An Efficient Dynamic and Distributed Cryptographic Accumulator

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

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

January 13, 2001

Abstract

One-way accumulators allow insecure directories to provide cryptographically secure answers to membership queries on a set maintained by a trusted third party (TTP). Such usage implements the authenticated dictionary abstract data type and it finds applications in certificate management for public key infrastructure, and the publication of data collections on the Internet. From the user's perspective, particularly in wireless applications, the optimal authenticated dictionaries provide small verifications derived from data signed by the TTP and involve computations that are simple to program and perform. We describe a new scheme for authenticated dictionaries that supports efficient incremental updates of the underlying set and optimal constant-time verification by the user. Our scheme is based on the dynamic maintenance of a one-way accumulator function over the set elements.

1 Introduction

Because of network latency and the risk of denial of service attacks, Internet services, such as Web servers, are often replicated to mirror sites. Thus, a user will in general be much closer to one of these mirror sites than to the source of the repository, and will therefore experience a faster response time from a mirror than it would by communicating directly with the source. In addition, by off-loading user servicing from the information source, this distributed scheme allows for load balancing across the mirror sites, which further improves performance.

An information security problem arising in the replication of data to mirror sites is the authentication of the information provided by the sites. Indeed, there are applications where the user may require that data coming from a mirror site be cryptographically validated as being as genuine as they would be had the response come directly from the source. For example, a financial speculator that receives NASDAQ stock quotes from the Yahoo! Web site would be well advised to get a proof of the authenticity of the data before making a large trade.

For all applications, and particularly for applications in wireless computing, we desire solutions that involve short responses from a mirror site that can be quickly verified with low computational overhead.

1.1 Problem Definition

More formally, the problem we address involves three parties: a trusted source, an untrusted directory, and a user. The *source* defines a finite set S of elements that evolves over time through insertions and deletions of items. The *directory* maintains a copy of set S. It receives time-stamped updates from the source together with *update authentication information*, such as signed statements about the update and the current elements

of the set. The *user* performs membership queries on the set S of the type "is element e in set S?" but instead of contacting the source directly, it queries the directory. The directory provides the user with a yes/no answer to the query together with *query authentication information*, which yields a proof of the answer assembled by combining statements signed by the source. The user then verifies the proof by relying solely on its trust in the source and the availability of public information about the source that allows to check the source's signature. The data structure used by the directory to maintain set S, together with the protocol for queries and updates is called an *authenticated dictionary* [27]. Figure 1 shows a schematic view of an authenticated dictionary.

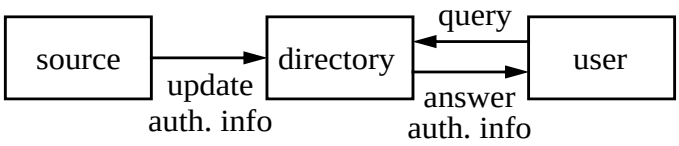

Figure 1: Authenticated dictionary.

The design of an authenticated dictionary should address several goals. These goals include low computational cost, so that the computations performed internally by each entity (source, directory, and user) should be simple and fast, and low communication overhead, so that bandwidth utilization is minimized. Since these goals are particularly important for the user, we say that an authenticated dictionary is *size oblivious* if the response and verification given to each user does not depend in any way on the number of items in the dictionary. We are most interested in this paper on solutions to the authenticated dictionary problem that are size oblivious. Such solutions are ideally suited for wireless applications, where user devices have low computational power and low bandwidth. In addition, size-oblivious solutions add an extra level of security, since the size of the dictionary is never revealed to users.

1.2 Applications

Authenticated dictionaries have a number of applications, including scientific data mining (e.g., genomic querying [18] and astrophysical querying [22, 10, 23]), geographic data servers (e.g., GIS querying), third-party data publication on the Internet [13], and certificate revocation in public key infrastructure [19, 26, 27, 1, 11, 17, 16]. They are also useful for time stamping online documents, provided the source publishes his signed summary information to a trusted and dated archive (such as the New York Times Classified Ads).

and used to process various types of queries. For example, genomic querying [18] tends to be comprised of text searches for various patterns, such as substrings or super-strings. Also, astrophysical querying, such as in the object catalog of the Sloan Digital Sky Survey [22, 10, 23], tends to be in the form of range searches for points lying within given geometric shapes. Given the significant scientific and economic benefits that can result from such querying, these users need to be certain that the results of their queries are accurate and current.

Another type of data replication problem arises in geographic information systems (GIS) applications, where a large collection of geographic data must be replicated to several web sites so as to provide data querying capability to a widely-dispersed population of users. Such queries are typically geographic or geometric in nature, and often need to be trustworthy, as critical navigation plans are often based on such queries. This application is well-suited for a size-oblivious authenticated dictionary, as the navigational device is likely to be a palm computer equipped with a Global Positioning Satellite (GPS) receiver.

In the third-party publication application [13], the source is a trusted organization (e.g., a stock exchange) that produces and maintains integrity-critical content (e.g., stock prices) and allows third parties (e.g., Web portals), to publish this content on the Internet so that it widely disseminated. The publishers

store copies of the content produced by the source and process queries on such content made by the users. In addition to returning the result of a query, a publisher also returns a proof of authenticity of the result, thus providing a validation service. Publishers also perform content updates originating from the source. Even so, the publishers are not assumed to be trustworthy, for a given publisher may be processing updates from the source incorrectly or it may be the victim of a system break-in.

In the certificate revocation application [19, 26, 27, 1, 11, 17, 16], 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 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.

The rest of this paper is organized as follows. In Section 2, we review previous work on authenticated dictionaries, especially in the context of certificate revocation, and review the exponential accumulator. We present our technique in Sections 3–5 through successive refinements of a simple scheme. Section 3 shows how a straightforward application of the exponential accumulator yields a simple authenticated dictionary with constant verification time but linear update and query time. An improvement of this scheme that gives constant query and verification but linear update time is described in Section 4. This improvement, called *precomputed accumulations*, consists of an efficient precomputation by the source of data used by the directories to speed-up query processing. In Section 5, we present our complete technique, which uses a second improvement, called *parameterized accumulations*, to achieve a variety of tradeoffs between the query and update times, while preserving constant verification time by the user. For example, we can balance the two times and achieve $O(\sqrt{n})$ query and update time. Finally, concluding remarks are given in Section 6.

2 Preliminaries

Before we present our technique for authenticated dictionaries, we review previous work on authenticated dictionaries and discuss some cryptographic concepts used in our approach.

Throughout the rest of this paper, we denote with n the current number of elements of the set S stored in the authenticated dictionary. Also, we describe the validation of positive answers to membership queries (i.e., validating $x \in S$). The validation of negative answers can be handled with a standard method, as discussed in Section 6.

2.1 Previous Work

Authenticated dictionaries are related to research in distributed computing (e.g., data replication in a network [5, 21]), data structure design (e.g., program checking [6, 8, 9, 29] and memory checking [7, 14]), and cryptography (e.g., incremental cryptography [2, 3, 14, 15]). The use of one-way accumulators (which we review in the next subsection) originates with Benaloh and de Mare [4]. They show how to utilize an exponential one-way accumulator, which is also known as an RSA accumulator, to summarize a collection of data so that user verification responses have constant-size. As we note in this paper, such a solution can be used to implement an authenticated dictionary, but in a dynamic setting, where items are inserted and deleted, the standard way of utilizing the exponential accumulator is inefficient. Several other researchers have also noted the inefficiency of this implementation in a dynamic setting (e.g., see [28]). Indeed, our solution can be viewed as refuting this previous intuition to show that a more sophisticated utilization of the exponential accumulator can be made to be efficient even in a dynamic setting.

Previous additional work on authenticated dictionaries has been conducted primarily in the context of certificate revocation. The traditional method for certificate revocation (e.g., see [19]) is for the CA (source) to sign a statement consisting of a timestamp plus a hash of the set of all revoked certificates, called *certificate revocation list* (CRL), and periodically send the signed CRL to the directories. A directory then just forwards that entire signed CRL to any user who requests the revocation status of a certificate. This approach is secure, but it is inefficient, for it requires the transmission of the entire set of revoked certificates for both source-to-directory and directory-to-user communication. This scheme corresponds to an authenticated dictionary where both the update authentication information and the query authentication information has size $\Theta(n)$. Thus, this solution is clearly not size-oblivious, and even more recent modifications of this solution, which are based on delta-CRLs [12], are not size-oblivious. Because of the inefficiency of the underlying authenticated dictionary, CRLs are not a scalable solution for certificate revocation.

Micali [26] proposes an alternate approach, where the source periodically sends to each directory the list of all issued certificates, each tagged with the signed timestamped value of a one-way hash function (e.g., see [28]) that indicates if this certificate has been revoked or not. This approach allows the system to reduce the size of the query authentication information to $O(1)$ words: namely just a certificate identifier and a hash value indicating its status. Unfortunately, this scheme requires the size of the update authentication information to increase to $\Theta(N)$, where N is the number of all nonexpired certificates issued by the certifying authority, which is typically much larger than the number n of revoked certificates. It is size-oblivious for immediate queries, but cannot be used for time stamping for archiving purposes, since no digest of the collection is ever made.

The *hash tree* scheme introduced by Merkle [24, 25] 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. A hash tree T for a set S stores the elements of S at the leaves of T and a hash value $h(v)$ at each node v, which combines the hash of its children. The authenticated dictionary for S consists of the hash tree T plus the signature of a statement consisting of a timestamp and the value $h(r)$ stored of the root r of T. An element x is proven to belong to S by reporting the values stored at the nodes on the path in T from the node storing x to the root, together with the values of all nodes that have siblings on this path. Thus, this solution is not size-oblivious, since the length of this path depends on n. Kocher [20] 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, by storing intervals instead of individual elements. As we note in Section 6, such an interval approach can also be applied to the exponential accumulator.

Using techniques from incremental cryptography, Naor and Nissim [27] 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, but the directory-to-user communication remains at $O(\log n)$. Thus, their solution is still not size oblivious.

Other certificate revocation schemes based on variations of hash trees have been recently proposed in [11, 16], but like the static hash tree, these schemes are also not size oblivious.

2.2 One-Way Accumulators

An important cryptography concept for our work is that of one-way *accumulator* functions [4, 28]. Such a function allows a source to digitally sign a collection of objects as opposed to a single document.

The most common form of one-way accumulator is defined by starting with a "seed" value y_0, which signifies the empty set, and then defining the accumulation value incrementally from y_0 for a set of elements $S = \{x_1, \cdots, x_n\}$, so that $y_i = f(y_{i-1}, x_i)$, where f is a one-way function whose final value does not depend on the order of the x_i's (e.g., see [4]). In addition, one desires that $|y_i|$ not be much larger to represent than

$|y_{i-1}|$, so that the final accumulation value, y_n, is not too large. Because of the commutative nature of f, a source can digitally sign the value of y_n so as to enable a third party to produce a short proof for any element x_i belonging to S—namely, swap x_i with x_n and recompute y_{n-1} from scratch—the pair (x_i, y_{n-1}) is a cryptographically-secure assertion for the membership of x_i in set S.

A well-known example of a one-way accumulator function f is the *exponential accumulator*,

$$\exp(y,x) = y^x \bmod N, \tag{1}$$

for suitably-chosen values of the seed y_0 and modulus N [4]. In particular, choosing $N = pq$ for two strong primes p and q makes the accumulator function exp as difficult to break as RSA cryptography [4].

The difficulty in using the function exp in the context of authenticated dictionaries is that it is not associative; hence, any updates to set S require significant recomputations. Indeed, some have mentioned the challenge of using the exponential accumulator function in an incremental setting, where items in the set S are inserted and removed over time.

2.3 Implications of Euler's Theorem

There is an important technicality involved with use of the exp function, namely in the choice of the seed $a = y_0$. In particular, we should choose this base of the exponent to be relatively prime with p and q. This choice is dictated by Euler's Theorem (e.g., see [28]), which states

Theorem 1 (Euler's Theorem): $a^{\phi(N)} \bmod N = 1$, *if* $a > 1$ *and* $N > 1$ *are relatively prime.*

Since a and N are relatively prime in our use of the accumulator function exp, the following well-known corollary to Euler's Theorem will prove useful.

Corollary 2: *If* $a > 1$ *and* $N > 1$ *are relatively prime, then* $a^x \bmod N = a^{x \bmod \phi(N)} \bmod N$, *for all* $x \geq 0$.

Proof: Suppose that $a > 1$ and $N > 1$ are relatively prime. Furthermore, let $x \geq 0$ be given. Define $z = x \bmod \phi(N)$ and let $y = x - z$. Thus, y is a multiple of $\phi(N)$; that is, $y = b\phi(N)$ for some integer $b \geq 0$. By Euler's Theorem, this implies that $a^y \bmod N = 1$. Therefore,

$$\begin{aligned} a^x \bmod N &= a^{y+z} \bmod N \\ &= a^y a^z \bmod N \\ &= (a^y \bmod N) a^z \bmod N \\ &= a^z \bmod N \\ &= a^{x \bmod \phi(N)} \bmod N. \end{aligned}$$

■

One implication of this corollary to the authenticated dictionary problem is that the source should never reveal the values of the prime numbers p and q. Such a revelation would allow a directory to compute $\phi(N)$, which in turn could result in a false validation at a compromised directory. So, our approach takes care to keep the values of p and q only at the source.

The challenge to using the exponential accumulator function, exp, for an authenticated dictionary is that the straightforward approach to its use, particularly for updates, is inefficient. In this paper we show how to significantly improve upon the performance of this straightforward approach. For completeness, however, let us first briefly review the straightforward approach before we describe improvements.

3 A Straightforward Scheme

Let $S = \{x_1, x_2, \ldots, x_n\}$ be the set of items stored at the source. The source chooses secure primes p and q that are suitably large. It then chooses a suitably-large base a that is relatively prime to $N = pq$. The source broadcasts the values of a and N to the directories, but it keeps the values p and q secret. Also, it computes the value $A = a^{x_1 x_2 \cdots x_n} \bmod N$ and broadcasts to the directories a signed message (A,t), where t is a current timestamp.

3.1 Query

To verify that some query item x_i is in S, the directory computes the value

$$A_i = a^{x_1 x_2 \cdots x_{i-1} x_{i+1} \cdots x_n} \bmod N. \tag{2}$$

That is, A_i is the accumulation of all the values in S besides x_i. After computing A_i, the directory then returns to the user A_i, N, and the signed pair (A,t).

3.2 Validation

The user checks that t is current and that (A,t) is indeed signed by the source. Then it computes $A_i^{x_i} \bmod N$ and compares it to A. If $A = A_i^{x_i} \bmod N$, then the user is reassured of the validity of the answer. Indeed, it is generally accepted to be computational infeasible for someone who does not know the values of p and q to compute a value B such that $A = B^{x_i} \bmod N$ when $x_i \notin S$. In particular, it is computational infeasible for the directory to provide a false justification for some element belonging to S when in fact this is not the case. Even correctly computing A_i for a user is no trivial task for the directory, for it must perform $n-1$ exponentiations to answer a query. Making the simplifying assumption that the number of bits needed to represent N is independent of n, the computation to answer a single query takes $O(n)$ time. Note that the message sent to the user has constant size; hence, this scheme is size oblivious.

3.3 Updates

For updates, this simple approach has an asymmetric performance, with insertions being much easier than deletions. To insert a new element x_{n+1} into the set S, the source simply computes $A^{x_{n+1}} \bmod N$, as its new A value, and the source sends a new (A,t) signed pair to the directories in the next time interval. That is, an insertion takes $O(1)$ time. The deletion of an element $x_i \in S$, on the other hand, will in general require the source to recompute the new value A by performing $n-1$ exponentiations. That is, a deletion takes $O(n)$ time. The performance of this straightforward use of the exponential accumulator is summarized in Table 1.

space	insertion time	deletion time	update info	query time	query info	verify time
$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(1)$

Table 1: Straightforward implementation of an authenticated dictionary using an exponential accumulator.

The above query time bound is generally considered too slow to be efficient for processing large numbers of queries. Fortunately, we describe an alternative approach that can answer queries much faster.

4 Precomputed Accumulations

We present a first improvement that allows for fast query processing. We require the directory to store each of the A_i accumulator values, as defined in (2). Thus, to answer a query, a directory looks up the A_i value, rather than computing it from scratch, and it then completes the transaction as described in the previous section. That is, a directory can under this precomputed accumulations scheme process any query in $O(1)$ time, with the computation for a user remaining unchanged.

Unfortunately, a standard way of implementing this approach is inefficient for processing updates. In particular, a directory now requires $O(n^2)$ time to process a single insertion or deletion, for after such an update the directory must recompute all the A_i values from scratch. That is, recomputing any single A_i at a directory after an update requires $n-1$ exponentiations. Thus, at first blush, this precomputed accumulations approach appears to be quite inefficient when updates to the set S are required.

We can process updates much faster than $O(n^2)$ time, however, by enlisting the help of the source. Our method in fact can be implemented in $O(n)$ time by a simple two-phase approach. The details for the two phases follows.

4.1 Phase One

Let S be the set of n items stored at the source after performing all the insertions and deletions required in the previous time interval. Build a complete binary tree T "on top" of the elements in S, so that each leaf of T is associated with an element of S. In the first phase, we perform a post-order traversal of T, so that each node v in T is visited only after its children are visited. The main computation performed during the visit of a node v is to compute a value $x(v)$. If v is a leaf of T, storing some x_i from S, then we compute

$$x(v) = x_i \bmod \phi(N).$$

If v is an internal node of T with children u and w (we can assume T is proper, so that each internal node has two children), then we compute

$$x(v) = x(u)x(w) \bmod \phi(N).$$

When we have computed $x(r)$, where r denotes the root of T, then we are done with this first phase. Since a post-order traversal takes $O(n)$ time, and each visit computation in our traversals takes $O(1)$ time, this entire first phase runs in $O(n)$ time.

4.2 Phase Two

In the second phase, we perform a pre-order traversal of T, where the visit of a node v involves the computation of a value $A(v)$. The value $A(v)$ for a node v is defined to be the accumulation of all values stored at nodes that are *not* descendents of v (including v itself if v is a leaf). Thus, if v is a leaf storing some element x_i from S, then $A(v) = A_i$. Recall that in a pre-order traversal we perform the visit action each node v before we perform the respective visit actions for v's children. For the root, r, of T, we define $A(r) = 1$. For any non-root node v, let z denote v's parent and let w denote v's sibling (and note that since T is proper, every node but the root has a sibling). Given $A(z)$, we can compute the value $A(v)$ for v as follows:

$$A(v) = A(z)^{x(w)} \bmod N.$$

By the corollary (2) to Euler's Theorem, we can inductively prove that each $A(v)$ equals the accumulation of all the values stored at non-descendents of v. Since a pre-order traversal of T takes $O(n)$ time, and each visit action can be performed in $O(1)$ time, we can compute all the A_i values in $O(n)$ time. Note

that implementing this algorithm requires knowledge of the value $\phi(N)$, which presumably only the source knows. Thus, this computation can only be performed at the source, who then must transmit all the new A_i values after any updates.

The performance of the precomputed accumulation scheme is summarized in Table 2.

space	insertion time	deletion time	update info	query time	query info	verify time
$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$

Table 2: Precomputed accumulation scheme for implementing an authenticated dictionary with an exponential accumulator.

Thus, this precomputed accumulations approach can be implemented to run in constant time for queries and in linear time for updates. If n is very large, however, and updates occur frequently but in small numbers, then even these linear-time computations at the source can take a while. Therefore, in the next section we describe how to combine the two above approaches to design a scheme that is efficient for both updates and queries.

5 Parameterized Accumulations

Suppose we are again interested in maintaining a set S as described above. We will use an integer parameter $1 \geq p \leq n$ to balance the processing between the source and the directories, depending on their relative computational power. The main idea is to partition the set S into p groups of roughly n/p elements each, performing the straightforward approach inside each group and the precomputed accumulations approach among the groups. The details are as follows.

5.1 Subdividing the Dictionary

Divide the set S into p groups, $Y_1, Y_2, \ldots, Y_p$, of roughly n/p elements each, balancing the size of the groups as much as possible. For group Y_j, let y_j denote the product of the items in Y_j modulo $\phi(N)$. Define B_j as

$$B_j = a^{y_1 y_2 \cdots y_{j-1} y_{j+1} \cdots y_n} \bmod N.$$

That is, B_j is the accumulation of all items that are not in the set Y_j. After any insertion or deletion in a set Y_j, the source can compute a new y_j in $O(n/p)$ time (and we show below, in Section 5.2, how with some effort this bound can be improved to $O(\log n/p)$ time). Moreover, since the source knows the value of $\phi(N)$, it can update all the B_j values after such an update in $O(p)$ time. Thus, the source can process an update operation in $O(p + n/p)$ time, assuming that the update does not require adjusting where the boundaries between the Y_j sets are.

Fortunately, maintaining the range of values in each Y_j is not a major overhead. We need only maintain the invariant that each Y_j has at least $\lceil n/p \rceil/2$ elements at most $2\lceil n/p \rceil/2$ elements. If a Y_j set grows too small, then we either merge it with one of its adjacent sets Y_{j-1} or Y_{j+1}, or (if merging Y_j with such a sets would cause an overflow) we "borrow" some of the elements from an adjacent set to bring the size of Y_j to at least $3\lceil n/p \rceil/4$. Likewise, if a Y_j set grows too large, then we simply split it in two. These simple adjustments will never take more than $O(n/p)$ time, and will maintain the invariant that each Y_j is of size $\Theta(n/p)$. Of course, this assumes that the value of n does not change significantly as we insert and remove elements. But even this condition is easily handled. Specifically, we can maintain the sizes of the Y_j's in a priority queue that keeps track of the smallest and largest Y_j sets. Whenever we increase n by an insertion, we can check the priority queue to see if the smallest set now must do some merging or borrowing to keep

from growing too small. Likewise, whenever we decrease n by a deletion, we can check the priority queue to see if the largest set now must split. A straightforward inductive argument shows that this approach keeps the size of the Y_j's to be $\Theta(n/p)$.

Keeping the Y_j's to have exactly size $\Theta(n/p)$ is admittedly an extra overhead. In practice, however, all this overhead can probably be ignored, as it is likely that the Y_j's will grow and shrink at more or less the same rate. Indeed, even if the updates are non-uniform, we can afford to completely redistribute the elements in all the Y_j's as often as every $O(\min\{p, n/p\})$ updates, amortizing the $O(n)$ cost for this redistribution to the previous set of updates that occurred since the last redistribution.

Turning to the task at a directory, then, we recall that a directory receives all p of the B_j values after an update occurs. Thus, a directory can perform its part of an update computation in $O(p)$ time. It validates that some x_i is in S by first determining the group Y_j that x_i belongs to, which can be done by table look-up. Then, it computes A_i as

$$A_i = B_j^{x_k x_{k+1} \cdots x_{i-1} x_{i+1} \cdots x_l} \bmod N,$$

where $[k,l]$ is the range of indices for the elements in Y_j. Thus, a directory can answer a query in $O(n/p)$ time.

The performance of the parameterized accumulation algorithm is summarized in Table 3.

space	insertion time	deletion time	update info	query time	query info	verify time
$O(n)$	$O(p+n/p)$	$O(p+n/p)$	$O(p)$	$O(n/p)$	$O(1)$	$O(1)$

Table 3: Parameterized scheme for implementing an authenticated dictionary using an exponential accumulator. We denote with p an integer such that $1 \leq p \leq n$.

The parameter p allows us to balance the performance between the source and the directories, and also between the cost for an update and the cost for performing queries. For example, we can balance performance equally by setting $p = \lceil \sqrt{n} \rceil$, which implies that both queries and updates in this scheme take $O(\sqrt{n})$ time. Note that for reasonable values of n, say for n between $10,000$ and $1,000,000$, $\sqrt{n}$ is between 100 and $1,000$. In many cases, this is enough of a reduction to make the dynamic exponential accumulator practical for the source and directories, while still keeping the user computation to be one exponentiation and one signature verification. Indeed, these user computations are simple enough to even be embedded in a smart card, a PDA, or cellphone.

5.2 Improving the Update Time for the Source

We describe in this section how the source can further improve the performance of an update operation in the parameterized scheme. Recall that in this scheme the set S is partitioned into p subsets, $Y_1, Y_2, \ldots, Y_p$, and the source maintains for each Y_j a value B_j, on behalf of the directories, that is the accumulation of all the values not in Y_j. Also recall that, for each group Y_j, we let y_j denote the product of the items in Y_j modulo $\phi(N)$. In the algorithm description above, the source recomputes y_j from scratch after any update occurs, which takes $O(n/p)$ time. In this section we describe how this can be done in $O(\log(n/p))$ time.

The method is for the source to store the elements of each Y_j in a balanced binary search tree. For each internal node w in T_j, the source maintains the value $y(w)$, which is the product of all the items stored at descendents of w, modulo $\phi(N)$. Thus, $y(r(T_j)) = y_j$. Any insertion or deletion will affect only $O(\log(n/p))$ nodes w in T_j, for which we can recompute their $x(w)$ values in $O(\log(n/p))$ total time. Therefore, after any update, the source can recompute a y_j value in $O(\log(n/p))$ time, assuming that the size of the Y_j's does not violate the size invariant. Still, if the size of Y_j after an update violates the size invariant, we can easily adjust it by performing appropriate splits and joins on the trees representing Y_j, Y_{j-1}, and/or Y_{j+1}.

Moreover, we can rebuild the entire set of trees after every $O(n/p)$ updates, to keep the sizes of the Y_j sets to be $O(n/p)$, with the cost for this periodic adjustment (which will probably not even be necessary in practice for most applications) being amortized over the previous updates. This performance of the resulting scheme is summarized in Table 4.

space	insertion time	deletion time	update info	query time	query info	verify time
$O(n)$	$O(p+\log(n/p))$	$O(p+\log(n/p))$	$O(p)$	$O(n/p)$	$O(1)$	$O(1)$

Table 4: Enhanced parameterized scheme for implementing an authenticated dictionary using an exponential accumulator. We denote with p an integer such that $1 \leq p \leq n$.

In this version of our scheme, we can achieve a complete tradeoff between the cost of updates at the source and queries at the directories. Tuning the parameter p over time, therefore, could yield the optimal balance between the relative computational powers of the source and directories. It could also be used to balance between the number of queries and updates in the time intervals.

6 Discussion and Conclusion

We have shown how to make the exponential accumulator function the basis for a practical and efficient scheme for authenticated dictionaries, which relies on reasonable cryptographic assumptions similar to those that justify RSA encryption. A distinctive advantage of our approach is that the validation of a query result performed by the user takes constant time and requires computations (a single exponentiation and digital signature verification) simple enough to be performed in devices with very limited computing power, such as a smart card or a cellphone. Our approach also achieves a complete 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, we can achieve $O(\sqrt{n})$ time for both updates and queries.

Our technique can be easily adapted to contexts, such as certificate revocation queries, where one needs to also validate that an item x is *not* in the set S. In this case, we use the well-known trick of storing in the dictionary not the items themselves, but instead the intervals $[x_i, x_{i+1}]$ in a sorted list of the elements of S (see, e.g., Kocher [20]). A query for an element x returns an interval $I = [x_i, x_{i+1}]$ containing, x plus a cryptographic validation of interval I. If x is one of the endpoints of this interval, it is in S; if it is strictly inside the interval, x is not in S. Note that this approach also requires that we have a way of representing some notion of $-\infty$ and $+\infty$. Even so, the overhead adds only a constant factor to all the running times for updates, queries, and validations.

Acknowledgments

We would like to thank Andrew Schwerin, Giuseppe Ateniese, and Douglas Maughan for several helpful discussions and e-mail exchanges relating to the topics 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] 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. http://ecf.hq.eso.org/iraf/web/ADASS/adass_proc/adass_95/brunnerr/brunnerr.html.

[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] D. A. Cooper. A more efficient use of delta-CRLs. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 190–202, 2000.

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

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

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

[16] 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.

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

[18] 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.

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

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

[21] 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.

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

[23] 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.

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

[25] 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.

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

[27] 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.

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

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