Efficient Content Authentication over Distributed Hash Tables

Roberto Tamassia* Nikos Triandopoulos*

November 6, 2005

Abstract

We study a new model for data authentication over peer-to-peer storage networks, where data is stored, queried and authenticated in a totally distributed fashion. The model captures the security requirements of emerging distributed computing applications. We present an efficient implementation of a *distributed Merkle tree*, which realizes a Merkle tree over a peer-to-peer network, thus extending a fundamental cryptographic authentication technique to a peer-to-peer distributed environment. We show how our distributed Merkle tree can be used to design an efficient *authenticated distributed hash table*. Our scheme is built on top of a broad class of existing distributed hash table implementations, is efficient, and achieves generality by only using the basic functionality of object location. We use this scheme to implement the first *distributed authenticated dictionary*.

1 Introduction

Peer-to-peer networks provide the basis for the design of fully decentralized distributed systems, where data and computing resources are shared among participating peers. Many distributed systems of this type have been designed and developed (e.g., [9, 12, 42, 44, 45, 48]). Properties of such systems include scalability, self-stabilization, data availability, load balancing, and efficient searching. As peer-to-peer networks become more mature and established, a growing number of new applications emerge for them, with a corresponding need for assuring security in such applications.

In this paper, we study data authentication in peer-to-peer networks, where data originated at a trusted source is shared and dispersed over remote and untrusted network nodes. We focus our study on peer-to-peer systems realizing a *distributed hash table* (DHT), i.e., a system supporting the basic put()-get() functionality over shared data objects. Various efficient implementations of DHTs [12, 42, 44, 45, 48] provide the core framework for designing and implementing more complex distributed applications. As more and more applications are built using DHTs, the demand of security for them increases. However, most of the existing authentication techniques for DHTs are static and centralized. For instance, existing DHTs that support data authentication use signatures and cryptographic hash functions on a per-object basis. In particular, there is currently no distributed implementation of the widely-used *Merkle tree* authentication scheme [36].

We introduce a new model for data authentication in peer-to-peer networks, based on DHTs, and present an efficient implementation of this model for performing dictionary operations. Our model extends the client-server model of authenticated data structures [17, 20, 34, 39]. Our scheme is based on the design of an efficient *distributed Merkle tree* — the first, to the best of our knowledge, distributed version of a Merkle tree [36].

Our distributed authentication scheme can be used also over different types of networks (e.g., ad-hoc, sensor and overlay networks) and enjoys properties that make it appropriate for realizing more general distributed trees over peer-to-peer systems.

*Brown University,{rt,nikos}@cs.brown.edu

1.1 Previous and Related Work

We overview previous and related work as follows.

Merkle Trees The Merkle tree [36] is a widely-used scheme in security applications and cryptographic constructions. The idea is to use a tree and a cryptographic collision-resistant hash function to produce a short cryptographic description of a large data set. Elements of the set are stored at the leaves of the tree and internal nodes store the result of applying a cryptographic hash function to the values of the children nodes. The authentication of an element is performed using a verification path, which consists of the sibling nodes of the nodes on the path from the leaf associated with the element to the root of the tree. The root value is signed and the collision-resistant property of the hash function is used to propagate authentication from the root to the leaves. This construction is simple and efficient and achieves *signature amortization*, where only one digital signature is used for signing a large collection of data. Updates in the Merkle tree are handled with complexity proportional to the height of the tree [39]. An extension to the symmetric-key setting is given in [21], where it is shown that verification along a path can be performed in parallel.

Authenticated Data Structures An authenticated data structure is a client-server model for data authentication [17, 20, 34, 39] where data is queried not from the trusted data source, but rather from a different, untrusted, entity. The cryptographic technique of signature amortization is used, similarly to the Merkle tree. A significant amount of work has been on developing efficient authenticated data structures, starting from the certificate revocation problem [5, 15, 26] and the design of authenticated dictionaries [1, 18, 19] and continuing with authenticated data structures for more general queries [4, 11, 20, 34]. Work related to authenticated data structures includes [6, 10, 16, 31, 32, 41, 49].

Distributed Hash Tables and P2P Storage Systems Distributed hash tables (DHTs) (see e.g., [25, 33, 40, 42, 44, 48]) are fundamental distributed data structures that support the basic functionality of put() and get() operations on key-value pairs. DHTs are based on randomized searching techniques in distributed environments. For a broad class of DHTs, an object is located with $O(\log n)$ expected communication steps, where n is the number of nodes of the DHT. Related distributed data structures are studied in [2, 3, 22] and, in a different setting, in [24, 28]. With advances in distributed object searching and the development of DHTs, several practical distributed storage systems over peer-to-peer networks have been designed and implemented. Examples include PAST [12], CAN [44], CFS [9] and OpenHDT [45].

Trees over DHTs After the development of distributed hash tables, many researches have designed search trees, aggregation trees and other type of trees over a DHT (see, e.g., [8, 13, 29, 43, 51]). However, these trees can not be used to implement a distributed Merkle tree. First, all of these trees are static, that is, they do not support dynamic updates on them. Second, these constructions are either search trees or special-purpose trees that are actually not appropriate to realize a distributed Merkle tree. Note that a distributed Merkle tree is very sensitive to node losses or structural changes because of the use of the cryptographic hash function. Unlike other trees, an authentication tree is sensitive in accuracy and correctness.

Security in P2P Systems Some security issues related to peer-to-peer systems are discussed in [47], where the authentication problem is considered solved. Work related to security issues and authentication on networks include [7, 27, 30, 37, 38, 46, 50]. Existing P2P storage systems,

e.g., [9, 12, 42, 44, 45], support an elementary authentication service for the stored data, where the retrieval of a stored data object is verified to be authentic by the requesting entity. This service adopts the so-called *self certified data* as introduced in [14]. The idea is that the data owner, before it inserts an object, digitally signs it using his private key (a PKI is assumed). The signature becomes part of the inserted object and when a data item is large and gets partitioned into blocks that are stored as separate objects in the system, then these blocks are cryptographically binded using collision-resistant hashing and some tree-like hierarchy among the blocks. Accordingly, in this case the root-block is digitally signed. Although, this authentication technique certainly resembles a Merkle tree, we observe that this mechanism is not as efficient as a Merkle tree. Signature amortization is performed only among a large data item and not among different data items. That is, data items are separately signed. As we discuss in the last section, this has certain disadvantages with respect to the cost of maintaining signed objects in the system, because signatures must by renewed at frequent time intervals. Additionally, the use of the authentication tree is static, meaning that no updates are performed. With our distributed Merkle tree, we propose a different method for authenticating objects in distributed storage systems, which has many advantages for data integrity in distributed systems.

1.2 Our Contributions

Our contributions are summarized as following:

- We introduce a new model for distributed data authentication that extends previous models based on the client-server computing paradigm. Our model captures the security requirements for data authentication that arise in peer-to-peer distributed storage systems.
- We present the first efficient scheme for implementing a *distributed Merkle tree* using the primitive object location functionality exported by a peer-to-peer distributed system. Our scheme is based on the $BB[\alpha]$ tree, a weight-balanced binary tree, and has certain properties that allow it to be efficiently distributed over a peer-to-peer network. We analyze its performance and compare it with other naive implementations of a distributed Merkle tree.
- We present an efficient *authenticated distributed hash table* (ADHT), which extends existing (non-authenticated) DHTs in various ways. We compare our ADHT with other DHT-based distributed storage systems with respect to the cost of data authentication.

1.3 Organization of the Paper

The rest of the paper is organized as follows. In Section 2 we describe a new model for distributed data authentication which extends the one of authenticated data structures and we briefly discuss the techniques that we use to implement this model. In Section 3 we present our main result, the implementation of a distributed Merkle tree over a peer-to-peer network and we analyze its performance. In Section 4 we show how our distributed Merkle tree can be used to realize an authenticated distributed hash table, which in turn can support a more general data authentication scheme. We conclude and discuss open problems in Section 5.

2 Model

In this section, we present our new distributed data authentication architecture. Our model extends the one of authenticated data structures. Informally, data is stored and queried in a totally distributed fashion, where answers to queries are accompanied by proofs verifying their authenticity. In particular, the model consists of:

- a *data source* S maintaining a (possibly structured) set D of elements;

- a distributed peer-to-peer network $\mathcal{N}$ that stores set D on behalf of the source and supports queries about D by providing both the *answer* to a query and a *proof* of the validity of the answer; and
- users who issue queries about D by accessing the peer-to-peer network $\mathcal{N}$ and verify the validity of the answer using the proof.

Data set D is dynamic, that is, it evolves in time through update operations submitted by the source to the the network $\mathcal{N}$. We are interesting in secure authentication schemes that impose low computational, communication and storage overhead to the participating parties and the underlying network. In particular, the cost parameters of the authentication scheme are:

- Storage cost: the amount of information stored at the source, the network and a user;
- Update cost: the computational and communication costs incurred at the source and the network when updates to data elements occur;
- Query cost: the computational and communication cost incurred by the network to answering queries;
- Verification cost: the computational cost incurred by a user to verify the validity of an anwer to a query.

Informally, we describe the security requirement that any authentication scheme should satisfy as follows. An authentication scheme is secure, if for any query issued by a user to the network $\mathcal{N}$, no polynomial-time adversary controlling network $\mathcal{N}$ and having oracle access to the authentication scheme has non-negligible (on some security parameter) advantage in causing a user to accept (verify as correct) an incorrect answer.

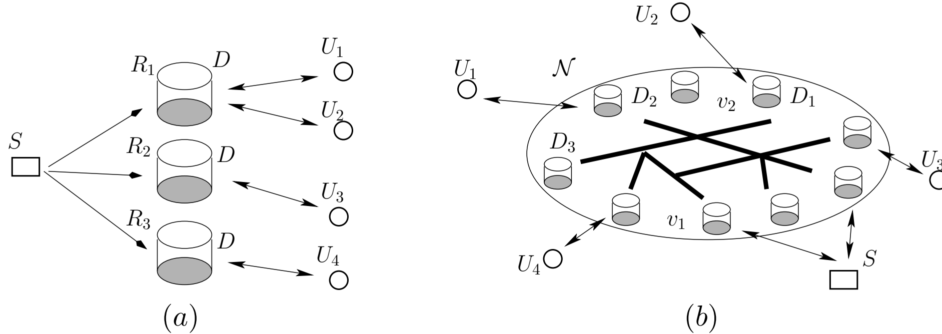

Figure 1: (a) Authenticated data structure: responders R_1, R_2 and R_3 store set D on behalf of source S and answer queries by users. (b) Distributed authentication over a peer-to-peer network $\mathcal{N}$: the data source and users contact any nodes of $\mathcal{N}$; data and authentication information is distributed in the network.

The above model differs from the authenticated data structures as follows (see Figure 1).

- Data elements and authentication information are distributed over the network nodes, whereas in an authenticated data structure, each responder node stores a copy of the entire data set D and all the authentication information.
- The user and the source do not have any access to the internal structure of the distributed network. They interact with it through a transparent specific minimal interface.

Our distributed architecture for data authentication enjoys all the advantages of distributed, peer-to-peer systems and also has additional benefits in comparison to authenticated data structures:

- The authentication structure effectively inherits all properties of the underlying peer-to-

peer system, including anonymity, fault tolerance against failures of participating nodes, and caching.

- The architecture is resilient against denial-of-service attacks since answering queries does not rely solely on a single network node.

Applying standard authentication techniques, such as signature amortization and hash trees, to our distributed authentication model is a challenging task. Consider, for instance, authentication with a standard Merkle hash tree. If the hash values are treated as regular data elements, they will be stored at some nodes of the network without any associated information that helps their efficient retrieval.

Distributed Hash Tables We use a *distributed hash table* (*DHT*) (see, e.g., [12, 42, 44, 45, 48]) as our underlying distributed peer-to-peer network. This choice has two advantages. First, we use a system widely accept by the computer systems community, which allows our scheme to leverage existing peer-to-peer architectures. Second, since DHTs have an elegant, minimalistic interface consisting of operations put() and get(), we build authentication on top of a simple functionality. In fact, our scheme is defined with respect to an even simpler primitive operation, locate(), which returns the id of a network node corresponding to a given abstract id.

Definition 1. *A* distributed hash table *(DHT) is a peer-to-peer distributed architecture that exports the basic functionality* node locate(key)*, where given a key (object identifier) a node (id) storing the key is returned.*

Most of the existing DHT implementations have the following properties:

- A DHT with n network nodes uses $O(\log n)$ storage per node and performs a location operation in $O(\log n)$ network hops (node-to-node communication steps through TCP connections) with high probability. As a result, the basic hash-table operations put() and get() (for storing and respectively retrieving a key-value pair) take $O(\log n)$ network hops with high probability.
- Node additions, deletions, and failures are handled dynamically through a distributed algorithm that incrementally updates the routing information. In particular, nodes deletions are eventually advertised to the entire network.
- Some form of redundancy is used, which replicates data objects to a constant number of neighboring nodes. Thus, node failures are tolerated also with respect to the data stored at them.
- Caching techniques are used to improve data retrieval.

Using the abstract functionality of a DHT, we extend the model of authenticated data structures to a distributed authentication model that operates over a peer-to-peer network. We design schemes that work for any DHT implementation and do not depend on the details of the implementation. However, when it is appropriate for the efficiency of our scheme, we take advantage of specific properties of the underlying functionality that are commonly present in most DHT implementations.

3 Distributed Merkle Tree

In this section, we show how our efficient implementation of a Merkle tree in a distributed peer-to-peer storage system.

Problem Description Our goal is to implement a *distributed Merkle tree* over a distributed hash table. Numerous security protocols and cryptographic constructions are based on Merkle trees. Thus, implementing a Merkle tree in a fully distributed manner yields distributed versions of such protocols and constructions. On the other hand, the design of a distributed Merkle tree

is a nontrivial task because of the heterogeneity of and strong structural relationships among the data items stored in it, whereas the underlying distributed hash table is a flat data structure.[1]

The following three properties are the primary requirements of a distributed Merkle tree:

1. The tree must be balanced and efficiently maintainable.
2. Verification paths (membership proofs) should be located in a distributed way.
3. Updates should be implemented in a distributed way.

Accordingly, the cost parameters of a distributed Merkle tree are the following:

- Location cost for a verification path (proof);
- Update cost to maintain the authentication structure after updates on the data set; and
- Storage cost.

Both the location and update costs each consists of (*i*) processing cost, i.e., computational cost for the participating nodes in the system, and (*ii*) communication cost, i.e., cost of location operations or direct communications between nodes.

3.1 Straightforward Solutions

We now describe some relatively straightforward solutions that yield schemes, but still give us an insight of what an efficient scheme should achieve. For the discussion below, we assume as a basic scenario that each node i has an object x_i and a distributed Merkle tree is built on top of data set $\{x_1, \ldots, x_n\}$. We assume the use of a balanced tree (thus, of verification paths of logarithmic size).

First Approach: Tree Replication The first approach would be to build a regular hash tree on top of the x_i values and then store the hash values in the tree as new "regular" objects in the system. The first problem to consider is how the hash values are indexed, i.e., with which keys they are stored in the system. The hash value is a value that is unknown to network nodes, thus the value itself cannot be used as a key. A straightforward solution to overcome this problem is to replicate (at least the structure of) the tree to all involved network nodes. Assuming that the tree itself has a canonical representation or that nodes know their structure, we can use the following convention about how hash values are stored in the system. Consider binary trees and a binary encoding of paths in the tree such that each distinct path in the tree has a distinct encoding (e.g., mark paths with labels such that the left (right) child of node with label ℓ has label $\ell 0$ ($\ell 1$)). Then we can use as a key of hash value stored at node v of the tree the label of this node. It turns out that, if nodes have a view of the current hash tree, then we have a functional distributed Merkle tree. The performance is as follows. The cost to construct a verification path is $O(\log n)$ locate operations, that is $O(\log^2 n)$ time and communication cost [2]. However, the cost to maintain the tree, after updates, at each network node is large: an update trigger information of $O(\log n)$ size to reach each network node, which requires the existence of a flooding-type broadcast capability over the distributed hash table and cost $O(n \log n)$. More importantly, the $O(n^2)$ total storage of this approach is prohibitive.

Second Approach: Path Replication Each node stores the entire verification path (verifying hash values) of the object it stores. Thus, $O(1)$ verification path cost is achieved, but now the update cost amounts to $O(n)$ locate operations (since each new hash needs to be given to $O(n)$ other nodes) or $O(n \log n)$ time and communication complexity. Of course, there are some difficulties, like how

[1] Actually, by its design goal, a distributed hash table should be as balanced and symmetric, thus unstructured, as possible.

[2] Recall, a location operation takes $O(\log n)$ time.

a new hash value is computed and by whom; this involves some specific protocol and some specific additional communication between nodes [3]. The total storage cost is $O(n \log n)$.

3.2 Our Approach: Route Distribution

We design an efficient dynamic distributed Merkle tree using route distribution. Here, we present the main ideas of our new scheme for implementing a distributed Merkle tree over a DHT, which is much more efficient than the straightforward solutions. The idea is as follows. Let T be a balanced binary tree defined on top of elements $x_1, \ldots, x_n$. Each tree node u has a tree id $id(u)$. Tree T is used also as a hashing structure, i.e., as a hash tree. That is, a cryptographic hash function is used to label each tree node u with a hash value $L(u)$ (the value that we get by applying the hash function to the labels of its children). The scheme is as follows.

Hash values (tree node labels) are stored as regular values keyed by the corresponding tree id; i.e., label $L(u)$ of tree node u is stored at the network node U corresponding to tree id $id(u)$. We augment this mapping, by additionally storing at U the labels of the children of u. Consider element x stored at leaf node [4] w and let $p = (w, u_1, \ldots, u_k, r)$ be the path from w to the root r of T. The node of the network storing element x is storing information related to path p of tree T. In particular, the stored information at w includes:

- the structural information of path p, i.e., left-right relation of nodes in the path p;
- the balancing information of nodes in path p, i.e., information that is used for restructuring the tree and maintaining its balance;
- sufficient information for locating the hash values of p, namely ids $id(u_1), \ldots, id(u_k), id(r)$. Note that the verification path is completely accessible by this information.

Note that this information does not include any hash values (tree labels).

Thus, this authentication structure allows queried nodes to report the $O(\log n)$ tree nodes storing the hash values in the path. Then the user has to contact $O(\log n)$ nodes, by performing $O(\log n)$ locate operations. Thus, the query cost is $O(\log^2 n)$. In terms of storage, the scheme uses $O(n \log n)$ total storage. Thus, it is a space optimal structure, since the $O(\log n)$ storage per network node overhead is asymptotically the same with the $O(\log n)$ per node storage overhead for the distributed hash table itself (i.e., routing information). With respect to the updates, we note the following. Regarding hash values, hashes along the path have to be recomputed and this can be done using $O(\log^2 n)$ communication cost ($O(\log n)$ location operations suffice in updating $O(\log n)$ hash values). Regarding the update of the tree itself, note that using the balancing information a node can update the structure of the tree generally in $O(\log n)$ time and accordingly advertise the changes to all involved nodes. However, note that although all hash values in p are changed for every update, not all nodes change balancing or structural information and we take advantage of this fact. Only nodes that need restructuring need to be updated and advertised. Using, $BB[\alpha]$ trees, which are weight balanced trees enjoying important properties, we can actually have that on *average* $O(1)$ *rotations* occur and they occur more often at nodes close to leaves than at nodes higher in the tree. Each such rotation involve communication cost proportional to $O(k \log k)$, where k is the size of the subtree rooted at the place where the rotation took place. We expect on overage a good performance. We next give the detailed description of our scheme and its complete analysis.

3.3 An Efficient Distributed Merkle Tree

In this paper and our exposition here, we consider the more general case, where an authentication structure over $m \leq n$ objects is built over the distributed hash table functionality. We start by

[3] Although, we can use ideas similar to techniques for Merkle tree traversal (see, e.g., [23]) to facilitate the update of hash values, still, the cost stays the same.

[4] We treat identically leaf nodes of the tree and corresponding network nodes.

first designing a distributed Merkle tree using the primitive locate() operation over a peer-to-peer network. We consider the basic case where m objects (data items) owned by a single source are stored in the network and, without loss of generality, we assume that objects are stored at distinct network nodes. Later we consider generalizations, where more than one objects are stored at nodes and also where more than one sources produce these objects.

Our scheme is described as follows, where we refer to Figure 2. For convenience, tree nodes are denoted by lower case letters and network nodes by capital ones. Let T a balanced binary tree build over the m objects $x_1, \ldots, x_m$. Tree T is used as a hashing structure in the standard way: each tree node u in T has a unique id $id(u)$ (drawn from a space of tree node ids) and is associated with (or stores, conceptually) a label $L(u)$, which equals to the cryptographic hash $h(L(v_1)\|L(v_2))$ of the hash values that are associated with (stored at) its children v_1 and v_2 and each leaf stores the hash value $h(x_i)$ of the corresponding object x_i. However, we augment the hashing structure in the following manner: we require that internal tree node u with children v_1 and v_2 also stores the hash values of v_1 and v_2.

Each tree node u is mapped to a network node $U = f(u)$ through a function f. Node U stores the (three) hash values associated with node u. Additionally, U stores the tree node ids of the parent tree node and the children of u and local structural information about node u. Moreover, a leaf node w_i, corresponding to object x_i, is also mapped to a network node $W_i = g(x_i)$ through function g [5]. Node W_i stores the following information:

- the object x_i,
- information related to path p_i in T from node w_i to the root r of T; in particular:
  - the ids of the tree nodes of path p in T; we denote this information as $id(p)$;
  - the structural and balancing information of tree nodes in p, that is, for each tree node u in p with children v_1 and v_2, U stores:
    1. whether v_1 or v_2 belongs in p;
    2. the balancing information of node u, which is basically a pair (b_1, b_2) of balancing information related to subtrees defined by v_1 and v_2 respectively.

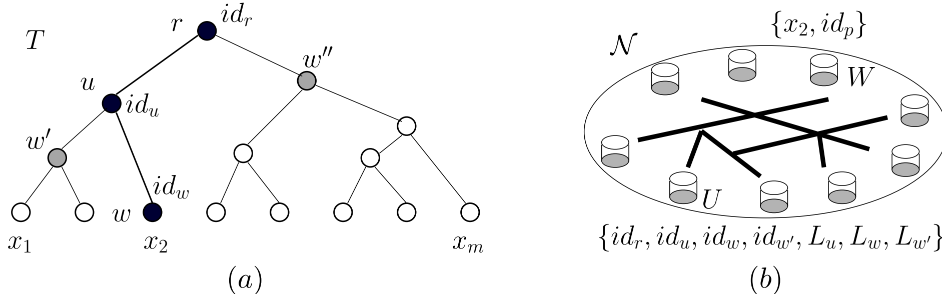

Figure 2: (a) Balanced hash tree T over data items $x_1, \ldots, x_m$. Tree node u is identified by id_u and mapped to network node U. Leaf node w corresponding to data item x_2 with verification path p is mapped to network node W. (b) Tree distribution over the network and information storage. Network node U corresponding to tree node u stores $\{id_r, id_u, id_w, id_{w'}, L_u, L_w, L_{w'}\}$ and local structural information about u. Network node W corresponding to w stores $\{x_2, id_p\}$ and structural and balancing information about p. Note that by contacting nodes in $id(p_2)$ all information about the verification path of x_2 can be retrieved.

[5]We note that we impose no restriction on functions $f(\cdot)$ and $g(\cdot)$. In general, $f = g$; they are the function that is used by the underlying peer-to-peer network. We use two distinct functions only to denote the possibility that more efficient schemes can be designed by having f and g satisfying some appropriate relation.

Note that our scheme, in fact, distributes internal tree nodes and verification paths over a peer-to-peer network $\mathcal{N}$. The tree is designed mainly for bottom-up use [6], which is appropriate for most security related and cryptographic applications. Finally, we choose out tree T to be a weight-balanced tree and, in particular, a $BB[\alpha]$ tree. Our choice is related to the efficiency of the scheme as described below.

Analysis We now discuss the scheme correctness, i.e., we show that the above scheme implements a distributed Merkle tree. We describe how the tree is efficiently accessed, how it is maintainable after updates and also the associated computational, communication and storage costs.

First, regarding the storage efficiency, we can easily see that our scheme requires $O(m \log m)$ storage. Indeed, internal tree nodes are stored in the network each using $O(1)$ information and m leaf nodes are represented each using $O(\log m)$ information, since T is balanced.

Now, assume that a tree T built over $X = \{x_1, \ldots, x_n\}$ is distributed over a network $\mathcal{N}$ and consider the task of accessing or locating the verification hash path corresponding to data item $x_i \in X$ and path p_i, initiated by network node M. Node M first locates the network node W_i that stores x_i (through mapping $g(\cdot)$), then W_i reports to M (through a direct connection) the ids of the tree nodes in the corresponding path p_i of T. Then M can locate the $O(\log m)$ network nodes storing hashing information related to the verification path of x_i; the mapping $f(\cdot)$ have to be used first to map tree ids to network nodes. After contacting $O(\log m)$ nodes, node M has retrieved all the verification information. Note that once a network node is located and contacted, not only the corresponding hash value of the tree node is retrieved but also the hash values of the children nodes. Thus, overall, retrieving the verification hashing path of an item takes $O(\log n \log m)$ cost. In particular, it involves $O(\log m)$ location operations and $O(\log m)$ communication cost (through direct network connections).

Next, consider the problem of maintaining the tree balanced after an update in the hash tree. The simplest update corresponds to simply changing an object x_i to a new object x'_i, where $g(x_i) = g(x'_i)$, that is when the storage location of x_i does not change [7]. We call these updates *hash updates*, since only the hash values are updated. It is easy to see that only $O(\log m)$ hash values need to recomputed, whereas the structure of the tree stays the same. This can be done by node $N = g(x_i)$ (or any other node that initiates the update, locates N and contacts N) locating and contacting the network node U corresponding to the lowest in T node u in p_i and notifying it about the change; then through $O(\log m)$ node locations and contacts every node in p_i updates the hash value it stores and notifies the network node storing its parent in T about the update. Thus, the update cost for this particular type of update is $O(\log n \log m)$, since locating a network node takes $O(\log n)$ communication cost.

Now consider the general case of an update, that is, an operation of the type $\mathsf{insert}(\cdot)$ or $\mathsf{delete}(\cdot)$ on the tree T. (Note that an $\mathsf{replace}(\cdot)$ operation when $g(x_i) \neq g(x'_i)$ corresponds a series of two such operations.) Then, not only $O(\log m)$ hash values need to be updated, but also the tree T needs structural update, due to re-balancing operations. The distributed update process is as follows. Network node W_i responsible for the update on data item x_i performs gradually the update and in a bottom-up fashion, according to its corresponding path p_i. That is, a leaf node is deleted or created and the path is checked using a bottom-up traversal of it, for any necessary restructuring operations (i.e., rotations). Note that all necessary information is available at node W_i for this check (the balancing information of p_i is stored at W_i). Node W_i traverses path p_i and if no

[6] Although it can be easily extended to support also top-down traversal, as a search tree. However, such a use is somehow in contrast with the flat structure of a peer-to-peer location system.

[7] This is not an extreme special case, but rather of typical that often occurs when for instance data objects are actually key-value pairs and only the value is being updated.

rotation is needed while examining tree node u, then the network node $U = f(v)$ corresponding to u is contacted (after it is first located) so that its hash value is recomputed. If additionally a rotation is needed at node u, then node U is notified appropriately by W_i and $f(v)$ executes the rotation by contacting (after first locating) the appropriate nodes among its neighboring in T network nodes. Node W_i is notified about the structural change, i.e., it learns the ids and the balancing information of the new nodes in p'_i. (Alternatively, once the rotation is complete, node W_i updates the balancing information of the affected by the rotation nodes by contacting them.) Then node W_i goes on to the node higher in p_i. Thus, the new path p'_i can be computed completely after $O(\log m)$ location operations and $O(\log m)$ communications between network nodes, that is, the cost for updating the verification path p_i to the new path p'_i is $O(\log n \log m)$.

However, since verification paths are distributed over the network $\mathcal{N}$ and if path p_i structurally changed to p'_i then it must be advertised to the network nodes which leaf nodes in T that belong in subtrees affected by the rotations are mapped to through $g(\cdot)$. We refer to this cost as *structural update cost*. Note that for general trees T the structural update cost can be of order $O(m \log m)$, involving $O(m)$ network node locations: indeed, a rotation at level k of T requires $O(2^k)$ location operations, proportional to the size of the affected subtree of T. This is because, the change due to a rotation must be distributed to the appropriate nodes of network in a top-down fashion by a series of node locations and communications. However, recall that our scheme uses a weight-balanced $BB[\alpha]$ tree as T, where the weight of a tree node is the number of leaves in the subtree defined by this node and α is a balance parameter. Our choice is justified by the following lemma. Over a linear number of update operations, our scheme requires only a logarithmic number of node locations.

Lemma 1. *For a series of $O(m)$ updates operations on an initially empty hash tree, a distributed Merkle tree T based on a weight-balanced $BB[\alpha]$ tree, with $\alpha \in (\frac{1}{4}, 1 - \frac{\sqrt{2}}{2})$, has $O(\log n \log m)$ amortized structural updated cost. In particular, during this series of tree updates, structural updates at level k of T with cost $O(2^k)$ occur with frequency $O(\frac{1}{2^k})$.*

Proof. The proof is based on the update technique in our scheme and the properties of $BB[\alpha]$ trees (e.g., see analysis in [35]). If path p_i is structurally updated to p'_i, let u^* be the node of p_i where a rotation took place and no other rotation occurred at an ancestor node of u. Let T^*_u be the subtree in T defined by u^*. Then all network nodes L^*_u corresponding to leaf nodes in T^*_u must update their paths, because for all these paths at least one tree node has changed (due to the rotation at u^*). This update in T can be done (once for all, for the entire update due to all rotations) by distributing the updates to nodes in L^*_u through network nodes corresponding to subtree T^*_u. The distribution occurs in a top-down fashion and by network nodes locating the nodes corresponding to their children and communicating to them the relevant updates [8]. The whole process is complete by using $O(|T^*_u|)$ node locations and $O(|T^*_u|)$ communication. Thus, overall we have that the structural update cost is $O(\log n \times (\log m + |T^*|))$, where $|T^*|$ is the size of the maximum subtree in T affected by the update. Again, the $\log n$ factor is due to location operations.

Since T is a weight-balanced $BB[\alpha]$ tree [9] with parameter α in an appropriate range and a rotation at node u in T incurs $O(|T_u|)$ node locations, using the analysis in [35], we get that the total node locations for updating all verification paths in our hash tree, for a sequence of t update operations (insertions or deletions) on an initially empty hash tree, is $O(t \log t)$. Thus, for the same series of update operations, the total structural update cost is $O(\log n \times t \times \log t)$. Then, for $t = O(m)$, we get that the amortized overall structural update cost is $O(\log n \log m)$ over a

[8] Alternatively, this can be done by threading the tree T such that leaf nodes are connected, that is, node W_i corresponding to leaf x_i storing the tree ids of the neighboring leaf nodes in T.

[9] Actually, the distributed Merkle tree is an *augmented* such tree where rotations at node u cost $O(|T_u|)$.

sequence of operations of size linear on m. Moreover, using the additional property shown in [35], namely that costly rotations at levels close to the root occur rarely, (in fact with frequency inversely proportional to the corresponding subtree size), the proof is completed. □

Overall, from the above analysis and Lemma 1, the following summarizes the efficiency of our scheme and our main result. By optimal network, we refer to a network where location operations take $O(\log n)$, where n the network size.

Theorem 1. *There exists a scheme for implementing a distributed Merkle tree T over a peer-to-peer network $\mathcal{N}$ with the following properties. If m is the size of the set over which tree T is built and n is the total number of nodes in the network $\mathcal{N}$, with $m \leq n$, then:*

1. *The distributed Merkle tree uses space $O(m \log m)$, distributed over $O(m)$ network nodes, and incurs $O(\log m)$ storage overhead per network node.*
2. *A verification path has size $O(\log m)$ and can be accessed with $O(\log m)$ locate operations; thus, for an optimal network $\mathcal{N}$, the expected computational and communication cost for accessing a verification path is $O(\log n \log m)$.*
3. *A hash update on the distributed Merkle tree involves $O(\log m)$ location operations; thus, for an optimal network $\mathcal{N}$, the expected computational and communication cost of a hash update is $O(\log n \log m)$.*
4. *A structural update on the distributed Merkle tree involves $O(m \log m)$ location operations, amortized over a series of $O(m)$ structural updates on an initially empty tree; thus, for an optimal network $\mathcal{N}$, the expected amortized computational and communication cost of a structural update is $O(\log n \log m)$.*

3.4 Improvement Through Caching

We now discuss a simple extension that can be used to improve the cost for path retrieval and the cost for updates of our scheme under a reasonable assumption.

Caching Network Nodes Assuming that network node failures occur less often than queries and updates on the distributed Merkle tree, we can improve the efficiency of our scheme by extending it as follows. The goal is to transform the multiplicative $O(\log n)$ factor into an additive term in the complexity of our scheme that is introduced by the need to locate network nodes storing hash values that need be retrieved or updated. Caching network node ids helps in this direction. The idea is to have each network node corresponding to a leaf of the tree to cache in its memory the $O(\log m)$ network nodes that store the hash values corresponding to the path the node stores. This can be easily achieved, since the node contacts these nodes as it updates its path, so only the used node ids need be remembered. In this way, no location operation is needed.

Of course, since network nodes can fail or go down, it is possible that cached nodes are no longer nodes of the network. In this case, we have a cache miss which will trigger a location operation. Although we can still use some techniques to avoid this overhead,[10] we can see that when the rate of network node failures is sufficiently small then we can actually amortize the $O(\log n)$ factor due to occasional location operations in the cost for operating on the tree. In particular, if network nodes fail independently with probability $O(\frac{1}{\log m})$ during the time interval of a tree traversal, then the expected number of network node failures that occur while a path retrieval of update is $O(1)$. Thus, the expected complexity for path retrieval and updates on the tree is $O(\log n + \log m)$.

[10] Indeed, node failures are typically tolerated by the peer-to-peer system by replicating data (key-value pairs) to neighboring (according to the identity space ordering) network nodes. Thus, one can actually use this feature to possibly eliminate a cache miss, by caching not only the exact network node but also their successor.

	Storage	Path Retrieval	Hash Update	Structural Update
Tree Replication	$O(m^2)$	$O(\log n \ \log m)$ *	$O(m \ \log n)$ *	$O(m \ \log n)$ *
Path Replication	$O(m \ \log m)$	$O(1)$	$O(m \ \log n)$ *	$O(m \ \log n)$ *
Route Distribution	$O(m \ \log m)$	$O(\log n \ \log m)$ *	$O(\log n \ \log m)^*$	$O(\log n \ \log m)^{**}$
Caching	$O(m \ \log m)$	$O(\log n)$ *	$O(\log n)^*$	$O(\log n)$ **

Table 1: Efficiency comparison of two straightforward schemes that use tree replication and path replication with our $BB[\alpha]$-based scheme that uses route replication and its extension through cashing. The comparison is performed with respect to the storage cost, the cost to retrieve a verification path and the costs to perform a hash and structural update respectively, where n the network size, m the data set size and $m \leq n$. Path retrieval and update costs refer to computational and communication complexity to perform the corresponding operation. We denote expected complexity using * and amortized expected complexity using **.

Table 1 summarizes the comparison between the various schemes for implementing a distributed Merkle tree. We see that our scheme provides an efficient solution to the problem of implementing a distributed Merkle tree and under certain assumptions about the rate of node failures, using caching, we can have an asymptotically optimal scheme in an amortized sense.

4 Authenticated Distributed Hash Table

We now show how to use the distributed Merkle tree described in the previous section to design distributed authentication techniques that can accordingly have various applications. We first use our distributed Merkle tree to authenticate the basic two operations of distributed hash tables, namely put() and get(). That is, we design an efficient *authenticated distributed hash table* (*ADHT*).

As we have seen in Section 2, the model of data authentication considers a data source S that produces m data items and stores them in a distributed hash table. The distributed hash table supports the insertion key-value pairs through operation put() and retrieval of a value given its key through operation get(). We use our distributed Merkle tree to augment the functionality of a distributed hash table as follow. The system should support:

Authenticated put() Any key-value pair can be inserted in the distributed hash table by source S in a way that both the system authenticates S's identity and source S is assured about the validity of the insertion.

Authenticated get() Any user of the system can retrieve the value that corresponds to an existing (stored in the system) given key in an authenticated way; that is he is given a proof that can be used to verify the authenticity of the data.

Authenticated remove() A previously inserted in the system key-value pair can be removed in the distributed hash table by source S in a way that both the system authenticates S's identity and source S is assured about the validity of the removal [11].

As in the model of authenticated data structures, the existence of a PKI is assumed. In particular, the users that the system know the public key of data source S.

Implementation We implement an authenticated distributed hash table ADHT using the following standard authentication technique. Signature amortization is performed by the use of a

[11]Typically, distributed hash tables do not support a removal operation, but rather, when used to support distributed storage, usually a TTL time interval is assigned to every inserted data item, which is used to determine an automatic deletion of the item from the system. With respect to this, our design uses a different approach.

Merkle tree (using a cryptographic collision-resistant hash function) built on top the data items owned by the S. The hash of the root of tree serves as the data digest and is signed by the data source. A data item is verified to be owned by S if the signed root hash value is verified to be authentic (signed by the source) and a verification path binds the item with the signed digest. The security of the technique follows for the security properties of the signature scheme in use and the collision-resistant hash function in use.

The only non-trivial part of the above design is the fact that now the system is a distributed hash table over a peer-to-peer network. We make use of the distributed Merkle tree to compute the digest of the data owned by S and to realize signature amortization: one signature over m data items. Now, we add an additional level of hashing in the tree: at a leaf node of the tree the hash stored is the hash of the concatenation of the hash of the of the key and hash of the corresponding value. We briefly describe some of the details of the scheme. We assume that using standard bootstrapping techniques both the data source and the user have access (through direct connection) to a valid (i.e., running) node of the underlying peer-to-peer system. Assuming that a distributed Merkle tree is already used to as described above to amortize one signature of data source S over m items and that the source keeps a copy of the signed digest of its data, we have that about functionality of the system:

Updates Updates are performed by the source S first contacting a node of the network and then issuing an update request, where the signed digest is also submitted to the system. The system then performs a path retrieval operation on the distributed Merkle tree and verifies the authenticity of the requester (i.e., that the requester in the valid source of the data associated with the distributed Merkle tree). If the verification rejects, so does the system. Otherwise, the system reports to the source the verification path. We augment the verification paths to contain all the additional information needed by the source in order to execute (simulate) locally both the hash and the structural update in order to recompute the correct new root hash value of the updated tree. This is feasible because both hash updates and structural tree adjustments (rotations) only happen along a path in a bottom-up fashion. Once the source implements the update and computes the new root hash, he signs it and returns a copy to the contacted network node. Then a regular hash tree update is performed by the system to execute the put() or remove() operation. Note that by this interaction the source need only keep $O(1)$ authentication information, namely, only the current signed digest. We note that the above technique to make this feasible, i.e., the interaction with the system as querier is similar to the one used in [16]. Note that asymptotically no additional computational or communication cost is introduced by this extra interaction between the system and the source.

Queries Queries are handled as follows. A user contacts a network node and requests the value of a key. A path retrieval query is executed over the tree by the system and what is returned to the user is: 1) the corresponding value, 2) the verification path (collection of hash values and relative information for computing the root hash) and 3) the signed digest. The user accepts the answer (value) if and only if the signed digest is valid and hashing over the value and the verification path results in a hash value that equals the root hash.

Note that in a way, the source updates the data set by effectively first querying it in a similar way that a user would do.

Using Theorem 1, we can prove the following.

Theorem 2. *There exists an authenticated distributed hash table over a peer-to-peer network of n nodes that supports authenticated operations put(), get() and remove() on a data set of size $m \leq n$, such that:*

1. *The authentication scheme is secure;*
2. *The storage at the source is $O(1)$; the storage at the network is $O(m \log m)$;*
3. *The query cost is $O(\log m)$, that is, $O(\log m)$ locate operations; or, equivalently, the expected time and communication complexity to answer a query is $O(\log n \log m)$;*
4. *The amortized update cost is $O(\log m)$, that is, $O(\log m)$ locate operations; or, equivalently, the amortized expected time and communication complexity of a query is $O(\log n \log m)$.*

Security We briefly discuss the security property of ADHT. This follows using standard reductions to the security of the underlying cryptographic primitives that is used in our authentication scheme, under standard hardness assumption. That is, by using a family of collision-resistant hash functions and a signature scheme secure against adaptive chosen-message attacks, we can prove the security of ADHT. Note that as described above, the security of the source against adversarial behavior by the DHT (or the underlying network) is still captured, since the interaction between the source and the DHT is treated as a special type of querying.

Discussion Existing designs for distributed hash tables use the "sign all" approach for storing data items of the same data source. Although some systems use a signature amortization through hashing, such that a large data item, e.g., a file or an entire file system, is divided into blocks which are binded together through hashing and only the root block is signed, this is performed within the data item itself not for the entire data collection that a source stores in the system. Thus, our scheme is the first to amortize one signature over any collection of large data items (even large files or entire file systems). One first advantage of this is in terms of storage, since typically the size of a cryptographic hash value is less than the size of a digitally signature. But an additional and even more important disadvantage of the sign-everything technique is the following.

Distributed hash tables typically do not support a removal operation, but instead they introduce a time-to-live (TTL) mechanism, so that old stored data items expire and are automatically deleted. In this TTL-based approach, if the source needs to renew the stored data, it needs to insert it and thus to sign it again. Additionally, even when stored data never expires the issue of data freshness is critical. An old signed value may be not valid anymore with respect to the application that uses it. A signed statement may be copied and be forgotten even when it is not valid anymore. Thus, a signed statement should be a freshly signed statement. This can be done by signing a time-stamped data and requiring that not only a signature is verified but also it is fresh. In a system storing m data items where $O(m)$ signatures are used, the signing cost for updating them is also $O(m)$. Signing typically involves expensive computations, thus the introduced computational overhead is high. Instead, in our scheme only one statement (the root hash) must be refreshed. The signing cost is $O(1)$, at the cost of increasing the query and update complexity by a logarithmic factor.

	Storage	Signing Cost	Query Cost	Update Cost
Existing Schemes	$O(m)$	$O(m)$	$O(\log n)$ *	$O(\log n)$ *
ADHT	$O(m \log m)$	$O(1)$	$O(\log n \ \log m)$ *	$O(\log n \ \log m)$ **
ADHT w/ Caching	$O(m \log m)$	$O(1)$	$O(\log n + \log m)$ *	$O(\log n + \log m)$ **

Table 2: Comparison of our authenticated distributed hash table ADHT with other schemes that use no signature amortization. Here, n is the number of network nodes and $m \leq n$ is the size of data items stored. If caching is used and under reasonable assumptions about low rates for network node failures, the query and update costs of our scheme are improved by a logarithmic factor. We denote expected complexity using * and amortized expected complexity using **.

Table 2 compares with respect to various costs our authenticated distributed hash table ADHT with other existing authenticated storage schemes. Our schemes realize an efficient technique that achieves *signature amortization*, where only one digital signature is used for a large collection of data items. In particular, we see that our authentication scheme that uses caching is as efficient as other existing schemes and achieves more scalability: for authenticating m data items, the signing cost is $O(1)$ rather than $O(m)$, at the slight overhead of increasing the storage needs by a logarithmic factor. Some other differences between our scheme and existing schemes that use the "self-certified data" authentication technique [14] are summarized as follows:

- The self-certified technique is designed and developed only for file systems. Files systems tend to be large, relatively flat structures and they are certainly not balanced. Our technique is designed to store and authenticate any type of data elements, including high-volume data, collections of files or relatively small pieces of information.
- Self-certified data is not dynamic; instead, static data is being authenticated. In contrast, our technique is totally dynamic and supports the authentication of collections of data that rapidly change.
- Self-certified data is accessed in a top-down fashion. That is, files are being retrieved and authenticated starting from the root directory, a root entry that is signed by the source. Our technique creates a bottom-up authentication path. Generally, top-down hash-based authentication approaches use more space and are computationally more expensive.

Data Authentication in Peer-to-Peer Systems An immediate application of the authenticated distributed hash table is a distributed authenticated dictionary. That is, membership queries are authenticated about the set of data items of a source. Suppose that keys are drawn from a totally ordered space. The distributed Merkle tree is built on top of key-value paired in a sorted sequence according to theirs keys. Additionally, to support authentication of negative answers, the source inserts in the system pairs of key-value pairs such that the keys are consecutive in the ordering used in the Merkle tree. The distributed authenticated dictionary has asymptotically the same performance as the authenticated distributed hash table ADHT described above, given by Theorem 2.

Additional Applications Note that our distributed tree is mainly designed for bottom-up access and updates. But this is not restrictive, as our tree can also be used in a top-down fashion (depending on the application). We believe that our tree can be used in other (not necessarily security-related) applications.

5 Conclusions and Future Work

In this paper, we consider the problem of data authentication and data integrity in peer-to-peer distributed storage networks. We extend the model of authenticated data structures to capture the security needs of these type of systems with respect to data authentication. We design the first efficient implementation of a distributed Merkle tree for peer-to-peer systems. We identify inefficiencies in the authentication techniques used by current peer-to-peer distributed storage systems and we show how our distributed Merkle tree can be used in combination with any DHT to implement an efficient authenticated distributed hash table (ADHT). Using an ADHT, we present an efficient distributed authenticated dictionary.

We plan to implement our distributed Merkle tree and experimentally test its efficiency. We leave as open problems the design of authenticated distributed data structures for more general queries and additional security issues in the model, such as DOS attacks and Byzantine behavior.

Also, a related issue is load-balancing. All existing techniques for achieving authentication in distributed hash tables, including our technique, introduce congestion at certain network nodes. New techniques and new machinery should be discovered for the design of load-balance distributed authentication.

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.

[2] L. Arge, D. Eppstein, and M. T. Goodrich. Skip-webs: Efficient distributed data structures for multi-dimensional data sets. In *24th ACM Symp. on Principles of Distributed Computing (PODC)*, 2005.

[3] J. Aspnes and G. Shah. Skip graphs. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, pages 384–393, 2003.

[4] E. Bertino, B. Carminati, E. Ferrari, B. M. Thuraisingham, and A. Gupta. Selective and authentic third-party distribution of XML documents. *IEEE Transactions on Knowledge and Data Engineering*, 16(6):1263–1278, 2004.

[5] 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.

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

[7] M. Castro, P. Drushel, A. Ganesh, A. Rowstron, and D. Wallach. Secure routing for structured peer-to-peer overlay networks. In *In Proceedings of Usenix Symposium of Operating Systems Design and Implementation (OSDI)*, 2002.

[8] A. Crainiceanu, P. Linga, J. Gehrke, and J. Shanmugasundaram. Querying peer-to-peer networks using P-trees, 2004.

[9] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, Oct. 2001.

[10] 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*, pages 136–145, 2001.

[11] P. Devanbu, M. Gertz, C. Martel, and S. G. Stubblebine. Authentic data publication over the Internet. *Journal of Computer Security*, 11(3):291 – 314, 2003.

[12] P. Druschel and A. Rowstron. Past: A large-scale, persistent peer-to-peer storage utility. In *HOTOS '01: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, page 75, Washington, DC, USA, 2001. IEEE Computer Society.

[13] M. J. Freedman and R. Vingralek. Efficient peer-to-peer lookup based on a distributed trie. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS02)*, Cambridge, MA, March 2002.

[14] K. Fu, M. F. Kaashoek, and D. Mazieres. Fast and secure distributed read-only file system. *Computer Systems*, 20(1):1–24, 2002.

[15] 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.

[16] 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.

[17] 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`.

[18] 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.

[19] 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.

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

[21] E. Hall and C. S. Julta. Parallelizable authentication trees. In Cryptology ePrint Archive, Dec 2002.

[22] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: A scalable overlay network with practical locality properties. In *USENIX Symp. on Internet Technologies and Systems*, Lecture Notes in Computer Science, 2003.

[23] M. Jakobsson, T. Leighton, S. Micali, and M. Szydlo. Fractal merkle tree representation and traversal. In *Proc. RSA Conference—Cryptographers' Track*, volume 2612 of *LNCS*, pages 314–326. Springer, 2003.

[24] T. Johnson and P. Krishna. Lazy updates for distributed search structure. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 337–346, New York, NY, USA, 1993. ACM Press.

[25] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, pages 654–663, 1997.

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

[27] M. Krohn, M. Freedman, and D. Mazieres. On-the-fly verification of rateless erasure codes for efficient content distribution. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 226–240, May 2004.

[28] B. Kroll and P. Widmayer. Distributing a search tree among a growing number of processors. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD94)*, pages 265–276, 1994.

[29] J. Li, K. Sollins, and D.-Y. Lim. Implementing aggregation and broadcast over distributed hash tables. *SIGCOMM Comput. Commun. Rev.*, 35(1):81–92, 2005.

[30] A. Lysyanskaya, R. Tamassia, and N. Triandopoulos. Multicast authentication in fully adversarial networks. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 241–255, May 2004.

[31] 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.

[32] P. Maniatis and M. Baker. Secure history preservation through timeline entanglement. In

Proc. USENIX Security Symposium, 2002.

[33] G. S. Manku, M. Naor, and U. Wieder. Know thy neighbor's neighbor: the power of lookahead in randomized P2P networks. In *Proceedings of the 36th ACM Symposium on Theory of Computing (STOC)*, pages 54–63, 2004.

[34] C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong, and S. G. Stubblebine. A general model for authenticated data structures. *Algorithmica*, 39(1):21–41, 2004.

[35] K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*, volume 1 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Heidelberg, Germany, 1984.

[36] R. C. Merkle. A certified digital signature. In G. Brassard, editor, *Proc. CRYPTO '89*, volume 435 of *LNCS*, pages 218–238. Springer-Verlag, 1989.

[37] R. Morselli, S. Bhattacharjee, J. Katz, and P. Keleher. Trust-preserving set operations. In *23rd Conference of the IEEE Communications Society (Infocom)*, March 2004.

[38] E. Mykletun, M. Narasimha, and G. Tsudik. Authentication and integrity in outsourced databases. In *Proceeding of Network and Distributed System Security (NDSS)*, 2004.

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

[40] M. Naor and U. Wieder. Know thy neighbor's neighbor: Better routing in skip-graphs and small worlds. In *3rd Int. Workshop on Peer-to-Peer Systems*, 2004.

[41] G. Nuckolls, C. Martel, and S. Stubblebine. Certifying data from multiple sources [extended abstract]. In *Proceedings of the 4th ACM conference on Electronic commerce*, pages 210–211, New York, NY, USA, 2003. ACM Press.

[42] C. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of ACM SPAA*, June 1997.

[43] S. Ramabhadran, J. Hellerstein, S. Ratnasamy, and S. Shenker. Prefix hash tree - an indexing data structure over distributed hash tables, 2004.

[44] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of SIGCOMM '01*, pages 161–172, San Diego, California, August 2001.

[45] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. OpenDHT: A public DHT service and its uses. In *Proceedings of the 2005 ACM SIGCOMM Conference*, 2005.

[46] E. Shi, A. Perrig, and L. V. Doorn. Bind: A fine-grained attestation service for secure distributed systems. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 154–168, Washington, DC, USA, 2005. IEEE Computer Society.

[47] E. Sit and R. Morris. Security considerations for peer-to-peer distributed hash tables, 2002.

[48] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

[49] R. Tamassia and N. Triandopoulos. Computational bounds on hierarchical data processing with applications to information security. In *Proc. Int. Colloquium on Automata, Languages and Programming (ICALP)*, volume 3580 of *LNCS*, pages 153–165. Springer-Verlag, 2005.

[50] Y. Xie, D. O'Hallaron, and M. K. Reiter. A secure distributed search system. In *Proceedings of the 11th IEEE Int. Symp. on High Performance Distributed Computing (HPDC)*, pages 321–332, 2002.

[51] C. Zhang, A. Krishnamurthy, and R. Wang. Brushwood: Distributed trees in peer-to-peer systems. In *Proceedings of the 4th International Workshop on Peer-to-Peer Systems (IPTPS05)*, 2005.