An Oblivious Data Structure and its Applications to Cryptography

Daniele Micciancio

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

June 1996

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 2-3 trees and update algorithms to insert and delete sequences of contiguous leaves, in such a way that the only information conveyed by an oblivious 2-3 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 oblivious 2-3 trees to solve the open problem of "private" incremental digital signatures raised by Bellare, Goldreich and Goldwasser (1995). A digital signature system is incremental if a document for which a digital signature has been produced can be edited and its digital signature can be efficiently updated to reflect the changes in the document. An incremental signature system is private if the digital signature produced by the system for the final version of a document that has undergone a sequence of edit operations, does not yield any information on intermediate versions of the document.

Keywords: Oblivious Data Structures, 2-3 Trees, Incremental Cryptography, Digital Signatures

1 Introduction

The idea of incremental cryptography, as outlined in [1], is to take advantage of the knowledge of the result of applying a cryptographic transformation to a document D, to compute the cryptographic transformation of a different but related document D' quicker than performing it from scratch.

In particular, [1] proposes a digital signature method for which the signature algorithm is incremental. Namely, the cost of updating a signature when the document is modified by a basic edit operation (e.g. the insertion or deletion of a sequence of blocks of text), is polynomial in a security parameter s (which is logarithmic in the size of the document), rather than proportional to the size of the entire document.

We shall call the digital signature of [1] a *tree signature* as it works essentially as follows. The blocks of a document are stored at the leaves of a tree. Each internal node contains a (standard) digital signature of its children. For each basic edit operation (insertion or deletion of a sequence of blocks), the signature of the document can be updated with changes that are local to a path from the root to the leaf just inserted or deleted. So, the cost of updating the tree signature of the document is proportional to the height of the tree. The height of the tree is kept logarithmic in the number of leaves through the use of 2-3 tree [5].

From the security point of view, the tree signing algorithm achieves tamper proof security (see section 4 for more details). An open problem remains: the privacy of signatures.

1.1 The Privacy Problem

The application that we have in mind is a text editor that maintains in the background signed copies of the documents being written using an incremental signing algorithm. The advantage of using such an editor is that when your document is finished, a digital signature of it is immediately available.

The signature of each version of a document is obtained as a function of a previous version of the same document and the previous version's signature. Some information on the way the document has been obtained as a sequence of edit operations, can be computed from the signature of the final document obtained by the incremental signature algorithm. Even though there is no secrecy about the final document, it may be undesirable for the signature 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.

In this paper we solve this problem. We do this by introducing oblivious 2-3 trees, an implementation of 2-3 trees [5] in which the operations are defined as probabilistic algorithms and satisfy the intuitive property of hiding the sequence of operations that has been applied to a tree.

1.2 Oblivious 2-3 tree

Our solution to the privacy problem consists of the definition of new insert and delete operations for 2-3 trees with the remarkable property that the topology of the tree obtained by applying any sequence of operations yields no information on the particular sequence of operations used[1]. We call this property obliviousness, and the resulting data structure oblivious 2-3 tree. 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 a leaf-to-root path. Nevertheless, we can prove that the probability distribution on 2-3 trees induced by a sequence of operations, is independent from the sequence of operations used.

This data structure solves the private signature problem introduced in [1].

Perhaps, more interestingly, oblivious 2-3 trees offer advantages over other deterministic and probabilistic data structures, even from a purely algorithmic point of view.

Algorithmic improvements to standard 2-3 tree: The expected height of an oblivious 2-3 tree is $\log_{2.5} n$, slightly improving the $\log_2 n$ bound offered by deterministic 2-3 trees.

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 operations on 2-3 trees is made only with respect to the coin tosses of the operation being executed, without any assumption on the input tree, the global sequence of operations or the coin tosses made during the execution of operations in the past. This is in contrast with the use of randomization that is made in most probabilistic data structure (see section 1.3). 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. Therefore the running times of the operations on oblivious 2-3 trees can be bounded independently of each other.

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.

[1] This is certainly not true for the usual insert and delete operations on 2-3 tree: for example, if a tree is built by inserting all leaves in order from left to right, all internal nodes (exception made for those along the rightmost path of the tree) will have degree two.

1.3 Related work on Randomized Data Structures

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 how in these data structures obliviousness (although not even defined) is achieved and used for the purpose of analyzing the running time of the algorithms.

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 single operation is $O(\log n)$.

However, the expected running time behavior of randomized search trees and skip lists is different from the one exhibited by our oblivious 2-3 trees. The running time of the operations on randomized search trees and skip lists depends not only on the coin tosses that are made during the operation being analyzed, but also on those made during the previous insertion and deletion operations. The expectation is computed with respect to all coin tosses made since the creation of the data structure.

Thus a malicious user can cause the data structure to become unbalanced and perform poorly, if the sequence of operations he executes on the data structure is not independent from the coin tossed during the execution of the previous operations. Notice that oblivious 2-3 trees are not subject to this weakness because they have *worst case* (over the inputs) $O(\log n)$ expected running time.

We illustrate the "malicious user" problem on randomized search trees. This data structure is defined in [4] as a rooted binary tree whose nodes have associated a *key* and a *priority* such that the nodes form a search tree with respect to their *keys*, and a *heap* with respect to their priorities. Priorities are chosen at random, so that the tree is kept balanced with high probability. In [4] it is pointed out that in order to maintain the tree probabilistically balanced, the priorities of the nodes must be kept hidden from the "user". In a distributed environment in which some users can be malicious (as it is often the case in cryptographic applications), this is far from being a realistic assumption because the priorities of the nodes can be detected by analyzing the running time of the access operations. Note that a malicious user do not even need to bias its own coin tosses: knowing their outcomes is enough to create a very "non-random" and unbalanced tree by a polynomial number of updates. Similar remarks apply to skip lists.

In conclusion, oblivious 2-3 tree is the first data structure that achieves obliviousness, not as a tool to prove other properties, but as an important property itself. Even from a purely algorithmic point of view, oblivious 2-3 tree achieves better performance than other data structures achieving obliviousness as a side effect proposed in the literature, as oblivious 2-3 tree exhibits *worst case* (over the inputs) $O(\log n)$ expected running time.

1.4 Outline

The rest of the paper is organized as follows. In section 2 we give some basic definitions. In section 3 oblivious 2-3 trees are defined and analyzed. In section 4 we show how oblivious 2-3 trees solve the privacy problem for incremental signature. Section 5 concludes with some remarks on the general notion of oblivious data structure.

2 Notation and Terminology

A 2-3 tree is a rooted tree in which all internal nodes have either two or three children and all leaves are at the same level. The leaves of a 2-3 tree store values taken from a totally ordered set of keys K. The keys stored at the leaves of a 2-3 tree are all distinct and appear in increasing order when the leaves are visited from left to right.

In the representation of 2-3 trees that we will use, the nodes at each level of a tree are organized in linked lists to allow an easy traversal of the levels. Each internal node n has the following fields:

- a key $n.key$ storing the minimum key of the subtree rooted at n,
- an integer $n.deg \in \{2, 3\}$ storing the degree of node n,
- a pointer $n.child$ to the first child of n,
- a pointer $n.next$ to the next node at the same level.

A new node with $n.deg = d$, $n.child = c$ and $n.next = n$ is created by the operation *new-node*(d, c, n). The value of field $n.key$ needs not to be specified explicitly because $n.key = (n.child).key$. We assume that each time the pointer $n.child$ is changed, also the field $n.key$ is suitably modified. The ith successor of a node n is defined by $n[0] = n$, $n[i + 1] = (n.next)[i]$. For all internal nodes n such that $n.next \neq$ NIL we have $n.child[n.deg] = n.next.child$.

A 2-3 forest is an ordered list of 2-3 trees all having the same height. If N is a pointer to a node in a 2-3 tree, N can be thought as pointing to a node, pointing to a 2-3 tree (the subtree rooted at N), pointing to a list of nodes ($[N[0], N[1], N[2], \ldots]$) or pointing to a forest (the list of trees rooted at $N[0]$, $N[1]$,$\ldots$). All these different sets of nodes are denoted by *Node*(N), *Tree*(N), *List*(N) and *Forest*(N) respectively. *length*(N) denotes the length of *List*(N): *length*$(NIL) = 0$, otherwise *length*$(N) = 1 +$ *length*$(N.next)$.

3 Oblivious 2-3 tree

In this section we define a set of update algorithms for 2-3 trees that are both efficient and oblivious, as defined below. The operations we consider are insertion and deletion of a key. The algorithms implementing the operations, INSERT(k,T) and DELETE(k,T), are probabilistic and have expected running time $O(\log n)$ where the expectation is taken on the internal coin tosses of the algorithm only, and not on the possible values of k and T.

Definition 1 *Let O be a set of operations that act over 2-3 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 $p_1 \ldots p_n$*

```
BuildLevel(L):
    if length(L) = 1 then return L
    if length(L) = 2 then return new-node_0(2, L[0], NIL)
    if length(L) = 3 then return new-node_0(3, L[0], NIL)
    if length(L) = 4 then
        return new-node_0(2, L[0], new-node_0(2, L[2], NIL))
    if length(L) ≥ 5 then
        toss a coin d ∈_R {2, 3}
        return new-node(d, L[0], BuildLevel(L[d]))
```

Figure 1: The BuildLevel Algorithm

and $q_1 \dots q_m$ generate trees storing the same set of leaves L, then the execution of the sequence of algorithms in S implementing $p_1 \dots p_n$ and the execution of those implementing $q_1 \dots q_m$ define identical probability distributions over 2-3 trees with leaves L.

In our case, the operations are the insertion and deletion of a key. The corresponding algorithms, Insert(k,T) and Delete(k,T), are defined and proved oblivious in sections 3.2 and 3.3. The running time is analyzed in section 3.4. We will prove obliviousness by defining, for any set of keys L, a probability distribution μ_L over the set of trees with leaves L. We then show that for any key k the probability distribution over the trees with leaves $L \cup \{k\}$ given by Insert(k, μ_L) coincides with $\mu_{L\cup\{k\}}$. Analogously, the probability distribution over the trees with leaves $L \setminus \{k\}$ defined by Delete(k, μ_L) is $\mu_{L\setminus\{k\}}$.

It follows that if a tree is built up using exclusively the two algorithms Insert(k,T) and Delete(k,T), the probability distribution defined by the final output of the algorithms executed, yield no information on the sequence of operations performed, other than the final set of leaves.

3.1 The probability distribution

The probability distribution μ_L over the set of trees with leaves L, is defined by an algorithm BuildTree(L) that given an ordered list of leaves L returns a 2-3 tree with leaves L. The algorithm is probabilistic and induces a family of probability distributions

$$\mu_L(T) = \Pr[\textsc{BuildTree}(L) = T].$$

We add to internal nodes a new field *n.random* storing a single bit. *n.random* is set to 1 iff the degree of node n has been randomly chosen between 2 and 3 by a coin flip. Unless otherwise stated the field *n.random* is always set to 1. A new internal node with field *random* set to 0 is created by the operation *new-node*$_0$(*deg*, *child*, *next*).

The tree is built up level by level using the subroutine BuildLevel shown in figure 1. The list of nodes at level i is obtained traversing the list of nodes at level $i+1$ and grouping them in groups of either two or three elements. The nodes in each group

```
BuildTree(L):
    if L is unordered then sort L
    while length(L) > 1 do
        L ← BuildLevel(L)
    return (L)
```

Figure 2: The BuildTree Algorithm

become the children of a node n in the upper level. The degree of n is the size of the associated group of nodes at level $i+1$, and is chosen uniformly at random between 2 and 3, provided that level $i+1$ contains at least 5 nodes. If we are left with two, three or four nodes at level $i+1$, there is only one way to group them in contiguous subsets of size two and three. So, in this case no randomization is involved in the construction of level i and the field *random* of node n is set to 0. Notice that if $n.random = 0$ then n is one of the last two nodes of its level.

The subroutine BuildLevel(N) takes as input a pointer N to a 2-3 forest. If N points to a forest with n trees, BuildLevel(N) return a forest of at most $n/2$ trees. The algorithm BuildTree, shown in figure 2, takes a list of leaves L and returns a 2-3 tree with leaves L. This is accomplished by repeatedly calling BuildLevel until L is reduced to a single tree.

3.2 Insert

We want to define an insertion algorithm Insert(k, T) such that the probability distribution over the trees with leaves $L \cup \{k\}$ defined by Insert(k, μ_L) is equal to $\mu_{L\cup\{k\}}$.

We first define a subroutine Ins(k, N) which takes as input a key k to be inserted, and a pointer N to a 2-3 forest. The inputs to Ins(k, N) must satisfy the condition $N.key \leq k$.

Ins(k, N) inserts the key k in the forest pointed to by N and returns a key k'. Ins(k, N) visits and possibly modifies the nodes of an initial sublist of *List*(N). The execution of Ins(k, N) may result in the insertion of a new node in *List*(N). The key of the last visited node is returned: if Ins(k, N) returns the key k', then all nodes in *List*(N) after the call with key greater than k' are guaranteed to be as they were before the execution of Ins(k, N).

ALGORITHM Ins(k,N):

1. If N points to a leaf node, then insert the new key k in the ordered list pointed to by N and terminate with return value k.
2. Advance the pointer N ($N \leftarrow N.next$) until either $N.random = 0$ or ($N.next$) has key greater than k.
3. Initialize a pointer M to the first child of N ($M \leftarrow N.child$).
4. Call recursively Ins(k, M) and store the returned value in k'.

```
INSERT(k,T):
{ T.next = NIL and T.key = −∞ }
    call INS(k, T)
    if (T.next = NIL) then return (T)
    otherwise return (new-node(0, T, NIL))
```

Figure 3: The INSERT Algorithm

5. If no coin has been tossed for the node pointed to by N ($N.random = 0$), then run the algorithm BUILDLEVEL on M. Replace the list of nodes pointed to by N with the result of BUILDLEVEL(M) and terminate with return value $+\infty$.
6. If the key of M is greater then k' do
   (a) If the first child of N is M terminate with return value $N.key$.
   (b) Toss a coin $d \in_R \{2, 3\}$.
   (c) If the first child of N is $M[d]$ then insert, immediately before N, a new internal node of degree d and with first child M. Terminate with return value $N.key$.
   (d) otherwise, set the degree of N to d and go on.
7. Set $N.child$ to M. Advance M of $N.deg$ positions ($M \leftarrow M[N.deg]$). Advance N of one position ($N \leftarrow N.next$) and go back to step 5.

The INSERT(k, T) algorithm, shown in figure 3, calls INS(k, T) as a subroutine. T is assumed to point to a tree, that is $T.next$ = NIL. To ensure that $T.key < k$, we assume that the tree contains a leaf with key $-\infty$. If the execution of INS(k, T) results in the insertion of a new node at level 0, then a new root node with children T and $T.next$ is created.

A proof of the correctness of the algorithm is implicit in the proof of obliviousness.

Proposition 1 *For any set of leaves L and for any leaf k, the following equality holds between probability distributions:*

$$\text{INSERT}(k, \mu_L)) = \mu_{L \cup \{k\}}$$

where μ_L is the probability distribution, over 2-3 trees storing the set of leaves L, generated by BUILDTREE(L).

The proof of the above Proposition is based on the following Lemma. Let $\text{INS}_k(F)$ the probability distribution over 2-3 forests resulting from the execution of INS(k, F).

Lemma 1 *For any 2-3 forest F with more than one tree and for any key k,*

$$\text{BUILDLEVEL}(\text{INS}_k(F)) = \text{INS}_k(\text{BUILDLEVEL}(F))$$

is an equality between probability distributions over 2-3 forests.

Proof: (Sketch) Consider an execution of $\textsc{Ins}(k, N)$ with N pointing to the result of running $\textsc{BuildLevel}(F)$. Clearly N does not point to a list of leaves, so Step 1 is skipped. Now observe that the execution of Step 2 only affects the running time of the Algorithm. Therefore, we can assume that Step 2 is not executed and the pointer M is initialized to the first node of F.

So, the call to $\textsc{Ins}(k, M)$ at step 4 generates a 2-3 forest with probability distribution $\textsc{Ins}_k(F)$. The proof proceeds by computational induction, showing that the execution of Steps 5-7 is equivalent to the assignment $N := \textsc{BuildLevel}(M)$. □

The proof of Proposition 1 easily follows from Lemma 1.

3.3 Delete

The delete algorithm is defined along the same lines as INSERT. First a routine to delete a key from a 2-3 forest is defined.

ALGORITHM DEL(k,N):

1. If N points to a leaf node, then delete the new key k in the ordered list pointed to by N and terminate with return value k.
2. Advance the pointer N ($N \leftarrow N.next$) until either $N.next.random$ equals 0 or $N.next.key \geq k$.
3. Initialize a pointer M to the first child of N ($M \leftarrow N.child$).
4. Call recursively $\textsc{Del}(k, M)$ and store the returned value in k'.
5. If $N.next.random = 0$, then run the algorithm BUILDLEVEL on M. Replace the list of nodes pointed to by N with the result returned by $\textsc{BuildLevel}(M)$ and terminate with return value $+\infty$.
6. If the key of M is greater then k' do
   (a) If the first child of N is M terminate with return value $N.key$.
   (b) Toss a coin $d \in_R \{2, 3\}$.
   (c) If the first child of $N.next.next$ is $M[d]$ then set $N.child = M$ and $N.deg = d$. Remove the node $N.next$ and return ($N.key$).
   (d) otherwise, set the degree of N to d and go on.
7. Set $N.child$ to M. Advance M of $N.deg$ positions ($M \leftarrow M[N.deg]$). Advance N of one position ($N \leftarrow N.next$) and go back to step 5.

The $\textsc{Delete}(k, T)$ Algorithm is shown in figure 4. It is assumed that T points to a single tree ($T.next =$ NIL) and the minimum key in T is strictly smaller than k. As before, this last condition is ensured by having a leaf in the tree with key $-\infty$.

The proof of correctness and obliviousness is analogous to that for the INSERT algorithm.

Proposition 2 *For any set of leaves L and for any leaf k, the following equality holds between probability distributions:* $\textsc{Delete}(k, \mu_L)) = \mu_{L \setminus \{k\}}$.

DELETE(k,T):
{ $T.next = NIL$ and $T.key = -\infty$ }
    call DEL(k, T)
    if $(T.child.next =$ NIL$)$ then return $(T.child)$
    otherwise return (T)

Figure 4: The DELETE Algorithm

3.4 Running time Analysis

In this section we prove that the expected running time of INSERT(k, T) is $O(h)$ where h is the height of the tree T. Since the leaves in a 2-3 tree are all at the same level, this implies a $O(\log n)$ bound, where n is the number of leaves of T. The analysis of the DELETE algorithm is analogous and yields similar results.

Consider an execution of Algorithm INSERT(k, T). The running time is proportional to the number of nodes visited. We will give an estimate to this number. The INS(k, N) procedure is called h times, once for each level of the tree T. Each call to INS visits a sequence of contiguous nodes, all at the same level. Let l_i the number of nodes visited at level i and consider the corresponding call to INS(k, N).

It is easily seen that the number of nodes visited during the execution of step 2 (or step 1 if N is a leaf) is at most 4.

After that, M is initialized to the first child of N and INS(k, M) is called. INS(k, M) visits l_{i+1} nodes at level $i + 1$ and returns the key k' of the last visited node. A new node is visited at level i for each iteration of steps 5-7. Notice that at step 7 the pointer M is advanced of at least two positions. So, after at most $(l_{i+1}/2 + 1)$ iterations M points to a node with key greater than k'.

For all subsequent iterations of steps 5-7 the execution of INS(k, N) terminates within two iterations with probability at least 1/4: if $N.child = M$ we stop immediately; if $N.child = M[2]$ or $N.child = M[1]$ and $N.deg = 2$ we stop in one iteration with probability 1/2 (when $d = 2$ and $d = 3$ respectively); finally, if $N.child = M[1]$ and $N.deg = 3$, the sequence of coin tosses $d = 2$, $d = 3$ make us stop in two more iterations with probability 1/4.

Therefore the number of nodes visited at level i can be bounded by

$$l_i \leq 4 + \frac{1}{2} l_{i+1} + 1 + 2X_i$$

where X_i is a random variable with geometric distribution of parameter 1/4. The total running time is given by

$$\text{Time} = \sum_{i=1}^{h} l_i \leq \sum_{i=1}^{h} (5 + \frac{l_{i+1}}{2} + 2X_i)$$

Subtracting Time/2 from both sides and multiplying by 2 we get the upper bound Time $\leq 10h + 4X$, where $X = \sum_{i=1}^{h} X_i$ is the sum of h independent random variables, all with geometric distribution of parameter 1/4. In particular, we have $E[X] = 4h$.

Proposition 3 *The expected running time of algorithm* $\textsc{Insert}(k, T)$ *and algorithm* $\textsc{Delete}(k, T)$ *is* $O(h)$*, where* h *is the height of the tree* T*. Moreover, the probability for the running time to deviate from its expected value by more than* α *is exponentially decreasing both in* α *and in* h*.*

4 Incremental Signatures

In this section we define in more detail the private signature problem and show how our data structure solves it.

Definition 2 *A signature scheme is specified by a triple* $(\mathcal{G}, \mathcal{S}, \mathcal{V})$ *of probabilistic polynomial time algorithms.*

- *Algorithm* $\mathcal{G}$ *is called the key generator.* $\mathcal{G}$ *takes as input a security parameter* 1^s *(i.e.* s *expressed in unary) and outputs a pair* (K_S, K_V) *of keys called the secret key and the verification key.*
- *Algorithm* $\mathcal{S}$ *is called the signature algorithm. It takes as input a secret key* K_S *and a message* m *and outputs a string* $\mathcal{S}(K_S, m)$ *called* digital signature *of* m *under key* K_S*.*
- *Algorithm* $\mathcal{V}$ *is called the verification algorithm. It takes as input the verification key* K_V*, a message* m *and a string* σ*, and tests whether* σ *is a valid signature for message* m *(i.e.* $\mathcal{V}(K_V, m, \sigma) = 1$ *iff* σ *is a possible output of* $\mathcal{S}(K_S, m)$*).*

Let $\mathcal{M}$ be a set of text modification operations (e.g. $\mathcal{M} = \{\texttt{insert}(b, i), \texttt{delete}(i)\}$, 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). If $p_1 \dots p_n$ is a sequence of such operations, $p_1 \dots p_n[D]$ denotes the result of applying the operations $p_1 \dots p_n$ sequentially to the initial document D.

Definition 3 *Let* $(\mathcal{G}, \mathcal{S}, \mathcal{V})$ *be a signature scheme, and let* $\mathcal{M}$ *be a set of text modification operations. An* $\mathcal{M}$-incremental signature system for $(\mathcal{G}, \mathcal{S}, \mathcal{V})$ *is an interactive machine* $\mathcal{I}$ *operating as follows.*

- $\mathcal{I}$ *is initialized with a pair of keys* (K_S, K_V)*, obtained by running* $\mathcal{G}(1^s)$*.*
- *In response to a* $\texttt{create}(D)$ *command, with parameter an initial document* D*,* $\mathcal{I}$ *returns two strings* α *and* σ*.* α *is called* document identifier *and can be used to later refer to the document.* σ *is the* current signature *of document* α *and it can be used to issue edit commands to* $\mathcal{I}$*.*
- *In response to a* $\texttt{edit}(\alpha, \sigma, p)$ *command, with parameters a document identifier* α*, the current signature* σ *of document* α *and a text modification operation* p*,* $\mathcal{I}$ *updates the current signature* σ *to reflect the application of operation* p *and returns the new current signature* σ' *of document* α*.*

Furthermore, for any sequence of operations $p_1 \dots p_n$*, if* $\mathcal{I}$ *receives the sequence of commands* $\texttt{create}(D), \texttt{edit}(\alpha, \sigma_0, p_1), \dots, \texttt{edit}(\alpha, \sigma_{n-1}, p_n)$ *(possibly interspersed with other commands not referring to document* α*) where* (α, σ_0) *is the value returned by* $\mathcal{I}$ *in response to the request* $\texttt{create}(D)$ *and for all* $i = 1, \dots, n$*,* σ_i *is the value returned by* $\mathcal{I}$ *in response to the request* $\texttt{edit}(\alpha, \sigma_{i-1}, p_i)$*, then* σ_n *is a valid signature of the document* $p_1 \dots p_n[D]$*, i.e.* $\mathcal{V}(K_V, p_1 \dots p_n[D], \sigma_n) = 1$*.*

In practice, the signature σ is not passed to and returned from the commands issued to $\mathcal{I}$. Rather, σ resides in some form of memory support and is modified in place by $\mathcal{I}$. We made σ an explicit parameter to the commands to emphasize that σ resides externally to $\mathcal{I}$ and a malicious user could alter the incrementable signature σ before issuing a command to $\mathcal{I}$ in the attempt of breaking the system.

We consider a user A interacting with $\mathcal{I}$ to have requested a signature of document D iff A issued to $\mathcal{I}$ a sequence of commands $\mathtt{create}(D_0)$, $\mathtt{edit}(\alpha, \sigma_0, p_1)$, $\ldots$, $\mathtt{edit}(\alpha, \sigma_{n-1}, p_n)$, (possibly interspersed with other commands not referring to document α) such that α is the document identifier returned by $\mathtt{create}(D_0)$ and $D = p_1 \ldots p_n[D_0]$.

We say that A produces a forgery iff A, after interacting with $\mathcal{I}$, outputs a valid signature for a document D whose signature has not been requested by A during the interaction with $\mathcal{I}$.

The definition of tamper proof security follows.

Definition 4 *An incremental signature system $\mathcal{I}$ is* tamper proof *secure iff for any probabilistic polynomial time algorithm A which may interact with $\mathcal{I}$, the probability that A produce a forgery is negligible with respect to 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 coin tosses of algorithm A and those of the system $\mathcal{I}$ (which include the coin tosses used by the key generator $\mathcal{G}$ to produce the initialization keys (K_S, K_V)).*

Definition 5 *An incremental signature system $\mathcal{I}$ is* private *iff for all possible pairs of keys (K_S, K_V) obtained by running $\mathcal{G}(1^s)$, for any initial document D and for any sequence of text modification operations $p_1, \ldots, p_n$ the following is true.*

If $\mathcal{I}$ is initialized with the keys (K_S, K_V), the probability distribution on signatures σ obtained by issuing the sequence of commands $\mathtt{create}(D)$ (with answer (α, σ)), $\mathtt{edit}(\alpha, p_1, \sigma_0)$ (with answer σ_1), $\ldots$, $\mathtt{edit}(\alpha, p_n, \sigma_{n-1})$ (with answer σ) (possibly interspersed with other commands not referring to document α), is identical to the probability distribution defined by $\mathcal{S}(K_S, p_1 \ldots p_n[D])$, i.e. running the signature algorithm directly on the final document.

Slightly different, but equivalent, definitions are given in [1] where it is also defined an incremental signature system, called the *tree scheme*, that uses 2-3 tree to implement all edit operations in logarithmic time. The tree scheme is built on top of a standard (non-incremental) signature scheme (G, S, V) and achieves tamper proof security under the assumption that (G, S, V) is secure under chosen message attack.

We now show how to define a similar system using oblivious 2-3 trees, meeting the additional requirement of privacy of signatures. Our definition is essentially the same as in [1], with ordinary 2-3 trees replaced by oblivious ones.

Let (G, S, V) be an ordinary signature scheme. We define a new signature scheme $(\mathcal{G}, \mathcal{S}, \mathcal{V})$ on top of (G, S, V). The key generator $\mathcal{G}$ is G itself. The algorithms $\mathcal{S}$ and $\mathcal{V}$ use S and V as subroutines with the keys generated by $\mathcal{G}$, and are defined as follows.

Algorithm $\mathcal{S}$ on input key K_S and document D, produces a 2-3 tree. Each node n of the tree contains an authentication tag $n.tag$ and an integer $n.size$ storing the number of leaves in the subtree rooted at n. (To avoid ambiguities, we will use the term "tag-tree" to refer to the signatures produced by $\mathcal{S}$, while the term "signature" will always refer to an ordinary signature produced by S.) The leaves of the tag-tree produced by $\mathcal{S}$ correspond to the blocks of document D. The field $n.size$ equals one if n is a leaf, otherwise it is computed as the sum of the sizes of the children of n ($n.size = \sum_{i=0}^{n.deg-1} n.child[i].size$). The authentication tag is computed as follows. If n is the ith leaf of the tree, then $n.tag = S(K_S, D[i])$ where $D[i]$ is the ith block of the document. If n is an internal node then $n.tag = S(K_S, (n.child[0], \ldots, n.child[n.deg-1], n.size))$. If n is the root, then $n.tag = S(K_S, (n.child[0], \ldots, n.child[n.deg-1], n.size, \texttt{root}))$, where `root` is a special symbol used to distinguish the tag-tree of a whole document from a subtree associated to part of a document. The topology of the tree is defined using the procedure BuildTree defined in section 3.

The verification algorithm $\mathcal{V}$ works in the obvious way. It takes as input key K_V, document D and a tag-tree σ, and uses V to check that all tags of the nodes in σ are valid signatures of the appropriate strings, as defined by $\mathcal{S}$.

We can now define an incremental signature system $\mathcal{I}$ for $(\mathcal{G}, \mathcal{S}, \mathcal{V})$. The system $\mathcal{I}$ is initialized with a pair of keys (K_S, K_V) obtained by running $\mathcal{G}$, and operates as follows.

- In response to a $\texttt{create}(D)$ command, $\mathcal{I}$ generates a fresh document identifier α, associates with it an internal register R_α, computes the tag-tree $\sigma = \mathcal{S}(K_S, D)$, initializes R_α to the contents of the *tag* field of the root of σ, and returns the pair (α, σ).
- In response to an $\texttt{edit}(\alpha, \sigma, \texttt{insert}(b, i))$ command, $\mathcal{I}$ checks that the value in the register R_α is equal to the *tag* field of the root of σ. If so, $\mathcal{I}$ inserts a leaf with tag equal to $S(K_S, b)$ in the tag-tree σ at position i using the oblivious 2-3 tree insertion algorithm modified as follows. The fields *size* of the nodes are used to locate where the new leaf must be inserted. Each time a new node n is accessed, a partial validity check is performed. The validity of node n is checked by running the verification algorithm V with parameters K_V, $n.tag$ and the appropriate string as defined by $\mathcal{S}$. The field $n.size$ is also checked to be equal to $\sum_{i=0}^{n.deg-1} n.child[i].size$. Any time a node is modified, the fields *size* and *tag* are recomputed.

  Then, the register R_α is updated to contain the new tag of the root of σ.
- $\texttt{edit}(\alpha, \sigma, \texttt{delete}(i))$ commands are treated analogously.

The above system 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 $\mathcal{I}$ described above is tamper proof secure.*

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

Theorem 2 *All* `edit` *operations are performed by $\mathcal{I}$ in time logarithmic in the length of the document being signed.*

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 3. □

Theorem 3 *The incremental signature system $\mathcal{I}$ achieves privacy.*

Proof: It follows immediately from the obliviousness of the tree operations used and from the fact that all calls to algorithm S are made with independent coin tosses. $\square$

5 Discussion

We have defined efficient algorithms to insert and delete nodes in 2-3 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 over 2-3 trees. We call the resulting data structure oblivious 2-3 tree (supporting insertion and deletion operations).

An efficient incremental digital signature system is defined based on oblivious 2-3 tree. The incremental signature system achieves tamper proof security and privacy, thus solving an open problem raised in in [1].

Oblivious algorithms for other tree operations, such as *split* and *merge* of 2-3 trees, can be defined following essentially the same ideas used in the definition of oblivious insert and delete. An incremental signature system which support *cut* and *paste* text modification operations can be easily defined using oblivious split and merge of 2-3 trees, essentially in the same way we did here for insert and delete operations.

It is clear that the definition of obliviousness for 2-3 tree, can be generalized to arbitrary data structures.

Definition 6 *Consider two data structures $(\mathcal{A}, \Sigma_{\mathcal{A}})$ and $(\mathcal{B}, \Sigma_{\mathcal{B}})$ implementing the same set of operations Σ. The operations $f_{\mathcal{A}}$ in $\Sigma_{\mathcal{A}}$ are deterministic algorithms. The operations $f_{\mathcal{B}}$ in $\Sigma_{\mathcal{B}}$ are probabilistic algorithms.*

Let ϕ be a function from $\mathcal{B}$ to $\mathcal{A}$ such that for all operation $f \in \Sigma$ of arity n, and for any n-tuple $\mathbf{B} \in \mathcal{B}^n$, we have $\phi(f_{\mathcal{B}}(\mathbf{B})) = f_{\mathcal{A}}(\phi(\mathbf{B}))$, where $f_{\mathcal{B}}(\mathbf{B})$ denotes any possible output of $f_{\mathcal{B}}$ on input $\mathbf{B}$.

We say that $(\mathcal{B}, \Sigma_{\mathcal{B}})$ is an oblivious implementation of $(\mathcal{A}, \Sigma_{\mathcal{A}})$ with respect to ψ, if ψ is a probabilistic algorithm such that for all $a \in \mathcal{A}$, $\phi(\psi(a)) = \{a\}$, and for all operation $f \in \Sigma$ of arity n, and for any n-tuple $\mathbf{A} \in \mathcal{A}^n$, $\psi(f_{\mathcal{A}}(\mathbf{A}))$ and $f_{\mathcal{B}}(\psi(\mathbf{A}))$ define identical probability distributions on $\phi^{-1}(f_{\mathcal{A}}(\mathbf{A}))$.

For example oblivious 2-3 trees are an oblivious implementation of the associated sets of leaves, where the probabilistic function ψ is given by BuildTree.

We believe that the applicability of the notion of oblivious data structure extends far beyond the particular problem solved here (privacy of incrementally generated digital signatures), in particular to the area of cryptography and cryptographic protocols.

6 Acknowledgements

The author is grateful to Shafi Goldwasser for suggesting the problem and for her help and encouragement to write this paper. Thanks also to Oded Goldreich for useful discussions.

References

[1] M. Bellare, O. Goldreich, 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, 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, CRYPTO 94.

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

[4] C. R. Aragon, 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, J. Ullman *The design and analysis of computer algorithms.* Addison Wesley, 1974.