On the Cost of Authenticated Data Structures

Roberto Tamassia Nikos Triandopoulos

rt@cs.brown.edu nikos@cs.brown.edu

Department of Computer Science
Brown University

Abstract

Authenticated data structures provide a model for data authentication, where answers to queries contain extra information that can produce a cryptographic proof about their validity. In this paper, we study the *authentication cost* that is associated with this model when authentication is performed through *hierarchical cryptographic hashing*. We introduce measures that precisely *model* the computational overhead that is introduced due to authentication. We study the theoretical limitations of the model for authenticated structures that solve the *dictionary problem* of size n and prove (i) an intrinsic *equivalence* between authentication through hashing and searching by comparison, (ii) a $\Omega(\log n)$ *lower bound* for the authentication cost and (iii) the optimality of *tree structures*. In view of the logarithmic lower bound, we (i) analyze and study the performance of existing authenticated structures with $O(\log n)$ authentication cost and (ii) describe a *new authentication scheme* based on skip-lists that achieves a performance very close to the theoretically optimal. Through the relation between authentication cost and number of comparisons, we finally get a *new skip list version* achieving expected performance that is very close to the *theoretically optimal* with respect to the logarithmic constant. Namely, a search operation takes on average $1.25 \log_2 n + O(1)$ comparisons, which must be compared with the $1.5 \log_2 n + O(1)$ previous best scheme.

Keywords Authenticated Data Structures, Data Authentication, Dictionary Problem, Skip Lists

1 Introduction

The design and implementation of efficient data structures for various query problems has a long history in computer science. Data structures are typically designed having as goal to organize a collection of data, so that searching in the collection and answering queries about the data are performed efficiently. Implicitly, the owner and the user of the data structure are assumed to be the same entity. However, an important security problem arises when this assumption is abandoned.

With the advent of Web services and pervasive computing, a data structure can be controlled by an entity different than the owner or the user of the data. Data replication applications achieve computational efficiency by caching data at servers near users, but they present a major security challenge. Namely, how can a user verify that the data items replicated at a server are the same as the original ones from the data source? For example, stock quotes from the New York Stock Exchange are distributed to brokerages and financial portals that provide quote services to their customers. An investor that gets a stock quote from a web site would like to have a secure and

efficient mechanism to verify that this quote is identical to the one that would be obtained by querying directly the New York Stock Exchange.

A simple mechanism to achieve the authentication of replicated data consists of having the source digitally sign each data item and replicating the signatures in addition to the data items themselves. However, when data evolves rapidly over time, as is the case for the stock quote application, this solution is inefficient.

Authenticated data structures are a model of computation where an untrusted *directory* (also called *responder*) answer queries on a data structure on behalf of a trusted source and provides a proof of the validity of the answer to the user. The data source ideally signs only a single digest of the data. On a query, along with the answer, the signed digest and some information that relates the answer to this digest are also given to the user and these are used for the answer verification.

In this paper, we study the computational and communication overhead of authentication on the performance of a data structure. We focus on the *dictionary problem*, i.e., answering membership queries on a set of elements. We model the authentication overhead so that all cost parameters can be easily expressed. Using this model, we relate authentication by hashing to searching by comparisons and prove a logarithmic lower bound on the authentication overhead. We provide a novel analysis of existing authentication schemes and design a new scheme that achieves authentication overhead close to the theoretical optimal. Finally, through the relation between authentication and search by comparisons, we present a new version of a skip list where on average searches can be performed with $1.25 \log_2 n + O(1)$ comparisons, which is optimal up to an additive constant term.

1.1 Model

The *authenticated data structure* model involves a structured collection S of objects (e.g., a set or a graph) and three parties: the *source*, the *directory*, and the *user* (see Figure 1) . A repertoire of *query operations* and optional *update operations* are assumed to be defined over S. The role of each party is as follows:

- The *source* holds the original version of S. Whenever an update is performed on S, the source produces *update authentication information*, which consists of a signed time-stamped statement about the current version of S.
- The *directory* maintains a copy of S. It interacts with the source by receiving from the source the updates performed on S together with the associated update authentication information. The directory also interacts with the user by answering queries on S posed by the user. In addition to the answer to a query, the directory returns *answer authentication information*, which consists of (i) the latest update authentication information issued by the source; and (ii) a *proof* of the answer.
- The *user* poses queries on S, but instead of contacting the source directly, it contacts the directory. However, the user trusts the source and not the directory about S. Hence, it verifies the answer from the directory using the associated answer authentication information.

The data structures used by the source and the directory to store collection S, together with the algorithms for queries, updates, and verifications executed by the various parties, form what is called an *authenticated data structure*.

In a practical deployment of an authenticated data structure, there would be several geographically distributed directories. Such a distribution scheme reduces latency, allows for load balancing, and reduces the risk of denial-of-service attacks. Scalability is achieved by increasing the number of directories, which do not require physical security since they are not trusted parties.

Figure 1: The three-party authentication model.

1.2 Authentication Through Hashing

In this paper, we focus on authentication schemes that use an *one-way, collision-resistant cryptographic hash function*, which will be referred to as *hash function* for brevity. The alternative technique of one-way accumulators, used in [3, 10], is significantly less efficient in practice.

Let S be a data set owned by the source. We use a hash function h to produce a *digest* of set S, which is signed by the source. The digest is computed through a *hashing scheme* over a directed acyclic graph (DAG) that has a single sink node t and whose source nodes store the elements of S (see [12], [16]). Each node u of G stores a *label* $L(u)$ such that if u is a source of G, then $L(u) = h(e_1, \ldots, e_p)$, where $e_1, \ldots, e_p$ are elements of S, else (u is not a source of G) $L(u) = h(L(w_1), \ldots, L(w_l), e_1, \ldots, e_q)$, where $(w_1, u), \ldots, (w_l, u)$ are edges of G and $e_1, \ldots, e_q$ are elements of S (p, q and l are some constant integers). We view the label $L(t)$ of the sink t of G as the digest of S, which is computed via the above DAG G.

The authentication technique is based on the following general approach. The source and the directory store identical copies of the data structure for S and maintain the same hashing scheme on S. The source periodically signs the digest of S together with a timestamp and sends the signed timestamped digest to the directory. When updates occur on S, they are sent to the directory together with the new signed time-stamped digest. In this setting, the update authentication information has constant size. When the user poses a query, the directory returns to the user (1) the signed timestamped digest of S, (2) the answer to the query and (3) a proof consisting of a small collection of labels from the hashing scheme (or of data elements if needed) that allows the recomputation of the digest. The user validates the answer by recomputing the digest, checking that it is equal to the signed one and verifying the signature of the digest; the total time spent for this process is called the *answer verification time*.

1.3 Previous and Related Work

Throughout this section, we denote with n the size of the collection S maintained by an authenticated data structure. Early work on authenticated data structures was motivated by the *certificate revocation* problem in public key infrastructure and focused on *authenticated dictionaries*, on which membership queries are performed. The *hash tree* scheme introduced by Merkle [17, 18] can be used to implement a static authenticated dictionary. A hash tree T for a set S stores cryptographic hashes of the elements of S at the leaves of T and a value at each internal node, which is the result of computing a cryptographic hash function on the values of its children. The hash tree uses linear space and has $O(\log n)$ proof size, query time and verification time. A dynamic authenticated dictionary based on hash trees that achieves $O(\log n)$ update time is described in [19]. A dynamic authenticated dictionary that uses a hierarchical hashing technique over skip lists is presented in [9]. This data structure also achieves $O(\log n)$ proof size, query time, update time and verification time. Other schemes based on variations of hash trees have been proposed in [2, 6, 13]. The software architecture and implementation of an authenticated dictionary based on skip lists is presented in [11].

An alternative approach to the design of authenticated dictionary, based on the *RSA accumulator*, is presented in [10]. This technique achieves constant proof size and verification time and provides a tradeoff between the query and update times. For example, one can achieve $O(\sqrt{n})$ query time and update time.

In [1], the notion of a *persistent authenticated dictionary* is introduced, where the user can issue historical queries of the type "was element e in set S at time t". A first step towards the design of more general authenticated data structures (beyond dictionaries) is made in [5] with the authentication of relational database operations and multidimensional orthogonal range queries.

In [16], a general method for designing authenticated data structures using hierarchical hashing over a search graph is presented. This technique is applied to the design of static authenticated data structures for pattern matching in tries and for orthogonal range searching in a multidimensional set of points.

Efficient authenticated data structures supporting a variety of fundamental search problems on graphs (e.g., path queries and biconnectivity queries) and geometric objects (e.g., point location queries and segment intersection queries) are presented in [12]. This paper also provides a general technique for authenticating data structures that follow the *fractional cascading* paradigm.

A distributed system realizing an authenticated dictionary, is described in [7]. This paper also provides an empirical analysis of the performance of the system in various deployment scenarios. The authentication of distributed data using web services and XML signatures is investigated in [20]. *Prooflets*, a scalable architecture for authenticating web content based on authenticated dictionaries, are introduced in [24]. Work related to authenticated data structures includes [3, 4, 8, 14, 15].

Skip lists were introduced in [21, 22] where it is shown that the expected number of comparisons for a search is $(\log_2 n)/(p \log_2 \frac{1}{p}) + O(1)$, where p is a probability parameter. An improved skip list version gives $\frac{1-p^2}{p \log_2 \frac{1}{p}} \log_2 n + O(1)$ expected comparisons.

1.4 Our Results

In this paper, we study the authentication overhead of authenticated data structures based on a hashing scheme. Our main contributions are as follows.

1. We model the authentication cost using properties of the corresponding hashing scheme. Using this model, we introduce precise measures of the computational overhead and we express all the involved cost parameters that appear in the design of authenticated data structures by these properties.

2. Focusing on authenticated dictionaries, we study the theoretical limits of authentication through hashing. We deploy an equivalence between authentication through hashing and searching by comparison and prove an $\Omega(\log n)$ logarithmic lower bound for the authentication overhead of a dictionary. In [19] the open question of whether authenticated dictionaries based on hash tables with roughly $O(1)$ complexity can be constructed. We prove a negative answer for this question in the hierarchical hashing model of authentication. Moreover, we show that tree structures are optimal as hashing schemes. Also, we examine the case where k hash values are signed by the source and show what is the optimal hashing scheme for this scenario.

3. We design a new authentication scheme based on skip-lists and show that its performance is close to the theoretical optimal.

4. Through the relation between authentication cost and number of comparisons we develop a new version of skip list where the expected number of comparisons in a search is $1.25 \log_2 n + O(1)$, which is optimal up to an additive constant factor and improves the previous best bound of $1.5 \log_2 n + O(1)$n.

5. We analyze existing authentication schemes for the dictionary problem.

In Section 2, the authentication overhead of authenticated data structures is discussed and we examine the cost components of authentication through hashing. In Section 3 we model the authentication overhead of any authenticated data structure by introducing new metrics which are related to specific properties of the hashing scheme. In Section 4 we describe the dictionary problem and in Section 5 we examine the theoretical limits of authentication through hashing for this problem. We prove a lower bound and that tree structures are better. In view of this results we then focus on tree hashing schemes and study the authentication overhead. In Section 6 we design and analyze a new authentication scheme based on skip lists and through this we describe an improved version of skip list with respect to the expected number of comparisons. Section 7 studies existing structures with respect to the authentication cost. Section 8 provides a comparison of various schemes.

2 The Authentication Overhead

In this section, we study the performance overhead due to authentication-related computations in an authenticated data structure based on a hashing scheme. This overhead is called the *authentication overhead* and consists of the following cost parameters.

1. **Time Overhead:** Includes any cost that adds to the time performance of the data structure. The *rehashing overhead* is the time spend by the source or the directory to rehash over the data in order to recompute the digest of the data structure after an update in the data set. The *query answering overhead* is the extra time consumption for the directory to create the answer authentication information. The *verification time* is the time spent by the user in order to verify the answer given by the directory. In cases where the data structure is created from scratch, we also consider the *complete hashing* overhead, i.e., the time to hash a given data set using a given hashing scheme.

2. **Communication Overhead:** Corresponds to the communication cost introduced by the model, i.e., the size of the update authentication information for the source-to-directory communication and the size of the answer authentication information for the directory-to-user communication. We are particularly interested in the size of the proof that is given to the user which is part of the answer authentication information.

3. **Storage Overhead:** The total number of hash values used by the authentication scheme.

Even with the most efficient implementations, the time for computing a hash function is an order of magnitude larger than the time for a comparison of basic number types (e.g., integers or floating-point numbers). Thus, the rehashing overhead dominates the update time and the practical performance of an authenticated data structure is characterized by the authentication overhead. For efficient authenticated data structures we want to minimize the above cost parameters. We will see that there is a major trade-off between time and communication overhead.

The authentication overhead heavily depends on the actual hashing scheme. Furthermore, for a given hashing scheme, the authentication overhead is affected by the hash function h adopted and by the mechanism used to realize a multivariate hash function from h.

2.1 Hashing Scheme

The major cost component of the authentication overhead is the hashing scheme G that is used to produce the digest of the data structure. It must be chosen so that for any data element in the data structure an update or verification process has the minimum possible authentication cost.

In the literature, it has been always the case the hashing scheme G for a data structure $\mathcal{DS}$ coincides with the data structure. This has the advantage that known efficient data structuring techniques can be used to guarantee that the authentication cost that is involved will be asymptotically equal with the time performance of operations on the data structure.

However, we can consider the *separation* of the two above *partially* or *completely*. According to the first technique, the hashing scheme does not have to completely overlap with the data structure or can be defined independently of the data structure. In Section 6, we use this to derive a new authentication scheme based on skip lists. By partially separating the hashing scheme G from the data structure $\mathcal{DS}$, in principle we can achieve less expensive hashing overhead and less used storage by shortening the hashing scheme G, so that one hash value is stored for more than one nodes of $\mathcal{DS}$. We also gain flexibility in reducing or eliminating the query answering overhead by carefully storing hash values of G in the appropriate nodes of $\mathcal{DS}$. Finally, we can even completely separate G from $\mathcal{DS}$ by storing G in its own data structure. This scheme may be more efficient in the static case where no updates are performed.

2.2 Cryptographic Hash Functions

The basic cryptographic primitive for authenticated data structures is a *collision resistant* hash function. A collision resistant hash function is a function h satisfying the following conditions:

1. The argument x can be of arbitrary length and the result $h(x)$ has a fixed length of n bits (with $n \geq 128...160$).

2. The hash function must be *one-way* in the sense that given a y is in the image of h, it is computationally hard to find a message x such that $h(x) = y$.

3. The hash function must be *collision resistant*: it is computationally hard to find distinct x and y that hash to the same result, i.e., $h(x) = h(y)$.

Most collision resistant hash functions used in practice are *iterated hash functions*. An iterated hash function h is based on a *compression function* f that maps two input strings, a string of N bits and a string of B bits, to an output string of N bits. The input string x is preprocessed using a *padding rule* to a string y whose length is a multiple of B. If $y = y_1 \| y_2 \| ... \| y_k$, then $h(x) = z_k$, where z_k is given by the following computation:

$$\begin{aligned} z_0 &= IV \\ z_1 &= f(z_0, y_1) \\ z_2 &= f(z_1, y_2) \\ \vdots &\ \vdots \ \vdots \\ z_k &= f(z_{k-1}, y_k). \end{aligned}$$

Above, IV is an *initial value* N bit string that is public. The padding rule must be such that it is unambiguous. To that end the length of the binary representation of the length $|x|$ of x is

part of the padded string. Figure 2 gives a schematic description of the construction of a iterated hash function h based on a compression function f. Given this model, we observe that the time complexity for the application of h on x is in general proportional to $|x|$ (since k is proportional to $|x|$).

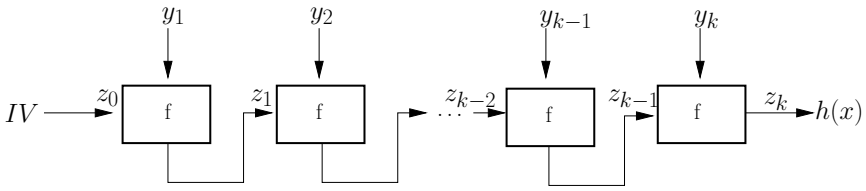

Figure 2: The iterated hash function model.

In more detail, the time complexity of an iterated hash function h for the operation $h(x)$ can be described as a function of its input size $\ell = |x|$ by the function

$$T(\ell) = c_1 \left(\left\lfloor \frac{\ell}{D} \right\rfloor + 1 \right) + c_2,$$

where D is some constant less than N. By setting $g(\ell) = \left\lfloor \frac{\ell}{D} \right\rfloor$ and $c = c_1 + c_2$, we have that

$$T(\ell) = c_1 g(\ell) + c.$$

Thus, $T(\ell)$ is a *staircase* function and the following holds.

Lemma 1 *The time complexity of an iterated hash function is a staircase function of its input size.*

Note that up to $\lfloor \frac{D-1}{N} \rfloor$ hash values can be hashed in time c, and, in general, up to $D-1$ bits can be hashed in time c (the minimum hashing time). We define M to be $\lfloor \frac{D-1}{N} \rfloor$.

Two candidate iterated hash functions that are commonly used in practice are MD5 and SHA-1. For MD5, $N = 128$ and $M = 3$, for SHA-1, $N = 160$ and $M = 2$, while for both, $B = 512$ and $D = 448$. Figure 3 depicts the time performance for MD5 and SHA-1.

2.3 Multivariate Hash Functions

Let u be a node of G having as children nodes $w_1, ..., w_l$. According to the hashing scheme, the label $L(u)$ equals the hash value $h(L(w_1), ..., L(w_l), e_1, ..., e_q)$, where e_is are data elements. That is, h is assumed to operate on a finite number of binary strings. However, the hash function h actually operates on one binary string rather than a set of a strings. Thus, we need *multivariate* hash functions. Suppose we want to implement a *d-variate* hash function h.

The straightforward way to realize $h(x_1, ..., x_d)$ is by using string concatenation, i.e., by hashing the concatenated string $x_1 \| ... \| x_d$. We call this multivariate hash function the *concatenation* hash function and we denote it as h_C. But this is not the only possible possible. In general, we can consider any realization where the concatenation hash function h_C is applied more than once recursively. Note that any realization can be computed in a unique way, since neither commutativity nor associativity hold for h_C. It should be noted that the collision-resistant property of such a recursively defined hash function is preserved.

Two other implementations for multivariate hash functions are the following. The *tree* hash function $h_T(x_1, ..., x_d)$ is implemented using recursively function h_C and a binary hash tree on

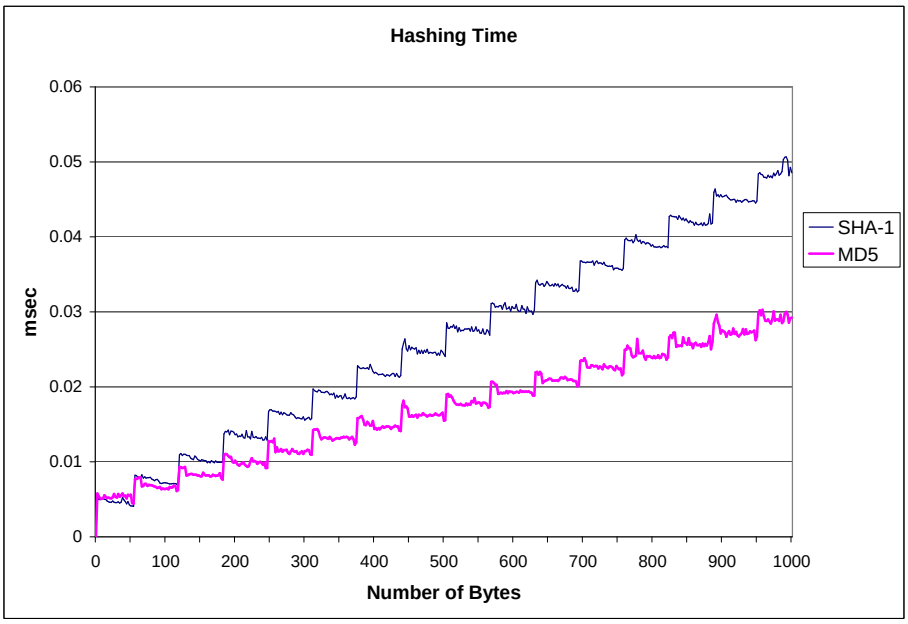

Figure 3: Hashing time of MD5 and SHA-1 as a function of the input size. Experimental results showed that for MD5 $c_1 \simeq 0.0014895$ and $c_2 \simeq 0.003837$ and that for SHA-1 $c_1 \simeq 0.002822$ and $c_2 \simeq 0.001898$ (all denote time in millisecond).

top of elements x_i with logarithmic in d associated time and communication cost. That is, h_T is implemented using h_C and a hash tree on x_is. This idea has been used in [16] and in [12]. Finally, the *sequential* hash function $h_S(x_1, ..., x_d)$ is implemented using h_C as $h_S(x_1, ..., x_d) = h_C(h_C(h_C(x_1, x_2), x_3), ..., x_d)$. Equivalently, $h_S(x_1, ..., x_d) = h_d$, where $h_i = h_C(h_{i-1}, x_i)$, $2 < i \leq d$, and $h_2 = h_C(x_1, x_2)$.

Any implementation follows an intrinsic time-communication trade-off.

Lemma 2 *Time-communication trade-off for multivariate hash functions: time overhead always increases by applying h_C more than once and communication overhead is always possible to decrease by applying h_C more than once;*

Proof Sketch: Time overhead increase follows by the definition of the time complexity of h and the fact that for a, b positive numbers, $0 \leq \lfloor a + b \rfloor - \lfloor a \rfloor - \lfloor b \rfloor \leq 1$. For the decrease in the communication overhead, consider the tree hash function h_T. If it is implementing a d-variate hash function the communication overhead is proportional to $\log d + 1$ compared to the communication overhead for h_C which is proportional to $d - 1$. □

The above lemma suggests the following. In terms of hashing overhead, $h(x_1, ..., x_d)$ is optimally implemented as $h(x_1 \| ... \| x_d)$. In terms of communication overhead, a different implementation of h, like h_T can help. Which of the multivariate hash function should be used when a hashing scheme G is already defined depends on G itself and on the degree d of the multivariate hash function. In general, unless G has specific properties, a different one must be used for each node of G so that the hashing overhead is minimized or the time-communication trade-off is appropriately tuned. In the case where G is not "balanced", we can use the *biased* multivariate hash function h_B, so that the cost is distributed so that commonly used queries have less communication cost. For

example, if $u \in G$ has predecessor nodes $v_1, ..., v_d$ in G, nodes v_i are assigned a weight $w(v_i)$ and h_B is implemented so that the authentication cost is distributed inversely proportionally to weights $w(v_i)$. However, we can limit our study only to the concatenation hash function h_C and always modify the hashing scheme G to accommodate the efficiency of a hash function implementation.

Lemma 3 *Any alternative realization of a d-variate function $h(x_1\|...\|x_d)$ based on the iterative applications of h_C can be expressed through a hashing scheme G.*

3 Two Metrics for Modeling the Authentication Overhead

We next introduce the concept of *authentication cost*, a *pair of properties* of the hashing scheme that is used. The authentication cost consists of the following two components: the *node size* and the *degree size*. Both components are simply properties of some connected subgraph of the hashing scheme and capture the efficiency of the hashing scheme in terms of the authenticated overhead that is added to the performance of the data structure. More importantly, for a given update or verification operation, each cost parameter of the authentication overhead can be precisely expressed as a *linear combination* of the node size and the degree size of some (corresponding to the operation) subgraph of the used hashing scheme.

Let G be the hashing scheme that is used for the data structure $\mathcal{DS}$. If u is a vertex of G, by $\text{indeg}(v)$ we denote the *in-degree* of vertex v, that is the number of predecessors of v in G. Let G' be a connected subgraph of G. The two components of the authentication cost for G' are defined as follows. The *node size* N_H of G' is the number of its vertices. The *degree size* N_L of G' is the sum of the in-degrees of its vertices, that is, $N_L = \sum_{v \in G'} \text{indeg}(v)$.

The choice of these two components is justified as follows. Every update, query of verification operation corresponds to some subgraph G' of the hashing scheme G. Then, the node size N_H corresponds to the number of *hash operations* that are involved at some of the three parties (source, directory or user) and the degree size N_L corresponds to the total number of hash values that participate as operands in these hash operations. Each cost parameter of the authentication overhead for this operation depends linearly on N_H and N_L.

First consider the hashing overhead for some update or verification operation. Lemma 1 states that for a string x of length ℓ, the time complexity of the hash operation $h(x)$ is of the form $T(\ell) = c_1 \left\lfloor \frac{\ell}{D} \right\rfloor + c$, where D, c_1 and c are constants and $D < N$. Thus, considering a vertex v in the hashing scheme G with $\text{indeg}(v) = d$ and using the concatenation multivariate hash function (Lemma 3 justifies this choice), we see that the time complexity for the hash operation that corresponds to vertex v is of the form $c + c'd$ for some new constant c'. For a subgraph G' of G, the hashing overhead that corresponds to all the involved hash operations is of the form $cN_H + c'N_L$. For the communication overhead of a query operation, we observe that in order the user to compute the hash value that is stored at a vertex v of G', exactly $\text{indeg}(v) - 1$ hash values need to be sent by the directory. Thus, in total, $N_L - N_H$ labels are needed. For the storage overhead, clearly N_H hash values are stored in the data structure at the source (the same holds for the directory), where N_H is the node size of G itself.

Using the above discussion we can express precisely the authentication overhead for any authenticated data structure $\mathcal{DS}$ as follows. Let G be the hashing scheme of $\mathcal{DS}$ before an operation or query. Any update operation U in $\mathcal{DS}$ generally changes G to G_U. A node in G_U is called a *cost* node if the set of its predecessors in G_U is different than the set of its predecessors in G, or if it is a new node, or the set of data elements that are hashed at this node has changed. Let V_U be the set of all cost nodes in G_U. Let $G' = (V, E)$ be the subgraph of G_U that contains all nodes of G_U that are reachable in G_U from nodes in V_U. Then, the rehashing overhead is given by a linear combination of

the node size and the degree size of G', that is, of formula $c|V|+c'\sum_{v\in G'}\text{indeg}(v) = cN_H + c'N_L$ (c and c' are constants, and N_H, N_L are the node size and the degree size of G'). Moreover, consider a query Q. The answer of Q is a subset S_Q of the the S that is stored in $\mathcal{DS}$. Let V_Q be the set of nodes in G whose hash label depend on S_Q. Let $G' = (V, E)$ be the subgraph of G that contains all nodes in G that are reachable in G from nodes in V_Q. The verification time is then a quantity of the form $c|V| + c'\sum_{v\in G'}\text{indeg}(v)$ (with c, c', N_H, N_L as before). The communication cost is the difference $\sum_{v\in G'}\text{indeg}(v) - |V| = N_L - N_H$. If $G = (V_G, E_G)$, then the storage overhead is clearly the number V_G of nodes of G, i.e., the node size of G and the complete hashing overhead is $c|V_G| + c'\sum_{v\in G}\text{indeg}(v) = c|V_G| + c'|E_G|$ (here, the node size and the degree size of G are V_G and E_G respectively).

For the rest of the paper, we focus our discussion on the *dictionary* problem. We analyze the authentication cost for authenticated dictionaries.

4 The Dictionary Problem

We consider a dictionary S storing key-element pairs (k, e) that supports the following update operations

- replace(k,e): replace the (existing in S) element of k with e;
- insert(k,e): inserts the (non existing in S) pair (k, e) in S;
- delete(k): deletes the (existing in S) pair that corresponds to k.

We are interested in authenticating answers to the following queries:

- contains(k): a yes/no answer about whether $(k, \cdot) \in S$ or not;
- get(k): returns the (existing in S) element of k.

For the user part, we also define a new operation that corresponds to the verification of an answer given by the directory: operation verify(k) verifies the answer received by the user to a query contains(k) or get(k).

We give an example of an authentication solution for the static case (see Figure 4). A binary tree is built on top of the key-element pairs. The hashing scheme is considered to be the tree itself having directed edges from children to parent. A leaf u corresponding to (k, e) stores the hash value $L(u) = h(k, e)$ and an internal node v with children w_1 and w_2 stores $L(v) = h(w_1, w_2)$. Here, h denotes the concatenation 2-variate hash function. The answer authentication information for query contains(k) or get(k) consists of the hash values of the siblings of the nodes in the path from u to the root, where leaf u corresponds to key k. We assume that element e is part of the answer authentication information for query contains(k).

Let $\mathcal{D}$ be a dictionary for data set S. In our model, a *hashing scheme* G is a directed acyclic graph (DAG) that defines a systematic way to produce a digest of S.

The authentication overhead of hashing scheme G is characterized by the following parameters:

1. $U(k)$: rehashing overhead due to the hashing operations and label recomputations needed to be performed by the source or the directory for an update operation U on k; U can be one of I (insert), D (delete) or R (replace);
2. $V(k)$: verification time spent by the user for the verification operation V on k;

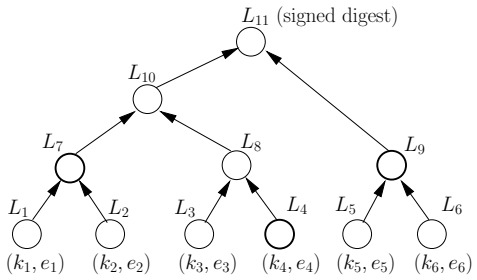

Figure 4: An example of a hashing scheme. Hash values of solid nodes and their left-right position correspond to the answer authentication information for query $\mathsf{get}(k_3)$.

3. $Q(k)$: communication cost, i.e., number of labels that need to be sent to the user by the directory as part of the answer authentication information for query Q on k; Q can be one of (contains) C or (get) G;

4. S: storage overhead, i.e., number of labels that need to be stored in the data structure $\mathcal{D}$ by the source and the directory.

5 The Limits of Authentication Though Hashing

Here, we study the limits of data authentication through hierarchical hashing for the dictionary problem. For our results we define the concept of the *authentication scheme* which is used to describe any authenticated dictionary.

The Authentication Scheme Model Let S be a set $\{x_1, ..., x_n\}$ of n elements such that $x_1 \leq x_2 \leq ... \leq x_n$ and let h be the concatenation multivariate hash function. An *authentication scheme* for S is a triple $\text{Auth}(G, n, k)$, where $G = (V, E)$ is a directed acyclic graph (DAG) with n *source* nodes and up to k *sink* nodes. G does not have parallel edges. Each source of G corresponds to a unique element x_i of S. Source nodes can be assigned elements of S in *any* possible way. Each node v of G is associated with a *label* $L(v)$. If s_i is the source node for element x_i, then $L(s_i) = h(x_i)$. If node v has predecessors $u_1, ..., u_l$, then $L(v) = h(L(u_1), ..., L(u_l))$, where the order of nodes u_i corresponds to some fixed topological order of G. Graph G has k distinguished nodes, called *signature nodes*, which include the sinks of G. Thus, if $k = 1$, G has a unique sink node.

Let G' be a weakly connected subgraph of G, i.e., a subgraph of G that is connected when one ignores edge directions. We define the *authentication cost* of G' as $C(G') = \sum_{v \in G'}(1 + \text{indeg}(v))$, where $\text{indeg}(v)$ is the number of incoming edges of v.

Let s be a source node of G, G_s be the subgraph of G that is reachable from s, and π be a path in G that connects node s with a signature node and has the *minimum* authentication cost. We define the following cost parameters associated with s: *update cost* $\mathcal{U}(s) = C(G_s)$; *verification cost* $\mathcal{V}(s) = C(\pi)$; and *communication cost* $\mathcal{C}(s) = \mathcal{V}(s) - 2|\pi|$.

Let $\text{Auth}(G, n, k)$ be an authentication scheme. The *update cost* $\mathcal{U}_G$ is the maximum update cost of a source node of G; the *verification cost* $\mathcal{V}_G$ is the maximum verification cost of a source node of G and the *communication cost* $\mathcal{C}_G$ is the communication cost that is associated with the node of the maximum verification cost.

Authentication by Hashing vs. Search by Comparisons We fist prove the equivalence between authentication by hashing and searching by comparisons on tree-based authentication schemes.

Lemma 4 *Let T be a single-sink tree and Auth$(T, n, 1)$ be an authentication scheme on set S. Tree T can be transformed into a search tree T' such that an element x_i of S can be located in T' with $\mathcal{C}(s_i) + 1$ comparisons.*

Proof: We reduce the authentication scheme Auth$(T, n, 1)$ on a sorted sequence S to a search tree T'. Consider the authentication scheme Auth$(n, T, 1)$ and any topological ordering of it. We traverse the nodes in T in reverse topological order and as we encounter source nodes of T we assign to them elements of S in increasing order. After elements are assigned to source nodes, we augment T by performing the following key assignment to all nodes in T, using again the topological ordering $t(T)$. Each source node is assigned as key the element that it stores. Each no source node v with predecessor nodes $u_1, ..., u_l$ is assigned keys $K_2, ..., K_l$, where for $2 \leq i \leq l$, K_i is the largest element that has been assigned to the source nodes of the subtree in T having as root node u_i. Graph T along with the assigned keys is a search tree T' for set S. For node v in the search path of x_i in T with cost $1 + d_v$, $d_v - 1$ comparisons are performed if $d_v > 0$, or one comparison otherwise. The number of comparisons follows from the definition of the communication cost. □

Theorem 5 *Any authentication scheme Auth$(T, n, 1)$ such that T is a tree has $\Omega(\log n)$ update, verification and communication costs.*

Proof: It follows directly from Lemma 4 and the $\Omega(\log n)$ lower bound on searching an ordered sequence in the comparison model. Note that, since T is a directed tree its verification cost is equal to its update cost. □

Next, we show that trees have optimal authentication overhead among authentication DAGs.

Theorem 6 *Let Auth$(G, n, 1)$ be an authentication scheme. There exists an authentication scheme Auth$(T, n, 1)$ such that T is a tree and $\mathcal{U}_T \leq \mathcal{U}_G$, $\mathcal{V}_T \leq \mathcal{V}_G$, and $\mathcal{C}_T \leq \mathcal{C}_G$.*

Proof: Fix a topological order $t(G)$ of G. For source node s_1 of G we find and mark the path π_1 having the minimum authentication cost, where we use the topological order $t(G)$ to search and traverse G. Using $t(G)$, we proceed by finding the minimum authentication cost path π_2 of node s_2 of G and marking the traversing path, but we stop the search if a marked node is reached, i.e., a node of path π_1. We proceed as above with all nodes s_i and we stop the traversal of G at the first marked node. At the end the marked subgraph is a tree T with and the authentication cost of any s_i in T is equal to the minimum authentication cost in G. The optimality of the authentication paths in T is proven with induction on n: optimality of the authentication cost for nodes $s_1, ..., s_{i-1}$ guarantee the optimality of the authentication cost of s_i. □

Finally, we show how signing more that one hash label affects the authentication cost. We have that a hashing scheme with k signature nodes has minimum authentication cost when the roots of k distinct trees are signed.

Lemma 7 *Let Auth$_G(G, n, k)$ be an authentication scheme. There is an authentication scheme Auth$_G(F, n, k)$ such that F is a forest and $\mathcal{U}_F \leq \mathcal{U}_G$, $\mathcal{V}_F \leq \mathcal{V}_G$, and $\mathcal{C}_F \leq \mathcal{C}_G$.*

Proof: The proof is similar to the proof of Theorem 6. For $i = 1, ..., n$, we find and mark the minimum verification paths $\pi_1, ..., \pi_n$ in G from source nodes $s_1, ..., s_n$ to a signature node or a previously marked node in G. The resulting marked subgraph of G is a forest: no marked path connecting two distinct signature nodes exists. □

Theorem 8 *Any authentication scheme Auth(G, n, k) has $\Omega(\log \frac{n}{k})$ update, verification and communication costs.*

Proof: It follows from Theorems 5 and 6 and Lemma 7. □

The above results apply to all authenticated dictionaries. This holds even if the authentication schemes as defined: (i) model static dictionaries (ii) do not support authentication of negative answers to the contain(k) query and (iii) use the concatenation hash function. It should be clear that these do not limit the generality of lower bound results: the authentication overhead is always increased when dynamic operations or authentication of negative answers to membership queries are supported. Also, Lemma 3 justifies the use of the concatenation multivariate hash function.

Authentication Cost for Tree Structures We now limit our study to hashing schemes with tree structure. Let T be a directed towards its root tree that is the hashing scheme for a dictionary $\mathcal{D}$ and let k be a key of $\mathcal{D}$. Then with respect to k and in accordance with the authentication cost measure defined in Section 1.1, we assign an authentication cost to each to update operations and queries on k.

We define path π_k to be the path in T from the node storing key k up to the single sink of T and let $C(\pi_k)$ be the authentication cost of π_k. $C(\pi_k)$ can be written as $C(\pi_k) = N_H + N_L$, where $N_H = |\pi|$ is the number of nodes in π and $N_L = \text{indeg}(T, \pi)$ is the number of the predecessor nodes in G of the nodes of π. We call N_H and N_L, the *hash path* and *hash size* of k respectively. Both N_H and N_L characterize the authentication overhead for operations on key k.

A commonly used technique for authenticating negative answers to a contains(k) query is to include in the hash computation at the leaf of the tree that corresponds to key k_i, key k_{i+1}, i.e., by storing at the leaf that corresponds to k_i the hash value $h(k_i, e_i, k_{i+1})$. In this way, any negative answer for contains(k), with $k_i < k < k_{i+1}$, is authenticated by providing the proof of existence of pair (k_i, k_{i+1}). We assume that this technique is used in our analysis. It is then the case than node of two rather than one paths in T change hash values on an update operation. For instanse, an update operation on key k_i causes the recomputation of the two paths in T that start with hash values (k_{i-1}, e_{i-1}, k_i) and (k_i, e_i, k_{i+1}). The two paths share a lot of nodes in general and one of them is path π_k. Let ρ_k be the *extra* path from a key k up to first node of π_k, that is the portion of the second path that does not belong to π_k. The authentication cost of ρ_k is $C(\rho_k)$.

Finally, for an update operation on k let $Cost(k)$ be the set of cost nodes, and σ_k be the subgraph of G that consists of nodes that are reachable from the nodes of $Cost(k)$ in T but are not nodes of π_k or ρ_k. Let $C(\sigma_k)$ the authentication cost of σ_k. $C(\sigma_k)$ typically corresponds to the rehashing cost for rebalancing the hashing scheme after an insertion or deletion.

Then, the authentication cost that is introduced by an update operation on k is $C(\pi_k) + C(\rho_k) + C(\sigma_k)$ and the authentication cost introduced by a query is $C(\pi_k) = N_H + N_L$. As we will see, for both trees and skip lists, ρ_k and σ_k are on average subgraphs of constant authentication cost and since we will be interenting in the logarithmic factor constants of the authentication overhead we can ignore them. That is both update operations and queries have authentication cost $C(\pi_k) = N_H + N_L$. The communication cost $Q(k)$ for a query is always $N_L - N_H$. We see that the hash path and hash size characterize the authentication overhead.

We can then only calculate the rehashing overhead $U(k)$ or the verification time $V(k)$ that corresponds to path π_k. Recall that the hashing time for a string x is given by $T(x) = c_1 g(|x|) + c$, where $g(x) = \lfloor \frac{x}{D} \rfloor$, $D = 448$ and $c = c_1 + c_2$. Let L be the size of a label and d_i be the degree of the ith node in path π. Then we have for $V(k)$

$$\begin{aligned} V(k) &= \sum_{i=1}^{N_H} T(L_{i_1} \| L_{i_2} \| \ldots \| L_{i_{d_i}}) + \sum_{i=1}^{N_H} T(L_{i_1} \| L_{i_2} \| \ldots \| L_{i_{d_i}}) \\ &= \sum_{i=1}^{N_H} [c_1 g(|L_{i_1} \| L_{i_2} \| \ldots \| L_{i_{d_i}}|) + c] \\ &= \sum_{i=1}^{N_H} c_1 g(d_i L) + c N_H \\ &= \sum_{i=1}^{N_H} c_1 \left\lfloor \frac{d_i L}{D} \right\rfloor + c N_H \\ &\leq c_1 \frac{N_L L}{D} + c N_H, \end{aligned}$$

and in general that $U(k) \geq V(k) + O(1)$. We see that both $U(k)$ and $V(k)$ are expressed as a linear combination of the hash path and the hash size, so both parameters need to minimized when possible. Finally, the storage overhead S is clearly equal to the number of nodes in T. In the sequel we will assume that the parameters N_H and N_L are either the same for any key k (considering an average case analysis for the data structure of interest) or that they correspond to the expected values for any key k (for structures that use randomness as skip lists).

6 A New Skip List Authentication Scheme

In this section, we describe a new authentication scheme for authenticated dictionary based on skip lists [21], the *multi-way skip list scheme*. Authenticated skip lists were introduced in [9]. We refer to the *standard scheme* as the one described in that work and analyze it in Section 7. Based on the idea of separating the hashing scheme from the actual data structure, we describe our new multi-way skip list scheme based on the skip list and study its performance in terms of the authentication costs N_H and N_L. That is, in our scheme the data structure is *unchanged* and only the hashing scheme is appropriately designed. We show that our new scheme has low authentication cost and how, using the equivalence between authentication through hashing and searching by comparison, we get an new optimized skip list with an average search cost close to optimal. We also describe an alternative scheme that achieves even better performance under certain assumptions.

6.1 Skip Lists and Bridges

We briefly describe the notation that we will use. A skip list is a set of lists $L_1, ..., L_h$, where L_1 stores all the element of a set S of size n and, for each i, each of the elements of list L_i is independently chosen to be contained in L_{i+1} with some fixed probability p. Lists are viewed as *levels* and we consider all elements of the same value that are stored in different levels to form a *tower*. That is, a tower consists of nodes of lists that store the same element. The *level* of a tower is the level of its top-most element. Each node of a tower has a forward pointer to the successor element in the corresponding list and pointer to the element one level below it. A node of the skip

list is a *plateau* node if it is the top-most node of its tower. We introduce the notion of a *bridge*. A *bridge* is a set of towers of the same level, where no higher tower is interfering them (i.e., the plateau nodes of the towers are all reachable in a sequence using forward pointers). The bridge size of a tower is the number of towers in the bridge that the tower belongs to . We now compute the expected bridge size of a tower t of level k. On average $\frac{1}{p}$ towers are consecutive having the same height, thus, the expected bridge size of a tower (of any level) is $\frac{1}{p}$. More formally, observe that if Y is the size of (any) bridge, then $Y \sim G(p)$ and $E[Y] = \frac{1}{p}$.

6.2 Multi-way Authenticated Skip List

We now describe our scheme, the *multi-way authenticated skip list* which can be viewed as a multi-way extension of the authenticated version of the skip-list data structure. The new scheme is constructed by *separating* the hashing scheme from the actual data structure, i.e., by defining the hashing scheme independently from the data structure. In particular, the data structure itself stays the *same*, but the hashing scheme G is now different from the data structure: a node in G generally corresponds to *more than one* nodes of the skip list. Furthermore, the hash value (label) that corresponds to a node of G is stored in some carefully chosen node of the skip list so that the query answering overhead is kept low. In the sequel, we give the details of our new scheme, by describing the hashing scheme G, how this is updated in insertions and deletions and how the hash values are collected in answering a query.

Hashing Scheme The notion of a bridge is essential in the new scheme. For each bridge b in the skip list, a corresponding hash value $H(b)$ is computed. We call $H(b)$ the hash of b. $H(b)$ in essence is computed by the hashes of all the *child* bridges that are contained under b and between its left-most and right-most towers. We define the hashing scheme G in a recursive way: the hash $H(b)$ of a bridge b is defined from the hashes of the child bridges of b (see Figure 5).

The *plateau towers* of a tower t are the towers on its right whose plateau nodes can be reached by t in one step using forward pointers. The hash of a bridge b is defined as follows. First, suppose that the size of b is one, i.e., b is simply a tower T. Let $t_1, ..., t_l$ be the *plateau* towers of T in increasing order with respect to their level. Note that the level of t_l is less than the level of T. If plateau tower t_i belongs in bridge b_i, then let $H(b_1), ..., H(b_l)$ be the corresponding bridge hashes. Then we define $H(b)$ to be $H(b) = h_S(h(x_T, s(x_T)), H(b_1), ..., H(b_l))$, where x_T is the element stored in tower T and $s(x_T)$ is its successor (element that is stored in the tower right to T) and h_S is the sequential multivariate hash function. If the size of b is more than one, say k, then, let $T_1, ..., T_k$ be the towers of b. For each such tower T_i with, say, $l+1$ plateau towers, we consider its, l lowest plateau towers $t_{i1}, ..., t_{il}$ (where, for $i < k$, tower T_{i+1} is *omitted* from this sequence; that is, T_{i+1} is not considered to be a plateau tower of T_i). Let $b_{i1}, ..., b_{il}$ be the child bridges that $t_{i1}, ..., t_{il}$ belong in. We define $H(T_i)$ to be the *representative* hash value of tower T_i. We define $H(T_i) = h_S(h(x_{T_i}, s(x_{T_i})), H(b_{i1}), ..., H(b_{il}))$, implemented sequentially as above. Finally, we define $H(b)$ to be the hash value $H(b) = h_C(H(T_1), ..., H(T_k))$ implemented using concatenation. In this way the complete hashing scheme G is defined recursively. The digest of the skip list is the hash of the highest bridge, i.e., the bridge of size two that consists of the left-most and right-most towers (see also Figure 6).

Storage Hash values in hashing scheme G described above are stored in such a way so that the query answering overhead, i.e., the process of collecting the answer authentication information when answering a query, is minimized. Let b be a bridge. If b consists of only one tower t and if $t_1, ..., t_l$ are

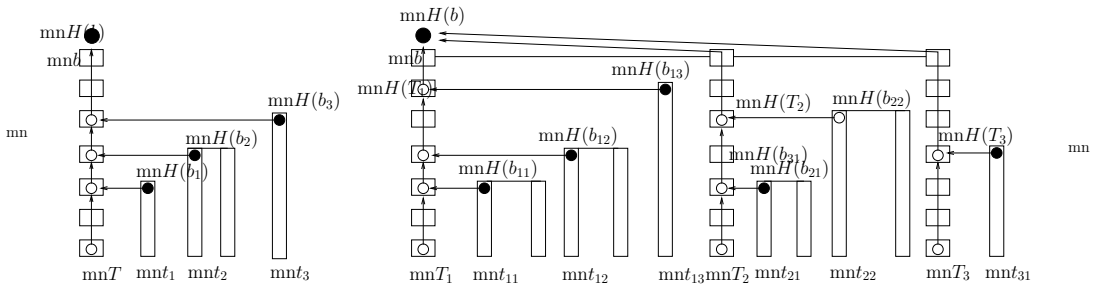

Figure 5: Hashing scheme G: $H(b)$ is recursively computed by the hashes of the child bridges.

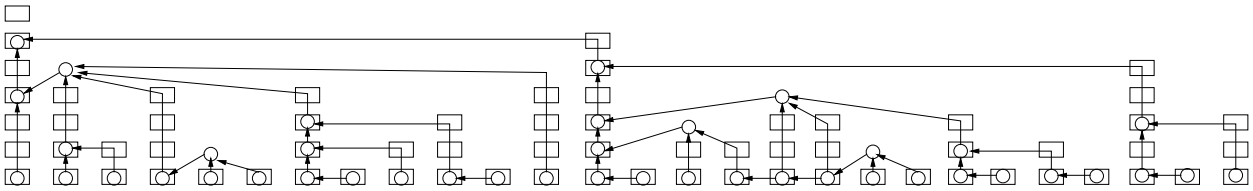

Figure 6: The hashing scheme G for multi-way authenticated skip list.

the plateau towers of t that belong in bridges $b_1, ..., b_l$, then hash value $h_i = h(h_{i-1}, H(b_i))$ is stored at the node of the same level as tower t_i, with value $h_1 = h(h(x_t, s(x_t)), H(b_1))$ is stored at the node of the same level as tower t_1. In particular, the hash of the bridge $H(b) = h_l = h(h_{l-1}, H(b_l))$ is stored at the node of height as the level of tower b_l. If b has size k, then for each of the k towers of b we proceed as above in storing the hash values h_{ij}, $1 \leq j < l$ where tower T_i has height l. The only difference is that both the hash of the bridge $H(b)$ and the representative hash values of all the towers in b are stored in the top-left node of b.

Queries Answering queries is performed as normal by searching for an element in a skip list. All needed hash values can be collected as the search is performed. In particular, by storing all the representative labels of a bridge of size more than one at the top-left node of it, we ensure that all the information that is needed to be part of the answer authentication information is present when needed. Indeed, the entry node of a new bridge is its top-left node, so all the hash values needed for the recomputation by the user of the hash value of the bridge are present at this node. We only need to take care for one thing. If the i tower of bridge b is followed by the search procedure, we need to omit the representative hash value $H(T_i)$ of this tower from the answer authentication information in order to save communication cost. This can be achieved by implementing the skip list search using recursion and returning value i accordingly or by cashing this information and performing a second pass.

Updates Any update operation of the skip list is performed as normal; recall that the actual skip list is not changed at all. The only operations that need be performed is the recomputation of the hash values that are affected by any change in the structure of the skip list. An insertion of a tower of level k may separate up to $k-1$ bridges and either creates a new bridge of size one or just increases the size of a bridge by one. In any case, all the rehashing can be easily performed by augmenting the insertion procedure. Similarly, a deletion of a tower of level k may merge up to

$k-1$ bridges and either delete a bridge or decrease its size by one. Again, in one pass (the deletion procedure) all the necessary rehashing operations can be performed.

Theorem 9 *Let $L(n) = \log_{\frac{1}{p}} n$, where p is the probability with which an element at level k of the skip list is copied in the list of level $k+1$. For any fixed element x in the skip list, the multi-way skip list authentication scheme has the following expected performance: (i) $E[N_H] \leq 2(1-p)L(n)+O(1)$; (ii) $E[N_L] \leq (1-p)(2p+2+\frac{1-p^2}{p})L(n)+O(1)$; (iii) $E[S] \leq \frac{n}{1-p}$.*

Proof: We consider traveling on the search path π of an element x backwards. We assume a worst case analysis, where π reaches level $L(n)$. We split π in two parts π_1 and π_2: the part of path π that take us to level $L(n) = \log_{\frac{1}{p}} n$ and the part of π that completes the backward search. We then bound the expected value of the quantity (authentication cost) under consideration studying separately the quantities that correspond to the subpaths π_1 and π_2. For the authentication costs that correspond to π_1, we assume an infinite skip list, that is no header tower is present.

For both authentication costs N_H and N_L, again, we consider the subcosts N_1, N_2 that correspond to subpaths p_1 and p_2.

For the number of hash computations N_H and N_1, we proceed as follows. If $C_k(t)$ counts the authentication cost for element x when k upwards moves remain to be taken and we are performing the tth step, then if we move up $C_k^t = X_1 + C_{k-1}^{t+1}$, otherwise (we move to the left) $C_k^t = X_2 + C_k^{t+1}$, where X_1, X_2 are 0-1 random variables that count if a hash computation is added to the cost when we move up or left respectively. Now, $\Pr[X_1 = 1] = p(1-p)$, because with probability $1-p$ the node that the forward pointer points to is a plateau node and with probability p the node that we move to is not a plateau node (otherwise we are at the top of a bridge). Moreover, $\Pr[X_2 = 1] = p+p(1-p)$, because with probability p we just leave a bridge and with probability $(1-p)$ the bridge has size more that one. Using conditional expectation, we get that $E[C_k] = p(E[C_{k-1}] + E[X_1]) + (1-p)(E[C_k] + E[X_2])$ and finally that $E[C_k] = 2(1-p)k$. So, since $N_1 \leq_{\text{prob}} C_{L(n)-1}$, we have that $E[N_1] \leq 2(1-p)L(n) + O(1)$. For N_2, and using the same upper bounds as in the proof of the standard scheme, we have that $E[N_2] \leq \frac{1}{p} + 1 = O(1)$. So,

$$E[N_H] \leq 2(1-p)L(n) + O(1).$$

For the number of hash labels N_L and N_1 we have that, if $C_k(t)$ counts the authentication cost for element x when k upwards moves remain to be taken and we are performing the tth step, then if we move up $C_k^t = X_1 + C_{k-1}^{t+1}$, otherwise (we move to the left) $C_k^t = X_2 + C_k^{t+1}$. Here X_1 and X_2 counts the number of hash labels that we have to add when moving up or left respectively. We have that $E[X_1] = 2p(1-p)$ because with probability $p(1-p)$ we have 2 hash labels while moving up. Also $E[X_2] = p(2 + \frac{1-p^2}{p})$, because with probability $p(1-p)$ we have Y hash labels while moving left and $E[Y] = \frac{1-p^2}{p}$ and with probability p we have another two hash labels. Using conditional expectation and putting everything together, $E[C_k] = p(E[C_{k-1}] + E[X_1]) + (1-p)(E[C_k] + E[X_2])$ and $E[C_{L(n)-1}] = (1-p)(2p+2+\frac{1-p^2}{p})L(n) + O(1)$. Once again $E[N_2] \leq \frac{1}{p} + 1$. So,

$$E[N_L] \leq (1-p)(2p+2+\frac{1-p^2}{p})L(n) + O(1).$$

Finally, for the number of labels stored in the data structure we use the trivial bound of the expected total number of nodes in the skip-list, so $S < \frac{n}{1-p}$. □

From Lemma 4 and Theorem 9, we have a version of skip list with optimal expected number of comparisons, up to an additive constant factor.

Theorem 10 *There is a skip list version such that the expected number of comparisons for a fixed element x is $E[N_L] - E[N_H] \leq \frac{(1-p)(p^2+1)}{p \log \frac{1}{p}} \log_2 n + O(1)$, which is $1.25 \log_2 n + O(1)$ for $p = \frac{1}{2}$.*

Proof Sketch: Follows from Lemma 4 and Theorem 9. The idea is that for a bridge of size k, $k-1$ (instead of k) comparisons are necessary for the search to go on. □

7 Analysis of Other Authentication Schemes

We now examine the authentication cost for authenticated dictionaries based on using balanced search trees, where the authentication scheme G is defined in the straightforward way, by directing towards its root the search tree. We then analyze the authentication cost of the authenticated dictionary based on skip list appeared in [9].

Balanced Trees We focus on the red-black trees and (a, b)-trees that implement multi-way tree structures. First we consider the *data organization* issue, i.e., whether it is better data items to be stored in the internal nodes of the tree or in the leaves. A tree is *node-oriented* if it stores data elements in internal nodes and *leaf-oriented* if it stores data elements in the leaves. In terms of authentication cost, *leaf organizations* seem to be preferable in general. Each node-oriented T_n tree has a leaf-oriented counterpart T_l which has L_n more nodes, where L_n is the number of leaves in T_n. Even though, paths in T_l are now longer, the authentication cost in T_n is on average greater, since the elements that are stored in interval nodes of the tree are also hashed. Even when elements have short binary length, given that on average a search path is closer to the leaves of T_n, the authentication extra overhead is big. The overhead increases when elements have large binary lengths. Hashing once more large elements to a smaller hash value, but thus, increasing the storage overhead, may help depending on the relative size of the elements.

Considering leaf-oriented trees, another issue is the extra authentication overhead that is added when update operations (insertions/deletions) that restructure the tree and then rebalancing operations are performed. It turns out that for both operations the overhead is *constant* for red-black trees and on average *constant* for (a, b)-trees. This authentication cost is proportional to the number of specific type of rotations for red-black trees and of split operations in (a, b)-trees. Red-black trees seem to be more efficient than (a, b)-tress is this respect. Red-black trees have also the advantage of achieving close to theoretically optimal average performance: for randomly built trees, about $1.002 \log n$ comparisons are performed for a search operation [23].

Skip List We now analyze the "standard hashing scheme" for skip lists described in [9]. Figure 7 illustrates the hashing scheme, where only the hash computations are shown and not the actual hash values that are stored in the data structure. We call this hashing scheme as *standard* skip list scheme.

Theorem 11 *Let $L(n) = \log_{\frac{1}{p}} n$, where p is the probability with which an element at level k of the skip list is copied in the list of level $k+1$. For any fixed element x in the skip list, the standard hashing scheme for skip lists has the following expected performance: $E[N_H] \leq \frac{1-p^2}{p} L(n) + O(1)$, $E[N_L] \leq \frac{2(1-p^2)}{p} L(n) + O(1)$, and $E[S] = \frac{n}{1-p}$.*

Proof: We use ideas and techniques as in [21, 22]. Let p be the probability with which an element at level k of the skip list is copied in the list of level $k+1$. Let $L(n) = \log_{\frac{1}{p}} n$. Consider the search path

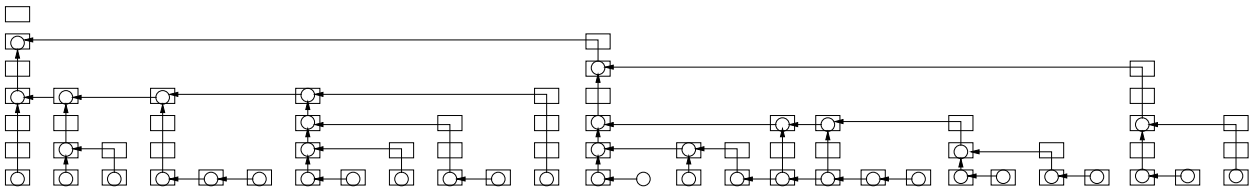

Figure 7: The standard hashing scheme for authenticated skip lists [9].

in a backward fashion. Assuming a worst case scenario, suppose that the search path π from x up to the top node of the header reaches and possibly exceeds level $L(n)$. Then $N_H \leq_{\text{prob}} N_1 + N_2 + N_3$, where N_1 is the number of hash computations that correspond to the subpath of π until the search path reaches level $L(n)$, N_2 is the number of hash computations that correspond to the leftward moves in the search after level $L(n)$ is reached and N_3 is the number of hash computations that correspond to the upward moves in the search after level $L(n)$ is reached. Using ideas as in [22], we can compute the expected values for N_1, N_2 and N_3.

Assume that we have a skip list of infinite size; that is, in our backward search from element x up to level $L(n)$ we never reach the header of the skip list. Let C_k^t be the random variable that equals the authentication cost under consideration (number of hash computations) for element x when k upwards moves remain to be taken and we are performing the tth step. Then, at the tth step, if we move up $C_k^t = X_{k-1} + C_{k-1}^{t+1}$, where X_i is a 0-1 random variable that is 1 when the forward pointer of the new node in the skip list leads to a plateau element, otherwise (we move to the left) $C_k^t = 1 + C_k^{t+1}$. We first observe that since we have an infinite skip list, $C_k^t \sim C_k^{t+i} \sim C_k$ for any $i > 0$ and that $X_i \sim Bern(1-p)$. Thus, using conditional expectation

$$\begin{aligned} E[C_k] &= E[E[C_k|\text{move}]] \\ &= E[p(X_{k-1} + C_{k-1}) + (1-p)(1 + C_k)] \\ &= p(E[X_{k-1}] + E[C_{k-1}]) + (1-p)(1 + E[C_k]) \\ &= \frac{1-p^2}{p}k \end{aligned}$$

Since, $N_1 \leq_{\text{prob}} C_{L(n)-1}$, we get that $E[N_1] \leq \frac{1-p^2}{p}(L(n)-1)$.

On the other hand, $N_2 \leq_{\text{prob}} Bin(n, \frac{1}{np})$; the number of hash computations that correspond to leftward moves at level $L(n)$ or higher is exactly the number of these leftward moves, with the latter being less than the number Y of towers of the skip list with level $L(n)$ or higher. But $Y \sim Bin(n, \frac{1}{np})$, since with probability $p^{L(n)-1} = \frac{1}{np}$ a tower has level $L(n)$ or higher. Thus, $E[N_2] \leq \frac{1}{p}$.

For $N_3 \leq_{\text{prob}} Bin(G(1-p), 1-p)$; the number of hash computations that correspond to upward moves at level $L(n)$ or higher is less than the number Y of hash computations that correspond to moving upwards in the highest skip list towers starting at level $L(n)$. $Y \sim Bin(G(1-p), 1-p)$, because the length of of this path is geometrically distributed with parameter $1-p$ and the we add one hash computation when, with probability $1-p$, the forward node is a plateau node. Thus, $E[N_2] \leq 1$ and, finally, $E[N_H] \leq \frac{1-p^2}{p}(L(n)-1) + \frac{1}{p} + 1$.

Finally, note that $N_L = 2N_H$ and that S is the exactly the storage of the skip list, thus $E[S] = \frac{n}{1-p}$. □

8 Comparison

Table 1 summarizes our main results on the authentication overhead of hashing schemes for authenticated dictionaries. To simplify the comparison, we choose parameter $p = \frac{1}{2}$ for the skip lists. All numbers correspond to expected values of the corresponding cost parameters where the lower order additive terms are omitted.

	N_H	N_L	C	S
red-black tree	$\log n$	$2 \log n$	$\log n$	$2n$
standard skip list	$1.5 \log n$	$3 \log n$	$1.5 \log n$	$2n$
multi-way skip list	$\log n$	$2.25 \log n$	$1.25 \log n$	$2n$

Table 1: Theoretical comparison of the schemes for $p = 0.5$. N_H: number of hash computations, N_L: number of hash labels that are hashed, C: communication cost (number of hash labels), S: storage (number of hash labels). Recall that hashing time is $T \leq c_1 \frac{L}{448} N_L + cN_H$ ($L = 160$ or $L = 128$). All numbers correspond to expected values of the corresponding cost parameters where the lower order additive terms are omitted.

We have conducted experiments that measure the space overhead S and the cost parameters N_H and N_L for authenticated dictionaries based on red-black trees, standard skip lists and multi-way slip lists. Recall that S is the number of hash values that are stored in the data structure and correspond to the number of nodes of the associated hashing scheme G. Also, recall that for a given element x of the dictionary, N_H and N_L are respectively the number of hash computations and the number of hash values that participate in these computations needed for a verification process on a query about x; they correspond respectively to the number of nodes and the sum of their degrees of the path in G from element x up to the single sink of G.

For the experiments, a leaf-oriented data organization for red-black trees was chosen. This choice is justified by the discussion of Section 7, but it also leads to a better comparison, given the fact that skip lists are also "node-oriented" data structures. Also, for the skip list schemes, the value $p = \frac{1}{2}$ was used as the probability with which a node of the skip list at level i is copied to level $i + 1$.

Storage Overhead S Figure 8 shows the storage overhead S for the three schemes as a function of the data set size n. For red-black trees, the storage overhead is always exactly $2n - 1$, since there are exactly $2n - 1$ nodes in the tree and each node is storing a hash value. This value for red-black trees is compared with the average storage overhead that each of the skip list schemes achieves. In Figure 8, the value of the storage overhead S for each skip list scheme correspond to an average of 20 different realizations of this scheme. We computed these values for dictionaries of up to 1.5 million elements.

From Figure 8, we see that the storage overhead for the standard skip list scheme is close to expected value $\frac{n}{p}$ of the total storage of the skip list data structure. Instead, the multi-way skip list scheme achieves a much lower storage overhead, which for $p = \frac{1}{2}$, is close to $1.41n$. Note that in Theorem 9 (and also in Table 1) the stated expected value of S for our new skip list scheme is clearly an *overestimate*. In fact, the hashing scheme G of the multi-way skip list saves a significant amount to hash values. To see why, consider a tower t of level k. In the standard skip list exactly k hash values are stored in t. We define the *plateau degree* d of t to be the number of plateau nodes that are reachable in one step from nodes of t. In the multi-way skip list, if t belongs in a bridge on

size 1, then only $d \leq k$ hash values are stored; otherwise, for each bridge of size $b > 1$, we save $b-2$ hash values. We conclude that the multi-way skip list is superior in terms of storage overhead.

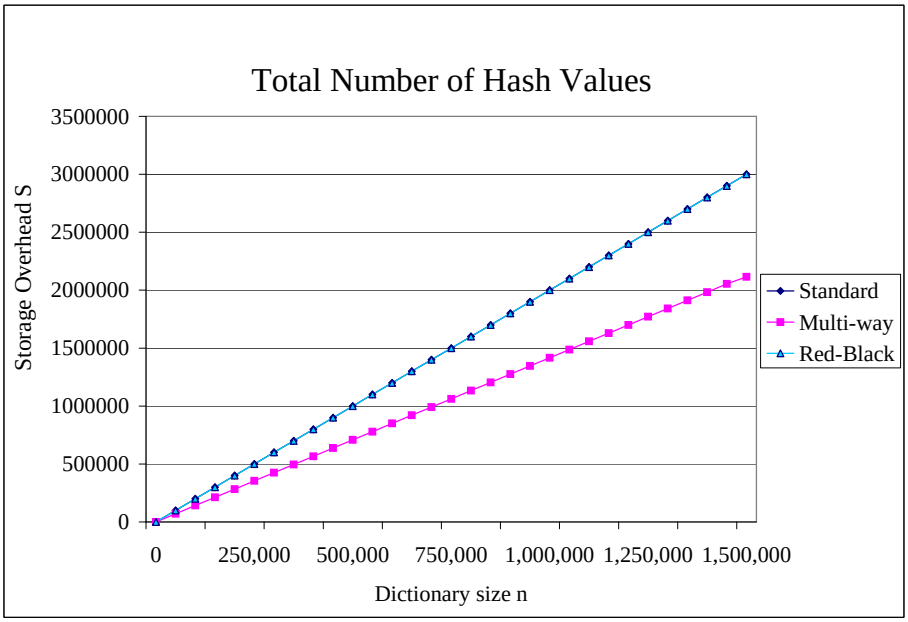

Figure 8: Storage Overhead S for authenticated dictionaries based on Standard skip list, Multi-way skip list and Red-Black trees. Multi-way skip lists use significantly less storage overhead. For multi-way skip lists, experimentally we have that $S \simeq 1.41n$; the value $2n$ in Table 1 is only a crude overestimation.

Authentication Cost Finally, Figures 9 and 10 show the authentication cost for the three authentication schemes in terms of the cost measures N_H and N_L, where we computed the *total average authentication cost*, that is, the average values $\bar{N}_H$ and $\bar{N}_L$ of N_H and N_L, over *all elements* in the dictionary. In essence, values $\bar{N}_H$ and $\bar{N}_L$ correspond to *average path authentication cost* of the authenticated data structures that we study. For skip lists we computed the average of $\bar{N}_H$ and $\bar{N}_L$ over a series of 20 realizations. We computed the authentication cost for dictionaries of up to 1.75 million elements.

From Figure 9, we can see that with respect to cost $\bar{N}_H$, the multi-way skip list and the red-black tree achieve cost close to the theoretical value of $\log n + O(1)$. They both have good performance. Figure 10 suggests that with respect to cost $\bar{N}_L$ in practice red-black trees perform better. Indeed, the multi-way skip list scheme achieve values for $\bar{N}_L$ that are away from $2\log n$. The reason is that. since $\bar{N}_L$ counts the average path authentication cost, elements under fat bridges they all contribute a high cost to N_L. Also the corresponding constant factor seem to be larger, i.e., in formula $2\log n + c$, c is larger for multi-way skip lists.

References

[1] A. Anagnostopoulos, M. T. Goodrich, and R. Tamassia. Persistent authenticated dictionaries and their applications. In *Proc. Information Security Conference (ISC 2001)*, volume 2200 of *LNCS*, pages 379–393. Springer-Verlag, 2001.

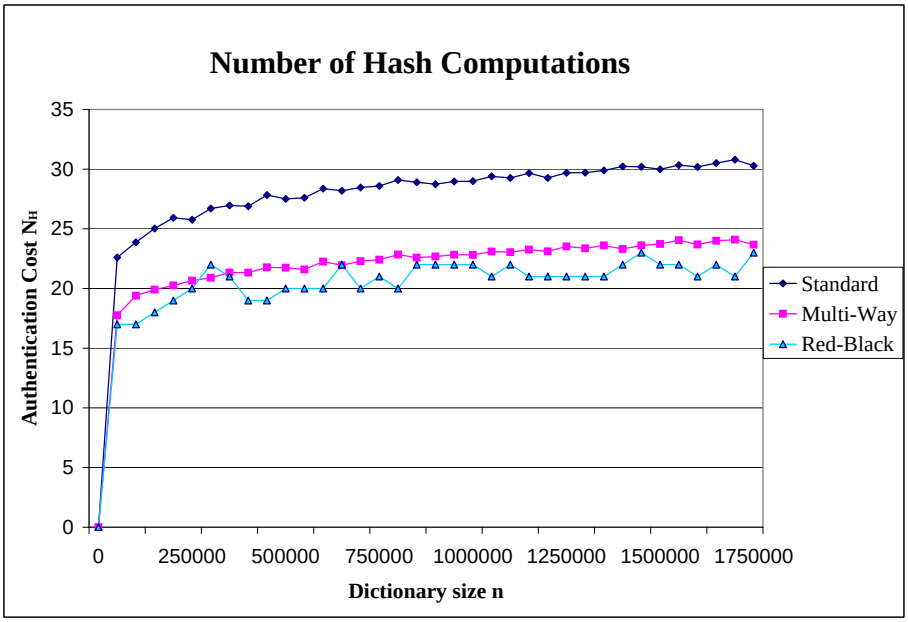

Figure 9: The total average authentication cost $\bar{N}_H$.

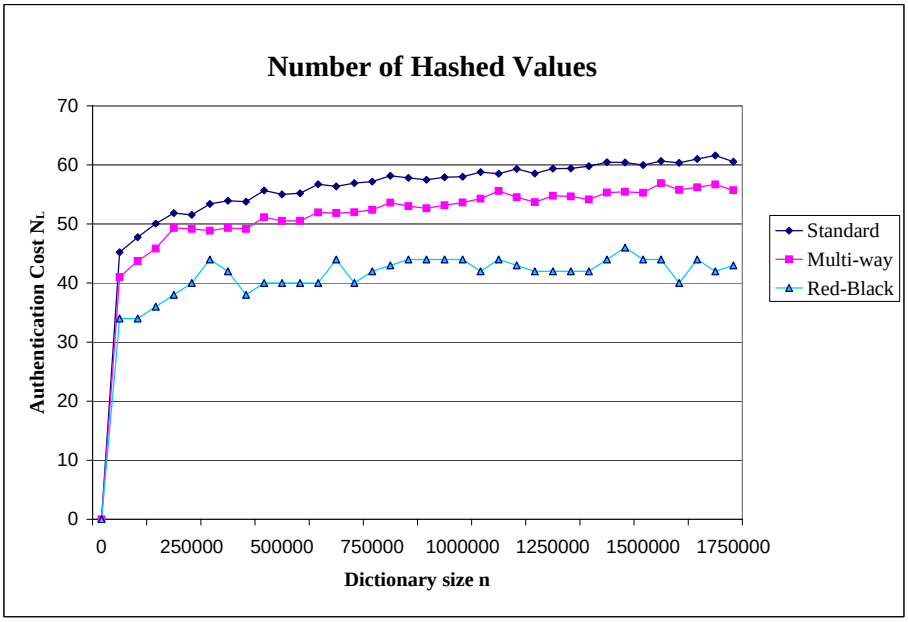

Figure 10: The total average authentication cost $\bar{N}_L$.

[2] A. Buldas, P. Laud, and H. Lipmaa. Accountable certificate management using undeniable attestations. In *ACM Conference on Computer and Communications Security*, pages 9–18. ACM Press, 2000.

[3] J. Camenisch and A. Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *Proc. CRYPTO*, 2002.

[4] P. Devanbu, M. Gertz, A. Kwong, C. Martel, G. Nuckolls, and S. Stubblebine. Flexible authentication of XML documents. In *Proc. ACM Conference on Computer and Communications Security*, 2001.

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

[6] I. Gassko, P. S. Gemmell, and P. MacKenzie. Efficient and fresh certification. In *Int. Workshop on Practice and Theory in Public Key Cryptography (PKC '2000)*, volume 1751 of *LNCS*, pages 342–353. Springer-Verlag, 2000.

[7] M. T. Goodrich, J. Lentini, M. Shin, R. Tamassia, and R. Cohen. Design and implementation of a distributed authenticated dictionary and its applications. Technical report, Center for Geometric Computing, Brown University, 2002. Available from `http://www.cs.brown.edu/cgc/stms/papers/stms.pdf`.

[8] M. T. Goodrich, M. Shin, R. Tamassia, and W. H. Winsborough. Authenticated dictionaries for fresh attribute credentials. In *Proc. Trust Management Conference*, volume 2692 of *LNCS*, pages 332–347. Springer, 2003.

[9] M. T. Goodrich and R. Tamassia. Efficient authenticated dictionaries with skip lists and commutative hashing. Technical report, Johns Hopkins Information Security Institute, 2000. Available from `http://www.cs.brown.edu/cgc/stms/papers/hashskip.pdf`.

[10] M. T. Goodrich, R. Tamassia, and J. Hasic. An efficient dynamic and distributed cryptographic accumulator. In *Proc. of Information Security Conference (ISC)*, volume 2433 of *LNCS*, pages 372–388. Springer-Verlag, 2002.

[11] 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.

[12] M. T. Goodrich, R. Tamassia, N. Triandopoulos, and R. Cohen. Authenticated data structures for graph and geometric searching. In *Proc. RSA Conference—Cryptographers' Track*, pages 295–313. Springer, LNCS 2612, 2003.

[13] P. C. Kocher. On certificate revocation and validation. In *Proc. Int. Conf. on Financial Cryptography*, volume 1465 of *LNCS*. Springer-Verlag, 1998.

[14] P. Maniatis and M. Baker. Enabling the archival storage of signed documents. In *Proc. USENIX Conf. on File and Storage Technologies (FAST 2002)*, Monterey, CA, USA, 2002.

[15] P. Maniatis and M. Baker. Secure history preservation through timeline entanglement. In *Proc. USENIX Security Symposium*, 2002.

[16] C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong, and S. Stubblebine. A general model for authentic data publication, 2001. Available from `http://www.cs.ucdavis.edu/~devanbu/files/model-paper.pdf`.

[17] R. C. Merkle. Protocols for public key cryptosystems. In *Proc. Symp. on Security and Privacy*, pages 122–134. IEEE Computer Society Press, 1980.

[18] R. C. Merkle. A certified digital signature. In G. Brassard, editor, *Proc. CRYPTO '89*, volume 435 of *LNCS*, pages 218–238. Springer-Verlag, 1990.

[19] M. Naor and K. Nissim. Certificate revocation and certificate update. In *Proc. 7th USENIX Security Symposium*, pages 217–228, Berkeley, 1998.

[20] D. J. Polivy and R. Tamassia. Authenticating distributed data using Web services and XML signatures. In *Proc. ACM Workshop on XML Security*, 2002.

[21] W. Pugh. Skip list cookbook. Technical Report CS-TR-2286, Dept. Comput. Sci., Univ. Maryland, College Park, MD, July 1989.

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

[23] R. Sedgewick. *Algorithms in C++*. Addison-Wesley, 2001.

[24] M. Shin, C. Straub, R. Tamassia, and D. J. Polivy. Authenticating Web content with prooflets. Technical report, Center for Geometric Computing, Brown University, 2002. `http://www.cs.brown.edu/cgc/stms/papers/prooflets.pdf`.