Implementation of an Authenticated Dictionary with Skip Lists and Commutative Hashing*

Michael T. Goodrich
Department of Computer Science
Johns Hopkins University
Baltimore, Maryland 21218
goodrich@cs.jhu.edu

Roberto Tamassia Andrew Schwerin
Department of Computer Science
Brown University
Providence, Rhode Island 02912
{rt,schwerin}@cs.brown.edu

Abstract

We present the software architecture and implementation of an efficient data structure for dynamically maintaining an authenticated dictionary. The building blocks of the data structure are skip lists and one-way commutative hash functions. We also present the results of a preliminary experiment on the performance of the data structure. Applications of our work include certificate revocation in public key infrastructure and the publication of data collections on the Internet.

1. Introduction

We present the software architecture and implementation of an efficient and practical data structure for dynamically maintaining a distributed collection of elements in an authenticated manner. Applications of our work include certificate revocation in public key infrastructure and authenticated publication of data collections on the Internet.

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 elements. 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 *answer 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* [18, 29]. Figure 1 shows a schematic view of an authenticated dictionary.

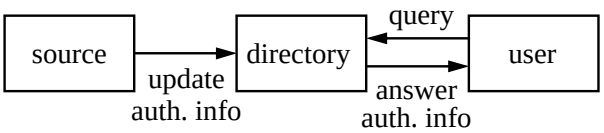

Figure 1. Authenticated dictionary.

1.1 Design Goals

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;
- *low communication overhead:* source-to-directory communication (update authentication information) and directory-to-user communication (answer 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.

*Research supported in part by DARPA Grant F30602–00–2–0509.

1.2 Metrics and Applications

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

1. space used by the data structures maintained by the source, directory, and user;
2. 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);
4. time spent by the directory to answer a query and return the answer authentication information as a proof of the answer;
5. size of the answer authentication information sent by the directory together with the answer (directory-to-user);
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 [20] and astrophysical querying [24, 10, 25]), geographic data servers (e.g., GIS querying), third-party publication on the Internet [12], and certificate revocation in public key infrastructure [21, 28, 29, 1, 11, 19, 15].

In the third-party publication application [12], 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 party publishers (e.g., Web portals), to publish this content on the Internet so that it is widely disseminated. The publishers store copies of the content produced by the source. They perform content updates originating from the source and process queries on such content made by the users. However, 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. Thus, in addition to returning the result of a query, a publisher should also return a proof of authenticity of the result.

In the certificate revocation application [21, 28, 29, 1, 11, 19, 15], the source is a *certification authority* (CA) that digitally signs certificates binding entities (e.g., identities or attributes) to their public keys, thus guaranteeing this binding. 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.

1.3 Organization of the Paper

The rest of this paper is organized as follows: Section 2 overviews previous work on authenticated dictionaries, especially in the context of certificate revocation. Our software architecture for authenticated dictionaries is described in Section 3. Our prototype implementation of an authenticated dictionaries based on skip lists and commutative hashing is outlined in Section 4. In Section 5, we report the results of a preliminary experiment on the performance of our data structure, and we conclude in Section 6.

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.

Authenticated dictionaries are related to research in distributed computing (e.g., data replication in a network [5, 23]), data structure design (e.g., program checking [6, 8, 9, 32] and memory checking [7, 13]), and cryptography (e.g., incremental cryptography [2, 3, 13, 14]).

2.1. Certificate Revocation

Previous work on authenticated dictionaries has been conducted primarily in the context of certificate revocation in public-key infrastructure (PKI). The traditional method for certificate revocation (e.g., see [21]) 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 answer authentication information has size $O(n)$. Because of the inefficiency of the underlying dictionary, CRLs are not a scalable solution for certificate revocation.

Micali [28] 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 [31]) that indicates if this certificate has been revoked or not. This approach allows the system to reduce the size of the answer

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 $O(N)$, where N is the number of all non-expired certificates issued by the certifying authority, which is typically much larger than the number, n, of revoked certificates.

2.2. Hash Trees

The *hash tree* scheme introduced by Merkle [26, 27] 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 label $f(v)$ at each node v, defined as follows:

- if v is a leaf, $f(v) = x$, where x is stored at v;
- else (v is an internal node), $f(v) = h(f(u), f(w))$, where u and w are the left and right child of v, respectively, and h 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 label $f(r)$ stored at the root r of T. An element x is proven to belong to S by reporting the labels of the nodes on the path in T from the leaf storing x to the root, together with the values of all nodes that have siblings on this path. Each node in this path must be identified as a left or right child, and the path must be 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 [22] 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 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.

2.3. Dynamic Hash Trees

Using techniques from incremental cryptography, Naor and Nissim [29] 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 sends insert and remove instructions to the directory, together with a signed statement 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. 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 answer 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 [11, 15], as well, but do not deviate significantly from the above approaches.

method	space	update time	update info	query time	answer info	validation time
CRL's	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Micali [28]	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(1)$	$O(t)$
Naor-Nissim [29]	$O(n)$	$O(\log n)$	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Goodrich-Tamassia [18]	$O(n)$	$O(\log n)$	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Goodrich-Schwerin-Tamassia [16]	$O(n)$	$O(p + n/p)$	$O(p)$	$O(n/p)$	$O(1)$	$O(1)$

Table 1. Comparison of data structures for authenticated dictionaries. 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. We denote with p an integer such that $1 \leq p \leq n$. The time and information size bounds of the Goodrich-Tamassia scheme are expected with high probability, while they are worst-case for the other schemes.

2.4. Skip Lists

Goodrich and Tamassia [18] have devised a data structure for an authenticated dictionary based on skip lists [30]. They introduce the notion of commutative hashing and show how to embed in the nodes of a skip list a computational DAG (directed acyclic graph) of cryptographic computations based on commutative hashing. This data structure matches the asymptotic performance of the Naor-Nissim approach [29], while simplifying the details of an actual implementation of a dynamic authenticated dictionary. In particular, the choice of a skip list and commutative hashing to implement an authenticated dictionary has the following benefits over approaches based on hash trees:

- It replaces the complex details of 2-3 trees with the easy-to-implement details of skip lists.
- It avoids the complication of storing intervals at leaf nodes [22], and instead returns to the intuitive concept of storing actual items at the leaf nodes.
- It greatly simplifies the verification process for a user, while retaining the basic security properties of signing a collection of values via cryptographic hashing.

2.5. One-Way Accumulators

The authors [16] have recently developed a data structure for authenticated dictionaries based on one-way accumulators [4, 31]. An advantage of this approach is that the validation of a query result performed by the user takes constant time and requires computations simple enough to be performed in devices with very limited computing power, such as a smart card or a wireless phone. This approach achieves a tradeoff between the cost of updates at the source and queries at the directories, with updates taking $O(p + \log(n/p))$ time and queries taking $O(n/p)$ time, for any fixed integer parameter $1 \leq p \leq n$. For example, one can achieve $O(\sqrt{n})$ time for both updates and queries.

We compare the asymptotic performance of data structures for authenticated dictionaries in Table 1.

3. Software Architecture

We have designed a general object-oriented software architecture for authenticated dictionaries and we have implemented it in Java. A high-level view of the software architecture is shown in Figure 2. In our architecture, an entity, called *certification authority*, or CA, has been added to the participants of the authenticated dictionary protocol. The CA is the only trusted entity in the system. It initiates updates and provides a signed statement to authenticate each update. This statement is modeled by an object called the Basis. In our formalization, the source acts as the intermediary between the CA and the directory. It forwards to the directory each update and its associated basis. The directory replies to queries made by the user by returning an object called AuthenticResponse, whose data fields provide the answer authentication information.

We use six interfaces (APIs) to describe our authenticated dictionary system: AuthenticatedDictionary, with its subinterfaces MirrorAuthenticatedDictionary and SourceAuthenticatedDictionary, AuthenticResponse, Update, and Basis.

3.1. Queries

Interfaces AuthenticatedDictionary, AuthenticResponse, and Basis relate to querying. At the heart of the query system is the AuthenticatedDictionary. Its principal methods are

- AuthenticResponse contains(Object o): queries the membership of an element and retrieves the answer as an AuthenticResponse object;
- Basis getBasis(): requests the Basis object providing the answer authentication information.

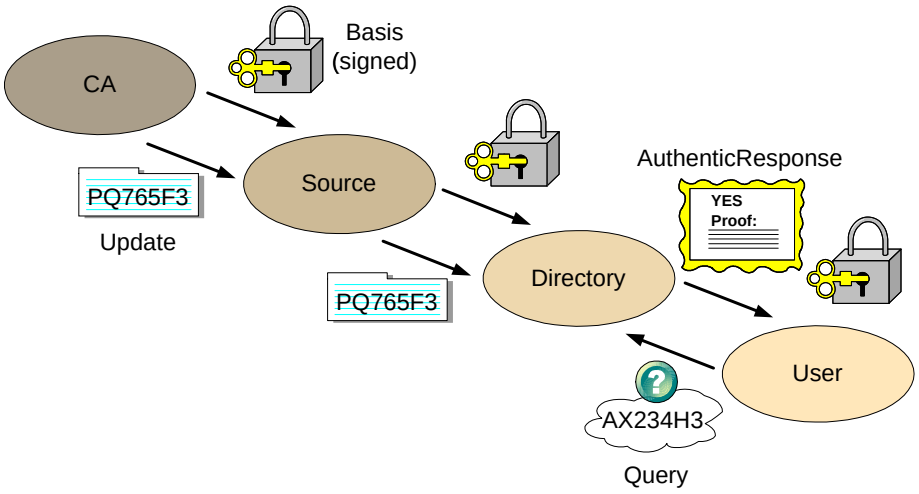

Figure 2. High-level view of our object-oriented software architecture for authenticated dictionaries.

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

The user should trust that the answer about the membership of the object returned by subject in the dictionary provided that subjectContained is correct and the following are verified:

1. the user trusts that the *data* stored in the instance of Basis has not been tampered with, e.g., because it has been signed by the CA.;

2. the user trusts that the *code* executed by the methods of the AuthenticResponse has not been tampered with, e.g., because it has been signed by the CA;

3. method validatesAgainst returns *true*.

A schematic interaction diagram for a query is shown in Figure 3. Note that method verifyBasis() is not part of the interfaces discussed above.

The data represented by the Basis and AuthenticResponse objects are implementation-dependent. For example, in the hash tree data structure, the basis is the label of the root of the tree, and the AuthenticResponse object for an element in the set contains the sequence of labels (and associated left-child/right-child) indicators, for the siblings of the nodes in the path from the leaf containing the element to the root. Method validatesAgainst recomputes the label of the root by hashing the labels in the sequence in the appropriate order and compares the value so obtained with the one provided by the basis.

3.2. Updates

Interfaces Update, MirrorAuthenticatedDictionary and SourceAuthenticatedDictionary, relate to updating an authenticated dictionary.

The SourceAuthenticatedDictionary interface describes the updates to the authenticated dictionary maintained at the source. It allows the CA to add or remove items from the dictionary. It has two methods: insert and remove. Both methods have a single parameter, the element, and return an Update object. The Update object is used to transmit changes in the dictionary to the directory. The Update interface contains an execute method that carries out the action of the update on a directory, which could be a single insert/remove operation or a sequence of them. The MirrorAuthenticatedDictionary is the view given to an object of type Update of the authenticated dictionary maintained at the directory. Its only method is used to initialize the directory.

It is assumed that a transport mechanism exists form distributing Update objects and their associated Basis objects to the directory. A schematic interaction diagram for updates is shown in Figure 4. Note that methods signBasis() and distribute() are not part of the interfaces discussed above.

Because specific implementations of authenticated dic-

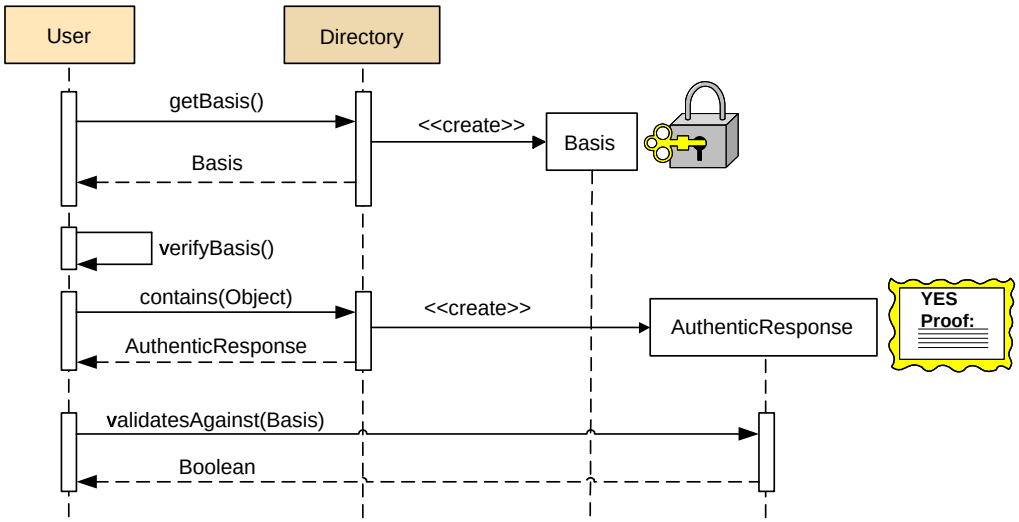

Figure 3. Query interaction diagram.

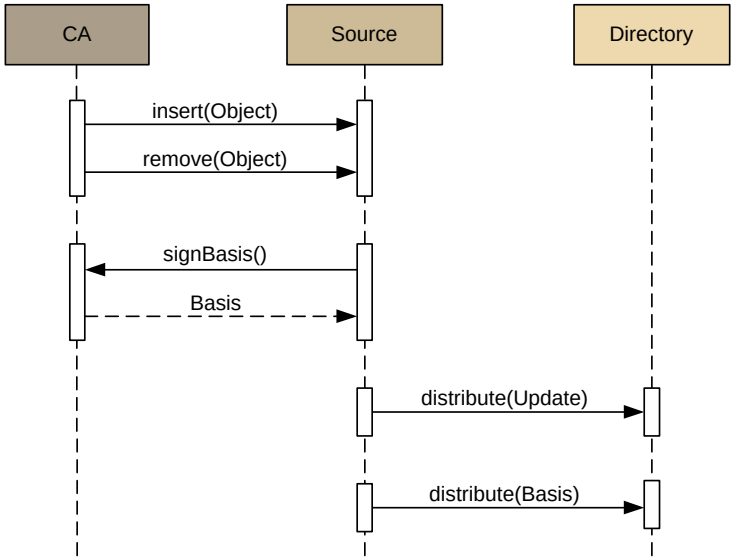

Figure 4. Update interaction diagram.

tionary systems may 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 show the source code for the above interfaces at the end of this paper, in Figures 10 through 15.

4. Implementation

To validate our software architecture for authenticated dictionaries, we have done a prototype implementation of an authenticated dictionary based on skip lists.

4.1. Skip Lists

In this section, we review the *skip list* data structure [30], which is an efficient means for storing a set S of elements from an ordered universe. It supports the following opera-

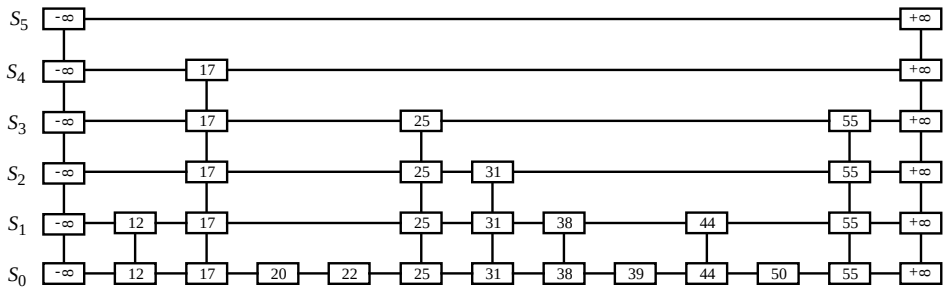

Figure 5. Example of a skip list.

tions:

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

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}. To define the sample from one level to the next, we choose each element of S_{i-1} at random with probability $1/2$ to be in the list S_i. The sentinel elements $-\infty$ and $+\infty$ are always included in the next level up, and the top level, t, is maintained to be $O(\log n)$. 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. The expected number of plateau elements between two tower elements is one. (See Figure 5.)

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

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 elem(right(v)) $< x$, we update $v =$ right(v)
- *Drop down*: If 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 $v =$ down(v).

The outer loop of the search process continues while 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 down(v) = **null**. If, at this point, 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, right(v) is the a node of the base list with the smallest element greater than x, that is, elem(v) $< x <$ elem(right(v)). Figures 6 shows an example of a search in the skip list of Figure 5.

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 (e.g., see [17]). Moreover, experimental studies (e.g., see [30]) have shown that skip lists often outperform 2-3 trees, red-black trees, and other deterministic search tree structures.

To insert a new element x, 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 $O(\log n)$ with high probability. To delete an existing element x, we remove all the nodes that contain the element x. This takes time is $O(\log n)$ with high probability.

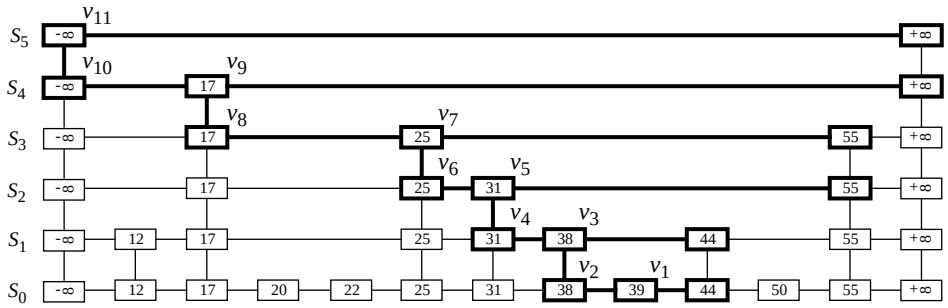

Figure 6. Search for element 39 in the skip list of Figure 5. The nodes visited and the links traversed are drawn with thick lines. This successful search visits the same nodes as the unsuccessful search for element 42.

4.2. 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 $h(x_1, h(x_2, \ldots h(x_{m-2}, h(x_{m-1}, x_m)) \cdots))$.

To simplify the verification process that a user has to do in an authenticated dictionary scheme, Goodrich and Tamassia introduce *commutative cryptographic hash functions* [18]. A hash function h is *commutative* if $h(x, y) = h(y, x)$, for all x and y. Such a function 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 $(a, b) \neq (d, c)$.

Given a cryptographic hash function h that is collision resistant in the usual sense, we construct a candidate commutative cryptographic hash function, h', as follows [18]:

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

It can be shown that h' is commutatively collision resistant [18].

4.3. Authenticated Dictionary Based on a Skip List

The authenticated dictionary approach introduced in [18] consists of a skip list where each node v stores a label computed accumulating the elements of the set with a commutatively cryptographic hash function h. For completeness, let us review how hashing occurs. See [18] for details.

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 $\text{right}(v) =$ **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. $u =$ **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. $u \neq$ **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 7. Note that the computation flow defines a directed acyclic graph, not a tree.

After performing the update in the skip list, the hash values must be updated to reflect the change that has occurred. The additional computational expense needed to update all these values is expected with high probability to be $O(\log n)$.

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 answer authentication information is a single sequence of values, together with the signed, timestamped, label $f(s)$ of the start node s.

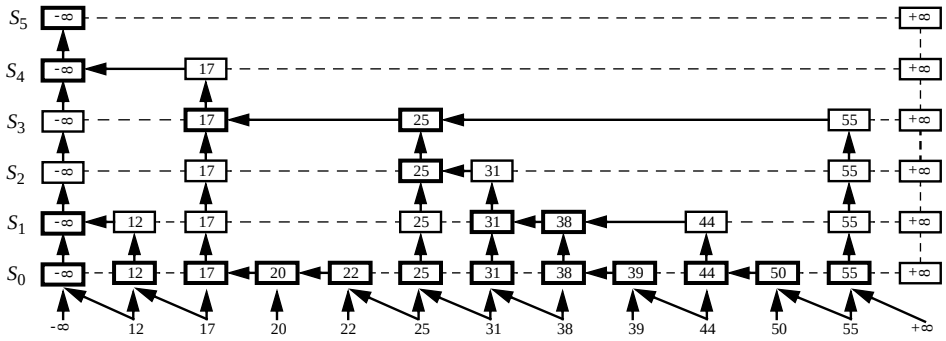

Figure 7. Flow of the computation of the hash values labeling the nodes of the skip list of Fig. 5. Nodes where hash functions are computed are drawn with thick lines. The arrows denote the flow of information, not links in the data structure.

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. 6, we have $P(39) = P(42) = (v_1, \cdots, v_{11})$. 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 $Q(x) = (y_1, \cdots, y_m)$ of values such that:

- $y_m = f(s)$, the label of the start node;
- $y_m = h(y_{m-1}, h(y_{m-2}, h(\cdots, y_1) \cdots)))$

The computation of the node sequence $P(x)$ can be done by pushing onto 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 easily construct the sequence $Q(x)$ of node labels.

The user verifies the answer for element x by simply hashing the values of the returned sequence $Q(x)$ 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.

4.4. Implementation Details

The six interfaces described in Section 3 have been implemented as Java classes. Additional auxiliary classes have been used. Some implementation details are overviewed below.

- A class CommutativeHash serves as a wrapper that adds commutativity to a standard java.security.MessageDigest.
- The class implementing the Basis interface stores the label of the start node of the skip list and a reference to the CommutativeHash used by the data structure.
- The class implementing the AuthenticResponse interface stores the sequence of label values $Q(x)$ and an integer flag to distinguish among the various cases of validation of the answer.
- Two classes are used to implement the Update interface. One represents insertion updates and stores the height of the tower associated with the newly inserted element. The other represents deletion updates.
- The class implementing the AuthenticatedDictionary interface uses finite sentinel values. Also, it limits to a given value the height of any tower.

5. Performance

We have conducted a preliminary experiment on the performance of our data structure for authenticated dictionaries 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 signing of the basis by the CA and the signature verification by the user were omitted from the experiment. The highest level of a tower was limited to 20.

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. Thus, we feel the use of skip lists and commutative hashing is a scalable solution for the authenticated dictionary.

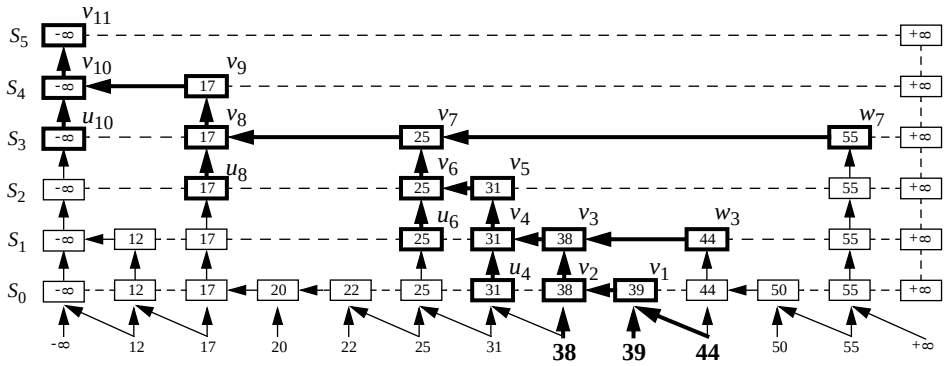

Figure 8. The answer authentication information for the presence of element $x = 39$ (and for the absence of element 42) consists of the signed time-stamped value $f(v_{11})$ of the source element and the sequence $Q(x) = (44, 39, 38, f(w_3), f(u_4), f(u_6), f(w_7), f(u_8), f(u, 4), f(u_{10}))$. The user recomputes $f(v_{11})$ by accumulating the elements of the sequence with the hash function h, and verifies that the computed value of $f(v_{11})$ is equal to the value signed by the source. As in Figure 7, the arrows denote the flow of information, not links in the data structure.

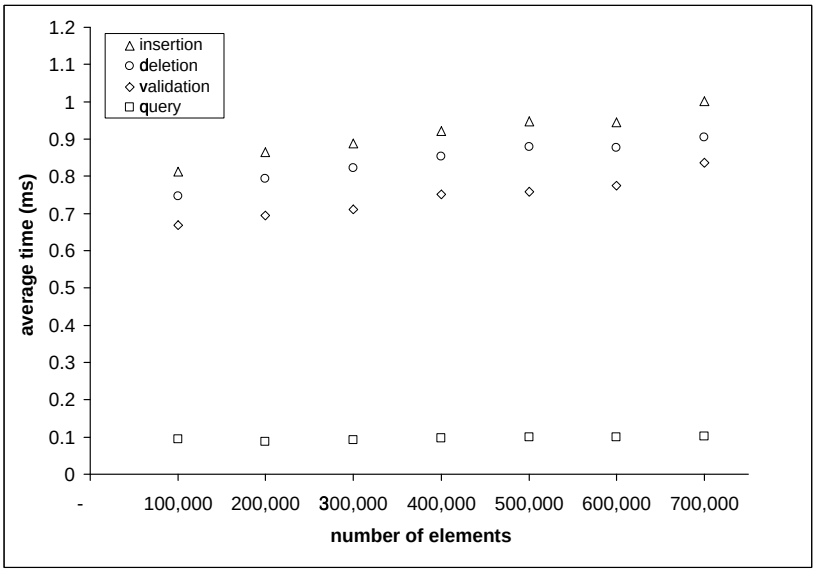

Figure 9. Average time per operation (in milliseconds) of our Java implementation of an authenticated dictionary using a skip list.

6. Conclusion

We presented an object-oriented software design of an authenticated dictionary and a prototype implementation of an efficient and practical data structure for realizing an authenticated dictionary. Preliminary experiments show we are able to retain the basic security properties of 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 and smart cards.

Acknowledgments We would like to thank Giuseppe Ateniese and Robert Cohen for helpful discussions on the topics of this paper, Benety Goh for assisting in the implementation of the data structure, and James Lentini for conducting runtime experiments. We also thank Jeremy Mullendore, Joel Sandin, and Michael Shin for additional software support.

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] M. Bellare, O. Goldreich, and S. Goldwasser. Incremental cryptography: The case of hashing and signing. In *Advances in Cryptology—CRYPTO '94*, volume 839 of *Lecture Notes in Computer Science*, pages 216–233. Springer-Verlag, 1994.

[3] M. Bellare, O. Goldreich, and S. Goldwasser. Incremental cryptography and application to virus protection. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on the Theory of Computing*, pages 45–56, 1995.

[4] J. Benaloh and M. de Mare. One-way accumulators: A decentralized alternative to digital signatures. In *Advances in Cryptology—EUROCRYPT 93*, volume 765 of *Lecture Notes in Computer Science*, pages 274–285, 1993.

[5] J. J. Bloch, D. S. Daniels, and A. Z. Spector. A weighted voting algorithm for replicated directories. *Journal of the ACM*, 34(4):859–909, 1987.

[6] M. Blum. Program result checking: A new approach to making programs more reliable. In S. C. Andrzej Lingas, Rolf G. Karlsson, editor, *Automata, Languages and Programming, 20th International Colloquium*, volume 700 of *Lecture Notes in Computer Science*, pages 1–14. Springer-Verlag, 1993.

[7] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. *Algorithmica*, 12(2/3):225–244, 1994.

[8] M. Blum and S. Kannan. Designing programs that check their work. *J. ACM*, 42(1):269–291, Jan. 1995.

[9] M. Blum and H. Wasserman. Program result-checking: A theory of testing meets a test of theory. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 382–393, 1994.

[10] 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.

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

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

[13] Fischlin. Incremental cryptography and memory checkers. In *EUROCRYPT: Advances in Cryptology: Proceedings of EUROCRYPT, LNCS 1233*, pages 393–408, 1997.

[14] M. Fischlin. Lower bounds for the signature size of incremental schemes. In *38th Annual Symposium on Foundations of Computer Science*, pages 438–447, 1997.

[15] 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.

[16] M. T. Goodrich, A. Schwerin, and R. Tamassia. An efficient dynamic and distributed cryptographic accumulator. Technical Report, Johns Hopkins Information Security Institute, 2000.

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

[18] M. T. Goodrich and R. Tamassia. Efficient authenticated dictionaries with skip lists and commutative hashing. Technical Report, Johns Hopkins Information Security Institute, 2000.

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

[20] 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.

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

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

[23] B. Kroll and P. Widmayer. Distributing a search tree among a growing number of processors. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 23(2):265–276, 1994.

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

[25] 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.

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

[27] 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.

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

[29] 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.

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

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

[32] G. F. Sullivan, D. S. Wilson, and G. M. Masson. Certification of computational results. *IEEE Trans. Comput.*, 44(7):833–847, 1995.

```
/**
 * Interface implemented by objects that represent the bases of the validatable
 * responses returned by AuthenticatedDictionary objects.  Implementations of
 * this class should have a public constructor that takes no parameters, and
 * a means to distinguish between initialized and uninitialized instances.
 *
 **/
  public interface Basis {
  /**
   * Get the encoded form of this basis.
   **/
    public byte[] getEncoded() throws NotYetInitializedException;
  /**
   * Read the basis from an array of bytes, initializing this instance of
   * the basis.  This should at least be able to read the encodings produced
   * by the getEncoded() method.
   **/
      public void readEncoded(byte[] encoding)
        throws AlreadyInitializedException, IncompatibleDataException;
}
```

Figure 10. Basis **interface.**

```
/**
 * Interface implemented by all objects representing updates to
 * instances of MirrorAuthenticatedDictionary .
 **/
  public interface Update extends java.io.Serializable {
  /**
   * An udpate that doesn't do anything at all.  Never throws an exception,
   * and may be executed upon any MirrorAuthenticatedDictionary.
   **/
  public static final Update NOOP = new NoUpdate();
    static class NoUpdate implements Update {
      public void execute(MirrorAuthenticatedDictionary dict)
      throws IncompatibleDataException, InconsistentUpdateException, NotYetInitializedException
      { /* do nothing */ }
      public String toString() { return "NOOP"; }
    };

  /**
   * Update the given MirrorAuthenticatedDictionary dict
   * Throws IncompatibleDataException if this update object cannot update
   * the given data structure because of an incompatible implementation.
   * Throws InconsistentUpdateException if this update object cannot update
   * the given data structure because the data structure does not have the
   * appropriate initial state, or if the data structure has not yet been initialized.
   * Throws NotYetInitializedException if the mirror is uninitialized.
   **/
    public void execute(MirrorAuthenticatedDictionary dict)
      throws IncompatibleDataException, InconsistentUpdateException, NotYetInitializedException;
}
```

Figure 11. Update **interface.**

```
public interface AuthenticatedDictionary {
/* Include the basic dictionary methods size() and isEmpty() */
   public int size() throws NotYetInitializedException;
   public boolean isEmpty() throws NotYetInitializedException;
/**
 * Responds as to whether or not the given object o is contained in the
 * dictionary.    The response takes the form of an instance of the
 * AuthenticResponse class which, whenever possible, answers the question
 * of containment in the dictionary in a verifiable manner.
 * Returns A response as to whether or not <code>o</code>
 * is in the dictionary.
 * Throws IncompatibleDataException If o is not a valid
 * object to store in this instance of the AuthenticatedDictionary
 * Throws NotYetInitializedException if this instance is a mirror,
 * and has yet to be initialized
 **/
   public AuthenticResponse contains(Object o)
     throws IncompatibleDataException, NotYetInitializedException;
/**
 * Get's the basis of the verifiable responses returned by this data
 * structure.    If the user trusts the basis, then the user may trust all
 * validatable responses concluded
 * from that basis.    The user might come to trust a particular basis by
 * receiving it from a trusted source over secure channels, or by receiving
 * a copy of it that has been signed by a trusted source.
 * Throws NotYetInitializedException if this instance is a mirror,
 * and has yet to be initialized.
 **/
   public Basis getBasis() throws NotYetInitializedException;
/**
 * Determines whether or not the basis b of the validatable responses created
 * by this AuthenticatedDictionary is the same as the parameter to the method.
 * To be the same, the two bases need to be able to validate exactly the
 * same set of responses.  If the basis of the data structure validates
 * responses that b does not, or b validates responses that the data
 * structure's basis does not, then the two bases are not the same;
 * otherwise, they are.
 * Throws NotYetInitializedException if this instance is a mirror,
 * and has yet to be initialized.
 **/
   public boolean validatesAgainst(Basis b) throws NotYetInitializedException;
/**
 * Gets data that may be used to initialize a compatible mirror
 * authenticated dictionary via its
 * initialize(AuthenticatedDictionaryInitialization) method.
 * Throws NotYetInitializedException if this instance is a mirror,
 * and has yet to be initialized.
 **/
    public AuthenticatedDictionaryInitialization getInitializationData() throws NotYetInitializedException;
}
```

Figure 12. AuthenticatedDictionary **interface.**

```
public interface AuthenticResponse {

/**
 * Returns the object whose membership in a particular
 * AuthenticatedDictionary this response authentically confirms or denies.
 * Returns The subject of the response.
 **/
public Object subject();

/**
 * Return <code>true</code> iff the subject of this response is contained
 * within the authenticated dictionary that issued this response.
 **/
public boolean subjectContained();

/**
 * Checks to see if the response is actually validatable.      It is possible
 * that a given instance of AuthenticatedDictionary might only be able to
 * provide validatable responses when the result of a query is postive.
 * Such instances must still be able to supply a response when the result
 * is negative, even if that response isn't validatable.
 *
 * Returns true iff this response is validatable
 **/
public boolean isValidatable();

/**
 * Checks to see of the response is a valid conclusion from the given basis.
 * If the response is a valid conclusion from the given basis, and if the
 * user trusts the basis, then the user may also trust the validity of the
 * response.
 *
 * @param b The basis against which to check this response.
 * @return true iff the response is a valid conclusion from the given basis.
 * Must return false if <code>isValidatable()</code> returns false.
 **/
public boolean validatesAgainst(Basis b);
}
```

Figure 13. AuthenticResponse **interface.**

```
  public interface SourceAuthenticatedDictionary
    extends AuthenticatedDictionary
{

  /**
   * Add o to the AuthenticatedDictionary;
   * returns an update object describing how mirror copies of this data
   * structure should modify themselves in order to be consistent with
   * this insertion
   * @param o The object to insert
   * @throws IncompatibleDataException if o is not an object
   * that may be stored in this AuthenticatedDictionary
   **/
    public Update insert(Object o)
      throws IncompatibleDataException;

  /**
   * Remove o from the AuthenticatedDictionary,
   * returns an update object describing how mirror copies of this data
   * structure should modify themselves in order to be consistent with
   * this removal
   * @param o The object to remove
   * @throws IncompatibleDataException if o is not an object
   * that may be stored in this AuthenticatedDictionary
   **/
    public Update remove(Object o)
      throws IncompatibleDataException;
}
```

Figure 14. SourceAuthenticatedDictionary **interface.**

```
  public interface MirrorAuthenticatedDictionary
    extends AuthenticatedDictionary {

  /**
   * Initialize the otherwise uninitialized data structure according to the
   * given initialization data, which was presumably issued by the
   * getInitializationData() method of a compatible
   * implementation of SourceAuthenticatedDictionary.
   * @param initData The initialization data
   * @exception AlreadyInitializedException If the data structure has already
   * been initialized
   * @exception IncompatibleDataException if the initialization data object is
   * not compatible with this implementation of
   * MirrorAuthenticatedDictionary.
   * @see SourceAuthenticatedDictionary#getInitializationData()
   **/
    public void initialize(AuthenticatedDictionaryInitialization initData)
      throws AlreadyInitializedException, IncompatibleDataException;
}
```

Figure 15. MirrorAuthenticatedDictionary **interface.**