Oblivious Data Structures: Applications to Cryptography

Daniele Micciancio*
Laboratory for Computer Science
Massachusetts Institute of Technology
email: miccianc@theory.lcs.mit.edu

Abstract

We introduce the notion of oblivious data structure, motivated by the use of data structures in cryptography. Informally, an oblivious data structure yields no knowledge about the sequence of operations that have been applied to it other than the final result of the operations. In particular we define Oblivious Tree, a data structure very similar to 2-3 Tree, but with the additional property that the only information conveyed by an Oblivious Tree is the set of values stored at its leaves. This property is achieved through the use of randomization by the update algorithms.

We use the Oblivious Tree data structure to solve the *privacy* problem for incremental digital signatures raised by Bellare, Goldreich and Goldwasser. An incremental signing algorithm is *private* if the digital signature it outputs does not give any information on the sequence of edit operations that have been applied to produce the final document. We show how the incremental signature scheme of Bellare, Goldreich and Goldwasser can be made achieve privacy using Oblivious Trees instead of 2-3 Trees.

1 Introduction

We introduce the notion of oblivious data structure, motivated by the use of data structures in cryptography. Informally, a data structure is oblivious if it yields no knowledge about the sequence of operations that have been applied to it other than the final result of the operations. In particular we reduce the *privacy* problem for incremental digital signatures raised in [1] to the data structuring problem of designing *efficient* and *oblivious* search trees.

We design *Oblivious Tree*, a data structure similar to *2-3 Tree* [5], but with the additional property of being oblivious. In particular the nodes of an Oblivious Tree have bounded degree, all leaves are at the same level, the height of the tree is logarithmic in the number of nodes and update operations can be performed essentially at the same cost of the corresponding operations on 2-3 Trees: Oblivious Trees can be created in $O(n)$ time and leaves can be subsequently inserted or deleted in $O(\log n)$ expected running time. (The expectation computed with respect to the coin tosses of a single execution of the update algorithm.)

The rest of the paper is organized as follows. In Section 2 we give a brief introduction to incremental cryptography and the *privacy* problem for incremental digital signatures. This problem leads to the notion of oblivious search tree which is introduced in Section 3 and compared to related data structures in Section 4. The Oblivious Tree data structure is defined and analyzed in Section 5.

*Partially supported by DARPA contract DABT63-96-C-0018, "Security for Distributed Computer Systems". Appeared on the Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC '97)

In Section 6 we describe a private incremental digital signature scheme based on the tree signature scheme of [1] which uses the Oblivious Tree data structure to achieve privacy. Section 7 concludes with possible extensions to this work.

2 Incremental Cryptography

The idea of incremental cryptography, as outlined in [1], is to take advantage of the knowledge of the result of applying a cryptographic transformation (e.g. producing a digital signature) to a document D, to compute the cryptographic transformation of a different but related document D' quicker than performing it from scratch. The application that we have in mind is a text editor that maintains in the background a digital signature of the document being written (see Figure 1). Each time an edit operation f is applied to a document D, a digital signature σ of D is already available to the text editor and the incremental signing algorithm can be used to efficiently compute a new digital signature of the modified document $D' = f(D)$ as a fast function of σ, D and f. The advantage of using an incremental signing algorithm is clear: as soon as we finish writing the letter, a digital signature of it is immediately available.

There are several settings in which a considerable amount of time could be saved using an incremental signing algorithm. For example, assume that you want to send similar documents to different people. With an incremental signing algorithm you would sign the document just once, and then quickly update the signature to reflect the different recipient names. Another example where a lot of information is sent repeatedly with small differences from time to time is a surveillance video camera which time stamps and signs each image frame before sending it. The real time constraints of this application would make impractical to sign each image from scratch, but the difference between successive frames is usually little, so an incremental signing algorithm can be profitably used.

Incremental cryptography offers advantages over conventional cryptography, but it also brings in new concerns. The are two fundamental problems that arise in the design of an incremental digital signature scheme: the need of a new definition of *security* and the novel issue of *privacy*.

Security: The most important issue concerns the definition of security against attacks from an outsider (the *adversary*) who does not know the secret key. In the standard notion of security [7] the adversary can get signatures to messages of its choice and then must produce a signature for a new message (see Section 6 for more details). In the incremental setting it is natural to allow the adversary not only to obtain signatures for messages of its choice, but also to issue text modification commands to existing documents and obtain signatures of the modified ones. In some applications it is also prudent to assume that the adversary can tamper with the incremental signatures before the incremental signature algorithm is applied to them. In [1] is defined a strong notion of security (*tamper proof security*) against these more powerful adversaries.

Privacy: A second issue concerns the notion of privacy with respect to the intended recipient of the signed messages. Notice that a digital signature σ generated by an incremental signing algorithm, does not depend only on the document D being signed, but it is a function of the sequence of operations by which D has been obtained (see Figure 1). Some information on the way the document D has been obtained as a sequence of edit operations, can be computed from the signature of the final document produced by the incremental signing algorithm. Consider for example the following straightforward incremental signature scheme.

A Straightforward Scheme. In order to sign a document D from scratch, just use some standard (non incremental) signing algorithm S to compute $\sigma = S(D)$. If the document D is the modification $f(D')$ of some document D' of which we already know a digital signature σ',

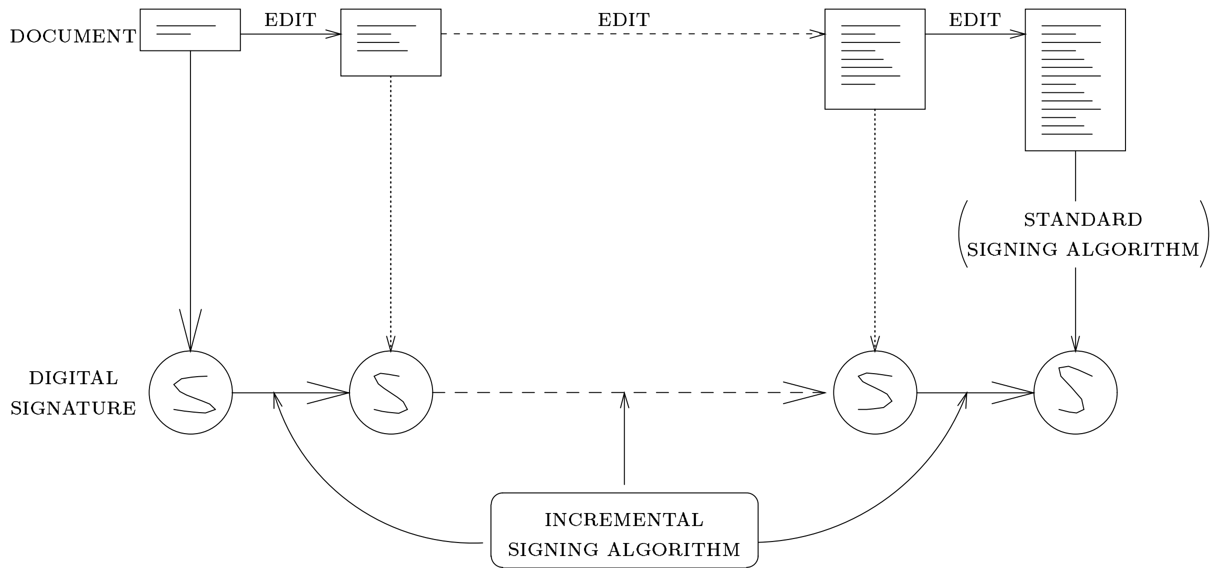

Figure 1: Incremental Signing Editor

compute the (non incremental) signature $\sigma = S(\sigma' \cdot f)$ and output the tuple (σ, σ', f, f') where f' is some edit operation such that $f'(D) = D'$. In order to check that (σ, σ', f, f') is a valid signature of document D, one verifies that σ' is a valid incremental signature of $f'(D)$, σ is a valid (non incremental) signature of $\sigma' \cdot f$ and the document D equals $f(f'(D))$.

The time required to sign $f(D')$, when a signature of D' is already available, is essentially proportional to the amount of modification f applied to D' instead of being proportional to the size of the whole document $f(D')$. Therefore, the above signing algorithm is incremental. Probably it is also secure, but *it is not private*: an incremental signature σ of document D gives to the recipient full information on all previous versions of the document D.

Even though there is no secrecy about the final document, you don't want to reveal information about intermediate documents that led to the final one. For example, suppose you are drafting a sensitive and important letter using the above mentioned text editor with incremental signature generation. When the final letter is complete, you certainly don't want the intermediate versions to be revealed through the signature.

3 Privacy and Oblivious Data Structure

In [1], Bellare, Goldreich and Goldwasser propose a digital signature method for which the signature algorithm is incremental. We shall call the digital signature of [1] a *tree signature* as it works essentially as follows. The document to be signed is first divided into blocks $D[1], D[2], \ldots, D[n]$ which are signed using a standard (i.e., non incremental) digital signature algorithm and the resulting signatures stored at the leaves of a search tree. Then each internal node is tagged with a (standard) digital signature of its children and an integer counting the number of leaves in the subtree rooted at that node. In particular they use 2-3 Trees [5] to maintain the tree balanced. For each basic edit operation (insertion or deletion of a sequence of blocks), the signature of the document can be updated using the corresponding update algorithm for 2-3 Tree and recomputing the signatures of the internal nodes that have been modified. Since 2-3 Trees have update algorithm that run

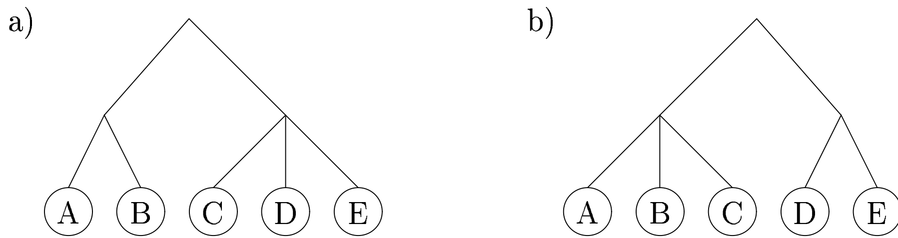

Figure 2: 2-3 Trees

in time $O(\log n)$, the tree signature algorithm is efficient: a tree signature can be updated after a basic edit operation in time proportional to the logarithm of the size of the whole document.

From the security point of view, the signing algorithm achieves tamper proof security (see Section 6 for more details). However the tree scheme *is not private*: a tree signature gives information on how the corresponding document has been obtained. This is not the first time that the use of tree based signatures causes information leakage (see for example [8] where tree based signatures are shown inappropriate as "blind signatures" in Damgaard's payment scheme).

We make the following *key observation*: the information about the previous versions of the signed document is given by the structure of the 2-3 Tree. For example if we build a 2-3 Tree storing the sequence of "ABCE" and then insert a "D" after the "C" we get the tree shown in Figure 2a, while if we build a 2-3 Tree storing "ACDE" and then insert a "B" after the "A" we get the tree in Figure 2b. So, the structure of the tree gives some information about the order in which nodes have been inserted. However, if the underlying standard signature scheme is state free, this is the only way tree signatures leak information about the previous versions of the document. If we replaced 2-3 Tree with a search tree data structure satisfying the property that the final result of applying a sequence of update operations does not give any information on the sequence of operations performed other than the set of leaves of the final tree, the tree signature scheme would be private.

Clearly, if we want the new signing algorithm to be incremental, we have to do that without increasing the running time of the update operations.

Therefore the privacy problem can be reduced to a purely data structuring problem: is it possible to define a search tree with efficient insert and delete algorithms such that the final result of applying any sequence of operations yields no information on the particular sequence other than the final set of leaves? We call this last property obliviousness.

Notice that data structures partially solving this problem have already appeared in the literature (see Section 4). In this paper we give a more satisfactory solution to this problem. We define a new data structure, called *Oblivious Tree*, very similar to 2-3 Tree, but with the additional property of being oblivious. In particular Oblivious Trees can be created in $O(n)$ time and leaves can be subsequently inserted or deleted in $O(\log n)$ expected running time, thus maintaining essentially the same performance of 2-3 Trees.

Insert and delete are defined as randomized algorithms: when a leaf is inserted or deleted, we make local changes to the topology of the tree based on the outcomes of a sequence of coin tosses. Essentially, we toss a coin for each internal node to decide its degree. The crucial point is that when the tree undergoes a local modification, we need to toss again the coins only for a small number of nodes, in the neighborhood of the path from the root to the inserted or deleted leaf. Nevertheless, we can prove that the final probability distribution is independent from the sequence of operations

performed.

This data structure solves the private signature problem introduced in [1]. Perhaps, more interestingly, Oblivious Trees offer advantages over other deterministic and probabilistic data structures, even from a purely algorithmic point of view.

Algorithmic improvements: The expected height of an Oblivious Tree is $\log_{2.5} n$, slightly improving the $\log_2 n$ bound offered by 2-3 Tree. As far as the running time is concerned, we prove that the insert and delete operations have $O(\log n)$ cost. The probabilistic analysis of our algorithms is made only with respect to the coin tosses of a single operation. This is in contrast with the use of randomization that is made in most probabilistic data structures (see Section 4). Usually the running time of an operation is analyzed with respect to all coin tosses made during the construction of the data structure. These include not only the coins tossed during the operation being analyzed, but also the coin tosses of previous executions of the update algorithms. Conversely, the running time of Oblivious Tree is analyzed without making any assumption on the coin tosses of the previous operations applied to the tree. Even in this "worst case" probabilistic analysis, we prove that the expected running time of the algorithms is $O(\log n)$, with negligible probability to deviate from the expected value. This means that the running times of the operations on Oblivious Trees can be bounded independently of each other even if the state of the data structure is known to the user.

Applications in distributed environments: Bounding the running time of the operations independently of each other, is of fundamental importance in certain applications. Consider a distributed environment in which the same data structure is accessed by several users. It is conceivable that each user, although willing to accept a probabilistic estimate on the cost of the operations he performs, wants the expected running time to be small with respect only to its own coin tosses. The possibility of the running time cost of the operations performed by one user being strongly influenced by those made by another one is undesirable. With our data structure the possibility of a user being slowed down by the malicious behavior of another user accessing the same data structure, is not present.

4 Related work on Data Structures

Oblivious Tree offers a probabilistic solution to the "uniquely represented dictionary" problem for which a deterministic $\Omega(n^{1/3})$ lower bound has been shown in [9]. Therefore randomization is necessary to achieve a $O(\log(n))$ running time for all update operations.

The idea of using randomization in performing tree operations has apparently appeared in the data structure literature before (see [4] and [3]) in order to improve on the algorithmic aspects of the tree operations.

Both *randomized search trees* ([4]) and *skip lists* ([3]) achieve $O(\log n)$ expected running time for insert and delete operations. It is interesting to note that obliviousness is achieved by these data structures, and in fact useful for the purpose of analyzing the running time of the algorithms as follows.

In randomized search trees and skip lists, the cost of an insert or delete operation essentially depends on the balance of the data structure. Randomization is used to keep the data structure balanced with high probability. The balance of the data structure is independent from the sequence of operations being applied, and in this sense the data structure is oblivious. This property is used to prove that the expected running time for each update operation is $O(\log n)$. However, this only works if the state of the data structure is kept hidden from the user issuing the update commands. If the user sees the coins tossed by the update algorithms, he can easily issue a sequence of operations

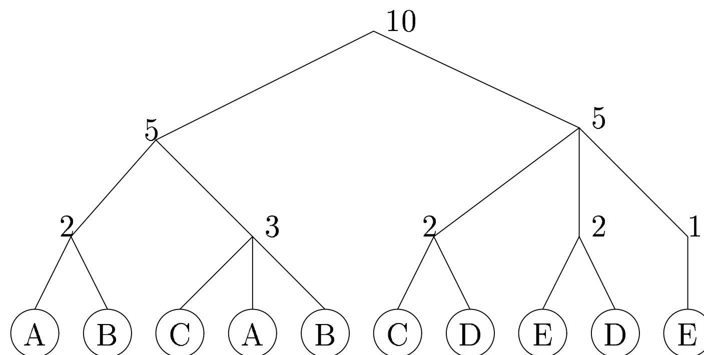

Figure 3: An Oblivious Tree

that makes the data structure unbalanced. This is because the expected $O(\log n)$ cost of each update operation is computed over all coins tossed since the creation of the data structure, and not just the coin tosses of the last update operation.

In a distributed environment in which some users can be malicious (as it is often the case in cryptographic applications), we don't want this (see last paragraph of Section 3).

The probabilistic running time analysis of Oblivious Tree is different. The expected running time of each operation is computed only with respect to the coin tosses of the update operation being analyzed. Therefore each operation has cost $O(\log n)$ even if the state of the data structure and the coins tossed during the execution of the previous operations are known to the user.

In conclusion, Oblivious Tree is the first data structure in which obliviousness is achieved not as a tool to prove other properties, but as an important property itself. However, even from a purely algorithmic point of view, Oblivious Tree achieves better performance than other oblivious data structures proposed in the literature, as Oblivious Tree exhibits *worst case* (over the previous coin tosses) $O(\log n)$ expected running time.

5 Oblivious Tree

An *Oblivious Tree* is a rooted tree such that

- all leaves are at the same level,
- all internal nodes have degree at most 3, and only the nodes along the rightmost path in the tree may have degree one.

Oblivious Trees differ from 2-3 Trees only because the nodes along the rightmost path of the tree may have degree one. This is not essential to the data structure but it does help simplifying the description of the update algorithms.

The *degree* of a node n, denoted $deg(n)$, is the number of children of n. The *size* of n, denoted $size(n)$, is the number of leaves in the subtree rooted at n, and is equal to the sum of the sizes of the children of n. A sequence of values $b_1 b_2 \dots b_n$ is stored at the leaves of an Oblivious Tree. Each internal node n stores $size(n)$. This information is used to efficiently locate the leaf storing each b_i.

For example, an Oblivious Tree storing the string "ABCABCDEDE" is shown in Figure 3.

Before going further in the definition of the algorithms that operate on Oblivious Trees, we formalize the requirement for this data structure of being "oblivious".

Definition 1 *Let $\mathcal{M}$ be a set of operations that act over search trees, and S be a set of algorithms implementing them. The set of algorithms S is* oblivious *iff for any two sequences of operations $p_1, \ldots, p_n$ and $q_1, \ldots, q_m$ the following is true. If $p_1 \ldots p_n$ and $q_1 \ldots q_m$ generate trees storing the same set of values L, then the execution of the sequence of algorithms in S implementing $p_1 \ldots p_n$ and the execution of those implementing $q_1 \ldots q_m$ have identical output probability distributions.*

The operations we consider are the following:

1. CREATE(L): build a new tree storing the sequence of values L at its leaves.
2. INSERT(b,i,T): insert a new leaf node storing the value b as the ith leaf of the tree T.
3. DELETE(i,T): remove the ith leaf from T.

The algorithms implementing the operations are probabilistic. A tree is produced first running CREATE(L) with a (possibly empty) initial sequence of leaves L and successively applying a sequence of modification operations INSERT and DELETE.

The algorithms are defined and analyzed in the following subsections.

We will prove obliviousness by showing that probability distribution obtained by running CREATE(L) and then applying a sequence of modification operations, is the same as we had directly run CREATE on the final sequence of leaves. To prove this is clearly sufficient to show that

- INSERT($b, i,$ CREATE(L)) outputs the same probability distribution as CREATE(L'), where L' is the sequence obtained from L by inserting a new element b at position i
- DELETE($i,$ CREATE(L)) outputs the same probability distribution as CREATE(L'), where L' is the sequence obtained from L by removing the ith element.

5.1 Create

Given a sequence of values L, we want to build a tree with leaves L in a randomized way, such that it is possible to efficiently insert or delete leaves preserving that probability distribution.

CREATE(L) works as follows. The tree is built up level by level starting from the leaves. The list of nodes at the ith level is obtained traversing the list of nodes at level $i+1$ from left to right and repeatedly doing the following:

1. Choose $d \in \{2, 3\}$ uniformly at random.
2. If there are less than d nodes left at level $i+1$, set d equal to the number of nodes left.
3. Create a new node n at level i with the next d nodes at level $i+1$ as children and compute the *size* of n as the sum of the sizes of its children.

For example, if we execute algorithm CREATE on input "ABCABCDEDE" and the outcomes of the coins $d \in \{2, 3\}$ are 2, 3, 2, 2, 3, 2, 3, 3 we get the tree shown in Figure 3.

It is easy to see that if a given level has k nodes, the next higher level will have at most $\lceil k/2 \rceil$ nodes. Therefore, after $O(\log |L|)$ iteration, the list of leaves L will be reduced to a single node. This node is the root of a tree storing at its leaves the sequence of values L.

5.2 Insert

We want to define an insertion algorithm such that for any list of leaves L the output distribution obtained by running the algorithms INSERT$(b, i,$ CREATE$(L))$ is the same as of CREATE(L') where L' is the list obtained from L by inserting a new element b at position i.

INSERT(b, i, T) locates the ith leaf of the tree using the size information associated to internal nodes. Then, the new leaf is inserted and the nodes following it are grouped as was previously done by CREATE but using new independent coin tosses. The way leaves are grouped will eventually synchronize with the previous grouping, at which point we can stop, possibly with a new node to be inserted at the previous level. Details follow.

1. Locate the ith leaf of the tree using the *size* information stored at the nodes and insert a new leaf storing the value b. Let $u_0, \ldots, u_h$ be the path from the root of the tree u_0 to the new leaf u_h.

2. Repeat the following steps for $l = h-1, h-2, \ldots, 0$. At each iteration $u_0, \ldots, u_l$ is the path from the root of the tree to a newly inserted node u_{l+1}.

   (a) If u_l is the last node at level l do the following.

      i. If u_l has degree two then go to step 3.

      ii. If u_l has degree three then select uniformly at random $d \in \{2, 3\}$ and if $d = 3$ go to step 3.

   (b) Initialize the variable $w = 1$ and repeat the following steps until $w = 0$.

      i. Let $u'_l = u_l$ and select uniformly at random $d \in \{2, 3\}$. d will be the degree of the node following u'_l at level l.

      ii. If $d = w$ or u'_l is the last node at level l, insert a new node u_l right after u'_l, change the parent of the last w children of u'_l to u_l, set $w = 0$ and skip to step 2(b)v.

      iii. Otherwise, let u_l be the node following u'_l at level l and let $t = deg(u_l)$. This node can be found as follows. Find the largest $j < l$ such that u_{j+1} is not the last child of u_j, set u_{j+1} to the next child of u_j and for $k = j+1, \ldots, l-1$ set u_{k+1} to the first child of u_k.

      iv. Change the parent of the last w children of u'_l to u_l and set $w = \max(0, t + w - d)$.

      v. Recompute the *size* field of the nodes along the path from u'_l (included) to u_j (excluded).

3. If $l \geq 0$, then recompute the size information of the nodes $u_l, u_{l-1}, \ldots, u_0$. Otherwise, increase the height of the tree by one creating a new root node r with children the previous root u'_0 and the new node u_0.

5.3 Delete

The DELETE(i, T) algorithm is similar to INSERT(b, i, T).

1. Let $u_0, \ldots, u_h$ be the path from the root of the tree u_0 to the ith leaf u_h.

2. Repeat the following steps for $l = h-1, h-2, \ldots, 0$. At each iteration $u_0, \ldots, u_l$ is the path from the root to a node u_l to be removed.

   (a) Delete u_{l+1} from the children of u_l.

(b) If u_l is the last node at level l do the following. If u_{l+1} was the only child of u_l, continue with the next iteration of loop 2 otherwise go to step 3.

(c) Initialize $w = 1$ and repeat the following steps until $w = 0$.

i. Set $u'_l = u_l$ and move u_l forward. This is done analogously as in algorithm INSERT step 2(b)iii.

ii. Let $t = deg(u_l)$. If $w \geq t$, then change the parent of the children of u_l to u'_l, set $w = 0$ and go to step 2(c)v.

iii. Select uniformly at random $d \in \{2, 3\}$.

iv. Change the parent of the first w children of u_l to u'_l and set w equal to $d - t + w$.

v. Recompute the *size* field of the nodes along the path from u'_l (included) to u_j (excluded).

3. If $l \geq 0$, then recompute the size information of the nodes $u_l, u_{l-1}, \ldots, u_0$. Otherwise, delete the node u_0 and make u_1 the new root of the tree.

5.4 Analysis

Proposition 1 *For any Oblivious Tree T storing n leaves, any $i \leq n$ and value b, the algorithms* INSERT(b, i, T) *and* DELETE(i, T) *have $O(\log n)$ expected running time. (Probability computed with respect to the coin tosses of a single execution of the algorithm.) Moreover, the probability for the running time to deviate from its expected value by a factor of α is exponentially small in α and in the height of the tree.*

We will prove the Proposition for INSERT(b, i, T). The analysis of the DELETE algorithm is analogous.

The only critical part of the INSERT algorithm is the loop 2b. In order to bound the number of times that loop 2b is executed before the exit condition $w = 0$ is satisfied, we first need to prove the following fact.

Lemma 1 *During the execution of loop 2b the variable w may assume only values in* $\{0, 1, 2\}$.

Proof: This is certainly true at the first iteration, as w is initialized to 1. Now, w may be set either to 0 at step 2(b)ii, or to $\max(t + w - d, 0)$ at step 2(b)iv. The only thing to be proved is that at step 2(b)iv we always have $t + w - d \leq 2$. We know that $t \in \{1, 2, 3\}$ by definition of Oblivious Tree, $d \in \{2, 3\}$ and $w \in \{1, 2\}$ by induction hypothesis. Notice that step 2(b)iv is executed only if $d \neq w$. It follows that $w - d \leq -1$ and $t + w - d \leq t - 1 \leq 2$.

We can give a probabilistic estimate of the number of times that loop 2b is repeated before $w = 0$.

Lemma 2 *At each iteration of loop 2b there is at least $1/4$ probability to set $w = 0$ (and therefore exit the loop) within two iterations (the probability is over the random choices $d \in \{2, 3\}$ at step 2(b)i).*

Proof: We know from Lemma 1 that at each iteration w equals either 1 or 2. If u'_l is the last node at level l we set $w = 0$ at step 2(b)ii. If $w = 2$, then with probability $1/2$ at step 2(b)i we choose $d = 2$ and we set $w = 0$ at step 2(b)ii. If $w = 1$ we distinguish two cases. If $t = 2$, with probability $1/2$ we have $d = 3$ and we set $w = t + w - d = 0$ at step 2(b)iv. If $t = 3$, with probability $1/2$, we

have $d = 2$ and we set $w = t + w - d = 2$. At the next iteration we will set $w = 0$ with probability $1/2$.

This already gives a polylogarithmic bound on the running time: the loop 2 is executed at most $h = O(\log n)$ times. Each time the inner loop 2b is executed an expected constant number of times, and each step takes at most $O(\log n)$ time. The expected total running time is $O(\log^2 n)$.

In fact, the running time of INSERT can be bounded more tightly. Notice that each node is visited at most two times: the first time either at step 1 or at step 2(b)iii, and the second time either at steps 2(b)iii and 2(b)v or at step 3. Therefore the total running time of the algorithm is proportional to the number of nodes visited.

Let T_l the number of nodes visited at level l. $\{T_l\}_l$ is a family of random variables whose exact value depends on both the input and the coin tosses of the algorithm. The nodes visited at level l can be divided into two groups: the nodes visited during iterations $h-1, h-2, \ldots, l+1$, and the nodes visited during iteration l of the main loop.

Using Lemma 2 the number of nodes visited at iteration l can be bounded by $2X_l$ where X_l is a random variable with geometric distribution of parameter $1/4$ that depends only on the coin tosses made during that iteration. The number of nodes visited during the previous iterations can be bounded by $T_{l+1}/2 + 1$, because the only nodes at level l visited at iteration $j > l$ are ancestors of nodes visited at level j.

Notice $\{X_l\}_l$ is a family of totally independent random variables, defined on the coin tosses of INSERT, and independent from the input tree. The total running time is given by

$$\text{Time} = \sum_{i=1}^{h} T_i \leq \sum_{i=1}^{h} \left(1 + \frac{T_{i+1}}{2} + 2X_i\right)$$

Subtracting Time/2 from both sides and multiplying by 2 we get the upper bound

$$\text{Time} \leq 2h + 4X,$$

where $X = \sum_{i=1}^{h} X_i$ is the sum of h totally independent random variables, all with geometric distribution of parameter $1/4$. In particular, we have $E[X] = 4h$. Moreover, the probability for the running time to deviate from its expected value by more than α is exponentially small both in α and in h.

This concludes the running time analysis. Let's now get to obliviousness. We already said that the only thing to be proved is that the update algorithms preserve the probability distribution defined by CREATE.

Proposition 2 *For all $L = L[1]L[2]\cdots L[n]$, $i \leq n$ and b the following equalities hold between probability distributions:*

$$\begin{aligned}
\text{INSERT}(b, i, \text{CREATE}(L)) &= \text{CREATE}(L[1], \ldots, L[i], b, L[i+1], \ldots, L[n]) \\
\text{DELETE}(i, \text{CREATE}(L)) &= \text{CREATE}(L[1], \ldots, L[i-1], L[i+1], \ldots, L[n]).
\end{aligned}$$

Proof: (Sketch) We will prove only the first equality. The proof of the second one is analogous. Consider how the nodes at level l are grouped by INSERT. First of all notice that each iteration l of the loop 2 modifies only the topology of nodes at level l. Let u_{l+1} be the node inserted at level $l+1$. The way the nodes up to u_{l+1} are grouped is not changed by INSERT. The nodes after u_{l+1}

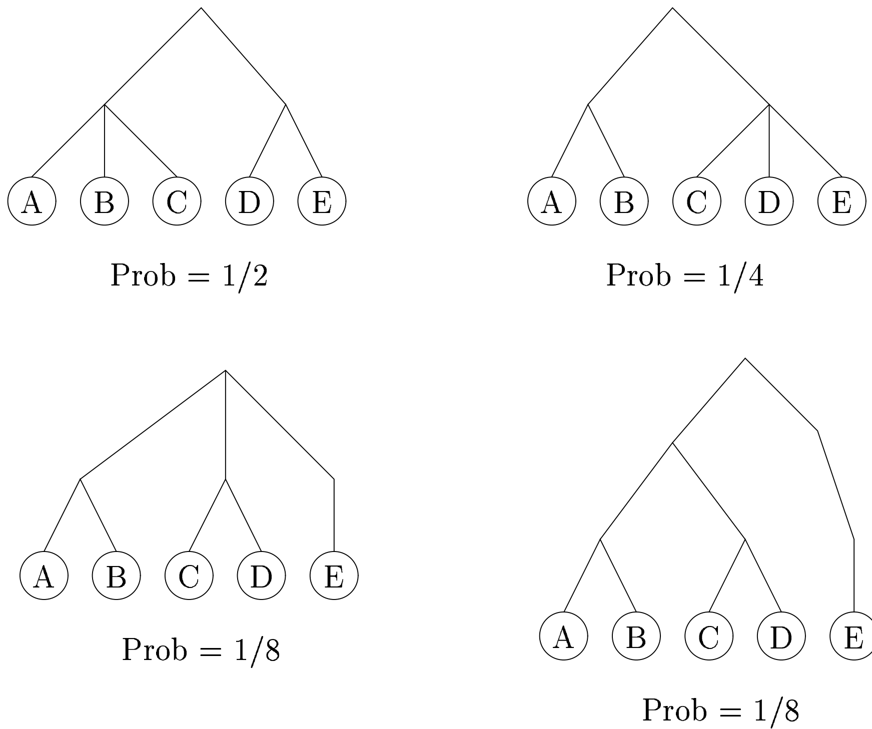

Figure 4: Result of CREATE(ABCDE)

are grouped exactly in the same way as algorithm CREATE would, but using new coin tosses. This defines the topology of the nodes up to u_l. The nodes after u_l are not modified.

Since the coin tosses used during the execution of the algorithms INSERT and CREATE(L) are independent, the final result is the same as we had run CREATE directly on the modified list of leaves.

For example if we run either CREATE($ABCDE$) or INSERT($B, 1,$ CREATE($ACDE$)), we will get one of the trees in Figure 4 with the shown probabilities.

6 A Private Incremental Signature Scheme

In this section we define in more detail the private signature problem and show how our data structure can be used to solve it. First we recall the definition of digital signature scheme.

Definition 2 *A* digital signature scheme *is a triple of probabilistic polynomial time algorithms* (KGEN, SIG, VF)*. The* key generation algorithm KGEN *takes as input a security parameter* 1^s *(i.e. s expressed in unary) and outputs a* secret key *Sk* *and a* public key *Pk*. *The* signing algorithm SIG *takes as input the secret key Sk and a message m and outputs a* digital signature $\text{SIG}_{Sk}(m)$. *The* verification algorithm VF *takes as input the verification key Pk, a message m and a string σ, and checks if σ is a valid signature of m (i.e.* $\text{VF}_{Pk}(m, \sigma) = 1$ *iff σ is a possible output of* $\text{SIG}_{Sk}(m)$*).*

The standard definition of security for a digital signature scheme (security against *chosen message attack* [7]) is that no probabilistic polynomial time algorithm A, capable of obtaining from

SIG signatures of messages of its choice, can produce a forgery with non-negligible probability, i.e. a valid signature for a message which has not been previously signed by SIG.

Let $\mathcal{M}$ be a set of text modification operations (e.g. let $\mathcal{M} = \{\texttt{insert}(b,i), \texttt{delete}(i)\}$, where $\texttt{insert}(b,i)$ is the operation of inserting a new block b at position i of a text and $\texttt{delete}(i)$ is the operation of deleting the ith block).

The following definitions are slightly different, but equivalent to those given in [1], the main difference being that in [1] INCSIG (see definition below) is defined as an interactive Turing machine, while here it is defined as a probabilistic algorithm.

Definition 3 *A signature scheme* (KGEN, SIG, VF) *is* incremental *with respect to a set of text modification operations $\mathcal{M}$, iff there exists a probabilistic polynomial time algorithm* INCSIG *(the* incremental signing algorithm*) which on input a pair of keys (Sk, Pk) generated by* KGEN*, a digital signature σ of a document D and an edit operation p, output a digital signature $\sigma' = \text{INCSIG}_{Sk,Pk}(\sigma, p)$ of the modified document $p(D)$.*

Another difference between our definition and that in [1] is that the current document is not input to INCSIG. In fact, INCSIG cannot read the whole document because otherwise it wouldn't be incremental. We assume that any information about the current document required to perform the signature update operation is contained in the description of the edit operation p.

Also the incremental signature σ could be too large to be read by INCSIG. In practice, the signature σ is not passed to and returned from the incremental algorithm INCSIG. Rather, σ resides in some form of memory support and is modified in place by INCSIG, which reads only a small portion of σ. We made σ an explicit parameter to INCSIG to emphasize that σ resides externally to INCSIG and a malicious user could alter the incrementable signature σ before issuing a command to INCSIG in the attempt of breaking the scheme.

As pointed out in [1], in order to define a notion of security for an incremental signature scheme it is necessary to refer to the documents by name. The reason is that forgery is defined as the ability to produce a signature to a new document. Therefore, we need to define of which documents we are requesting a signature when we call the incremental signing algorithm $\text{INCSIG}_{Sk,Pk}(\sigma, p)$.

We assume that the output of the signing algorithms SIG and INCSIG consists of a pair of strings (α, τ). The first string is a *document identifier*, and the second one is a digital signature of the document identified by α. We associate to each document identifier α a virtual document D_α defined as follows:

- If (α, τ) was obtained by calling $\text{SIG}(D)$, then $D_\alpha = D$.
- If (α, τ) was obtained by calling $\text{INCSIG}((\alpha', \tau'), p)$, then $D_\alpha = p(D_{\alpha'})$.

D_α is the document that the signer believes he has signed when he outputs (α, τ). Therefore, when we call the incremental algorithm $\text{INCSIG}((\alpha, \tau), p)$, this is considered as a request to sign document $p(D_\alpha)$.

Definition 4 *A* forger *is an algorithm A with oracle access to SIG_{Sk} and $\text{INCSIG}_{Sk,Pk}$, i.e., A can use $\text{INCSIG}_{Sk,Pk}$ and SIG_{Sk} as black boxes to obtain signatures $\text{INCSIG}_{Sk,Pk}((\alpha, \tau), p)$ and $\text{SIG}_{Sk}(D)$ of messages of its choice. We stress that the incremental signatures (α, τ) input to $\text{INCSIG}_{Sk,Pk}$ need not be valid signatures. We consider A to have requested signatures of D if it called algorithm $\text{SIG}_{Sk}(D)$ or $\text{INCSIG}_{Sk,Pk}((\alpha, \tau), p)$ and $D = p(D_\alpha)$. We say that A produces a forgery if it outputs a valid signature for a new document, i.e., a document of which it has not previously requested a signature to either $\text{INCSIG}_{Sk,Pk}$ or SIG_{Sk}.*

Security can now be defined following the standard paradigm.

Definition 5 *A incremental digital signature scheme* (KGEN, SIG, INCSIG, VF) *is* tamper proof secure *iff for any probabilistic polynomial time algorithm A, with oracle access to* $\text{INCSIG}_{Sk,Pk}$ *and* SIG_{Sk}*, the probability that A produces a forgery is negligible as a function of the security parameter s, i.e. it is less than $1/p(s)$ for any polynomial p and for all s large enough. The probability is computed with respect to the choice of (Sk, Pk) by* KGEN *and the coin tosses of algorithms A,* SIG *and* INCSIG.

We now define *privacy.* Informally, an incremental signature scheme is private if an adversary cannot tell whether a signature has been obtained by running the signature algorithm from scratch or by applying a sequence of edit operations.

Definition 6 *Let* (KGEN, SIG, INCSIG, VF) *be an incremental signature scheme and let (Sk, Pk) a pair of keys obtained by running* $\text{KGEN}(1^s)$. *Consider an adversary A operating in two stages as follows.*

- *First stage: A is given the public key Pk and outputs an initial document D and a sequence of text modification operations $p_1, \ldots, p_n$.*
- *Second stage: we flip a random bit $c \in \{0, 1\}$ and we give A an incremental digital signature of $D' = p_n(p_{n-1}(\ldots(p_2(p_1(D)))))$ obtained either by running $\text{SIG}_{Sk}(D')$ if $c = 0$, or by running $\text{INCSIG}_{Sk,Pk}(\text{INCSIG}_{Sk,Pk}(\ldots \text{INCSIG}_{Sk,Pk}(\text{SIG}_{Sk}(D), p_1) \ldots, p_{n-1}), p_n)$ if $c = 1$. A then outputs a bit b. A is successful if $b = c$.*

The incremental signature scheme achieves perfect privacy *iff for any (computationally unbounded) adversary A, the probability that A guesses c is $1/2$ (the probability is taken over the coin tosses of both the incremental signature scheme and the adversary A).*

A weaker notion of privacy can be defined with respect to a computationally bounded adversary.

Definition 7 *Let* (KGEN, SIG, INCSIG, VF) *and A be as in Definition 6. The incremental signature scheme achieve* computational privacy *if for any probabilistic polynomial time adversary A, the probability that A guesses c is negligible, i.e., for any polynomial p there exists an integer s_0 such that for all $s > s_0$ the probability of success of A is less than $1/2 + 1/p(s)$. (The probability computed over the random choice of $(Sk, Pk) \in \text{KGEN}(1^s)$ and the coin tosses of* SIG_{Sk}, $\text{INCSIG}_{Sk,Pk}$ *and A.)*

We now define an incremental signature scheme which achieves both tamper proof security and perfect privacy. The scheme is essentially the same as the *tree scheme* described in [1], with 2-3 Trees replaced by Oblivious Trees.

Let (G, S, V) be a standard (non incremental) signature scheme. We assume that the signature scheme (G, S, V) satisfies the following technical condition: for any two messages m_1 and m_2, if there is a valid signature σ of m_1 which is also a valid signature of m_2, then any valid signatures of m_1 are valid signatures of m_2.

We define an incremental digital signature scheme (KGEN, SIG, INCSIG, VF) on top of (G, S, V). The key generator KGEN is G itself. The algorithms SIG, INCSIG and VF use S_{Sk} and V_{Pk} as subroutines with the keys (Sk, Pk) generated by KGEN.

Algorithm SIG on input key Sk and document D, produces a pair (α, τ) where α is a binary string and τ is an Oblivious Tree with a label label(n) attached to each node n. The string α

is chosen at random and it is long enough to make the probability of a collision (i.e., that two executions of the algorithm generate the same string) negligible. Alternatively, IncSig and Sig can be defined as an interactive machine with an internal counter α which is incremented at each call.

If n is a leaf node the label is a digital signature of a block of text. If n is an internal node label(n) consists of a digital signature of the labels of the children of n concatenated with $size(n)$

The tree is obtained by first signing each block of the document D to get $\sigma_i = \text{Sig}_{Sk}(D[i])$. Then the algorithm Create($\sigma_1, \ldots, \sigma_n$) (see Section 5) is run modified as follows. When a new internal node n is created, the label label$(n) = \text{Sig}_{Sk}(l_1, l_2, l_3) \cdot size(n)$ is attached to the node, where l_1, l_2, l_3 are the labels of the children of n (if n has less then three children take $l_i = \epsilon$ for $i > deg(n)$). The label of the root has a special form. If n is the root, then label$(n) = \text{Sig}_{Sk}(l_1, l_2, l_3, \alpha) \cdot size(n)$.

The verification algorithm Vf works in the obvious way. It takes as input key Pk, document D and a signature (α, τ), and checks that all labels at the nodes of τ are valid. A label label$(n) = \sigma \cdot z$ is valid iff z is the sum of the sizes of the children of n and $V(msg, \sigma) = 1$ where msg is the concatenation of the labels of the children of n and the document identifier α if n is the root node.

We now define the incremental signing algorithm IncSig. On input (α, τ) and `insert`(b, i), IncSig generates a new document identifier α', signs the new block $\text{Sig}_{Sk}(b) = \sigma$, and runs the algorithm Insert(σ, i, τ) modified as follows. Each time a new node is visited, check if its label is valid, and when a node is modified recompute the label of its parent.

A more detailed description of the modifications to be made to algorithm Insert now follows.

- At step 1 use Vf_{Pk} to check that the labels of the nodes

$$u_0, \ldots, u_{l-1}$$

  are valid. In particular the root of τ must contain a valid digital signature of (l_1, l_2, l_3, α).

- At the end of step 2(b)iii check that the labels of $u_{j+1}, \ldots, u_l$ are valid.

- At steps 2(b)v recompute the labels of the nodes $u'_l, \ldots, u_j$ as well as their sizes.

- At step 3 recompute the labels of the nodes $u_l, \ldots, u_0$. The new label of the root node will contain a signature of the string (l_1, l_2, l_3, α') where α' is the new document identifier.

Let τ' be the new tree. The output of IncSig is (α', τ').

Edit operations `delete`(i) are treated analogously.

The above scheme meets all three requirements of being tamper proof secure, efficient and private.

Theorem 1 *If the signature scheme (G, S, V) is secure under chosen message attack, then the incremental signature scheme* (KGen, Sig, IncSig, Vf) *described above is tamper proof secure.*

The proof of this theorem is essentially the same as that in [1] and is sketched in Appendix A.

Theorem 2 *For any update operations, the expected running time of* IncSig *is $O(\log n)$.*

Proof: The running time of a document modification operation is proportional to the running time of the corresponding insert or delete tree operation. The theorem follows from Proposition 1.

Theorem 3 *The incremental digital signature scheme* IncSig *defined above achieves perfect privacy.*

Proof: Let A be an adversary as described in Definition 6 and let D and $p_1, p_2, \ldots, p_n$ be the initial document and the sequence of edit operations it outputs. A is given one of the two following signatures

$$\sigma_0 = \text{Sig}(p_n(\ldots(p_2(p_1(D)))))$$

$$\sigma_1 = \text{IncSig}(\ldots \text{IncSig}(\text{Sig}(D), p_1)\ldots, p_n).$$

Let's say, A receives σ_c where c is chosen at random from $\{0, 1\}$. We claim that the random variables σ_0 and σ_1 have identical probability distribution, and therefore σ_c and c are independent.

For $c = 0, 1$, σ_c consists of a pair (α_c, τ_c) where α_c is a randomly chosen binary string, and τ_c is a labeled Oblivious Tree constructed either by running Sig or by running IncSig. Clearly α_0 and α_1 have identical distribution. As regard the trees, it follows from Proposition 2 that the topology of τ_0 and τ_1 is distributed according to the same probability, independently from the sequence of operations used to build either tree. Finally, the label of each node in the trees is computed running the signing algorithm S_{Sk} with independent coin tosses. So, τ_0 and τ_1 have the same probability distribution.

This proves that σ_c and c are independent. Let c' the final output of A. Since c' is a function of σ_c and the coin tosses of A only, it is independent from c and the probability that $c = c'$ is $1/2$.

7 Discussion

We have defined efficient algorithms to insert and delete nodes in trees, satisfying the property that if two sequences of operations produce trees that have the same set of leaves, than the execution of the algorithms corresponding to the two sequences of operations produce identical probability distributions. We call the resulting data structure Oblivious Tree (supporting insertion and deletion operations).

An efficient incremental digital signature scheme is defined using the Oblivious Tree data structure. The incremental signature scheme achieves tamper proof security and perfect privacy, and thus solves an open problem raised in [1].

Perfect privacy (see Definition 6) is defined with respect to a computationally unbounded adversary which chooses the sequence of edit operations to issue to the incremental signing algorithm, but which only sees the final result of the operations. An interesting question which we leave open is whether privacy can be achieved with respect to an adversary (possibly computationally bounded, see Definition 7), which also gets partial information on the history of the data structure. As a scenario in which such a stronger notion of privacy is required consider the following: three signed copy of (different versions of) a document are sent to Alice, Bob and Charly. If a private incremental signing algorithm has been used to produce the three digital signatures, no single signature give any information about different versions of the signed document. However, Alice and Charly could collaborate and be able to get some information about the document sent to Bob by comparing the signatures they received.

Oblivious algorithms for other tree operations, such as *split* and *merge* of Oblivious Trees, can be defined following essentially the same ideas used in the definition of the insert and delete algorithms. For example, two Oblivious Trees can be merged by concatenating their nodes level by

level and flipping coins to choose again the degree of the nodes near the rightmost path of the first tree. An incremental signature scheme which supports *cut* and *paste* text modification operations can be easily defined using split and merge of Oblivious Trees, essentially in the same way we did here for insert and delete operations.

It is clear that the definition of obliviousness for search trees can be generalized to arbitrary data structures. An attempt to give a general definition of oblivious data structure is in Appendix B. We believe that the applicability of the notion of oblivious data structure extends beyond the particular problem solved here (privacy of incremental digital signatures), in particular to the area of cryptography.

8 Acknowledgements

I would like to thank Shafi Goldwasser for suggesting the problem and for her help and encouragement to write this abstract. Thanks also to Oded Goldreich, Ron Rivest and Marc Fischlin for useful discussions and comments.

References

[1] M. Bellare, O. Goldreich and S. Goldwasser, "Incremental Cryptography and Application to Virus Protection", *Proc. of the 27th Ann. ACM Symp. on the Theory of Computing.* 1995, pp 45-56.

[2] M. Bellare, O. Goldreich and S. Goldwasser, "Incremental Cryptography: The case of Hashing and Signing", *Advances in cryptology. Proceedings of the 14th Ann. International Conference.* pp 216-233. 1994 Springer-Verlag, LNCS 839.

[3] W. Pugh, "Skip Lists: A Probabilistic Alternative to Balanced Trees", *Univ. of Maryland, Tech. Report CS-TR-2190.* 1989.

[4] C. R. Aragon and R. G. Seidel, "Randomized Search Trees", *Proc. of the 30th Ann. IEEE Symp. on Foundations of Computer Science.* 1983. pp 540–545.

[5] A. Aho, J. Hopcroft and J. Ullman, "The design and analysis of computer algorithms", Addison Wesley, 1974.

[6] M. Wirsing, "Algebraic Specification", in *Handbook of Theoretical Computer Science.* Edited by J. van Leeuwen. Elsevier 1990. Vol. B, Chapter 13, pp 677-788.

[7] S. Goldwasser, S. Micali and R. L. Rivest, "A Digital Signature Scheme Secure Against Adaptive Chosen-Message Attacks", *SIAM Journal of Computing*, 17(2), 1988, pp. 281–308.

[8] B. Pfitzmann and M. Waidner, "How to Break and Repair a Provably Secure Untraceable Payment System", *Advances in Cryptology: Proceedings of the 11th Ann. International Conference.* 1991, LNCS 576, Springer-Verlag. pp. 338–350.

[9] A. Andersson and T. Ottmann, "Faster Uniquely Represented Dictionaries", *Proc. of the 38th Ann. IEEE Symp. on Foundations of Computer Science.* 1991. pp.642–649.

A Proof of Security (Sketch)

Assume for contradiction that the incremental digital signature scheme

$$(\text{KGen}, \text{Sig}, \text{IncSig}, \text{Vf})$$

is not tamper proof secure, i.e., there exists a forger algorithm $\mathcal{F}$ which succeeds with probability greater than ϵ. We will show that the standard digital signature scheme (G, S, V) is not secure againsts chosen message attack, contradicting our assumption.

For simplicity we assume that the document identifiers α produced by Sig and IncSig never repeat. This is true if the signing algorithms are defined as an interactive machine that keeps a counter α which is incremented at every call. If Sig and IncSig are state-free probabilistic algorithms, the length of α is large enough to make the probability of a repetition negligible.

We use $\mathcal{F}$ to define an algorithm A which forges S signatures. Algorithm A is given a public key Pk and has oracle access to S_{Sk}. A simulates $\mathcal{F}$ and answers its queries as follows.

When $\mathcal{F}$ ask for $\text{Sig}_{Sk}(D)$, A simulates Sig on input D using the oracle S_{Sk} to compute the labels of the nodes, and aswers $\mathcal{F}$ with the result of Sig.

A maintains a list M of the pairs message-signature obtained from oracle S_{Sk}. Before asking the oracle for a signature of message m, A first checks if the message m is not in the list M, but for some signature σ in M, $V_{Pk}(m, \sigma) = 1$. In such (m, σ) is a forgery and A stop with output (m, σ). Note that if this never happen, then all messages signed by the oracle have distinct valid signatures.

When $\mathcal{F}$ ask for $\text{IncSig}_{Sk,Pk}((\alpha, \tau), p)$, A first checks if τ contains a valid signature for some message not in M, i.e., a forgery. If so, A outputs it and stops. Otherwise A simulates IncSig on input $((\alpha, \tau), p)$ using the public key Pk to verify the labels of the nodes and using oracle S_{Sk} to sign the modified nodes, and answers $\mathcal{F}$ with the result of IncSig.

Eventually, $\mathcal{F}$ outputs an incremental signature (α, τ) of some new document D. A checks if τ contains a valid signature for some message not in M and outputs it.

Clearly A is polynomial time. We claim that A output a forgery with probability at least ϵ. We do this showing that if A does not output a forgery, then also $\mathcal{F}$ didn't succeed in forging the incremental signature scheme.

Assume that A does not output a forgery. A run the simulation of $\mathcal{F}$ till the end. Let (α, τ) the final output of $\mathcal{F}$. If (α, τ) is not valid, then also $\mathcal{F}$ failed. So, assume that all labels in τ are valid, but all messages signed in τ have already been signed by the oracle S_{Sk}, i.e., they are in M.

In particular, the root of τ contains a signature of (l_1, l_2, l_3, α) and this message has been signed to answer a query to $\mathcal{F}$. Since this message has a special form, at some point $\mathcal{F}$ must have made a query whose answer was (α, τ'). We will prove that $D = D_\alpha$, where D_α is the virtual document associated to the document identifier α, contradicting the hypothesis that D was a new message.

For each virtual document D_α we define a virtual tree τ_α. The definition mimics the one of a virtual document.

- If (α, τ) was output by $\text{Sig}(D)$, then the virtual tree of D_α is τ itself.
- If (α, τ) was output by algorithm $\text{IncSig}((\alpha', \tau'), p)$ then the virtual tree of D_α consists of a tree τ_α in which the new (or modified) nodes are the same as in τ and all other nodes are as in the corresponding subtrees of $\tau_{\alpha'}$.

The following important properties of virtual trees can be proved by induction on the definition of virtual documents and virtual trees.

Lemma 3 *For any document identifier α, all labels of the virtual tree τ_α have been created by the system.*

Lemma 4 *For any document identifier α, (α, τ_α) is a valid signature of D_α, i.e.,*

$$\mathrm{VF}_{Pk}((\alpha, \tau_\alpha), D_\alpha) = 1.$$

So, (α, τ_α) is a valid signature of D_α and (α, τ) is a valid signature of D. Moreover, all messages signed either in τ_α or in τ, have been signed by the system, and therefore are in M. Since all messages in M have distinct signatures, and the roots of the two trees τ and τ_α contains signatures of the same message, then the two trees τ and τ_α are the same at all nodes, as it can easily proved by induction on the level of the nodes. In particular, τ and τ_α are signatures of the same document $D = D_\alpha$.

B General Definition of Oblivious Data Structure

In order to define the notion of oblivious data structure, we first need to define what a data structure is. A common approach to the definition of data structures is that based on many-sorted algebras. For a general introduction to this topic the reader is referred to [6]. Here we will recall the basic definitions restricted to the case of a single sort to simplify the notation. The extension to the many-sorted case is straightforward.

A data structure can be defined as a set of objects which can be manipulated by a given set of operations Σ. More formally, the *syntax* of a data structure is specified by a *signature*, i.e. a set Σ of function symbols with an associated *arity* function $\alpha\colon \Sigma \to \mathbb{N}$. The meaning of the symbols in Σ is specified by a Σ-algebra. A Σ-algebra $\mathcal{A}$ is a pair (A, Σ_A) where A is a set, called the *carrier*, and Σ_A contains a function $f_A\colon A^n \to A$ for each function symbol $f \in \Sigma$ of arity $\alpha(f) = n$.

Compound expressions, called *terms*, can be built up from the symbols in Σ. For example if a, b and f are function symbols with arity $\alpha(a) = \alpha(b) = 0$ (i.e. a and b are constants) and $\alpha(f) = 2$ (i.e. f is a binary function) we can build the term $f(a, b)$. Given a Σ-algebra $\mathcal{A} = (A, \Sigma_A)$ we associate to each Σ-term t an element $t_A \in A$ defined by interpreting the symbols in t as the corresponding functions in Σ_A. A *congruence* over $\mathcal{A}$ is an equivalence relation over A such that for any $f \in \Sigma$ with $\alpha(f) = n$ and for any $a_1, \ldots, a_n, b_1, \ldots, b_n \in A$ such that $a_i \sim b_i$ for all i's, we have

$$f_A(a_1, \ldots, a_n) \sim f_A(b_1, \ldots, b_n).$$

For any $a \in A$, the equivalence class of a is defined by $[a] = \{b \mid b \sim a\}$.

In order to define oblivious data structure, we need to extend the above notions of Σ-algebra and Σ-congruence to allow the function symbols to be interpreted as probabilistic algorithms.

A probabilistic Σ-algebra is a pair (A, Σ_A) where A is a set, called the carrier, and Σ_A is a set of functions. For each function symbol $f \in \Sigma$ of arity $\alpha(f) = n$, Σ_A contains a randomized algorithm f_A from A^n to A. A *congruence* over $\mathcal{A}$ is an equivalence relation over A such that for all $f \in \Sigma$ with $\alpha(f) = n$ and for any $a_1, \ldots, a_n, b_1, \ldots, b_n \in A$ such that $a_i \sim b_i$ for all i's, we have $f_A(a_1, \ldots, a_n) \sim f_A(b_1, \ldots, b_n)$ for all possible random choices made during the computation of $f_A(a_1, \ldots, a_n)$ and $f_A(b_1, \ldots, b_n)$. In particular, $[f_A(a_1, \ldots, a_n)]$ does not depend on the random choices of f_A.

Given probability distribution D, we denote with $[D]$ the set of points with non zero probability.

Definition 8 *Let $\mathcal{A} = (A, \Sigma_A)$ be a probabilistic data structure and let $\sim$ be a congruence relation over $\mathcal{A}$.*

We say that $\mathcal{A}$ is oblivious with respect to $\sim$ if for any two terms t^1 and t^2 if $t^1_A \sim t^2_A$ then t^1_A and t^2_A define the same probability distribution.

For example Oblivious Trees are oblivious with respect to the equivalence relation $T_1 \sim T_2$ iff T_1 and T_2 have the same sequence of leaves.

The way obliviousness has been proved for our data structure in Section 5 suggests an alternative characterization of oblivious data structures.

Theorem 4 *Let $\mathcal{A} = (A, \Sigma_A)$ be a probabilistic data structure and let $\sim$ be a congruence relation over $\mathcal{A}$.*

$\mathcal{A}$ is oblivious with respect to $\sim$ iff there exists a family of probability distributions μ_C, one for each equivalence class $C \in \{[a] \mid a \in A\}$, such that and for all operation $f \in \Sigma$ of arity n, and for any n-tuple $(x_1, \ldots, x_n) \in A^n$ the probability distribution defined by $f_A(\mu_{[x_1]}, \ldots, \mu_{[x_n]}))$ is the same as $\mu_{[f_A(x_1,\ldots,x_n)]}$.

In the Oblivious Tree case the probability distribution μ_L is defined by $\textsc{Create}(L)$.