS and TTS. This implies a significant number of useless concatenations and hashing operations.

Table 1. Timestamping costs

	Processing n requests	Processing one request
MTS⁹	$[n]c^{10} - [n]h^{11}$	$[1]c - [1]h$
LLS¹²	$[n]c - [n]h$	$[1]c - [1]h$
BLS¹³	$[3n/2]_{\sim}c^{14} - [n]h$	$[3/2]_{\sim}c - [1]h$
TTS¹⁵	$[n(2 + \lg(n)/2) - 1]c - [2n - 1]h$	$[\lg(n)/2 + 2]_{\sim}c - [2]h$
Chronos	$[n]c - [n]h$	$[1]c - [1]h$

We also focus on the size of timestamps and on the verification cost. For each scheme, table 2 shows the number of digests included in a timestamp (n is the total number of timestamps delivered in one round) and table 3 shows the number of concatenations and hashing operations necessary to verify a given timestamp. We can see that our scheme offers the same optimal performances than MTS. For example, for 10^7 delivered timestamps, each timestamp produced by Chronos includes 25 digests in the worst case. The verification requires 25 concatenations and 25 hashing operations. In BLS scheme, for the same number of requests, each timestamp includes about 100 digests, and verifying a timestamp requires about 100 concatenation

⁹ MTS for Merkle Tree Scheme

¹⁰ $[i]c$: i is the exact number of concatenation operations.

¹¹ $[i]h$: i is the exact number of hashing operations.

¹² LLS for Linear Linking Scheme

¹³ BLS for Binary Linking Scheme

¹⁴ $[i]_{\sim}c$: i is the average number of concatenation operations.

¹⁵ TTS for Threaded Tree Scheme

operations and 67 hashing operations. In TTS scheme, timestamps contain 25 digests, and the verification needs about 37 concatenations and 25 hashing operations.

Table 2. Size of timestamps

	MTS	LLS	BLS	TTS	Chronos
Size of Timestamps	$\lg(n) + 1$	$n - 1$	$4 \lg(n)$	$\lg(n) + 1$	$\lg(n) + 1$

Table 3. Verification costs

	Verification
MTS	$\lceil \lg(n) + 1 \rceil c - \lceil \lg(n) + 1 \rceil h$
LLS	$\lceil n - 1 \rceil c - \lceil n - 1 \rceil h$
BLS	$\lceil 9/2 \lg(n) - 3 \rceil \sim c - \lceil 3 \lg(n) - 2 \rceil \sim h^{16}$
TTS	$\lceil 1 + 3/2 \lg(n) \rceil \sim c - \lceil \lg(n) + 1 \rceil h$
Chronos	$\lceil \lg(n) + 1 \rceil c - \lceil \lg(n) + 1 \rceil h$

We could also compare our scheme to existing authenticated dictionary based on skip lists [5][12][13][15][16][17]. The reader can refer to [5] for such a comparison.

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 [7][8], we suggest some directions to implement a multi-server timestamping system. The main idea used in [7][8] 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] Ansper, A., Buldas, A., Willemsen, J.: General linking schemes for digital time-stamping. Technical Report (1999).
- [2] 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.
- [3] Benaloh, J., De Mare, M.: Efficient Broadcast time-stamping. Technical report 1, Clarkson University Department of Mathematics and Computer Science (1991).
- [4] Benaloh, J., De Mare, M.: One-way accumulators: A decentralized alternative to digital signatures. Advances in Cryptology (1993).

¹⁶ $\lceil i \rceil \sim h$: i is the average number of hashing operations.

Kaouthar Blibech, Alban Gabillon

- [5] Blibech K., Gabillon A.: A New Timestamping Scheme Based on Skip Lists. Proc. Of the 2006 International Conference on Computational Science and its Applications (Applied Cryptography and Information Security Workshop). Mai 2006. Glasgow, UK.
- [6] 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).
- [7] Bonnecaze, A., Liardet, P., Gabillon, A., Blibech, K.: A Distributed time stamping scheme. Proc. of the conference on Signal Image Technology and Internet based Systems (SITIS '05), Cameroon (2005).
- [8] Bonnecaze, A., Liardet, P., Gabillon, A., Blibech, K.: Secure Time-Stamping Schemes: A Distributed Point of View. In Annals of Telecommunication, Vol. 61, n°5-6, 2006.
- [9] Buldas, A., Laud, P.: New Linking Schemes for Digital Time-Stamping. First International Conference on Information Security and Cryptology (1998).
- [10] Buldas, A., Lipmaa, A., Schoenmakers, B.: Optimally efficient accountable time-stamping. Public Key Cryptography (2000) 293–305.
- [11] 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.
- [12] Goodrich, M., Tamassia, R., Schwerin, A.: Implementation of an authenticated dictionary with skip lists and commutative hashing (2001).
- [13] Goodrich, M., Tamassia, R.: Efficient authenticated dictionaries with skip lists and commutative hashing. Technical report, J. Hopkins Information Security Institute (2000).
- [14] 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).
- [15] 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).
- [16] 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.
- [17] Maniatis, P.: Historic Integrity in Distributed Systems. PhD thesis, Computer Science Department, Stanford University, Stanford, CA, USA (2003).
- [18] Massias, H., Quisquater, J.J., Serret, X.: Timestamps : Main issues on their use and implementation. Proc. of IEEE 8th International workshop on enabling technologies: Infrastructure for collaborative enterprises (1999).
- [19] Merkle, R. C.: Protocols for public key cryptosystems. IEEE Symposium on Security and Privacy (1980).
- [20] Merkle., R. C.: A certified digital signature. Advances in Cryptology (1990).
- [21] Pugh, W.: Skip lists: a probabilistic alternative to balanced trees. Communications of the ACM (1990) 668–676.