Time and Space Efficient Algorithms for Two-Party Authenticated Data Structures*

Charalampos Papamanthou and Roberto Tamassia

Department of Computer Science, Brown University

Abstract. Authentication is increasingly relevant to data management. Data is being outsourced to untrusted servers and clients want to securely update and query their data. For example, in database outsourcing, a client's database is stored and maintained by an untrusted server. Also, in simple storage systems, clients can store very large amounts of data but at the same time, they want to assure their integrity when they retrieve them. In this paper, we present a model and protocol for two-party authentication of data structures. Namely, a client outsources its data structure and verifies that the answers to the queries have not been tampered with. We provide efficient algorithms to securely outsource a skip list with logarithmic time overhead at the server and client and logarithmic communication cost, thus providing an efficient authentication primitive for outsourced data, both structured (e.g., relational databases) and semi-structured (e.g., XML documents). In our technique, the client stores only a constant amount of space, which is optimal. Our two-party authentication framework can be deployed on top of existing storage applications, thus providing an efficient authentication service. Finally, we present experimental results that demonstrate the practical efficiency and scalability of our scheme.

1 Introduction

Data authentication has lately been very important due to the expansion of the Internet and the continuing use of it in daily transactions. Data authentication provides assurance for integrity of data, namely that data have not been corrupted (for example modified or deleted) by an adversary. Imagine for example the following scenario. There a lot of internet companies that provide cheap storage space. Clients that use this service are assigned a special account which they can use to store, query and update their data. They basically *outsource* their data in the storage servers provided by the companies. This is an amazing service but raises the following security issue. How can the client be assured that the data it is putting in the storage servers have not been tampered with? Each time the client issues a query, it would like to be assured that the data it receives is consistent with the previous state and nobody has changed something. Hence the server (the entity where data have been outsourced) and the client

* This work was supported in part by IAM Technology, Inc. and by the National Science Foundation under grants IIS–0713403 and OCI–0724806.

S. Qing, H. Imai, and G. Wang (Eds.): ICICS 2007, LNCS 4861, pp. 1–15, 2007.
© Springer-Verlag Berlin Heidelberg 2007

(the entity that outsources the data) have to engage in an efficient two-party authentication protocol during which the client can send updates and queries to the server and be able to verify the answers it is receiving. Ideally, we would like this protocol not only to be secure (the security definition will be given later) but also to involve low computational overhead at the client's side, low communication cost between the server and the client and possible low computational overhead at the server's side.

One application of this model is database outsourcing [9]. In database outsourcing, a third party database service provider offers software, hardware and network resources to host its clients' data. Since the clients outsource their databases in an environment which is susceptible to attacks, the security of the hosted data is a big issue. These attacks can be caused by malicious outsiders or by the service provider itself. In any case, the client should be able to verify the *integrity* of its data. Yet another important problem in database outsourcing, orthogonal to what we investigate in this paper (data integrity), is ensuring data secrecy and privacy [8, 10]. Another application of this model is converting the widely known authenticated data structures model [12, 19], which is a three-party model, to a two-party model, where we want to maintain the efficiency of the corresponding authenticated data structure and have the client execute both updates and queries.

In this paper, we develop and analyze time- and space-efficient algorithms for outsourcing an authenticated skip list [5]. We aim at providing integrity checking of answers received from the data structure (privacy is orthogonal to our approach). Our technique is further applicable to other hash-based authenticated data structures [1, 7, 13] (for example, Merkle trees or dynamic trees), given that we can develop the respective algorithms for the operations defined on the specific data structure. We present algorithms for outsourcing a skip list [18] that run (both on the client and on the server side) in logarithmic time (in the size of the outsourced data) and incur logarithmic communication cost. Our method requires constant space (a single hash value) at the client side, which is optimal.

1.1 Related Work

There is considerable previous work on authenticated data structures in the three-party model, where a data owner outsources the data to a server, which answers queries issued by clients on behalf of the data owner. See [19] for a survey. In particular, the authentication of outsourced databases in the three-party model, using an authenticated B-tree for the indices, is presented in [11]. Client storage bounds in the three-party model are discussed in [21].

In [4], data integrity in the two-party model and solutions for multi-authored dictionaries are explored, where users can efficiently validate a sequence of updates. The authentication of outsourced databases using signature schemes appears in [15, 16], where it is mentioned that this approach is inefficient due to the high cost of the client's computation and the fact that the client has to engage in multi-round protocol in order the perform an update. In [2], a method for

outsourcing a dictionary is presented, where a skip list is stored by the server into a table of a relational database management system (DBMS) and the client issues SQL queries to the DBMS to retrieve authentication information. A related solution is presented in [14]. This DBMS-based approach relies on external-memory storage of authentication information. Thus, it scales to large data sets but incurs a significant computational overhead.

1.2 Our Contributions

In this paper we provide a model and protocol for two-party authentication of data structures. Namely, a client outsources its data structure and verifies that the answers to queries are valid. Our framework is fairly general and can be extended to outsourcing a variety of authenticated data structures [1, 7, 13]. We focus on efficient algorithms to outsource a dictionary by using a skip list, which is a well known efficient randomized realization of a dictionary. Our authentication protocol is simple, and efficient. It is based on cryptographic hashing and requires the client to store only a single hash value. The computational overhead for the server and the client and the communication cost are logarithmic, which is optimal for hash-based authentication structures. We have fully implemented our scheme and we provide experimental results that confirm its efficiency and scalability in practice. Our protocol can be deployed on top of existing storage applications to provide a transparent authentication service.

1.3 Preliminaries

We briefly introduce some useful terminology for the authenticated skip list (the non-authenticated skip list was proposed by Pugh [18] and provides a dictionary functionality), since it will be the underlying data structure for our authentication protocol. In an authenticated skip list, every node of the skip list is associated with a hash value that is computed as a function of neighboring nodes according to a special DAG scheme [5]. The importance of the authenticated skip list comes in the verification of the result. When the client queries about the membership of an element x in the dictionary, the server returns a collection of hash values that allows the client to verify the result. For more details on authenticated data structures see [1, 5, 12].

2 Two-Party Authentication Model

In the two-party authentication model, there are two entities participating, an untrusted server $\mathcal{S}$ and a client $\mathcal{C}$. The server stores a data collection, which is owned (and has been outsourced) by the client $\mathcal{C}$, in a data structure (for example a dictionary) and the client issues updates (insertions, deletions) to its data. Also the client can issue queries (for example membership queries in a dictionary, connectivity queries in a graph) to the data structure. Since the server is untrusted, we want to make sure that if the server returns a wrong

answer to a client's query (namely if the server tampers with the data of the client), then the client rejects the answer w.h.p.

An obvious solution to this, which has been adopted in practice (see for example [17]), is to have the client store and update hash values of the data objects it wants to outsource. In this way the client can always tell if the answers that it gets back from the untrusted server are correct or not. However, this solution is very space inefficient. Ideally, we would like the client to hold a constant size state (digest of the data) and to execute very simple algorithms in order to update this state, whenever it issues an update. We will assume that initially the server and the client share the same digest that correspond to the data. This digest (which later we will be calling s) can be computed and updated using any existing hashing scheme [21]. Then we are going to present protocols and algorithms that securely update this digest whenever updates occur, for the case of the skip list data structure.

2.1 The Protocol and Its Security

In the following we describe the protocol that ensures authentication of operations (insertions, deletions, queries) issued by the client. Then we give the definition of security and also prove that it is satisfied by the presented protocol. Before presenting the protocol, we give some necessary definitions:

Definition 1 (Sequential Hashing). *Given an ordered sequence of hash values $\Lambda = \lambda_1 \circ \lambda_2 \circ \ldots \circ \lambda_m$ and a collision-resistant hash function $h(.,.)$, then the sequential hashing $S(\Lambda)$ maps Λ to a hash value such that $S(\lambda_1 \circ \lambda_2 \circ \ldots \circ \lambda_m) = h(S(\lambda_1 \circ \lambda_2 \circ \ldots \circ \lambda_{m-1}) \circ \lambda_m)$ for $m \geq 2$. For $m = 1$, we define $S(\lambda_1) = \lambda_1$.*

The complexity of sequential cryptographic hashing $S(\Lambda)$ is a function of its input size, namely a function of $|\Lambda|$ (if we assume that λ_i is of fixed length, then $|\Lambda| = O(m)$). Moreover, it can be proved [20] that the time needed to perform a sequential hashing of Λ is $O(|\Lambda|)$. This means that the hashing complexity is proportional to the size of the data being hashed. Based now on the definition of collision resistance of the function h, we can prove the following:

Lemma 1. *Given an ordered sequence of hash values $\Lambda = \lambda_1 \circ \lambda_2 \circ \ldots \circ \lambda_m$, there is no probabilistic polynomial-time adversary than can compute another sequence of hash values $\Lambda' = \lambda'_1 \circ \lambda'_2 \circ \ldots \circ \lambda'_m$ such that $S(\Lambda) = S(\Lambda')$ with more than negligible probability.*

In the following we present one execution of the authentication protocol (for either an update or a query), which we call Auth2Party. The protocol uses three main procedures, namely certify, verify and update:

- Let s be the state (digest) that corresponds to the current data at the server's side. We assume that the client stores s (for the case of the skip list the digest is the hash of the top-left node with key $-\infty$).

- The client issues an operation $o \in \{I(x), D(x), Q(x)\}$ with respect to an element x that can be either an insertion (I), a deletion (D) or a query (Q).
- The server runs an algorithm $(\pi, a(o)) \leftarrow \mathsf{certify}(o)$ that returns a proof π and an answer (with reference to the specific operation o) $a(o)$. Then it sends the proof and the answer to the client. If the operation is an update, the server executes afterward the update in its local data and $a(o) = \mathsf{null}$.
- The client runs an algorithm $(\{0,1\}, s') \leftarrow \mathsf{verify}(o, a(o), \pi, s)$ and we have the following cases:
  1. If $o \in \{I(x), D(x)\}$ (if the operation issued by the client is either an insertion or a deletion) then we distinguish the following cases for the output of verify:
     a. If the sequential hashing (which is the main body of verify)[1] of the proof π hashes to s, i.e., if $S(\pi) = s$, then the output is $(1, s')$. In this case the client accepts the update, it runs an algorithm update on input π and computes the new digest s'.
     b. If the sequential hashing of the proof π does not hash to s, i.e., if $S(\pi) \neq s$, then the output is $(0, \perp)$. In this case the client rejects and the protocol terminates.
  2. If $o = Q(x)$ (if the operation issued by the client is a query) then if the sequential hashing of the proof π hashes to s, i.e., if $S(\pi) = s$, then verify outputs $(1, \perp)$ and the client accepts. Otherwise it outputs $(0, \perp)$ and the client rejects. Note that in this case the state of the client is maintained, since there are no structural changes in the data.

To distinguish between the proof returned when the client issues a query or when the client issues an update, we define as *verification* proof to be the proof returned to a query and as *consistency* proof to be the proof returned to an update. In the following we present the security definition for our protocol.

Definition 2 (Security of Two Party Authentication Protocol). *Suppose we have a two-party authentication protocol $\mathcal{D}$ with server $\mathcal{S}$, client $\mathcal{C}$ and security parameter k. We say that $\mathcal{D}$ is secure if no probabilistic polynomial-time adversary A with oracle access to algorithm* certify*, given any query $Q(x)$, can output an answer $a'(Q(x))$ and a verification proof π', such that $a'(Q(x))$ is an incorrect answer that passes the verification test. That is, there exists negligible*[2] *function $\nu(k)$, such that for every probabilistic polynomial-time adversary A and every query $Q(x)$*

$$\Pr\left[(\pi', a'(Q(x))) \leftarrow A(Q(x)) \wedge (1, s') \leftarrow \mathsf{verify}(Q(x), a'(Q(x)), \pi', s)\right] = \nu(k).$$

[1] In [5], it is described how one can use sequential hashing on a carefully constructed proof in order to compute the digest of the skip list. We use this scheme without loss of generality for the description of the protocol but we note that the computation of the digest depends on the data structure and the algorithms used.

[2] Formally, $\nu : N \rightarrow \Re$ is negligible if for any nonzero polynomial p, there exists m such that $\forall n > m \ |\nu(n)| < \frac{1}{p(n)}$.

We can now state the result for the security of our scheme (its proof is deferred to the full version of the paper):

Theorem 1. *If algorithm* update *correctly computes the new digest (i.e., the new digest is consistent with the issued update) and algorithm* verify *is correct, then protocol* Auth2Party *is secure.*

3 Operations

In this section, we describe our authentication protocol for a skip list. We describe how to use the proof returned by a contains() query in order to compute the digest of the skip list resulting from the insertion of a new element x. For more details on authenticated skip lists, see Section 1.3.

3.1 Search Path and Proof Path

For every node v of a skip list (if $\mathsf{level}(v) > 0$ we consider as node every node that "makes" a difference in hashing, i.e., nodes with both right and down pointers not null) we denote with $\mathsf{key}(v)$ the key of the tower that this node belongs to, with $\mathsf{level}(v)$ the respective level of this node, with $f(v)$ the hash of the node of the skip list and with $\mathsf{right}(v)$ and $\mathsf{down}(v)$ the right and down pointers of the v. We finally write $u \leftarrow v$[3] if there is an actual link in the skip list from v to u (either a right or a down pointer). We also write $u \hookleftarrow v$ if there is no actual link between u and v but $\mathsf{level}(v) = \mathsf{level}(u) = 0$ and u is a successor of v (this means that u belongs to a tower of height > 0).

Definition 3 (Search Path). *Given a skip list SL and an element x, then the search path $\Pi(x) = v_1 \hookleftarrow v_2 \leftarrow \ldots \leftarrow v_{j-1} \leftarrow v_j \leftarrow v_{j+1} \leftarrow \ldots \leftarrow v_m$ is an ordered sequence of nodes in SL satisfying the following properties:*

- *v_m is the top-leftmost node of the skip list.*
- *$v_2 \leftarrow v_3 \leftarrow \ldots \leftarrow v_m$ is a path in the skip list consisting of the zero-level path $v_2 \leftarrow v_3 \leftarrow \ldots \leftarrow v_j$ and the path $v_{j+1} \leftarrow v_{j+2} \leftarrow \ldots \leftarrow v_m$ which contains nodes of level > 0.*
- *There is $2 \leq t \leq j$ such that $\mathsf{key}(v_t) < \mathsf{key}(x) \leq \mathsf{key}(v_{t-1})$.*
- *Index j is defined as the "boundary" index.*
- *Index t is defined as the "insertion" index.*

Definition 4 (Proof Path). *Given an authenticated skip list SL, an element x and the respective search path $\Pi(x) = v_1 \hookleftarrow v_2 \leftarrow \ldots \leftarrow v_m$ with boundary index j and insertion index t, we define the proof path $\Lambda(x) = \lambda(v_1) \circ \lambda(v_2) \circ \ldots \circ \lambda(v_m)$ to be the following ordered sequence of values with the following properties:*

- *For all $i \leq j$ $\lambda(v_i) = \mathsf{key}(v_i)$.*
- *For all $i > j$, $\lambda(v_i) = f(\mathsf{right}(v_i))$ if $\mathsf{right}(v_i) \neq v_{i-1}$ else $\lambda(v_i) = f(\mathsf{down}(v_i))$.*

[3] Although it is more intuitive to write $v \rightarrow u$, we use this notation because the hashing proceeds from the last to the first element of the proof. This means that there are no actual pointers from v to u in the skip list and this notation is only used for indicating the hashing procedure.

Lemma 2. *Given an authenticated skip list SL and an element x ($x \in SL$ or $x \notin SL$), then there exist unique search and proof paths $\Pi(x)$ and $\Lambda(x)$ for x.*

Proof. By contradiction. Suppose there is another search or proof path for x. Then there would be a cycle of links in SL. This is a contradiction since pointers in SL define a tree structure.

Definition 5 (Node Removal). *Given a search path $\Pi(x) = v_1 \hookleftarrow v_2 \leftarrow \ldots \leftarrow v_m$ and the respective proof path $\Lambda(x) = \lambda(v_1) \circ \lambda(v_2) \circ \ldots \circ \lambda(v_m)$, then, for every $1 < i \leq m$, we denote with $\mathsf{rem}(\Pi(x), v_i)$ and $\mathsf{rem}(\Lambda(x), \lambda(v_i))$ the search path $v_1 \hookleftarrow v_2 \leftarrow \ldots v_{i-1} \leftarrow v_{i+1} \leftarrow \ldots \leftarrow v_m$ and the proof path $\lambda(v_1) \circ \lambda(v_2) \circ \ldots \circ \lambda(v_{i-1}) \circ \lambda(v_{i+1}) \circ \ldots \circ \lambda(v_m)$ respectively.*

Theorem 2. *Given an authenticated skip list SL and an element x, then $S(\Lambda(x))$ (the sequential hashing of the proof path of x) is equal to the digest of SL.*

Suppose now we are given a search path $\Pi(x)$. For each element v_i of the search path we define $D_x(v_i)$ to be 1, if $\mathsf{key}(v_i) \geq x$, else 0. Also, we define $L_x(v_i)$ to be 0, if $i \leq j$ (j is the boundary index) else it is defined as the level of v_i. An example of an authenticated skip list implemented with pointers is shown in Figure 1. In the following, we describe algorithms for the computation of the new digest from the client side after operations $\mathsf{insert}(x)$ and $\mathsf{delete}(x)$. From the definition now of the search path we have:

Lemma 3. *Every search path $\Pi(x)$ is sorted in increasing L_x order. Moreover, any elements u and v such that $L_x(v) = L_x(u)$ are sorted in decreasing* key *order.*

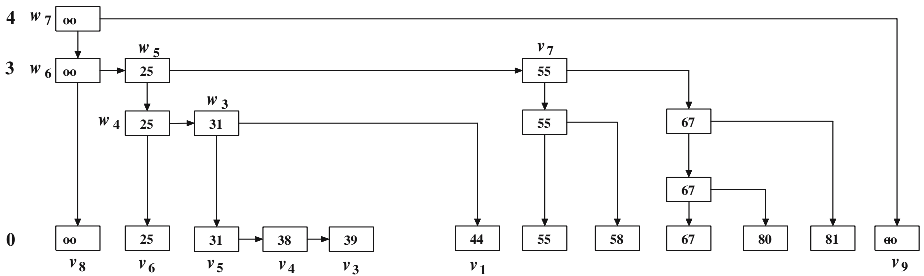


Fig. 1. An authenticated skip list implemented with pointers. For the non-existing element 41 (and also for the existing element 44), we have that the search paths are $\Pi(41) = \Pi(44) = v_1 \hookleftarrow v_3 \leftarrow v_4 \leftarrow v_5 \leftarrow w_3 \leftarrow w_4 \leftarrow w_5 \leftarrow w_6 \leftarrow w_7$ while the respective proof paths are $\Lambda(41) = 44 \circ 39 \circ 38 \circ 31 \circ f(v_1) \circ f(v_6) \circ f(v_7) \circ f(v_8) \circ f(v_9)$. Also note that $L_{41} = (0 \circ 0 \circ 0 \circ 0 \circ 2 \circ 2 \circ 3 \circ 3 \circ 4)$ and $D_{41} = (1 \circ 0 \circ 0 \circ 0 \circ 1 \circ 0 \circ 1 \circ 0 \circ 1)$. Note that the insertion index is 2 while the boundary index is 4.

3.2 Insertion and Deletion

Suppose now that $x \notin S$. The client wants to insert x in the skip list at level ℓ. The client gets the proof path $\Lambda(x)$ (non-membership proof) together with the vectors L_x and D_x. We show how it can compute the new proof path (the proof returned to a $\mathsf{contains}()$ query after the insertion of x). Suppose

$$\Pi(x) = v_1 \hookleftarrow v_2 \leftarrow \ldots \leftarrow v_{j-1} \leftarrow v_j \leftarrow v_{j+1} \leftarrow \ldots \leftarrow v_m$$

where j is the boundary index. Also

$$\Lambda(x) = \lambda(v_1) \circ \lambda(v_2) \circ \ldots \circ \lambda(v_{j-1}) \circ \lambda(v_j) \circ \lambda(v_{j+1}) \circ \ldots \circ \lambda(v_m)$$

is the initial proof path as defined in 4. Let $2 \leq t \leq j$ be the insertion index.

In the following we give an algorithm for the computation of the new search path $\Pi'(x)$ and the new proof path $\Lambda'(x)$ on input $\Pi(x)$, $\Lambda(x)$, D_x, L_x (and t). We have the following cases for $\mathsf{insert}(x, \ell)$.

1. if $\ell = 0$ we *output* the final paths

$$\Pi'(x) = v_1 \hookleftarrow \ldots \leftarrow v_{t-1} \leftarrow x \leftarrow v_t \leftarrow \ldots \leftarrow v_m \quad (1)$$
$$\Lambda'(x) = \lambda(v_1) \circ \ldots \circ \lambda(v_{t-1}) \circ \lambda(x) \circ \lambda(v_t) \circ \ldots \circ \lambda(v_m) \quad (2)$$

2. if $\ell > 0$ we set

$$\Pi(x) = x \hookleftarrow v_t \leftarrow \ldots \leftarrow v_{j-1} \leftarrow v_j \leftarrow v_{j+1} \leftarrow \ldots \leftarrow v_m \quad (3)$$
$$\Lambda(x) = \lambda(x) \circ \lambda(v_t) \circ \ldots \circ \lambda(v_{j-1}) \circ \lambda(v_j) \circ \lambda(v_{j+1}) \circ \ldots \lambda(v_m) \quad (4)$$
$$\mathsf{temp} = S(\lambda(v_1) \circ \lambda(v_2) \circ \ldots \circ \lambda(v_{t-1}) \circ \lambda(x)) \quad (5)$$

Then we sequentially process the hash values $\lambda(v_i)$ for $i = j+1, \ldots, m$ of the proof path and at each iteration we distinguish the following cases:
   - If $\lambda(v_i)$ is such that $\ell < L_x(v_i)$ or ($\ell = L_x(v_i)$ and $D_x(v_i) = 0$) then we create a new node r at level ℓ and we *output* the final paths (nodes v_{i-1} and v_i are linked through the newly created node r in the search path)

$$\Pi'(x) = x \hookleftarrow v_t \leftarrow \ldots \leftarrow v_j \leftarrow \ldots \leftarrow v_{i-1} \leftarrow r \leftarrow v_i \leftarrow \ldots \leftarrow v_m \quad (6)$$
$$\Lambda'(x) = \lambda(x) \circ \lambda(v_t) \circ \ldots \circ \lambda(v_j) \circ \ldots \circ \lambda(v_{i-1}) \circ \mathsf{temp} \circ \lambda(v_i) \circ \ldots \circ \lambda(v_m) \quad (7)$$

   - If $\lambda(v_i)$ is such that $\ell \geq L_x(v_i)$ and $D_x(v_i) = 1$ then we set

$$\Pi(x) = \mathsf{rem}(\Pi(x), v_i) \quad (8)$$
$$\Lambda(x) = \mathsf{rem}(\Lambda(x), \lambda(v_i)) \quad (9)$$
$$\mathsf{temp} = h(\mathsf{temp} \circ \lambda(v_i)) \quad (10)$$

Definition 6. *Let x be any element in a skip list at height ℓ. We define as $\mathsf{guard}(x)$ the first tower on the left of x of height $\geq \ell$.*

We can now have the following main results (their proofs are deferred to the full version of the paper):

Lemma 4. *Let x be an element in a skip list at height ℓ. Then there is always a node $s(x)$ called spy of x that belongs to $\mathsf{guard}(x)$ such that $\mathsf{right}(s(x))$ points to a node of the tower defined by x. Moreover, $s(x)$ always belongs to $\Pi(x)$.*

Lemma 5. *The sequence $\Pi'(x)$ is the correct search path after the insertion of element x at level ℓ and it can be computed in expected $O(\log n)$ time w.h.p..*

Lemma 6. *The sequence $\Lambda'(x)$ is the correct proof path after the insertion of element x at level ℓ and it can be computed in expected $O(\log n)$ time w.h.p., incurring expected $O(\log n)$ hashing complexity w.h.p..*

As we are going to show later, the proof path $\Lambda(x)$ should be extended to include information about the levels of the elements of the proof so that we could ensure security. Consider for example the following scenario. Suppose, the server in the beginning contains only the sentinel values $+\infty$ and $-\infty$. The client issues the command $\mathsf{insert}(x, \ell)$. Suppose the server chooses another level ℓ' and inserts x at ℓ'. Obviously, the digest of the data structure will be exactly the same since the level of the element does not matter at all. However, the efficiency of the data structure is not assured, since the level was not chosen by flipping a coin. Therefore, the server could insert elements at arbitrary levels while the client would not notice anything.

Definition 7 (Extended Proof Path). *Let $\Lambda(x)$ be a proof path with respect to an element x. Let L_x and D_x be the vectors as defined before. We define as extended proof path $Q(x)$ the ordered sequence of triples $Q_i = (\lambda(v_i), L_x(v_i), D_x(v_i))$ for $i = 1, \dots, |\Lambda(x)|$.*

The client, however, does not need to compute the new proof and search paths. All it needs to do is to update the digest. In the following we give a very simple algorithm that updates the digest. Algorithm update, shown in Figure 2, takes as input the *extended* proof path $Q = Q(x)$ as defined in 7 (in the case that the client wants to insert x in the data structure), the desired level of insertion ℓ and the element x the client wants to insert. Note that the levels of the nodes are also taken into consideration in the hash computations, since we use the extended proof path. In the following we outline the algorithm used for verification at the client's side, $\mathsf{verify}(Q(x))$: On input $Q(x)$, it sequentially hashes $Q(x)$ to see if the computed digest matches the existing one. If not, it rejects, else it replaces the current digest by the one returned by calling update (if the operation is an update) and finally accepts. We now reduce an authenticated deletion to an authenticated insertion as follows. Suppose at some point the client and the server share the common digest s. The client then executes $\mathsf{delete}(x)$, where element x is at level ℓ in the skip list. The server deletes x and **then** constructs a proof π' by issuing a $\mathsf{contains}(x)$ query. Next, the server sends π' and the level ℓ to the client. The client runs the update algorithm on input π', x, ℓ. If the output digest is s, then the deletion is accepted and the new digest is $s' = S(\pi')$.

3.3 Analysis

In the data authentication model through hashing, where a three-party model is used, any hashing scheme with k digest nodes that implements an authenticated dictionary of size n has $\Omega(\log(\frac{n}{k}))$ update, verification and communication cost

Algorithm $\mathsf{update}(Q, \ell, x, t)$

```
1: if ℓ == 0
2:   return S(Q_1 ∘ ... ∘ Q_{t−1} ∘ (key(x), 0, 1) ∘ Q_t ∘ ... ∘ Q_m);
3: else
4:   let r be the smallest index ≥ t such that ℓ < L_x(v_r) or ℓ = L_x(v_r) and D_x(v_r) = 0;
5:   if ℓ == L_x(v_r)
6:     set U = Q_{r+1} ∘ ... ∘ Q_m;
7:   else
8:     set U = Q_r ∘ ... ∘ Q_m;
9:   L = (key(x), ℓ, 1);
10:  R = S(Q_1 ∘ ... ∘ Q_{t−1}) ∘ (key(x), ℓ, 1);
11:  for i = t, ..., r
12:    if ℓ ≥ L_x(v_i) and D_x(v_i) = 1
13:      R = R ∘ Q_i;
14:    else
15:      L = L ∘ Q_i;
16:  return S(L ∘ S(R) ∘ U);
```

Fig. 2. Algorithm executed by the client to update the digest after the insertion of an element x. The inputs to the algorithm are the extended proof path Q, the insertion level ℓ, the inserted element x, and the insertion index t. Variable m denotes the length of the extended proof sequence Q.

(see [21]). Using this result, we can prove that our protocol is optimal. To see this, suppose there exists a two-party authentication scheme that uses hashing and achieves better performance than $O(\log n)$ (update, verification, communication). Then we can use these algorithms and implement a three party protocol in the data authentication model through hashing with the same bounds, violating the existing lower bounds [21]. In addition, by using Theorem 1, the proof of correctness of the insertion algorithm and the results on the complexity of authenticated skip list operations [5], we obtain the main result of our paper:

Theorem 3. *Assume the existence of a collision-resistant hash function. Our two-party authentication protocol for outsourcing a dictionary of n elements supports authenticated updates* insert() *and* delete() *and authenticated query* contains() *and has the following properties:*

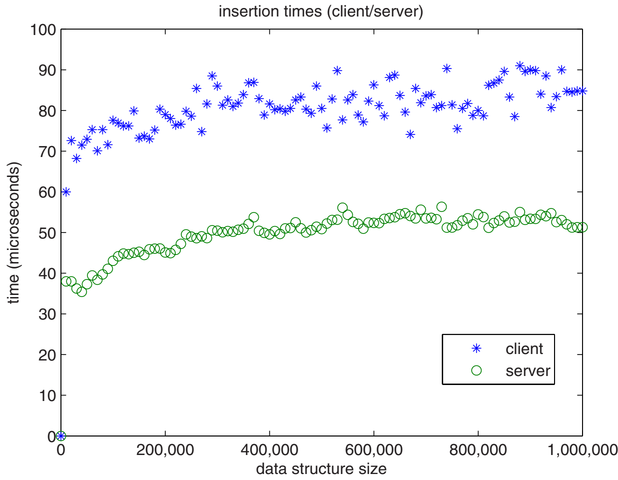
1. *The protocol is secure;*
2. *The expected running time of updates/queries is $O(\log n)$ at the server and at the client w.h.p.;*
3. *The expected communication complexity of updates/queries is $O(\log n)$ w.h.p.;*
4. *The expected hashing complexity of updates/queries is $O(\log n)$ w.h.p.;*
5. *The client uses space $O(1)$;*
6. *The server uses expected space $O(n)$ w.h.p.;*
7. *The protocol has optimal time, space, and communication complexity, up to a constant factor, over all hash-based authentication protocols.*

Taking into account constant factors (see the definitions in [21]), the communication and hashing complexity can be shown to be at most $1.5 \log n$ with high

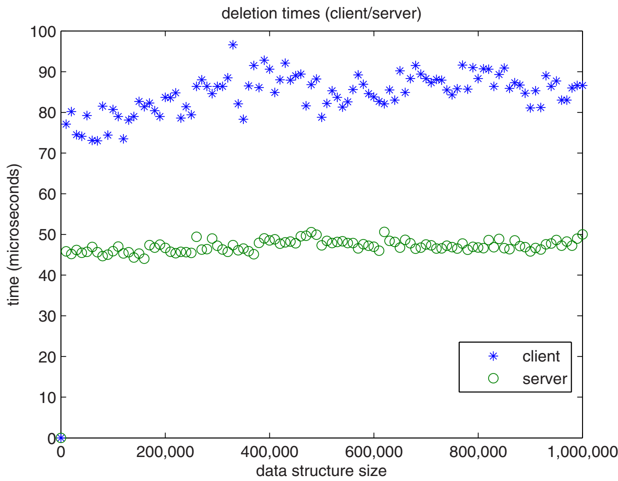
probability. Moreover, the client keeps a single hash value (e.g., a 128-bit or 256-bit digest, using standard hash functions).

4 Experimental Results

We have conducted experiments on the performance of the presented authentication protocol. The time needed for querying the server (construction of the proof) and for query verification have been reported extensively in [3]. Here we report times concerning the time needed for the client to securely update the digest (after doing a verification of the proof it receives) when it does an insertion or a deletion and also for the server to perform the respective update on the skip list. Concerning the client's side, insertion times include an initial verification of the proof and then the processing of the proof in order to update the digest. Also we give plots that indicate the size of the consistency and verification proof (bytes). From the plots we will see that the actual communication overhead of our protocol is very small; Only at most 1KB of information (for 1,000,000 elements) needs to be communicated from the server to the client so that the client can update the digest. For each operation, the average running time (or the average size in the case of proofs) was computed over 10,000 executions. The experiments were conducted on a 64-bit, 2.8GHz Intel based, dual-core, dual processor machine with 8GB main memory and 2MB cache, running Debian Linux 3.1 with Linux kernel 2.6.15 and using the Sun Java JDK 1.5. The Java Virtual Machine (JVM) was most of the times launched with a 7GB maximum heap size. Cryptographic hashing was performed using the standard Java implementation of the MD5 algorithm. We report the running times obtained in our experiments excluding the time consumed by the garbage collector. Our experimental results provide a measure of the computational and communication overhead. The results of the experiments concerning insertions and deletions are summarized in Figure 3. One can see that insertion/deletion at the client's side takes more than insertion/deletion at the server's side (Figures 3(a) and 3(b)). This is because for client insertions we count exactly the following time: The time needed for verifying that the consistency proof π hashes to the current digest and the time needed to run `update` on the current consistency proof in order to update the digest. For client-side deletions (see Figure 3(b)) we firstly have to run `update` and then do a verification. On the other hand, server side insertions and deletions do not have to do any verification at all. They are just a usual insertion/deletion in a skip list. This is enough to justify that difference in the execution times, since verification is a costly operation as it involves $O(\log n)$ hash computations. Finally, as we can see the times needed for insertion and deletion follow a logarithmic increase, verifying in this way our theoretical findings. Also, the actual time needed at the client's side is roughly 80μs which shows the efficiency of the method. The results of the experiments concerning the size of the proof are summarized in Figure 4. The size of the proof in bytes is computed as follows. First of all we compute the size of the structure Λ which is the input in `update`(). Let N be that size. It is easy to see that the maximum



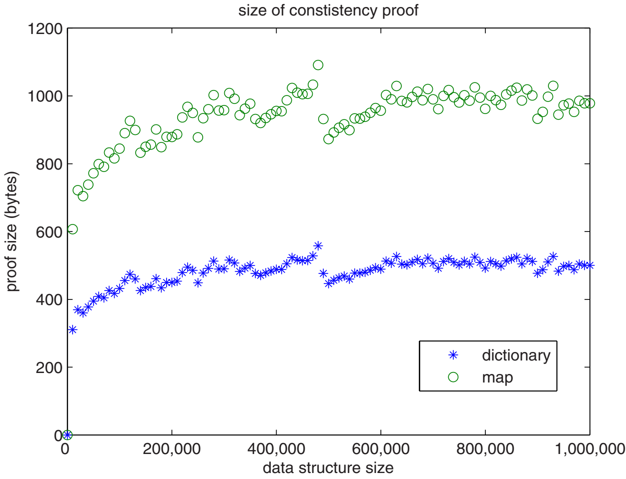
(a) Client/Server Insertion at the client's and server's side.



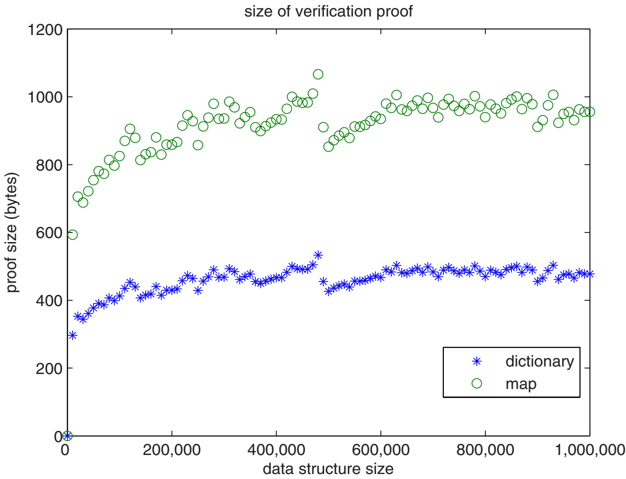
(b) Deletion at the client's and server's side.

Fig. 3. Insertion and deletion times. The times shown are the average of 10,000 executions on data structure sizes varying from 0 to 10^6 elements.

size (the size of vector Λ) of the proof in our experiments (for 10^6 elements) is 30 and it validates the theory since $1.5 \log(10^6) \simeq 30$. Since now for each element of Λ, Λ_i, it is $\Lambda_i = (\lambda(v_i), \ell_x(\lambda(v_i)), k_x(\lambda(v_i)))$, we have that each element of the extended proof needs exactly $(128 + \log(\mathsf{maxlevel}) + 1)$ bits. Here $\mathsf{maxlevel}$ is the maximum level of the skip list towers and for our experiments is 20. Hence for a consistency proof of size N we need exactly $\frac{N}{8} \times 134$ bytes. This holds for the case of the dictionary. If we want to use the map functionality (i.e., to bind a key with



(a) Size of the consistency proof for a dictionary and a map.



(b) Size of the verification proof for a dictionary and a map.

Fig. 4. Communication overhead (proof size). The sizes shown are the average of 10,000 executions on data structure sizes varying from 0 to 10^6 elements.

a value), which is something that has more applications, then the size of the proof is $\frac{N}{8}(2\times128+\log(\mathsf{maxlevel})+1)$ bytes, since we also need to hold a digest for the value. The plot for the size of the consistency proof is shown in Figure 4(a). One can observe the logarithmic increase. Also, for the case of the dictionary the average size of the proof is 0.5KB. The plot for the verification proof is shown in Figure 4(b). Finally the communication overhead of a consistency versus a verification proof is only a few bits (namely 6 per element).

5 Conclusions and Future Work

In this paper, we have presented an efficient protocol for two-party authentication based on cryptographic hash functions. Namely, we have given efficient, lightweight and provably secure algorithms that ensure the validity of the answers returned by an outsourced dictionary. We have implemented our protocol and we have provided experimental results that confirm the scalability of our approach. As future work, we envision developing a general authentication framework that can be applied to other data structures, such as dynamic trees [7]. Additionally, we could investigate realizations of two-party authenticated protocols based on other cryptographic primitives, (for example cryptographic accumulators [6]).

Acknowledgments

We thank Nikos Triandopoulos and Michael Goodrich for useful discussions.

References

[1] Anagnostopoulos, A., Goodrich, M.T., Tamassia, R.: Persistent authenticated dictionaries and their applications. In: Davida, G.I., Frankel, Y. (eds.) ISC 2001. LNCS, vol. 2200, pp. 379–393. Springer, Heidelberg (2001)

[2] Di Battista, G., Palazzi, B.: Authenticated relational tables and authenticated skip lists. In: Proc. Working Conference on Data and Applications Security (DBSEC), pp. 31–46 (2007)

[3] Goodrich, M.T., Papamanthou, C., Tamassia, R.: On the cost of persistence and authentication in skip lists. In: Proc. Int. Workshop on Experimental Algorithms (WEA), pp. 94–107 (2007)

[4] Goodrich, M.T., Shin, M., Tamassia, R., Winsborough, W.H.: Authenticated dictionaries for fresh attribute credentials. In: Nixon, P., Terzis, S. (eds.) iTrust 2003. LNCS, vol. 2692, pp. 332–347. Springer, Heidelberg (2003)

[5] Goodrich, M.T., Tamassia, R.: Implementation of an authenticated dictionary with skip lists and commutative hashing. In: Proc. DARPA Information Survivability Conference & Exposition II (DISCEX II), pp. 68–82. IEEE Computer Society Press, Los Alamitos (2001)

[6] Goodrich, M.T., Tamassia, R., Hasic, J.: An efficient dynamic and distributed cryptographic accumulator. In: Chan, A.H., Gligor, V.D. (eds.) ISC 2002. LNCS, vol. 2433, pp. 372–388. Springer, Heidelberg (2002)

[7] Goodrich, M.T., Tamassia, R., Triandopoulos, N., Cohen, R.: Authenticated data structures for graph and geometric searching. In: Joye, M. (ed.) CT-RSA 2003. LNCS, vol. 2612, pp. 295–313. Springer, Heidelberg (2003)

[8] Hacigümüş, H., Iyer, B., Li, C., Mehrotra, S.: Executing SQL over encrypted data in the database-service-provider model. In: Proc. Int. Conference on Management of Data (SIGMOD), pp. 216–227 (2002)

[9] Hacigümüş, H., Mehrotra, S., Iyer, B.: Providing database as a service. In: Proc. Int. Conference on Data Engineering (ICDE), p. 29 (2002)

[10] Hore, B., Mehrotra, S., Tsudik, G.: A privacy-preserving index for range queries. In: Proc. Int. Conference on Very Large Databases (VLDB), pp. 720–731 (2004)

[11] Li, F., Hadjieleftheriou, M., Kollios, G., Reyzin, L.: Dynamic authenticated index structures for outsourced databases. In: Proc. of ACM SIGMOD International Conference on Management of Data, pp. 121–132 (2006)
[12] Martel, C., Nuckolls, G., Devanbu, P., Gertz, M., Kwong, A., Stubblebine, S.G.: A general model for authenticated data structures. Algorithmica 39(1), 21–41 (2004)
[13] Merkle, R.C.: A certified digital signature. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 218–238. Springer, Heidelberg (1990)
[14] Miklau, G., Suciu, D.: Implementing a tamper-evident database system. In: Grumbach, S., Sui, L., Vianu, V. (eds.) ASIAN 2005. LNCS, vol. 3818, pp. 28–48. Springer, Heidelberg (2005)
[15] Mykletun, E., Narasimha, M., Tsudik, G.: Authentication and integrity in outsourced databases. In: Proceeding of Network and Distributed System Security (NDSS) (2004)
[16] Narasimha, M., Tsudik, G.: Authentication of outsourced databases using signature aggregation and chaining. In: Proc. of 11th International Conference on Database Systems for Advanced Applications, pp. 420–436 (2006)
[17] Oprea, A., Reiter, M.K.: Integrity checking in cryprographic file systems with constant trusted storage. In: Proc. USENIX Security Symposium (USENIX), pp. 183–198 (2007)
[18] Pugh, W.: Skip lists: a probabilistic alternative to balanced trees. Commun. ACM 33(6), 668–676 (1990)
[19] Tamassia, R.: Authenticated data structures. In: Di Battista, G., Zwick, U. (eds.) ESA 2003. LNCS, vol. 2832, pp. 2–5. Springer, Heidelberg (2003)
[20] Tamassia, R., Triandopoulos, N.: On the cost of authenticated data structures. Technical report, Center for Geometric Computing, Brown University, Available (2003), from `http://www.cs.brown.edu/cgc/stms/papers/costauth.pdf`
[21] Tamassia, R., Triandopoulos, N.: Computational bounds on hierarchical data processing with applications to information security. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 153–165. Springer, Heidelberg (2005)