Efficient Authenticated Dictionaries with Skip Lists and Commutative Hashing*

MICHAEL T. GOODRICH
Department of Computer Science
Johns Hopkins University
goodrich@cs.jhu.edu

ROBERTO TAMASSIA
Department of Computer Science
Brown University
rt@cs.brown.edu

January 13, 2001

Abstract

We present an efficient and practical technique for dynamically maintaining an authenticated dictionary. The main building blocks of our scheme are the skip list data structure and cryptographic associative hash functions. Applications of our work include certificate revocation in public key infrastructure and the the publication of data collections on the Internet.

1 Introduction

We present an efficient and practical data structure for dynamically maintaining an authenticated dictionary. Applications of our work include certificate revocation in public key infrastructure and the the publication of data collections on the Internet.

1.1 Problem Definition

The problem we address involves three parties: a trusted source, an untrusted directory, and a user. The *source* defines a finite set S of elements that evolves over time through insertions and deletions of items. The *directory* maintains a copy of set S. It receives time-stamped updates from the source together with *update authentication information*, such as signed statements about the update and the current elements of the set. The *user* performs membership queries on the set S of the type "is element e in set S?" but instead of contacting the source directly, it queries the directory. The directory provides the user with a yes/no answer to the query together with *query authentication information*, which yields a proof of the answer assembled by combining statements signed by the source. The user then verifies the proof by relying solely on its trust in the source and the availability of public information about the source that allows to check the source's signature. The data structure used by the directory to maintain set S, together with the protocol for queries and updates is called an *authenticated dictionary* [17]. Figure 1 shows a schematic view of an authenticated dictionary.

The design of an authenticated dictionary should address the following goals:

- *low computational cost:* the computations performed internally by each entity (source, directory, and user) should be simple and fast; also, the memory space used by the data structures supporting the computation should be as small as possible;

*This research was supported by DARPA Grant F30602-00-2-0509.

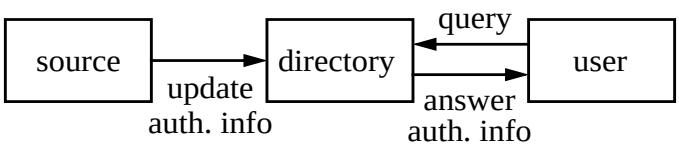

Figure 1: Authenticated dictionary.

- *low communication overhead:* source-to-directory communication (update authentication information) and directory-to-user communication (query authentication information) should be kept as small as possible;
- *high security:* the authenticity of the data provided by a directory should be verifiable with a high degree of reliability.

We can formalize the above goals as the algorithmic problem of minimizing the following cost parameters of an authenticated dictionary for set S.

1. space used by the data structure;
2. the time spent by the directory to perform an update initiated by the source;
3. size of the update authentication information sent by the source in an update (source-to-directory communication);
4. time spent by the directory to answer a query and return the query authentication information as a proof of the answer;
5. size of the query authentication information sent by the directory together with the answer (directory-to-user communication);
6. time spent by the user to verify the answer to a query.

Authenticated dictionaries have a number of applications, including scientific data mining (e.g., genomic querying [8] and astrophysical querying [11, 2, 12]), geographic data servers (e.g., GIS querying), third-party data publication on the Internet [4], and certificate revocation in public key infrastructure [9, 15, 17, 1, 3, 7, 5].

In the third-party publication application [4], the source is a trusted organization (e.g., a stock exchange) that produces and maintains integrity-critical content (e.g., stock prices) and allows third parties (e.g., Web portals), to publish this content on the Internet so that it widely disseminated. The publishers store copies of the content produced by the source and process queries on such content made by the users. In addition to returning the result of a query, a publisher also returns a proof of authenticity of the result, thus providing a validation service. Publishers also perform content updates originating from the source. Even so, the publishers are not assumed to be trustworthy, for a given publisher may be processing updates from the source incorrectly or it may be the victim of a system break-in.

In the certificate revocation application [9, 15, 17, 1, 3, 7, 5], the source is a *certification authority* (CA) that digitally signs certificates binding entities to their public keys, thus guaranteeing their validity. Nevertheless, certificates are sometimes revoked (e.g., if a private key is lost or compromised, or if someone loses their authority to use a particular private key). Thus, the user of a certificate must be able to verify that a given certificate has not been revoked. To facilitate such queries, the set of revoked certificates is distributed to *certificate revocation directories*, which process revocation status queries on behalf of users.

The results of such queries need to be trustworthy, for they often form the basis for electronic commerce transactions.

In this paper, we present a new scheme for authenticated dictionaries, based on the skip list data structure and on commutative collision-resistant hash functions. Our data structure is efficient and secure. It matches the theoretical performance parameters of the best previous approaches that attempt to optimize simultaneously all the above performance measures. In addition, our algorithms are simpler to implement and deploy in practical applications. With our technique, the computations performed by the user are very simple and can be easily done on devices with limited memory and computing power, such as PDAs, smart cards, and cellphones.

1.2 Previous and Related Work

In this section, and throughout the rest of this paper, we denote with n the current number of elements of the set S stored in the authenticated dictionary.

Previous work on authenticated dictionaries has been conducted primarily in the context of certificate revocation. The traditional method for certificate revocation (e.g., see [9]) is for the CA (source) to sign a statement consisting of a timestamp plus a hash of the set of all revoked certificates, called *certificate revocation list* (CRL), and periodically send the signed CRL to the directories. A directory then just forwards that entire signed CRL to any user who requests the revocation status of a certificate. This approach is secure, but it is inefficient, for it requires the transmission of the entire set of revoked certificates for both source-to-directory and directory-to-user communication. This scheme corresponds to an authenticated dictionary where both the update authentication information and the query authentication information has size $\Theta(n)$. Because of the inefficiency of the underlying authenticated dictionary, CRLs are not a scalable solution for certificate revocation.

Micali [15] proposes an alternate approach, where the source periodically sends to each directory the list of all issued certificates, each tagged with the signed time-stamped value of a one-way hash function (e.g., see [19]) that indicates if this certificate has been revoked or not. This approach allows the system to reduce the size of the query authentication information to $O(1)$ words: namely just a certificate identifier and a hash value indicating its status. Unfortunately, this scheme requires the size of the update authentication information to increase to $\Theta(N)$, where N is the number of all nonexpired certificates issued by the certifying authority, which is typically much larger than the number n of revoked certificates.

The *hash tree* scheme introduced by Merkle [13, 14] can be used to implement a static authenticated dictionary, which supports the initial construction of the data structure followed by query operations, but not update operations (without complete rebuilding). A hash tree T for a set S stores the elements of S at the leaves of T and a value $h(v)$ at each node v, defined as follows:

- if v is a leaf, $h(v) = x$, where x is the element stored at x;
- else (v is an internal node), $h(v) = f(h(u), h(w))$, where u and w are the left and right child of v, respectively, and f is a collision-resistant cryptographic hash function, such as MD5 or SHA1.

The authenticated dictionary for S consists of the hash tree T plus the signature of a statement consisting of a timestamp and the value $h(r)$ stored of the root r of T. An element x is proven to belong to S by reporting the values stored at the nodes on the path in T from the node storing x to the root, together with the values of all nodes that have siblings on this path. Each node in this collection must be identified as a left or right child, and the path given in order, so that the user can recompute the root's hash value and compare it to the current signed value. It is important that all this order and connectivity information be presented to the user, for without it the user would have great difficulty recomputing the hash value for the root. This hash tree scheme can be extended to validate that an item x is not in S by keeping the leaves of T sorted and

then returning the leaf-to-root paths, and associated hash values, for two elements y and z such that y and z are stored at consecutive leaves of T and $y < x < z$, or (in the boundary cases) y is undefined and z is the left-most leaf or z is undefined and y is the right-most leaf. Again, the user is required to know enough about binary trees to be able to verify from the topology of the two paths that y and z are stored at consecutive leaves.

Kocher [10] also advocates a static hash tree approach for realizing an authenticated dictionary, but simplifies somewhat the processing done by the user to validate that an item is not in the set S. In his solution, the leaves of the hash tree store the intervals defined by the consecutive elements in the sorted sequence of the elements of S. A membership query for an item x always returns a leaf v and and the interval $[y,z]$ stored at v such that $y \leq x < z$, together with the path from v to the root and all sibling hash values for nodes along this path. The user validates this path by recomputing the hash values of the nodes in this path, keeping track of whether nodes are left children or right children of their respective parents. Although there is a minor extra overhead of now having to have a way of representing $-\infty$ and $+\infty$, this method simplifies the verification for the case when an item is not in S (which will usually be the case in certificate revocation applications). It does not support updates of the set S, however.

Using techniques from incremental cryptography, Naor and Nissim [17] dynamize hash trees to support the insertion and deletion of elements. In their scheme, the source and the directory maintain identically-implemented 2-3 trees. Each leaf of such a 2-3 tree T stores an element of set S, and each internal node stores a one-way hash of its children's values. Hence, the source-to-directory communication is reduced to $O(1)$ items, since the source send insert and remove instructions to the directory, together with a signed message consisting of a timestamp and the hash value of the root of T. A directory responds to a membership query for an element x as follows: if x is in S, then the directory supplies the path of T from the leaf storing x to the root, together with all siblings of nodes on this path; else (x is not in S), the directory supplies the leaf-to-root paths from two consecutive leaves storing y and z such that $y < x < z$, together with all siblings of the nodes on these paths. By tracing these paths, the user can recompute the hash values of their nodes, ultimately recomputing the hash value for the root, which is then compared against the signed hash value of the root for authentication. One can apply Kocher's interval idea to this scheme as an alternative way of validating items that are not in the dictionary S. There are nevertheless some drawbacks of this approach. Dynamic 2-3 trees are not trivial to program correctly, as it is. In addition, since nodes in a 2-3 tree can have two or three children, one must take special care in the structuring of the query authentication information sent by the directory to the user. Namely, all sibling nodes returned must be classified as being left children, middle children (if they exist), or right children. Recomputing the hash value at the root requires that a user be able to match the computation done at the source as regards a particular leaf-to-root path.

Other certificate revocation schemes based on variations of hash trees have been recently proposed in [3, 5], as well, but do not deviate significantly from the above approaches.

1.3 Summary of Results

We introduce an efficient and practical authenticated dictionary scheme, which factors away many of the complications of the previous schemes while maintaining their asymptotic performance properties. Our approach is based on two new ideas. First, rather than hashing up a dynamic 2-3 tree, we hash in a skip list [18]. This choice has two immediate benefits:

- It replaces the complex details of 2-3 trees with the easy-to-implement details of skip lists.
- It allows us to avoid the complication of storing intervals at leaf nodes [10], and instead allows us to return to the intuitive concept of storing actual items at the base nodes.

Second, we introduce the use of commutative hashing as a means to greatly simplify the verification process for a user, while retaining the basic security properties of signing a collection of values via cryptographic

hashing. We summarize the asymptotic performance of our scheme in Table 1. Our methods therefore match the asymptotic performance of the Naor-Nissim approach [17], while simplifying the details of an actual implementation of a dynamic authenticated dictionary. Indeed, we show that the verification process for a user can now be simplified to a straightforward iterative hashing of a sequence of numbers, with no regards to notions such as leaf-to-root paths or left, middle, or right children. If the hash of this sequence matches the signed hash of the entire skip list, then the result (either a membership or its converse) is validated.

method	space	update time	update info	query time	query info	verify time
CRL's	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Micali [15]	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(1)$	$O(t)$
Naor-Nissim [17]	$O(n)$	$O(\log n)$	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Our scheme	$O(n)$	$O(\log n)$	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$

Table 1: Comparison of the main authenticated dictionary schemes with our new scheme. We use n to denote the size of the dictionary, t to denote the number of updates since a queried element has been created, and N to denote the size of the universe the elements of the dictionary come from. The time and information size bounds for our scheme are expected with high probability, while they are worst-case for the other schemes.

Our update info size is actually about $\log n$ bits, which we can encode in a single word (since $\log n$ bits is needed just to store the value of n). We provide the details of our method in the sections that follow. We begin by reviewing the skip list data structure and how it can be used to maintain a dictionary of elements subject to insertions, deletions, and searches. We also describe an implementation of our approach in Java, and some promising benchmarking data on its performance.

2 Skip Lists

The *skip list* data structure [18] is an efficient means for storing a set S of elements from an ordered universe. It supports the following operations:

- **find**(x): determine whether elements x is in S.
- **insert**(x): insert element x into S.
- **delete**(x): remove element x from S.

2.1 Definition

A skip list stores a set S of elements in a series of linked lists S_0, S_1, S_2, ..., S_t. The base list, S_0, stores all the elements of S in order, as well as sentinels associated with the special elements $-\infty$ and $+\infty$. Each successive list S_i, for $i \geq 1$, stores a sample of the elements from S_{i-1}. The method used to define the sample from one level to the next determines the kind of skip list being maintained. The default method is simply to choose each element of S_{i-1} at random with probability $1/2$ to be in the list S_i. But one can also define a deterministic skip list [16], which uses simple rules to guarantee that between any two elements in S_i there are at least 1 and at most 3 elements of S_{i-1}. In either case, the sentinel elements $-\infty$ and $+\infty$ are always included in the next level up, and the top level, t, is maintained to be $O(\log n)$. In both the deterministic and the randomized versions, the top level is guaranteed to contain only the sentinels. We therefore distinguish the node of the top list S_t storing $-\infty$ as the *start node* s.

An element that exists in S_{i-1} but not in S_i is said to be a *plateau* element of S_{i-1}. An element that is in both S_{i-1} and S_i is said to be a *tower* element in S_{i-1}. Thus, between any two tower elements, there are some plateau elements. In deterministic skip lists, the number of plateau elements between two towers is at least one and at most three. In randomized skip lists, the expected number of plateau elements between two tower elements is one. (See Figure 2.)

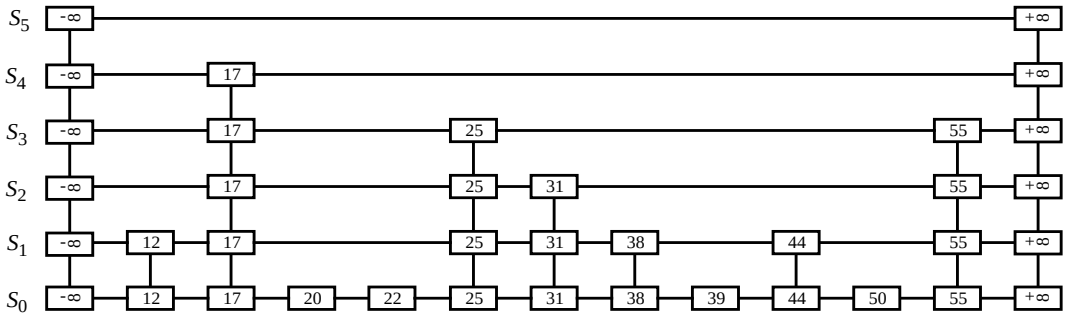

Figure 2: Example of a skip list.

For each node v of list S_i, we denote with $\text{elem}(v)$ the element stored at v. Also, we denote with $\text{down}(v)$ the node in S_{i-1} below v, which stores the same element as v, unless $i = 0$, in which case $\text{down}(v) =$ **null**. Similarly, we denote with $\text{right}(v)$ the node in S_i immediately to the right of v, unless v is the sentinel storing $+\infty$, in which case $\text{right}(v) =$ **null**.

2.2 Search

To perform a search for element x in a skip list, we begin at the start node s. Let v denote the current node in our search (initially, $v = s$). The search proceeds using two actions, *forward hop* and *drop down*, which are repeated one after the other until we terminate the search.

- *Hop forward*: We move right along the current list until we find the node of the current list with largest element less than or equal to x. That is, while $\text{elem}(\text{right}(v)) < x$, we update $v = \text{right}(v)$
- *Drop down*: If $\text{down}(v) =$ **null**, then we are done with our search: the node v stores the largest element in the skip list less than or equal to x. Otherwise, we update $v = \text{down}(v)$.

The outer loop of the search process continues while $\text{down}(p) \neq$ **null**, performing inside the loop one hop forward followed by one drop down. After completing such a sequence of hops forward and drops down, we ultimately reach a node v with $\text{down}(v) =$ **null**. If, at this point, $\text{elem}(v) = x$, then we have found element x. Otherwise, v is the node of the base list with the largest element less than x; likewise, in this case, $\text{right}(v)$ is the a node of the base list with the smallest element greater than x, that is, $\text{elem}(v) < x < \text{elem}(\text{right}(v))$.

Figures 3–4 show examples of searches in the skip list of Figure 2.

In a deterministic skip list, the above searching process is guaranteed to take $O(\log n)$ time. Even in a randomized skip list, it is fairly straightforward to show (e.g., see [6]) that the above searching process runs in expected $O(\log n)$ time, for, with high probability, the height t of the randomized skip list is $O(\log n)$ and the expected number of nodes visited on any level is three. Moreover, experimental studies (e.g., see [18]) have shown that randomized skip lists often outperform 2-3 trees, red-black trees, and other deterministic search tree structures.

Performing updates in a skip list is also quite simple.

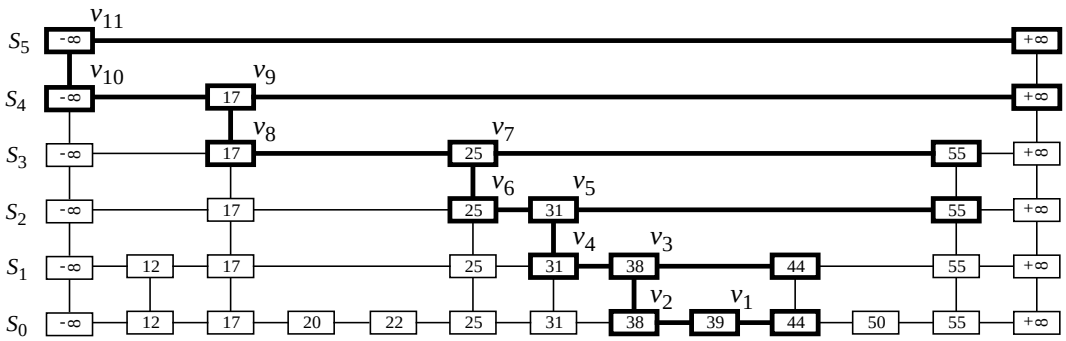

Figure 3: Search for element 39 in the skip list of Figure 2. The nodes visited and the links traversed are drawn with thick lines. This successful search visits the same nodes as the unsuccessful search for element 42.

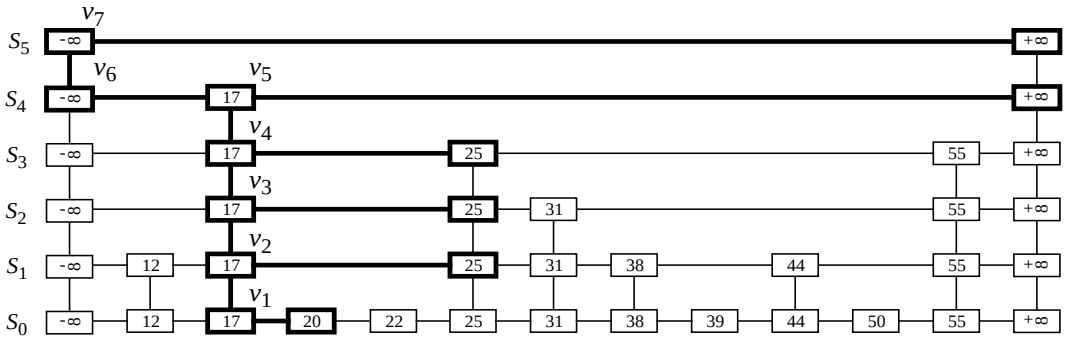

Figure 4: Search for element 17 in the skip list of Figure 2. The nodes visited and the links traversed are drawn with thick lines. This successful search visits the same nodes as the unsuccessful search for element 18.

2.3 Insertion

To insert a new element x, we begin, as we would in a search, at the start node s. Indeed, we proceed by performing a search for element x. In addition, each time we perform a hop forward or drop down from the current node v, we push a reference to v in a stack, A, indicating if this was a hop forward or a drop down. Once the search stops at a node v of the base level S_0, we insert a new node into S_0 right after v and store there the element x. In a deterministic skip list we then use the stack A (with pop operations) to update the skip list as necessary to maintain the size rules for plateau nodes between towers. In a randomized skip list, we determine which lists should contain the new element x by a sequence of simulated random coin flips. Starting with $i = 0$, while the coin comes up heads, we use the stack A to trace our way back to the position of list S_{i+1} where element x should go, add a new node storing x to this list, and set $i = i + 1$. We continue this insertion process until the coin comes up tails. If we reach the top level with this insertion process, we add a new top level on top of the current one. The time taken by the above insertion method is $O(\log n)$, worst-case for a deterministic skip list, and with high probability for a randomized skip list.

2.4 Deletion

To delete an existing element x, we begin by performing a search for a fictitious element smaller than x but larger than any element of S less than x. As in the insertion case, we use a stack A to keep track of the nodes

encountered during the search. The search ends at a node v on the bottom level such that $\text{elem}(\text{right}(v)) = x$. (If this is not the case, then there is no element x in the skip list and we should signal an error.) We remove $\text{right}(v)$ from the base level and, using the stack A, we remove all the nodes on higher levels that contain the element x. The time taken by the above removal method is $O(\log n)$, worst-case for a deterministic skip list, and with high probability for a randomized skip list.

3 Commutative Hashing

For this paper, we view a cryptographic hash function as a function that takes two integer arguments, x and y, and maps them to an integer $h(x,y)$ that is represented using a fixed number k of bits (typically fewer than the number of bits of x and y). Intuitively, $h(x,y)$ is a digest for the pair (x,y). We can also use the hash function h to digest a triple, (x,y,z), as $h(x,h(y,z))$. Likewise, we can use h to digest larger sequences. Namely, to digest a sequence $(x_1,x_2,\ldots,x_m)$ we can compute $h(x_1,h(x_2,\ldots h(x_{m-2},h(x_{m-1},x_m))\cdots))$.

3.1 Collision Resistance

The reason that cryptographic hash functions are useful for the task of authenticating dictionary responses is that given a pair (a,b) it is difficult to find a pair $(c,d) \neq (a,b)$ such that $h(a,b) = h(c,d)$. The concept of difficulty here is that it is computationally intractable to find such a pair (c,d) independent of the specific values for a and b. The value T, of time steps needed to compute such a colliding pair, should depend only on the number of bits in the hash value for h's range, and T should be so large as to be an infeasible computation for even a supercomputer to perform. This property of h is known as *collision resistance*.

To simplify the verification process that a user has to do in our authenticated dictionary scheme, we would like a cryptographic hash function that has an additional property. We want a hash function h that is *commutative*, that is, $h(x,y) = h(y,x)$, for all x and y. Such a function will allow us to simplify verification, but it requires that we modify what we mean by a *collision resistant* hash function, for the condition $h(x,y) = h(y,x)$ would normally be considered as a collision. We therefore say that a hash function is *commutatively collision resistant* if, given (a,b), it is difficult to compute a pair (c,d) such that $h(a,b) = h(c,d)$ while $(a,b) \neq (c,d)$ and $(a,b) \neq (d,c)$.

3.2 A Candidate Construction

Given a cryptographic hash function f that is collision resistant in the usual sense, we construct a candidate commutative cryptographic hash function, h, as follows:

$$h(x,y) = f(\min\{x,y\}, \max\{x,y\}).$$

This function is clearly commutative.

Its collision resistance derives from the collision resistance of f. Specifically, given (a,b), consider the computational difficulty of finding a pair (c,d) such that $h(a,b) = h(c,d)$ while $(a,b) \neq (c,d)$ and $(a,b) \neq (d,c)$. Without loss of generality, let us assume that $a < b$. If finding a collision pair (c,d) is computationally feasible for h, then this immediately implies (by a simple reduction) that it is computationally feasible to find numbers c and d such that $f(a,b) = f(c,d)$ and $(a,b) \neq (c,d)$ or $f(a,b) = f(d,c)$ and $(a,b) \neq (d,c)$. In either case we would have a collision in the usual cryptographic sense. Thus, if f is collision resistant in the usual sense, then h is commutatively collision resistant.

In the next section we show how to use a commutative cryptographic hash function to accumulate efficiently the values of a dictionary that are stored in a skip list.

4 Authenticated Dictionary Based on a Skip List

Let h be a commutative cryptographic hash function. Our authenticated dictionary for a set S consists of the following components:

- A skip list data structure storing the items of S.
- A collection of values $f(v)$ that label each node v of the skip list, computed accumulating the elements of S with the hash function h, as discussed below.
- A statement signed by the source consisting of the timestamp of the most recent label $f(s)$ of the start node of the skip list.

We use h to compute the label $f(v)$ of each node v in the skip list, except for the nodes associated with the sentinel value $+\infty$. The value $f(s)$ stored at the start node, s, represents a digest of the entire skip list. Intuitively, each label $f(v)$ accumulates the labels of nodes below v possibly combined with the labels of some nodes to the right of v.

4.1 Hashing Scheme

For each node v we define label $f(v)$ in terms of the respective values at nodes $w = \text{right})(v)$ and $u = \text{down}(v)$. If $\text{right}(v) = \textbf{null}$, then we define $f(v) = 0$. The definition of $f(v)$ in the general case depends on whether u exists or not for this node v.

1. $u = \textbf{null}$, i.e., v is on the base level:
   (a) If w is a tower node, then $f(v) = h(\text{elem}(v), \text{elem}(w))$.
   (b) If w is a plateau node, then $f(v) = h(\text{elem}(v), f(w))$.
2. $u \neq \textbf{null}$, i.e., v is not on the base level:
   (a) If w is a tower node, then $f(v) = f(u)$.
   (b) If w is a plateau node, then $f(v) = h(f(u), f(w))$.

We illustrate the flow of the computation of the hash values labeling the nodes of a skip list in Figure 5. Note that the computation flow defines a directed acyclic graph, not a tree.

4.2 Updates

The source maintains its own copy of the authenticated dictionary, and updates the authenticated dictionary of the directory by specifying the operation performed (insertion/deletion) and the element x involved, plus the following authentication information:

- a signed statement consisting of a timestamp and the new hash value of the start node s;
- if the skip list is randomized, the random bits used by the source in the update, which are $O(\log n)$ with high probability.

After performing the update in the skip list, the hash values must be updated to reflect the change that has occurred. In either an insertion or deletion, the stack A (see Sections 2.3–2.4) comes to our aid, to make the updating simple. For the nodes stored in the stack A (and possibly their right neighbors) are precisely the nodes whose hash values have changed. Thus, we can simple pop off nodes from the stack A and recompute the $f(v)$ labels for each one (plus possibly the f labels for v's right neighbor, if that node stores the element x that is being inserted or deleted, as the case may be). The additional computational expense needed to update all these values is proportional to the number of elements in A; hence, it is expected with high probability to be $O(\log n)$ in a randomized skip list and is guaranteed to be $O(\log n)$ in a deterministic skip list.

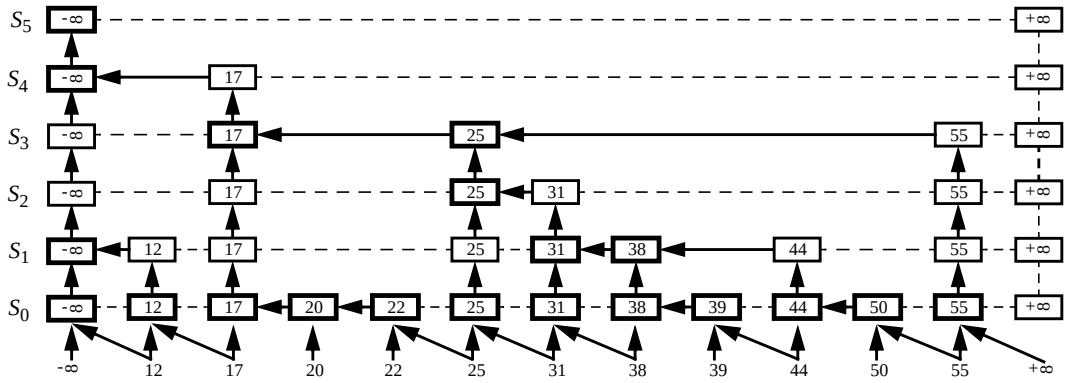

Figure 5: Flow of the computation of the hash values labeling the nodes of the skip list of Fig. 2. Nodes where hash functions are computed are drawn with thick lines. Also, please note that the arrows denote the flow of information, not pointer values in the data structure.

4.3 Query and Verification

The verification of the answer to a query is simple, thanks to the use of a commutative hash function. Recall that the goal is to produce a verification that some element x is or is not contained in the skip list. In the case when the answer is "yes," we verify the presence of the element itself. Otherwise, we verify the presence of two elements x' and x'' stored at consecutive nodes on the bottom level S_0 such that $x' < x < x''$. In either case, the query authentication information is a single sequence of values, together with the signed timestamp and value $f(s)$.

Let $P(x) = (v_1, \cdots, v_m)$ be the sequence of nodes that are visited when searching for element x, in reverse order. In the example of Fig. 3, we have $P(39) = P(42) = (v_1, \cdots, v_{11})$, and in the example of Fig. 4, we have $P(17) = P(18) = (v_1, \cdots, v_7)$ Note that by the properties of a skip list, the size m of sequence $P(x)$ is $O(\log n)$ with high probability. We construct from the node sequence $P(x)$ a sequence $Q(x) = (x_0, \cdots, x_p)$ of values using the method given in Figure 6.

The computation of the sequence $P(x)$ can be easily done by pushing into a stack the nodes visited while searching for element x. When the search ends, the stack contains the nodes of $P(x)$ ordered from top to bottom. Using this stack, we can construct sequence $Q(x)$ by following the method of Figure 6, where the nodes of $P(x)$ are obtained by popping them from the stack, one at a time.

We recall from Section 2 that node v_1 stores either x (for a "yes" answer) or the largest element less than x (for a "no" answer). In the first case (v_1 stores x), the directory returns as part of the authentication information for the membership of x in S the sequence $Q(x)$, as illustrated in Figures 7 and 8. In the second case (v_1 does not store x), let w_1 be the node to the right of v_1, and z be the node to the right of w_1, if any; we further distinguish three subcases:

1. w_1 is a tower node: sequence $Q(x)$ is returned, as illustrated in Figure 7;
2. w_1 is a plateau node and z_1 is a tower node: the sequence $(\text{elem}(z), \text{elem}(w_1)) \cdot Q(x)$ is returned, where $\cdot$ denotes concatenation;
3. w_1 is a plateau node and z_1 is a plateau node: the sequence $(f(z), \text{elem}(w_1)) \cdot Q(x)$ is returned, as illustrated in Figure 8.

In either case, the user verifies the answer by simply hashing the values of the returned sequence in the given order, and comparing the result with the signed value $f(s)$, where s is the start node of the skip list. If the two values agree, then the user is assured of the validity of the answer at the time given by the timestamp.

$w_1 \leftarrow \text{right}(v_1)$
if w_1 is a plateau node **then**
  $x_0 \leftarrow f(w_1)$
else
  $x_0 \leftarrow \text{elem}(w_1)$
end if
$x_1 \leftarrow x$
$j \leftarrow 1$
for $i \leftarrow 2, \cdots, m-1$ **do**
  $w_i \leftarrow \text{right}(v_i)$
  if w_i is a plateau node **then**
    $j \leftarrow j+1$
    if $w_i \neq v_{i-1}$ **then**
      $x_j \leftarrow f(w_i)$
    else
      if v_i is in the base list S_0 **then**
        $x_j \leftarrow \text{elem}(v_i)$
      else
        $u_i \leftarrow \text{down}(v_i)$
        $x_j \leftarrow f(u_i)$
      end if
    end if
  end if
end for
$p \leftarrow j$

Figure 6: Computation of the sequence $Q(x) = (x_0, \cdots, x_p)$ from the node sequence $P(x) = (v_1, \cdots, v_m)$.

Note that our scheme requires only the repeated accumulation of a sequence of values with a hash function. Unlike the previous best hash tree schemes [15], there is no need to provide auxiliary information about the order of the arguments to be hashed at each step, as determined by the topology of the path in a hash tree.

5 Implementation

We have developed a prototype implementation in Java of an authenticated dictionary based on skip lists. We use six interfaces combine to describe our authenticated dictionary system. Three relate to querying, and three relate to modifying an authenticated dictionary.

At the heart of the query system is the AuthenticatedDictionary interface, which describes membership queries made by a user. It allows the user to retrieve the query authentication data, represented by an instance of the Basis interface, and to request data to be used to initialize a new directory. It also declares the principal query method, contains. The contains method takes a single parameter, the element, and returns an instance of the AuthenticResponse interface that may be used to verify whether or not the element is contained in the dictionary.

An instance of AuthenticResponse has a method, subject, to determine the element of the query for which the response is issued, and a method, subjectContained, to determine whether or not the element is

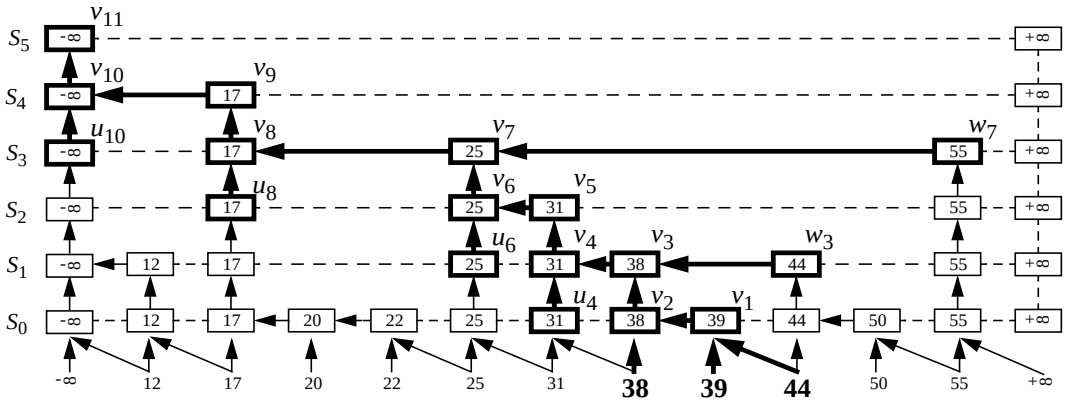

Figure 7: The query authentication information for the presence of element 39 (and for the absence of element 42) consists of the signed time-stamped value $f(v_{11})$ of the source element and the sequence $(44, 39, 38, f(w_3), f(u_4), f(u_6), f(w_7), f(u_8), f(u, 4), f(u_{10}))$. The user recomputes $f(v_{11})$ by accumulating the elements of the sequence with the hash function h, and verifies that the computed value of $f(v_7)$ is equal to the value signed by the source. Please note that, as in Figure 5, the arrows denote the flow of information, not pointer values in the data structure.

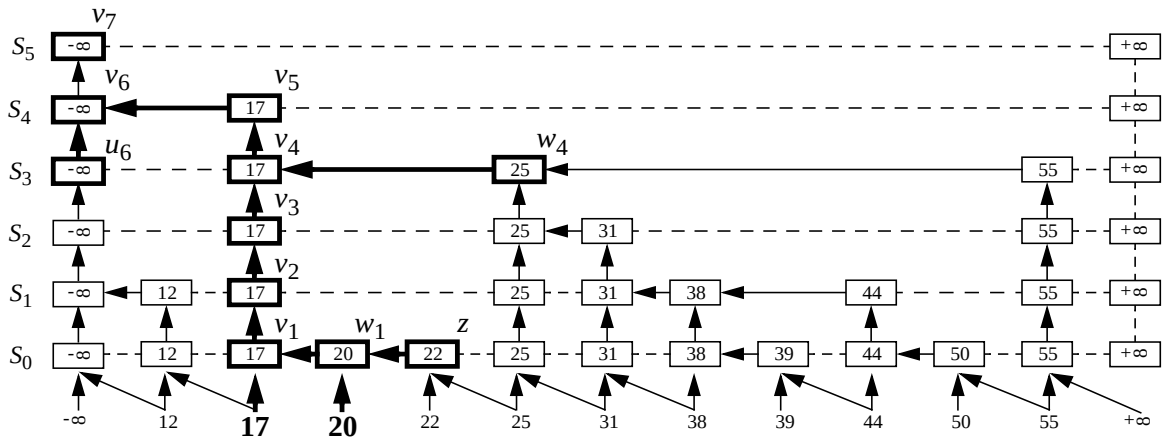

Figure 8: The query authentication information for the absence of element 18 consists of the signed time-stamped value $f(v_7)$ of the source element, and the sequence: $Y = (f(z), 20, 17, f(w_4), f(u, 6))$. The user recomputes $f(v_7)$ by accumulating the elements of sequence Y with the hash function h, and verifies that the computed value of $f(v_7)$ is equal to the value signed by the source. Please note that, as in Figure 5, the arrows denote the flow of information, not pointer values in the data structure.

contained by the dictionary. There is also a method for determining whether or not the response is valid, called validatesAgainst, which takes an instance of Basis as its parameter. If the user trusts that the data stored in the instance of Basis has not been tampered with, and if validatesAgainst returns *true*, then the user may trust that the values returned by subject and subjectContained are correct.

The data represented by the Basis and AuthenticResponse are implementation-dependent. In our system, which uses commutative hashing and skip lists, the basis is $f(s)$, that is, the label of the start node s of the skip list. The instance of AuthenticResponse for a contains query with parameter x is a sequence of values computed from the skip list including x if x is contained in the dictionary, or containing the values immediately less than and immediately greater than x in the dictionary, if x is not contained. To verify the response, the user checks to see that the appropriate values are in the sequence, and then uses the hashing function to recompute $f(s)$. If the recomputed value of $f(s)$ matches the value in the basis, then the response is valid. Otherwise, it is not.

The three interfaces involved in modifying an authenticated dictionary system include an interface to allow trusted entities to add or remove items from the dictionary, an interface for use in initializing the directories (mirror instances of the data structure), and an interface for transmitting to the directories updates based on changes made by the trusted source. The first interface is named SourceAuthenticatedDictionary, and has two methods: insert and remove. Both methods have a single parameter, the element, and return an instance of the third interface, Update. Copies of the Update instance may be sent to directories in order to inform them of changes in the contents of the dictionary. The Update interface contains an execute method that carries out the action of the update on a directory, so the only method in the MirrorAuthenticatedDictionary interface is the one used to initialize a new directory.

Because specific implementations of authenticated dictionary systems restrict the types of data that may be stored in the dictionaries, the contains method of AuthenticatedDictionary, as well as the insert and remove methods of SourceAuthenticatedDictionary and the initialize method of MirrorAuthenticatedDictionary may throw exceptions if the user attempts to insert incompatible data. Also, if a directory is not fed instances of Update in the order in which they were generated at the source, exceptions may arise, depending upon specific implementations.

We have conducted a preliminary experiment on the performance of our data structure on randomly generated sets of 128-bit integers ranging in size from $100,000$ to $700,000$. For each operation, the average was computed over 30,000 trials. The experiment was conducted on a 440MHz Sun Ultra 10 with 256M of memory running Solaris. The Java Virtual Machine was launched with a 200M maximum heap size. Cryptographic hashing was performed using the standard Java implementation of the MD5 algorithm.

The results of the experiment are summarized in Figure 9. Note that validations, insertions and deletions take less than 1ms, while queries take less than 0.1ms. We note that a simple ratio test, comparing the running time $t(n)$ in milliseconds of an operation to the function $\log_2 n$, estimates the ratio $t(n)/\log_2 n$ as 0.005 for queries, 0.04 for validations, and 0.05 for insertions and deletions. Thus, we feel the use of skip lists and commutative hashing is a scalable solution for the authenticated dictionary.

6 Conclusion

We have presented an efficient and practical technique for realizing an authenticated dictionary. Our methods achieve asymptotic performance bounds that match those of previous schemes, but do so using simpler algorithms. We are able to retain the basic security properties of the previous schemes but make the dynamic maintenance of an accumulated dictionary more practical, particularly for contexts where user computations must be performed on simple devices, such as PDAs, smart cards, or cellphones.

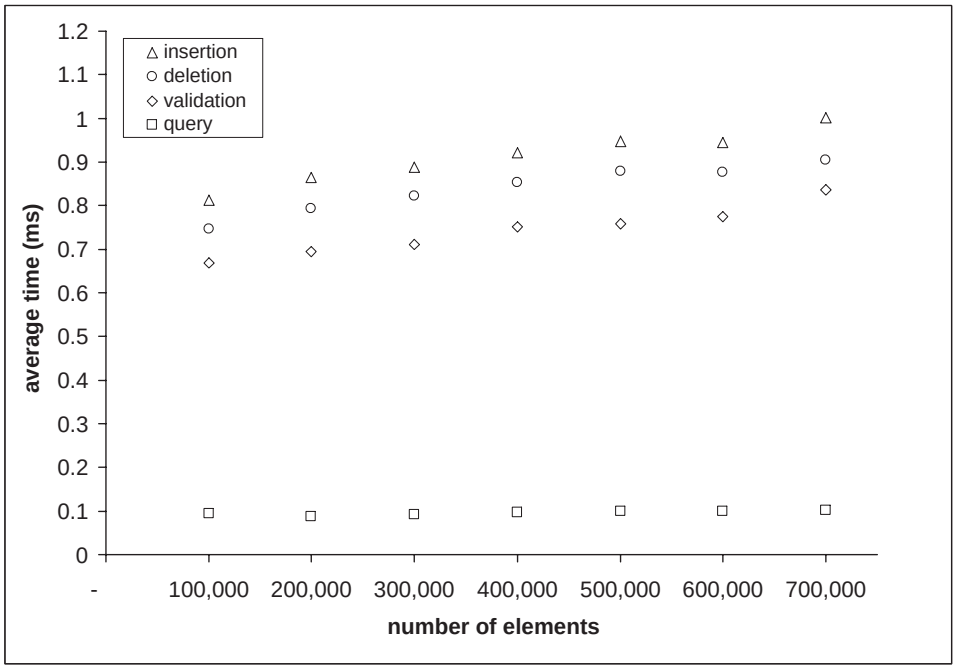

Figure 9: Average time per operation (in milliseconds) of our Java implementation of an authenticated dictionary using a skip list.

Acknowledgments

We would like to thank Giuseppe Ateniese for helpful discussions on the topics of this paper, Benety Goh for implementing the data structure, Andrew Schwerin for designing the Java interfaces for authenticated dictionaries, and James Lentini and Andrew Schwerin for conducting the runtime experiments..

References

[1] W. Aiello, S. Lodha, and R. Ostrovsky. Fast digital identity revocation. In *Advances in Cryptology – CRYPTO ' 98*, Lecture Notes in Computer Science. Springer-Verlag, 1998.

[2] R. J. Brunner, L. Csabai, A. S. Szalay, A. Connolly, G. P. Szokoly, and K. Ramaiyer. The science archive for the Sloan Digital Sky Survey. In *Proceedings of Astronomical Data Analysis Software and Systems Conference V*, 1996. http://ecf.hq.eso.org/iraf/web/ADASS/adass_proc/adass_95/brunnerr/brunnerr.html.

[3] A. Buldas, P. Laud, and H. Lipmaa. Accountable certificate management with undeniable attestations. In *ACM Conference on Computer and Communications Security*. ACM Press, 2000.

[4] P. Devanbu, M. Gertz, C. Martel, and S. Stubblebine. Authentic third-party data publication. In *Fourteenth IFIP 11.3 Conference on Database Security*, 2000.

[5] I. Gassko, P. S. Gemmell, and P. MacKenzie. Efficient and fresh certification. In *International Workshop on Practice and Theory in Public Key Cryptography ' 2000 (PKC '2000)*, Lecture Notes in Computer Science, pages 342–353, Melbourne, Australia, 2000. Springer-Verlag, Berlin Germany.

[6] M. T. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java*. John Wiley & Sons, New York, NY, 1998.

[7] C. Gunter and T. Jim. Generalized certificate revocation. In *Proc. 27th ACM Symp. on Principles of Programming Languages*, pages 316–329, 2000.

[8] R. M. Karp. Mapping the genome: Some combinatorial problems arising in molecular biology. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on the Theory of Computing*, pages 278–285, 1993.

[9] C. Kaufman, R. Perlman, and M. Speciner. *Network Security: Private Communication in a Public World*. Prentice-Hall, Englewood Cliffs, NJ, 1995.

[10] P. C. Kocher. On certificate revocation and validation. In *Proc. International Conference on Financial Cryptography*, volume 1465 of *Lecture Notes in Computer Science*, 1998.

[11] R. Lupton, F. M. Maley, and N. Young. Sloan digital sky survey. http://www.sdss.org/sdss.html.

[12] R. Lupton, F. M. Maley, and N. Young. Data collection for the Sloan Digital Sky Survey—A network-flow heuristic. *Journal of Algorithms*, 27(2):339–356, 1998.

[13] R. C. Merkle. Protocols for public key cryptosystems. In *Proc. Symp. on Security and Privacy*. IEEE Computer Society Press, 1980.

[14] R. C. Merkle. A certified digital signature. In G. Brassard, editor, *Advances in Cryptology—CRYPTO '89*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238. Springer-Verlag, 1990.

[15] S. Micali. Efficient certificate revocation. Technical Report TM-542b, MIT Laboratory for Computer Science, 1996.

[16] J. I. Munro, T. Papadakis, and R. Sedgewick. Deterministic skip lists. In *Proc. Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 367–375, 1992.

[17] M. Naor and K. Nissim. Certificate revocation and certificate update. In *Proceedings of the 7th USENIX Security Symposium (SECURITY-98)*, pages 217–228, Berkeley, 1998.

[18] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

[19] B. Schneier. *Applied cryptography: protocols, algorithms, and sourcecode in C*. John Wiley and Sons, Inc., New York, 1994.