Persistent Authenticated Dictionaries and Their Applications[⋆]

Aris Anagnostopoulos[1], Michael T. Goodrich[2], and Roberto Tamassia[1]

[1] Dept. Computer Science, Brown University. Email: {aris, rt}@cs.brown.edu
[2] Dept. Computer Science, University of California, Irvine. Email: goodrich@acm.org

Abstract. We introduce the notion of *persistent authenticated dictionaries*, that is, dictionaries where the user can make queries of the type "was element e in set S at time t?" and get authenticated answers. Applications include credential and certificate validation checking in the past (as in digital signatures for electronic contracts), digital receipts, and electronic tickets. We present two data structures that can efficiently support an infrastructure for persistent authenticated dictionaries, and we compare their performance.

1 Introduction

At its core, nonrepudiation involves making cryptographically strong statements about the past. Although we digitally sign statements in the present, we only worry about enforcing statements made in the past. Consider, for example, the following scenarios:

- Alice executed a digital mortgage two years ago and now is in default. The bank can only sue Alice if it can prove that she was in fact the one who digitally signed the mortgage contract. (Moreover, the exact history of Alice's digital certificates may be crucial here, for her signature should remain valid even if her private key was compromised and her digital certificate revoked even just a few weeks after she signed the contract.)
- Bob signed a digital receipt for Alice's electronic payment and then shipped her a defective item. Alice must be able to prove that Bob truly was the one who signed that receipt.
- A company, CryptoTicket.com, issued a signed digital ticket to Alice for the opera, but Alice is refused entry to the performance. She needs to be able to prove that indeed CryptoTicket.com issued that ticket and was also authorized to do so.
- A company, CryptoNet.com, has published an online catalog that advertised widgets at $1 a piece on the day they accepted a digital purchase order (PO) for 100 widgets from Alice. Now the company has raised the price to $100 and is demanding $10,000 from Alice. She needs to be able to prove that $1 was the valid price for widgets on the day they accepted her digital PO.

[⋆] Research supported in part by DARPA Grant F30602-00-2-0509.

Thus, in order to enforce nonrepudiation on important contractual statements, such as in these examples, we need to have in place a mechanism for checking credentials, certificates, and published information in the past. Ideally, we would like there to be a collection of potentially untrusted directories that can answer historical queries about such items.

1.1 Problem Definition and Applications

Put more abstractly, the problem we address involves three types of parties: a trusted source, untrusted directories, and users. The *source* defines a finite set S of elements that evolves over time through insertions and deletions of elements. Each *directory*, acting as a query agent for the source maintains a copy of set S and 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. A *user* performs membership queries on the set S of the type "was element e in S at time t?" but instead of contacting the source directly, it queries one of the directories. The contacted directory provides the user with a yes/no answer to the query together with *query authentication information*, which yields a cryptographic 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 the user to check the source's signature. We call the data structure used by the directory to maintain the set S, together with the protocol for queries and updates a *persistent authenticated dictionary* (*PAD*).

The PAD abstract data type extends the usual notion of an authenticated dictionary [15], where the queries are only of the form "is element e currently in S?". Thus, a PAD has the added burden of having to archive the entire history of updates sent by the source to the directories. Moreover, the directories must be able to provide answers and proofs for queries related to any time, past or present. We show, in Fig. 1, a schematic view of a persistent authenticated dictionary.

The design of a persistent 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. More importantly, the space needed to archive historical copies of S should be small.
– *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 certainty.

In addition to the motivating examples given above, applications of persistent authenticated dictionaries include third-party publication of historical data on the Internet [3] and historical certificate revocation checking in public key

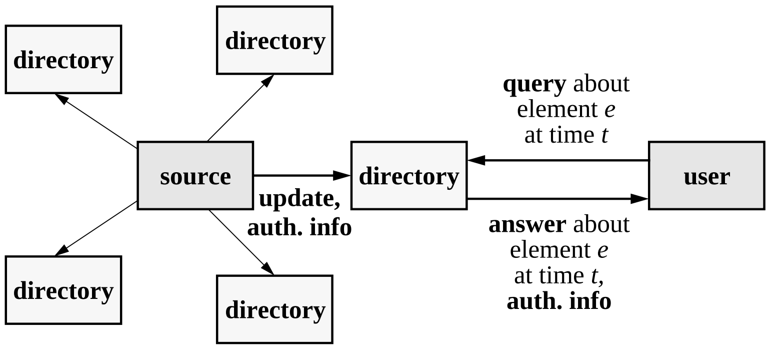
directory
directory
directory
directory
directory
source
update, auth. info
query about element e at time t
answer about element e at time t, auth. info
user

Fig. 1. Persistent authenticated dictionary.

infrastructures [9, 15, 1, 2, 8, 5]. In the third-party publication application [3], the source is a trusted organization (e.g., a stock exchange) that produces and maintains historical 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 becomes widely disseminated. The publishers store copies of the content produced by the source and process queries on such content made by the users.

In the certificate revocation application [9, 15, 1, 2, 8, 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 his authority to use a particular private key). Thus, the user of a certificate must be able to verify that a given certificate used to validate someone for a historic contract was not revoked at the time the contract was executed. To facilitate such queries, *certificate revocation directories* process historic 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 persistent authenticated dictionaries, based on efficient persistent data structures known as red-black trees and skip lists. Our structures are secure, as well as being fast and space efficient in terms of the parameters n, which denotes the current number of elements of the set S, and m, which denotes the total number of updates the source has performed.

1.2 Previous and Related Work

We are not aware of any previous work on persistent authenticated dictionaries. We summarize prior work on *ephemeral* authenticated dictionaries, where only the current version of S is maintained, and on persistent (non-authenticated) dictionaries.

Ephemeral Authenticated Dictionaries. Work has been done on ephemeral authenticated dictionaries 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 list 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: the update and query authentication information has size $\Theta(n)$. Moreover, to turn the CRL approach into a persistent authenticated dictionary requires that every CRL ever issued be archived at the directories. Thus, if m CRLs have been issued, this solution requires in the worst case quadratic $O(m^2)$ storage space at each directory. In other words, the CRL-based approach is a simple but very inefficient solution for the persistent authenticated dictionary problem.

There are other more space-efficient methods for implementing ephemeral authenticated dictionaries. But methods for adapting them to the persistent context are not obvious. The *hash tree scheme* introduced by Merkle [13] 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). Still, the only obvious way to make a hash tree persistent is to checkpoint the entire tree at each time quantum, which is clearly not space efficient.

Kocher [12] 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. Using techniques from incremental cryptography, Naor and Nissim [15] dynamize hash trees to support the insertion and deletion of elements using 2-3 trees. Other certificate revocation schemes based on variations of hash trees have been recently proposed in [2, 5, 10], as well, but do not deviate significantly from the above approaches.

Goodrich and Tamassia [6, 7] have proposed an authenticated-dictionary scheme based on the skip-list data structure that has asymptotically the same performance as [15] but it is simpler to implement. Still, like other previous solutions, their data structure is ephemeral—it only stores the most recent copy of the set S.

Persistent Data Structures. Researchers have worked on persistent data structures for other abstract data types besides authenticated dictionaries. The idea of path copying in a tree, for example, which is a component in some of our solutions, has been used in non-authenticated contexts by several researchers (e.g. Myers [14] and Reps, Teitelbaum and Demers [17]). Sarnak and Tarjan [18] proposed the node-copying method and use persistent trees to solve the planar point location problem, while Driscoll, Sarnak, Sleator and Tarjan [4] developed techniques for making linked data structures persistent. Nevertheless, none of these previous schemes for making data structures persistent have directly addressed the need for authentication or directory distribution.

1.3 Summary of Results

We present two data structures for implementing PADs, based on the dictionary data structures known as red-black trees and skip lists. Our solutions allow for element insertions and removals in the current set S to run in $O(\log n)$ time and queries in the past to run in $O(\log m)$ time. More importantly, our solutions use only $O(\log n)$ additional space per update. (Recall that n is the number of elements in the current set S and m is number of updates that have occurred so far.) Thus, our solutions are significantly more efficient than the CRL-based approach or checkpointing-based approaches, which require $\Theta(n)$ additional space for archiving S at each historic time quantum. We describe the theoretical foundations behind our solution to the persistent authenticated dictionary problem in Section 2.

In addition, we claim that our methods are simple, which is an often neglected, but important aspect of computer security solutions, for implementation correctness is as important as theoretic soundness. To support this claim, we have implemented our solutions and have performed a number of benchmarking tests, which we report on in Section 3.

2 Making Authenticated Dictionaries Persistent

We use and extend some ideas from previous work on persistent data structures to create our solutions to the persistent authenticated dictionary (PAD) abstract data type. Let us therefore begin with a quick review.

A Quick Review of Persistent Data Structures. Most of the data structures are *ephemeral* in the sense that whenever the user performs an update to them he destroys the previous version. If the previous version is retained and we can do queries to them we talk about *persistent* data structures. We can even talk about fully persistent data structures if we can make updates and not only queries to previous versions.

In more detail, the operations that a persistent data structure S supports are **find**(e, t), which determines whether element e was in S at time t, **insert**(e), which inserts element e into S (at current time), and **delete**(e), which removes element e from S (at current time).

In our case, we maintain the data structure in both the source and the directories. When the user queries a directory if an element e was in the source at some time t, the directory returns "yes" if e existed in the source at time t along with a proof of its existence, or "no" if e did not exist in the source at time t along with a proof of its nonexistence. The proof must permit the user to verify that the answer is authentic (i.e., it is as trustworthy as if it were directly signed by the source) and that it is current (i.e., it corresponds to time t).

Depending on the application we can use one of the two versions of the PADs which differ in the way they define time. In particular we can have:

- *Discrete time* where we can think that the dictionary holds several versions whose time is numbered sequentially with integers starting at 0. When a user performs a query, t equals the time of one of the versions, and the dictionary searches the corresponding version and returns an answer that allows the user to verify that indeed the dictionary searched version at time t.
- *Continuous time* where we can define the time over any set for which there exist a complete order. The versions of the dictionary are ordered according to that order. Whenever a user queries the directory about time t, the directory must provide an answer corresponding to the latest dictionary version at time t' that is earlier than t; moreover, it must provide information for the user to verify that indeed t' is the time of the version that should have been queried and that does not exist a version at time t'' such that $t' < t'' \leq t$.

We have developed two data structures that implement PADs, one based on the red-black tree and the other based on the skip-list data structures.

2.1 PADs Based on Red-Black Trees

We denote the element stored at a node v as elem(v).

The version of the red-black tree we use, has all the values stored only at the external nodes. We use the internal nodes to make queries to the tree and to store authentication information. The value stored at each internal node equals the maximum value of the left subtree; the values at the right subtree are greater than that value. When we want to make an insertion to the tree of an element with value e that does not exist into the tree, first we find the external node v containing the minimum element elem(v), such that elem(v) $> e$. We then create a new internal node w to replace v and we set as the left child of w a new external node u such that elem(u) $= e$ and as the right child the node v.[1] Finally we perform the necessary reconstructions and recolorings. A deletion of a node is performed in the opposite manner.

The persistent red-black tree is a modification of the red-black tree. Each node u has also a field time(u) where we store the timestamp of its creation. Each addition or deletion to the tree does not change the current tree nodes but instead it adds new nodes with timestamp value that of the current time.

In Fig. 2 we can see an example of an instance of a red-black tree. The bold lines denote black nodes and edges, while the normal ones denote red. The label at the top of each node (in Fig. 2 all are 0 for simplicity) denotes the timestamp value.

Suppose we want to add a new element ($e = 18$). We can see the new tree created in Fig. 3, where the timestamp of the new nodes has taken the current time value (here just 1 for simplicity). Because at each update operation we copy the whole path from a node up to the root, this method is called *path-copying* method. Note that we have as many signed roots as update operations. Details of the operations follow in the next paragraphs.

[1] There is an exception for the value that is larger than the largest element at the tree.

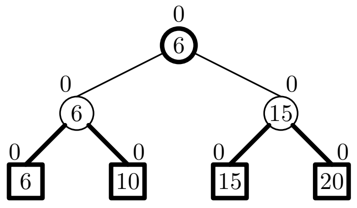

Fig. 2. The tree before the insertion.

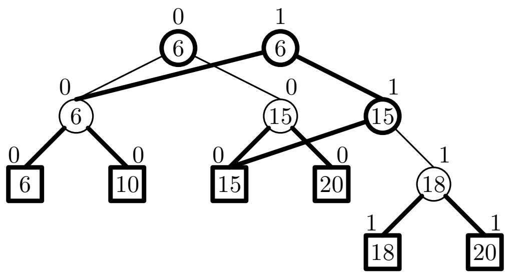

Fig. 3. The tree after the insertion of element 18.

In our scheme we use an authenticated persistent red-black tree. Each node u has stored one more value, auth(u), that is used for authentication; auth(u) equals elem(u) if u is an external node, and the concatenation of the auth() values of u's children if u is an internal node. h must be a collision-free hash function. Finally the root of every version of the tree (for the different update times) is signed. In summary, each node u contains three fields: elem(u), time(u), and auth(u).

We describe now the details of the operations.

Insertion. At every **insert** operation, the old version of the tree doesn't change at all (except for the colors of the nodes and edges as we will see later). We find the external node u that will be the father of the new node. Then we make a copy of the nodes on the path, from the root down to u to which we assign a timestamp value equal to the current time. We perform then the insertion by creating a new node as we saw before. We denote here a node x as x_0 for the old version, and x_1 for the new version that is copied. If along the path, a node v that is copied has a child w that is also copied, we add the edge from the node v_1 to w_1. For the rest of the edges of v_0 to some other node z_0 we add an edge from v_1 to z_0. We can see an example in Figures 2, 3.

Since we have a red-black tree, we must color the new nodes. Every node v_1 will be colored with the same color with v_0, and the insertion will be done normally. After the insertion we may have to perform some transformations on the tree, namely some rotations and recolorings. The recolorings are easy, since we can overwrite the old colors of any node of any previous version of the tree. The colors are used only for the update operations, so we do not need to keep track of the old ones—we do not make updates in the past. Rotations are also simple; they will be done only on the new version of the tree and they will affect only the new nodes and the links originating from them.

Finally we must compute the values elem() and auth() for all the new nodes created. These operations are straightforward and we can perform them as we create the new nodes, in $O(\log n)$ time. In particular, we must compute only two elem() values, the one of the new external node, and that of its parent (which has the same value). We must compute the auth() values of all the $O(\log n)$ new nodes. Finally, if r_1 is the root of the last version of the tree (just created) and r_0 is the root of the just previous version, then the source signs $(\text{auth}(r_1), \text{time}(r_1), \text{time}(r_0))$. (The previous timestamp is necessary only in the case of continuous time; we give more details later.)

Deletion. A **delete** operation is similar to an **insert**. We copy the path from the root down to the node to be deleted, we remove the nodes (one external and one internal node will be deleted), and then perform the necessary rotations and recolorings. Here we must be careful, since the rotations in this case may also affect nodes not existing in the new path created. So before the rotation, these nodes must also be copied.

During the deletion we may require to perform in the worst case at most a logarithmic number of recolorings and two rotations. In total, except for the path of the node deleted, it may be necessary to copy at most 4 additional nodes; therefore, we copy only a logarithmic number of nodes.

Finally we compute the necessary elem() and auth() values and sign the new root, by the method we described in the *Insertion* paragraph. Again, we must compute all the auth() values of the $O(\log n)$ new nodes, and the only internal elem() value that we must alter is the value of the node that followed the removed external node in the infix traversal of the tree.

Query. A **find** operation can query the PAD at any time in the past (or present) to find if an element e existed at that time. Say that we want to query for time t_1 and we have the two consecutive tree versions for times t_0 and t_2, where $t_0 \leq t_1 < t_2$, assuming that t_2 exists. In the following, assume that r_0 is the root with timestamp t_0, r_2 is the root with timestamp t_2 and r_3 is the root with the largest timestamp that is less than t_0, if it exists. In order to search for e we work on the tree originating from the root r_0 of time t_0. We distinguish two cases:

– **The element e exists in the tree at time t_0:** The dictionary returns to the user the answer "yes", the root r_0 signed by the source (i.e., the signature of $(\text{auth}(r_0), \text{time}(r_0), \text{time}(r_3)))$, the root r_2 signed by the source, that is, the signature of $(\text{auth}(r_2), \text{time}(r_2), \text{time}(r_0))$, the element e, and the sequence $Q(e)$ of the hashes of the siblings of the nodes of the path from the root of the tree to e. The user can verify that the answer is valid from the sequence $Q(e)$ and the source's signature of the root.

– **The element e does not exist in the tree at time t_0:** In this case, let e' be the maximum element that exists in the tree and is smaller than e, and e'' the minimum element that exists in the tree and is greater than e. The dictionary returns to the user the answer "no", the root r_0 with timestamp t_0 and the root r_1 with timestamp t_1, signed by the source like in the previous case, the elements e' and e'', and the sequences $Q(e')$ and $Q(e'')$ defined as before. The user can verify now that both e' and e'' exist in the tree and that they are consecutive external nodes (hence e is not between them).

The root r_2 has to be included only in the case of continuous time. The reason is to prove that there is not another version of the tree at time t_4 such that $t_0 < t_4 \leq t_1 < t_2$ which may have different information concerning C. The user can now verify that the version that immediately follows the one that corresponds to time t_0 is at time t_2 since it is included in the signature.

Summarizing the above results, we have the following theorem:

Theorem 1. *A persistent authenticated dictionary can be implemented with a persistent authenticated red-black tree. The time needed for an update operation is $O(\log n)$ and for a query operation is $O(\log m)$, where n is the number of elements of the last version and m is the total number of versions. The total space requirement is $O\left(\sum_{i=1}^{m} \log n_i\right)$, where n_i is the number of elements of version i.*

Proof. We perform an update to the last version of the tree, which has n elements. The update operation requires the addition of $O(\log n)$ new nodes, $O(\log n)$ changes to the tree structure (including recolorings) and the computation of $O(\log n)$ new hash values. Therefore, it can be performed in time $O(\log n)$.

A query on version i requires a binary search on the root of the trees to find the appropriate version of the tree, which takes time $O(\log m)$, a search in the tree to find the appropriate node(s), which needs $O(\log n_i)$ steps, and the creation and return of the response, which is done in $O(\log n_i)$ steps. Since n_i cannot be greater than m, the overall time requirement of a query is $O(\log m)$.

The space required for the i-th update is $O(\log n_i)$: the insert operations need to copy only the nodes of the path of the new node, while the delete operations may need to copy at most 4 additional nodes. By the property of red-black trees the total path length is $O(\log n_i)$, hence, the total space is $O\left(\sum_{i=1}^{m} \log n_i\right)$. $\square$

Note that the above time complexity results hold when we have continuous time. In the case of discrete time, we can find the appropriate root for a query just by a table lookup in constant time, and in this case the total query time is $O(\log n_i)$. Also, note that when we query the last version, the time is reduced to $O(\log n)$, as in the ephemeral authenticated dictionary.

2.2 PADs Based on Skip Lists

The *skip-list* data structure [16] is an efficient means for storing a set S of elements from an ordered universe. Briefly, a skip list consists of a collection of linked lists $S_0, S_1, \ldots, S_k$ (where S_i contains a randomly selected subset of the items in S_{i-1}, plus two additional values $-\infty$ and $+\infty$), and links between them. We can see an example in Fig. 4. With high probability (whp[2]), skip lists have the same asymptotic performance as red-black trees. However, experimental studies (e.g., see [16]) have shown that they often outperform in practice 2-3 trees, red-black trees, and other deterministic search tree structures.

[2] We use "whp" to refer to a probability that is at least $1 - 1/n^c$ for some constant $c \geq 1$.

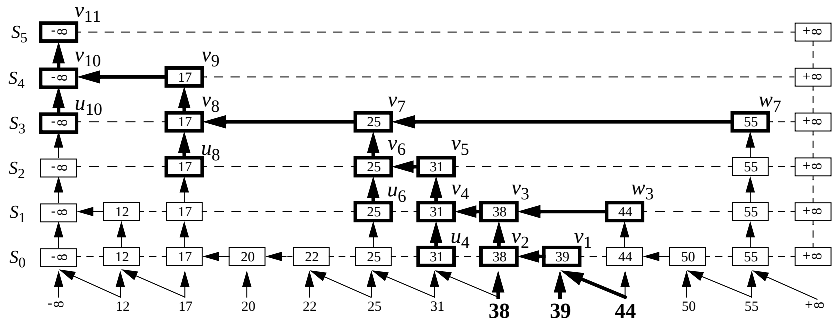

Fig. 4. (a) An example of a skip list. All the lines indicate links of the data structure. (b) The answer authentication information for the presence of element $x = 39$ (and for the absence of element 42) consists of the signed time-stamped value $f(v_{11})$ of the source element and the sequence $Q(x) = (44, 39, 38, f(w_3), f(u_4), f(u_6), f(w_7), f(u_8), f(u_{10}))$. 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 [6, 7]. The arrows denote the flow of authentication information.

Commutative Hashing. To simplify the verification process, *commutative cryptographic hash functions* are introduced in [6]. A hash function h is commutative if $h(x, y) = h(y, x)$ for all x and y. A commutative hash function is *commutatively collision resistant* [6] 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 g that is collision resistant in the usual sense, the commutative hash function, $h(x, y) = g(\min\{x, y\}, \max\{x, y\})$ is commutatively collision resistant if x and y have the same length [6].

Authenticated Dictionary Based on a Skip List. Using the skip-list data structure and commutative hashing, we can design a scheme for authenticated dictionaries, similar to the one based on balanced trees. It has the same asymptotic performance but it avoids many complications that arise in the implementation of the hash trees, leading to easier and less error-prone implementations. An example of the data structure and of an authenticated query is in Fig. 4. For more details refer to [6, 7].

PAD Based on a Skip List. We can apply the path-copying idea to the authenticated skip list and create a persistent authenticated skip list. First, we make the following observation, which allows us to have an efficient implementation: *for the operations supported by the skip list, some of the nodes and links are unnecessary and can be omitted.* The skip list then looks like that of Fig. 5. We can see now that in this form the skip list is essentially a binary tree (whose root is the node in the highest-level list with the value $-\infty$) and so we can apply the path-copying method. We have to be careful however to copy all the nodes whose out-links or authentication information change.

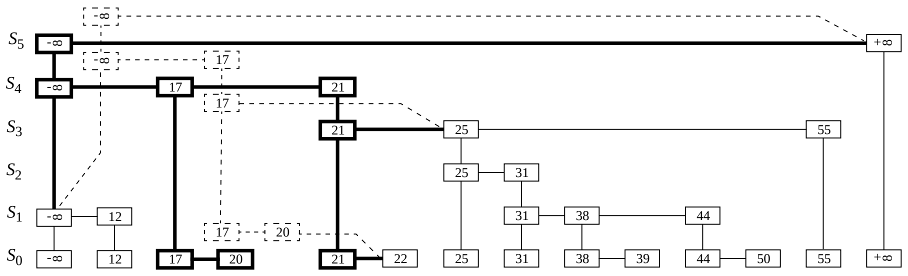

Fig. 5. (a) A tree-like representation of a skip list. (b) Insertion of element 21 with height 5. The new nodes and links are bold, while the dashed ones exist only in the previous version.

It is important to mention that when we do an update, we do not change the existing nodes of the skip list—we only add new nodes. Therefore, by storing links to the appropriate root nodes we can make queries to any version of the dictionary. We describe now in detail the operations.

Insertion. Assume that we want to insert element e and that e' is the largest element that is smaller than e. On the search path p we will reach element e' that belongs to the list S_0. We insert the element e and hence the authentication data of e' in S_0 changes; the change of the authentication data propagates up to the root node *along the path p that we followed to reach e'*. In other words, the nodes of p (and of course the nodes of the new element) are the only nodes whose authentication data changes. Finally, some of the links to the right of the nodes on p—basically, those that belong to nodes that are lower than the height of the new element—must be set to null and be replaced by links that originate from the new elements. Therefore, the new nodes that we create are exactly those that belong to p plus those of the new element e. Fig. 5 shows an example of an insertion.

The number of the new nodes that we create equals the length of the search path to the element e' plus the number of the new nodes for the new element. Both of these are $O(\log n)$ whp. The number of hashes that we have to compute is at most equal to the number of the new nodes (although for some of the nodes the authentication data is equal to that of the node below and does not have to be recomputed).

Deletion. Assume that we want to remove element e from the skip list and that e' is the element immediately preceding e. We follow the path p to element e'. Since we remove the element next to e' (i.e., e), the authentication data of e' in S_0 changes. Again, the change propagates up to the root node only along p, and the only links to the right that must change are in p. Therefore, the only nodes that we must duplicate are those in p, which with high probability are $O(\log n)$. Also, the number of the hashes to compute equals at most the number of the new nodes and so it is $O(\log n)$ whp.

Query. Assume that we want to find if element e existed in the skip list at time t. First, we find the appropriate root node r of the skip list. Like in the red-black trees implementation, in the case of discrete time, we can find the start node by a table lookup in constant time, while for continuous time we can find it in time $O(\log m)$ using, for example, binary search. Since in the update operations we never change existing nodes but we only add new ones, the skip list originating from r is exactly the one we had at time t. Therefore, we can perform the query simply by following the procedure of the ephemeral authenticated skip list; the time needed is $O(\log n_t)$, where n_t is the number of elements of the skip list of version at time t.

Thus, the total time needed is $O(\log n_t)$ for discrete time and $O(\log m)$ for continuous-time.

Theorem 2. *A persistent authenticated dictionary can be implemented with a persistent authenticated skip list. With high probability, the time needed for an update operation is $O(\log n)$ and for a query operation $O(\log m)$, where n is the number of elements of the last version and m is the total number of versions. After an update, the space used by the data structure increases by $O(\log n)$ whp.*

2.3 Security

The security of both techniques for PADs is based on Merkle's scheme for digital signatures [13] which is the cryptographic basis for previous approaches for authenticated dictionaries as well [12, 15, 2, 5, 10, 6].

One possible way for the directory to cheat is to try to return to the user a response corresponding to a different version than the specified. In the discrete-time setting, the response of the directory must contain a signed statement of the source that contains both the number of the version that the user specified at his query and the hash value of the corresponding root. In the continuous-time setting, as we explained in section 2.1, the response of the directory allows the user to verify that indeed the hash of the correct root is returned. Hence, in both cases, the user is assured (based only on the trust in the source) that the hash value of the root that he received is that of the correct version.

Assuming that the directory returns the authentication information corresponding to the correct version, our schemes for PADs are as secure as the corresponding ephemeral ones: In order for the directory to cheat, it has to find a sequence of $O(\log n)$ values, which, if are hashed in succession, produce the same output as the values that actually exist in the dictionary; if we use a collision-resistant cryptographic hash function, such as MD5 or SHA1, this task is infeasible. For more details, refer to [13, 15, 6].

2.4 Extensions

We present here three useful extensions of the preceding schemes that can be implemented with minor overhead.

First, instead of performing membership queries, we can have a value associated with each element to be returned by a query operation. The element defines the position of the pair in the data structure, and the authentication data depends on both the element and the value. The additional operation that we allow is changing the value that corresponds to an element and we can implement it easily by only copying the old nodes (no restructural operations are necessary).

Second, we can apply the path-copying technique to other types of authenticated balanced trees, such as AVL or 2-3 trees.

Finally, we can combine the advantages of the PADs with the efficiency offered by B-trees when we store them in secondary memory. This yields an efficient authenticated data structure that is persistent in two ways: it holds the whole history of updates, and remains in permanent, external memory—usually a disk. The idea is to allocate a whole new disk block when we create a new node, and in some time instances to compact older versions, by putting two small nodes in the same block, to save disk space.

3 Experimental Results

We have designed a comprehensive set of Java interfaces describing the PAD abstract data type and auxiliary data types (source, directory, user, etc.). Also, we have conducted an experiment on the performance of our data structures on randomly generated sets of 256-bit integers in dictionaries ranging in size from 0 to $500,000$. For each operation, the average was computed over $10,000$ trials. The experiment was conducted on a 400MHz Sun Ultra Enterprise with 2GB of memory running Solaris. The Java Virtual Machine was launched with a 1GB maximum heap size. Cryptographic hashing was performed using the standard Java implementation of the MD5 algorithm. The signing of the update authentication information by the source and the signature verification by the user were omitted from the experiment.

We compare the execution time of the ephemeral authenticated skip list with that of the persistent one. The highest level of a tower was limited to 20. The results are shown in Fig. 6. We can see that the query time is almost the same for both the ephemeral and persistent versions, while the insertion time is slightly lower for the ephemeral one, since it allows some optimizations in the implementation.

After noticing that most of the update time (about 94%) is consumed by the hashing computations, we have determined the number of hashes required for an update in a skip list (it is the same for both the ephemeral and the persistent versions) and in a red-black tree. We present the experimental results in Fig. 6. Furthermore, we observe that the number of hashes computed at each update equals the number of new nodes created in a red-black tree, and is about $0.25 \log_2 n$ less than the number of new nodes created in a skip list. Therefore, the number of hashes provides a measure of the space requirement of our data structures. From Fig. 6, we can see that the red-black tree requires fewer hashes than the skip list. This is justified theoretically, since for a skip

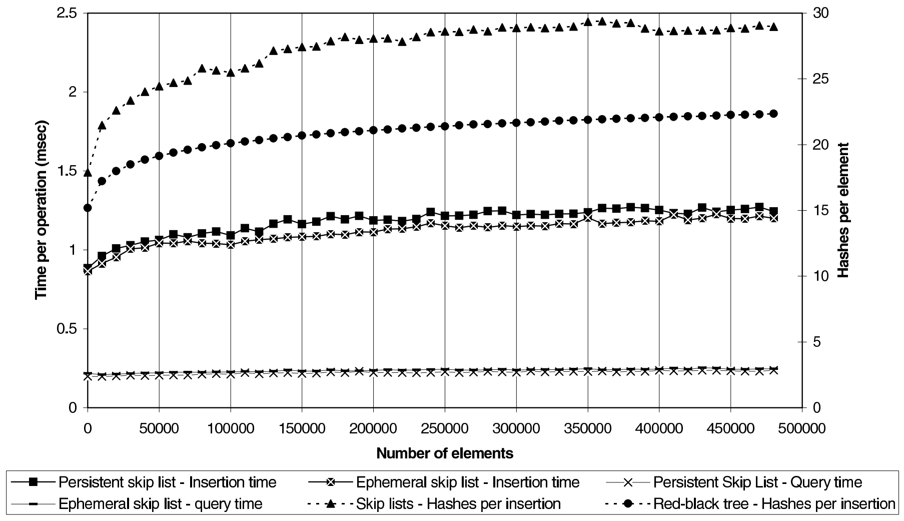

Fig. 6. (a) Average insertion and query times for ephemeral and persistent authenticated skip lists. The results for deletions are similar to those for insertions. (b) Average number of hash computations per insertions for skip lists (both persistent and ephemeral) and red-black trees.

list with n elements, the average number of comparisons performed in a search (which is a lower bound of the number of hashes that we have to compute for an update) is $\frac{3}{2}\log_2 n + \frac{7}{2}$ [16], while for a balanced tree with n external nodes, it is about $\log_2 n$ [11].

4 Conclusions

We have introduced the notion of persistent authenticated dictionaries and justified their usefulness. We have also presented two techniques for implementing persistent authenticated dictionaries that support updates in $O(\log n)$ time and queries in $O(\log m)$ time, and use $O(\log n)$ space per update (n is the number of elements in the current set S and m is the number of updates that have occurred so far). The technique based on red-black trees has better running time and space requirement than the one based on skip lists, but its implementation is more complex. We leave as an open problem whether persistent authenticated dictionaries can be implemented with constant space per update.

Acknowledgments

We would like to thank Andrew Schwerin for many useful conversations regarding the contents of this paper.

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

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

[4] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38:86–124, 1989.

[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. Efficient authenticated dictionaries with skip lists and commutative hashing. Technical Report, Johns Hopkins Information Security Institute, 2000.

[7] M. T. Goodrich, R. Tamassia, and A. Schwerin. Implementation of an authenticated dictionary with skip lists and commutative hashing. In *Proc. 2001 DARPA Information Survivability Conference and Exposition*, volume 2, pages 68–82, 2001.

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

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

[10] H. Kikuchi, K. Abe, and S. Nakanishi. Performance evaluation of certicate revocation using k-valued hash tree. In *Proc. ISW'99*, volume 1729 of *LNCS*, pages 103–117. Springer-Verlag, 1999.

[11] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, 1973.

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

[13] 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.

[14] E. W. Myers. Efficient applicative data types. In K. Kennedy, editor, *Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages*, pages 66–75, Salt Lake City, UT, Jan. 1984. ACM Press.

[15] 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.

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

[17] T. Reps, T. Teitelbaum, and A. Demers. Incremental context-dependent analysis for language-based editors. *ACM Transactions on Programming Languages and Systems*, 5(3):449–477, July 1983.

[18] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29(7):669–679, July 1986.