Authenticated Data Structures for Graph and Geometric Searching[*]

Michael T. Goodrich[†] Roberto Tamassia[‡] Nikos Triandopoulos[‡] Robert Cohen[§]

Abstract

Following in the spirit of data structure and algorithm correctness checking, authenticated data structures provide cryptographic proofs that their answers are as accurate as the author intended, even if the data structure is being maintained by a remote host. We present techniques for authenticating data structures that represent graphs and collection of geometric objects. We use a model where a data structure maintained by a trusted source is mirrored at distributed directories, with the directories answering queries made by users. When a user queries a directory, it receives a cryptographic proof in addition to the answer, where the proof contains statements signed by the source. The user verifies the proof trusting only the statements signed by the source. We show how to efficiently authenticate data structures for fundamental problems on networks, such as path and connectivity queries, and on geometric objects, such as intersection and containment queries. Our work has applications to the authentication of network management systems and geographic information systems.

1 Introduction

Verifying information that at first appears authentic is an often neglected task in data structure and algorithm usage. Fortunately, there is a growing literature on correctness checking that aims to rectify this omission. Following early work on program checking and certification [2, 29, 30], several researchers have developed efficient schemes for checking the results of various data structures [3, 5, 4, 14, 22], graph algorithms [18, 20], and geometric algorithms [11, 23]. These schemes are directed mainly at defending the user against an inadvertent error made during implementation. In addition, these previous approaches have primarily assumed that usage is limited to a single user on an individual machine.

With the advent of Web services and Internet computing, data structures and algorithms are no longer being used just by a single user on an individual machine. Indeed, with the development of content distribution services (e.g., Akamai) spreading content across the Internet, the machine responding to a user's query could be unknown to both the data structure author and its user. Bringing the lessons of recent world events into the domain of algorithmics, we must recognize that, although they benefit efficiency, such scenarios open the possibility that an agent hosting a data structure or algorithm could deliberately falsify query responses to users. If the information represented in a response has security or financial implications, such falsification could cause significant adverse consequences.

In this paper we are interested in studying a new dimension in data structure and algorithm checking—how can we design sophisticated data structures and algorithms so that their responses can be verified as accurately as if they were coming from their author, even when the response is coming from an untrusted host? Examples of the kind of information we want to authenticate include dynamic documents, online catalog entries, and the responses to queries in geographic information systems, financial databases, medical information systems, and scientific databases. Digital signatures can be used to verify simple static documents, but are inefficient for dynamic data structures. We therefore need new techniques for authenticating data structures.

In particular, we are interested in efficiently verifying paths and connectivity information in transportation and computer networks (that is, combinatorial graph structures), even when the network is changing. In addition, we are interested in verifying complex geometric queries in spatial data bases, such as ray shooting queries, point location queries, and range searching queries, which are used extensively in geographic

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

[†] University of California, Irvine, goodrich@acm.org

[‡] Brown University, {rt,nikos}@cs.brown.edu

[§] AlgoMagic Technologies, rfc@algomagic.com

information systems. The main challenge in providing an integrity service in such contexts is that the space of possible answers is much larger than the data size itself. For example, there are $O(n^2)$ different paths in a tree of n nodes, and each of these paths can have $O(n)$ edges. Requiring an authenticator to digitally sign every possible response is therefore prohibative, especially when the data is changing due to the insertion or deletion of elements in the set. Ideally, we would like our authenticator to sign just a single digest of our data structure, with that digest being built from the careful combination of cryptographic hashes of subsets of our data. If we can achieve such a scheme, then verifying the answer to a query in our data base can be reduced to the problem of collecting the appropriate partial hashes for a user to recompute the digest of the entire structure and compare that to the digest signed by the authenticator. Even when we follow this approach, however, we are faced with the challenge of how to subdivide the data in a way that allows for efficient assembly for any possible query. For simple data structures, such as dictionaries, this subdivision is fairly straightforward (say using a linear ordering and a Merkle hash tree [24, 25]; see also [15, 26]), but the subdivision method for complex structures, such as graphs, geometric structures, and structures built using the fractional cascading paradigm is far from obvious. For example, we have no linear ordering among the data in such cases upon which to build a hash tree.

1.1 A Model for Authenticated Data Structures

Our data structure authentication model involves three parties: a trusted source, an untrusted directory, and a user. The *source* holds a structured collection S of objects, where we assume that a set of *query operations* are defined over S. If S is fixed over time, we say that it is *static*. Otherwise, we say that S is *dynamic* and assume that a set of *update operations* are defined that modify S.

For example, S could be a network whose vertices and edges store data items and on which the following two query operations are defined: a *connectivity query* on S asks whether two given vertices of S are in the same connected component and a *path query* returns a path, if it exists, between two given vertices. We could also define update operations of S that add and/or remove vertices and edges. As a second example, S could be a collection of line segments in the plane forming a polygonal chain, where an *intersection query* returns all the segments intersected by a given query line. In this case we could define update operations that insert and/or remove segments.

The *directory* maintains a copy of the collection S together with *structure authentication information*, which consists of statements about S signed by the source. The *user* performs queries on S but instead of contacting the source directly, it queries a directory. The directory provides the user with an answer to the query together with *answer authentication information*, which yields a cryptographic proof of the answer. The answer authentication information should include a time stamp and information derived from signed statements in the structure authentication information. The user then verifies the proof relying solely on the time stamp and the information derived from statements signed by the source (subject to standard cryptographic assumptions). If S is dynamic, the directory receives together with each update *update authentication information*, which consists of signed time-stamped statements about the update and the current state of S.

The data structures used by the source and the directory to store collection S, together with the protocols and algorithms for queries, updates, and verifications executed by the various parties, form what we call an *authenticated data structure* [15, 21, 26]. In a practical deployment of an authenticated data structure, there would be various instances of geographically distributed directories. Such a distribution scheme reduces latency, allows for load balancing, and reduces the risk from denial-of-service attacks.

1.2 Previous Work

Previous work related to authenticated dictionaries is mostly concerned with *authenticated dictionaries*, which are authenticated structures for sets on which membership queries are performed.

The *hash tree* scheme introduced by Merkle [24, 25] can be used to implement a static authenticated dictionary. A hash tree T for a set S stores hashes of the elements of S at the leaves of T and a value $L(v)$ at each internal node v, which is the result of computing a one-way hash function on the values of its children. The authenticated dictionary for S consists of the hash tree T plus the signature of the value $L(r)$ stored of the root r of T. An element e is proven to belong to S by reporting the values stored at the nodes on the path in T from the node storing e to the root, together with the values of all nodes that have siblings on this path. With this approach, space is linear, and the query authentication information and the query and verification time are logarithmic in the size of the set S. Kocher [19] also advocates a static hash tree approach for realizing an authenticated dictionary, but simplifies somewhat the processing done by the user to verify that an item is not in the set S, by storing intervals instead of individual elements.

Using techniques from incremental cryptography, Naor and Nissim [26] dynamize hash trees to support the insertion and deletion of elements in logarithmic time, thus implementing a dynamic authenticated dictionary. In their scheme, the source and the directory maintain identically-implemented 2–3 trees. The update authentication information has $O(1)$ size and the query authentication information has logarithmic size.

Goodrich and Tamassia [15] present a data structure for an authenticated dictionary based on skip lists [27]. They introduce the notion of commutative hashing and show how to embed in the nodes of a skip list a computational DAG (directed acyclic graph) of cryptographic computations based on commutative hashing. This data structure matches the asymptotic performance of the Naor-Nissim approach [26], while simplifying the details of an actual implementation of a dynamic authenticated dictionary. In related works, Goodrich, Tamassia and Schwerin [17] present the software architecture and implementation of an authenticated dictionary based on the above approach and Anagnostopoulos, Goodrich and Tamassia [1] introduce the notion of *persistent authenticated dictionaries*, where the user can issue historical queries of the type "was element e in set S at time t".

Goodrich, Tamassia and Hasic [16] show how to use the RSA one-way accumulator to realize a dynamic authenticated dictionary for a set with n elements with $O(1)$ query authentication information size and verification time. Their scheme allows a tradeoff between the query and update times. For example, one can balance the two times and achieve $O(\sqrt{n})$ query and update time and $O(\sqrt{n})$ update authentication information.

A first step towards the design of more general authenticated data structures (beyond dictionaries) is made by Devanbu *et al.* [10]. Using an extension of hash trees, they show how to authenticate operations *select*, *project* and *join* in a relational database. Moreover, they present an authenticated data structure for a set of multidimensional points that supports orthogonal range queries. This latter result goes beyond simple authenticated dictionaries, but it is restricted to hashing in range trees.

Recently, Martel at al. [21] have initiated work that begins a study of authenticated queries beyond tree structures and skip lists. They consider the class of data structures such that (i) the links of the structure form a directed acyclic graph G of bounded degree and with a single source node; and (ii) queries on the data structure correspond to a traversal of a subdigraph of G starting at the source. They show that such data structures can be authenticated by means of a hashing scheme that digests the entire digraph G into a hash value at its source. With this scheme, the size of the answer authentication information and the verification time are proportional to the size of the subdigraph traversed. They show how this general technique can be 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. The also begin an initial treatment of authenticating fractional cascading structures, but only for range tree data structures, where catalogs are arranged as unions in a tree. Related work on the authentication of XML documents by Devanbu *et al.* appears in [9].

1.3 Our Results

In this paper we present general techniques for building authenticated data structures for a number of non-trivial query problems for a general graph G, including the following:

- areConnected(v, w): Are v and w in same connected component?
- areBiconnected(v, w): Are v and w in same biconnected component?
- areTriconnected(v, w): Are v and w in same triconnected component?
- path(v, w): Return a path from v to w.
- pathLength(v, w): Return the length of a path from v to w.

We also support efficient update operations that involve inserting vertices and edges in G. Our data structure uses linear space and supports connectivity queries and update operations in $O(\log n)$ time and path queries in $O(\log n + k)$ time, where k is the length of the path reported. The update authentication information has $O(1)$ size. The size of the answer authentication information and the verification time are each $O(\log n)$ for connectivity, biconnectivity and triconnectivity queries and $O(\log n + k)$ for path queries. These results have applications to the authentication of network management systems.

In addition, we address several geometric search problems, showing how to authenticate the full, general version of the powerful *fractional cascading* technique [6]. Namely, we can authenticate any query efficiently answered in a fractional-cascading structure via *iterative search*, where we have a collection of k dictionaries of total size n stored at nodes in a connected graph and we want to search for an element in each dictionary in a subgraph in this graph. A number of fundamental two-dimensional geometric searching problems arising in the implementation of geographic information systems can be solved with data structures based on this iterative search approach [7]. These problems include:

- *line intersection queries* on a polygon P, to report the edges of P intersected by a query line
- *ray shooting queries* on a polygon P, to report the first edge of P intersected by a query ray
- *point location* on a planar subdivision, to report the region containing a query point
- *orthogonal range search* on a set of points in $\mathbf{R}^2$, to report the points inside a query rectangle
- *orthogonal point enclosure* on a set of rectangles, to report the rectangles that contain a query point
- *orthogonal intersection queries* on a set of rectangles, to report the rectangles intersected by a query rectangle.

We show that our authenticated fractional cascading data structure can be extended to yield efficient authenticated data structures for all the above problems. These results have applications to the authentication of geographic information systems. We are unaware of previously known authenticated data structures for the above problems, with the exception of orthogonal range search, for which an authenticated data structure was given in [21]. The security of our scheme is based on standard cryptographic primitives, such as one-way hashing and digital signatures; hence, it is practical and does not need any new cryptographic assumptions. We leave open the problem of the dynamization of our authenticated geometric search structures based on fractional cascading.

1.4 Security Model

We provide a security model that aims to provide efficient, secure, dynamic, and trusted solutions to the problems we consider. Our work is motivated by our desire to improve over limitations of two naive approaches:

- *Central Server:* Queries to collection S are handled by a single trusted server S_T that gives a digitally signed response to every query.
- *Signed Collection:* Collection S is signed in its entirity by a trusted party and distributed to every client.

Central server solutions are prone to network delays and provide a single point of failure. Signed directories are expensive to make dynamic since upon update, the entire collection S needs to be signed and distributed to every client.

Our approach results in solutions with the following properties:

- Queries are handled by a collection of potentially geographically distributed directories.
- Responses include authentication information that allows clients to verify responses with the same trust as if the response was digitally signed by a trusted third-party.
- The size of update information sent to the directories is proportional to the number of updates, so update time is small.
- Directory computers can be in untrusted locations and provide trusted responses.

In this paper we show how to provide solutions that meet these criteria for important graph and geometric problems.

2 Cryptographic Preliminaries

In this section, we present the general cryptographic technique that is used in our authenticated data structures.

The basis of trust in our authentication model is the assumption that the user trusts the source. This is expressed by means of a digital signature scheme. Moreover, all the desired security results are achieved by means of the use of a *cryptographic hash function*. A cryptographic hash function h typically operates on a variable-length message M producing a fixed-length hash value $h(M)$. We assume some well-defined binary representation for any data element e, so that h can operate on e. Also, we assume that rules have been defined so that h can operate over any finite number of elements. A cryptographic hash function h is a *collision-resistant hash function*, if, given the value $h(x)$, it is computationally intractable to find x and, moreover, if, given y, it is computationally intractable to find $x \neq y$ with $h(x) = h(y)$. The collision resistance property will be used for our security results.

Let S be a data set owned by the source. In our authentication schemes, a collision-resistant hash function is used to produce a *digest*, i.e., a cryptographic hash value over the data elements of S. The digest is computed in a systematic way which can be expressed by means of directed acyclic graph (DAG) defined over S (similar technique is presented in [21]). We define a single-sink DAG G associated with S as follows. Each node u of G stores a label $L(u)$ such that if u is a source of G, then $L(u) = h(e_1, \ldots, e_m)$, where $e_1, \ldots, e_m$ are elements of S, else (u is not a source of G) $L(u) = h(e_1, \ldots, e_n, L(z_1), \ldots, L(z_k))$, where $(z_1, u), \ldots, (z_k, u)$ are edges of G and $e_1, \ldots, e_n$ are elements of S (here, both m and n 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.

We call the above scheme as *hashing scheme* of S using G. The answer to a query usually involves some elements. Then, the proof typically consists of the signed digest and all the information (labels of G) that is necessary for the recomputation, given the answer, of this digest by the user.

The authentication techniques presented in this paper are based on the following general scheme. The source and the directory store identical copies of the data structure representing 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 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 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.

3 Path Properties

In this section, we present an authenticated scheme for various types of queries on a sequence S.

3.1 Path Hash Accumulator

An abstract notion of a *path* is used to represent S. We use and extend the notation used in [8].

A *path* is an ordered sequence of one or more nodes. By *head*(p) and *tail*(p) we denote the first and last nodes of a path p. If p' and p'' are paths, the *concatenation* $p = p' \mid p''$ is a path formed by adding a directed edge from *tail*(p') to *head*(p''). A *subpath* $\bar{p}(v, u) = \bar{p}$ of a path p is the path consisting from the collection of consecutive nodes of p $v, w_1, \ldots, w_l, u$, with *head*$(\bar{p}) = v$ and *tail*$(\bar{p}) = u$.

A path stores a data set through *node attributes*, which are values stored at nodes, and *node properties* which are collections of node attributes. A node attribute $N(v)$ of node v can assume arbitrary values and occupies $O(1)$ storage. A node property $\mathcal{N}(v)$ is a sequence $N_1(v), \ldots, N_r(v)$ of node attributes, where r is a constant. Similarly, *path attribute* and *path property* are defined to extend the notion of node attribute and node property. $P(p)$ is the path attribute of p; it occupies only $O(1)$ storage and depends on the values $N(v)$ for every node v of p and on the order of nodes of p. The path property $\mathcal{P}(p)$ of p is a sequence of path attributes $P_1(p), \ldots, P_s(p)$, where s is a constant. We require that $\mathcal{P}(p)$ includes path attributes *head*(p), *tail*(p). The definition of path attribute and path property are naturally extended when subpaths of paths are considered.

Let $p = p' \mid p''$ be a path that is the concatenation of paths p' and p''. A path property $\mathcal{P}$ satisfies the *concatenation criterion* if $\mathcal{P}(p) = \mathcal{F}(\mathcal{P}(p'), \mathcal{P}(p''))$, where $\mathcal{F}$ is a function that can be computed in $O(1)$ time that is called the *concatenation function* of $\mathcal{P}$.

Given a path p and a query argument q, a *node selection query* Q_N maps p into a node $v = Q_N(p, q)$ of p. A node selection query is always associated with some path selection function. Given that $p = p' \mid p''$, a *path selection function* $\sigma(p, q)$ for Q_N determines in $O(1)$ time whether v is in p' or p'' using q and values $\mathcal{P}(p')$ and $\mathcal{P}(p'')$. A *path selection query* extends a node selection query using a *path advance function.* Given a path p and some query argument q, a *path selection query* Q_P maps p into a subpath $\bar{p} = Q_P(p, q)$ of p. A path selection query is characterized by a *path advance function.* Given that $p = p' \mid p''$, a *path advance function* $\alpha(p, q)$ for Q_P returns in $O(1)$ time the subpath(s) among p', p'' (possibly none) for which the query argument q holds (values $\mathcal{P}(p')$ and $\mathcal{P}(p'')$ are used by the path advance function).

Let p be a path and $\mathcal{P}$ be a path property that satisfies the concatenation criterion. We are interested in authenticating the following operations:

- property(subpath $\bar{p}(v,u)$)–report the value of path property $\mathcal{P}$ for subpath $\bar{p}(v,u)$ of p ($\bar{p}$ may be p).
- property(node v)–report the value of node property $\mathcal{P}$ for node v.
- locate(path p, path selection function σ, argument q)–find node v of p returned by the node selection query that corresponds to the path selection function σ.
- subpath(path p, path advance function α, argument q)–find the subpath of p returned by the path selection query expressed by the path advance function α.

We represent a path p with a balanced binary tree $T(p)$ as follows. A leaf of $T(p)$ represents a node of p. An internal node v of $T(p)$ represents the subpath $p(v)$ of p associated with the leaves in the subtree of v. Each leaf stores the corresponding node property and each non leaf node stores the corresponding path property.

Let h be a collision resistant hash function. The *path hash accumulator* for a path p is the hashing scheme over the node and path properties of p defined as follows. Consider the data set consisting of: (1) for each leaf node v of $T(p)$, the node property $\mathcal{N}(v)$ and (2) for each internal node u of $T(p)$, the path property $\mathcal{P}(p(u))$. Let G be the DAG obtained from $T(p)$ by directing each edge toward the parent node. For a node v of p, let $pred(v)$ and $succ(v)$ respectively denote the predecessor and the successor of v in p. In particular, $pred(head(p))$ and $succ(tail(p))$ are some special (nil) values. Using G and h, we compute a label L for each node of $T(p)$ as follows: (1) if u is a source vertex of G, i.e., a leaf of $T(p)$, then $L(u) = h(\mathcal{N}(pred(u)),\mathcal{N}(u),\mathcal{N}(succ(u)))$; (2) if w is a non source vertex of G and (z_1, w) and (z_2, w) are edges of G, then $L(w) = h(\mathcal{P}(w), L(z_1), L(z_2))$. The digest of the above data set is the label $L(r)$ of the sink r of G (r is the root of $T(p)$). This digest is called the *path hash accumulation* of path p.

Lemma 1 *Let p be a path of length n. There exists an authenticated data structure for p based on the path hash accumulator scheme with the following performance:*

- *query operations property(v), property(p), locate(p,σ,q) and subpath(p,α,q) take each $O(\log n)$ time;*
- *the update authentication information has size $O(1)$;*
- *the answer authentication information size and the answer verification time are each $O(\log n)$.*

4 Authenticated Graph Searching

In this section, we consider authenticated data structures for graph searching problems. We first develop a generic authenticated data structure for a forest of trees and then consider general graphs.

4.1 Tree of Paths

We develop an efficient and fully dynamic authenticated data structure that supports path property queries in a forest of trees. The data structure has fast, update, query, and validation times.

We use the path hash accumulator authentication scheme over a collection Π of paths that are maintained through the update operations on paths: split and concatenate. At a high level point of view, Π is organized by means of a rooted tree $\mathcal{T}$ of paths, meaning that each node of $\mathcal{T}$ corresponds to a path $p \in \Pi$. Neighboring paths in $\mathcal{T}$ are generally interconnected and share information. This is achieved by the definition of suitable node attributes and properties.

A tree of paths $\mathcal{T}$ is considered to be directed; the direction of an edge is from a child to a parent. Let μ be a node of $\mathcal{T}$, let $\mu_1, \ldots, \mu_k$ be its children in $\mathcal{T}$ and let p be the path that corresponds to μ. A node attribute $N(v)$ of a node v of p is extended so that it depends not only on v but possibly also on some path properties of the paths that correspond to nodes $\mu_1, \ldots, \mu_k$ of $\mathcal{T}$. We say that path p is the *parent path* of paths $\mu_1, \ldots, \mu_k$ and these paths are the *children paths* of p. Clearly, this extension makes the path property $\mathcal{P}(p)$ of path p that correspond to node μ to include information about paths in the subtree of $\mathcal{T}$ having as root node μ. We again consider path properties that satisfy the concatenation criterion.

The idea above can be generally extended using a directed acyclic graph as the high level graph for the organization of a path collection Π. Using such a graph we introduce a *hierarchy* over paths in Π. Path properties are thus extended to include information about other paths according to the underlying hierarchy.

4.2 Path Properties in a Forest

We now present the construction of our data structure that supports path property queries on a forest.

Let F be a forest of trees. F is associated with a data set by storing at each tree node a node attribute. Using the framework presented in Section 3.1, any path in F is associated with some path property $\mathcal{P}$. We assume that $\mathcal{P}$ satisfies the concatenation criterion. We study the implementation of the authenticated query operation $\mathsf{property}(u, v)$–return the path property of the path from u, v, if such a path exists, while the following update operations over trees of F are performed:

- *destroyTree*(w) - Destroys the tree with root w;
- *newTree*$()$ - Creates a new tree in F that consists of a new, single node;
- *link*(u, v) - Merge two trees by adding an edge from the root u of some tree to a leaf v of another tree;
- *cut*(u) - Separate a tree by removing the edge from non-root node u to its parent.

Note that any tree can be assembled or disassembled using these operations.

Our data structure is based on dynamic trees of [28]. A dynamic tree T is a rooted tree whose edges are classified as being either *solid* or *dashed*, with the property that any internal node has at most one child connected by a solid edge. This edge classification partitions the nodes of the tree into *solid paths* connected with each other by dashed edges (see Figure 1(a)). We view a solid path as directed toward the root of T.

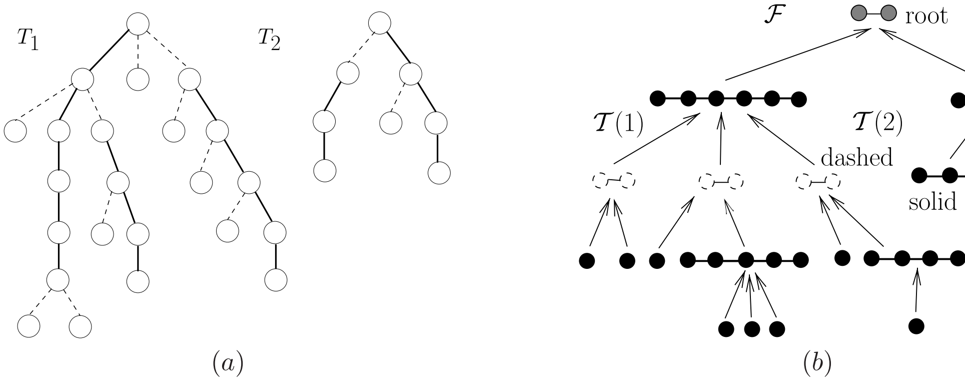

Figure 1: (a) The partition of trees into solid paths. (b) Trees of paths and the final tree $\mathcal{F}$.

Every non leaf node v of T has at most one child u_0 such that a solid edge connects them. Assume that v has more children. Consider all such children, say nodes $u_1, \ldots, u_k$ in T (that are connected with v through dashed edges). The *dashed path* $p = d(v)$ is an abstract path of length k, so that that there is one to one correspondence between edges (u_i, v) in T and nodes of p.

Let $T_1, \ldots, T_m$ be all trees in F. Let $\Pi(T_i)$ be the collection of all solid and dashed paths defined for tree T_i of F as explained above. Using the hierarchical path scheme discussed in Section 4.1, we can associate $\Pi(T_i)$ with a directed tree $\mathcal{T}(i)$ of paths. This is performed as follows:

- Each path p (solid or dashed) in $\Pi(T_i)$ corresponds to a vertex μ_p of $\mathcal{T}(i)$;
- if p is solid, for each node v of p that has only one child u in T_i such that u is node of path p' and $p \neq p'$ (and is connected with it through a dashed edge), the directed edge $(\mu_{p'},\mu_p)$ is edge of $\mathcal{T}(i)$;
- if $p' = d(v)$ is dashed with length k, that is, p' corresponds to the dashed edges of a node v in T_i, let p be the solid path that v belongs, let $u_1, \ldots, u_k$ be the corresponding children of v in T, and $p_1, \ldots, p_k$ the solid paths containing these children. Then, the directed edges $(\mu_{p'},\mu_p)$ and $(\mu_{p_i},\mu_{p'})$, $1 \leq i \leq k$, are edges of $\mathcal{T}(i)$.

Finally, given all directed trees $\mathcal{T}(i)$, we add a new *root* node ω that connects all the roots of trees $\mathcal{T}(i)$, thus, obtaining a new tree $\mathcal{F}$. We consider one last *root* path $\pi(\omega)$ that corresponds to ω: nodes of that path correspond to trees T_is of F. Any node ordering in $\pi(\omega)$ can be used.

Consider the collection of paths $\Pi(F)$ associated with the nodes of tree $\mathcal{F}$. The children of the root path $\pi(\omega)$ are solid paths. The children of a solid path are either solid of dashed paths. The children of a dashed path are solid paths. Figure 1(b) shows such a tree $\mathcal{F}$.

Using this path tree, we implement our data structure as follows. Each path (root, solid or dashed) is implemented through the path hash accumulator authentication scheme. The individual data structure that implements each path is chosen to be biased binary tree. A path property $\mathcal{P}$, a collection of path attributes, that satisfies the concatenation criterion is defined. By the implicit path interconnection, through the idea of setting path properties as node attributes, $\mathcal{P}(p)$ is able to include information about the children paths of p. We include path attributes in node properties, as follows. Let v is a node of path p and $L(p)$ denote the path hash accumulation of path p.

1. If p is root path or dashed path, then $L(p')$ and $tail(p')$ are included in $\mathcal{N}(v)$, where p' is the child solid path corresponding to node v.
2. If p is solid path, then $L(p')$ and $tail(p')$ are included in $\mathcal{N}(v)$, if v corresponds to a solid child path p', or $L(p')$ is included in $\mathcal{N}(v)$, if v corresponds to a dashed child path p'.

The above scheme yields a digest for the forest F consisting of the path hash accumulation of the root path $\pi(\omega)$ of $\mathcal{F}$. We can prove the following theorem:

Theorem 2 *Given a forest F with n nodes, there exists a fully dynamic authenticated data structure that supports path property queries with the following performance, (i) operations destroyTree and newTree take $O(1)$ time; operations link and cut take each $O(\log n)$ time; operation property takes $O(\log n)$ time; (ii) the query authentication information for operation property has size $O(\log n)$; (iii) the update authentication information is $O(1)$; (iv) the query verification time for operation property is $O(\log n)$; (v) the total space used is $O(n)$.*

Theorem 2 gives an authenticated data structure to answer path (reports the path, if any, connecting two nodes in F), areConnected (answers the question "is there a path between two nodes?"), pathLength (reports the length of the path, if any, between two nodes) and includesNodeType (answers the question "is there node of specific type in the path, if any, between two nodes?") queries on a dynamic forest of trees.

Theorem 3 *Given a forest F with n nodes, there exists a fully dynamic authenticated data structure that supports path and connectivity queries with the following performance, where k is the length of the path returned by operation path$()$: (i) operations destroyTree and newTree take $O(1)$ time; operations link and cut take each $O(\log n)$ time; operations areConnected, pathLength, includesNodeType take each $O(\log n)$ time; operation path takes $O(\log n+k)$ time; (ii) the query authentication information for operations areConnected, pathLength, includesNodeType has size $O(\log n)$ and for operation path() has size $O(\log n + k)$; (iii) the update authentication information is $O(1)$; and (iv) the query verification time for operations areConnected(), pathLength, type is $O(\log n)$ and for operation path() is $O(\log n + k)$.*

In the next subsections, we show how this result can be extended to give us authenticated schemes for more advanced graph queries. All these results have applications to the authentication of network management systems.

4.3 Authenticated Path and Connectivity Queries

We can apply Theorem 3 to design an authenticated data structure for path and connectivity queries in a graph that evolves through vertex and edge insertions with similar performance bounds. The main idea is to maintain a spanning forest of the graph. For embedded planar graphs, we can also support deletions using the data structure described in [13].

Theorem 4 *There exists an authenticated data structure for connectivity and path queries in a graph with n vertices that uses $O(n)$ space. A connectivity query takes $O(\log n)$ time and a path query takes $O(\log n + k)$ time, where k is the length of the path returned. Also, the answer verification information has size proportional to the query time. For a general graph, insertions are supported in $O(\log n)$ time each. For an embedded planar graph, insertions and deletions are supported in $O(\log n)$ time each.*

4.4 Authenticated Biconnectivity Queries

Let G be a graph that is maintained through operations: makeVertex(v) (create a new vertex v) and insertEdge(u, v, e) (add edge e between vertices u and v in G). We are interested in authenticating query operation areBiconnected(u, v) that determines whether u and v are in the same biconnected component.

We extend the data structure of [31]. We maintain the *block-cutvertex forest* $\mathcal{B}$ of G. Each tree T in $\mathcal{B}$ corresponds to a connected component of G. There are two types of nodes in T: *block nodes* that correspond to blocks (biconnected components) of G and *vertex nodes* that correspond to vertices of G. Each edge of T connects a vertex node to a block node. The block node associated with a block B is adjacent to the vertex nodes associated with the vertices of B. We have that two vertices u and v of G are in the same biconnected component if and only if there is a path between the vertex nodes of $\mathcal{B}$ associated with u and v and this path has length two. Thus, operation areBiconnected in G is reduced to operation pathLength in $\mathcal{B}$.

Theorem 5 *Given a graph G with n vertices, there exists a dynamic authenticated data structure for biconnectivity queries on G with the following performance: (i) query operation areBiconnected takes $O(\log n)$ time, update operation makeVertex takes $O(1)$ time and update operation insertEdge takes $O(\log n)$ amortized time; (ii) the query authentication information has size $O(\log n)$; (iii) the update authentication information has size $O(1)$; and (iv) the query verification time is $O(\log n)$.*

4.5 Authenticated Triconnectivity Queries

We now show how to authenticate operation areTriconnected(u, v) that reports whether there is a triconnected component containing vertices u and v.

We extend the data structure of [12], where a biconnected graph (or component) G is associated with an *SPQR tree* T that represents a recursive decomposition of G by means of separation pairs of vertices. Each S-, P-, and R-node of T is associated with a triconnected component C of G and stores a separation pair (s, t), where vertices s and t are called the *poles* of C. A Q-node of T is associated with an edge of G. Each vertex v of G is allocated at several nodes of T and has a unique *proper allocation node* in T.

Our authenticated data structure augments tree T with V-nodes associated with the vertices of G and connects the V-node of a vertex v to the proper allocation node of v in T. Also, it uses node attributes to store the type (S, P, Q, R, or V) of a node of T and its poles. We can show that operation areTriconnected can be reduced to a small number of pathLength and includesNodeType queries on the augmented SPQR tree.

Theorem 6 *Given a graph G with n nodes, there exists a dynamic authenticated data structure that supports triconnectivity queries with the following performance: (i) query operation areTriconnected takes $O(\log n)$ time, update operation makeVertex takes $O(\log n)$ time and update operation insertEdge takes $O(\log n)$ amortized time; (ii) the query authentication information has size $O(\log n)$; (iii) the update authentication information has size $O(1)$; and (iv) the query verification time is $O(\log n)$.*

5 Geometric Search

In this section, we consider authenticated data structures for geometric searching problems. Such data structures have applications to the authentication of geographic information systems.

5.1 Fractional Cascading

Fractional cascading, presented in [6], is a general algorithmic technique used in a broad class of geometric retrieval problems. It solves the *iterative search* problem which we briefly discuss.

Let U be an ordered universe and $\mathcal{C}= \{C_1, C_2, ..., C_k\}$ a collection of k catalogs, where each catalog C_i is an ordered collection of n_i elements chosen from U. For any element $x \in U$, the *successor* of x in C_i is defined to be the smallest element in C_i that is equal or greater than x. We say that we *locate* x in C_i when we find the successor of x in C_i. In the iterative search problem, given an element $x \in U$, we want to locate x in each catalog in $\mathcal{C}$. If $n = \sum_{i=1}^{k} n_i$ is the total number of stored elements, the fractional cascading technique succeeds in achieving an $O(k + \log n)$ time complexity to solve the problem, while keeping the storage linear.

The idea is to consider the catalogs as nodes in a connected graph (in the simplest case, the graph is a path) and to preprocess them, so that pairs of neighboring catalogs are correlated. Then, one can perform a binary search to locate x in some catalog, and then, using the underlying graph, to locate x in all the other catalogs by moving along neighboring ones and spending $O(1)$ time for each node transition.

$\mathcal{C}$ is associated to a graph as follows. Let G be a *single source* directed acyclic graph, that has bounded degree, i.e., each node of G has both in-degree and out-degree bounded by a constant d. Each node v of G is associated with a catalog C_v. G is called a *catalog* graph. Given G, we define $\mathcal{Q}(G)$ to be the family of all connected subgraphs $Q = (V, E)$ of G that contain s and do not contain any other node (except s) having zero in-degree in Q. The iterative search problem for the catalog graph G can then be restated as: given an element $x \in U$ and a member $Q = (V, E)$ of $\mathcal{Q}(G)$, locate x in C_v for all $v \in V$. We refer to Q as the *query* graph.

Each catalog C_v is augmented to a catalog A_v by storing some extra elements. In A_v, elements in C_v are called *proper* and the other (extra) elements are called *non-proper*. Augmented catalogs that correspond to adjacent nodes of G are connected via *bridges*. Let $e = (u, v)$ be an edge of G. A bridge connecting A_u and A_v is a pair (y, z) associating two non-proper elements y and z, where $y \in A_u$, $z \in A_v$ and $y = z$. Elements y and z have references to each other. Each non-proper element y belongs to exactly one bridge. Two neighboring catalogs A_u and A_v are connected through at least two extreme bridges that correspond to non-proper elements $+\infty$ and $-\infty$ respectively.

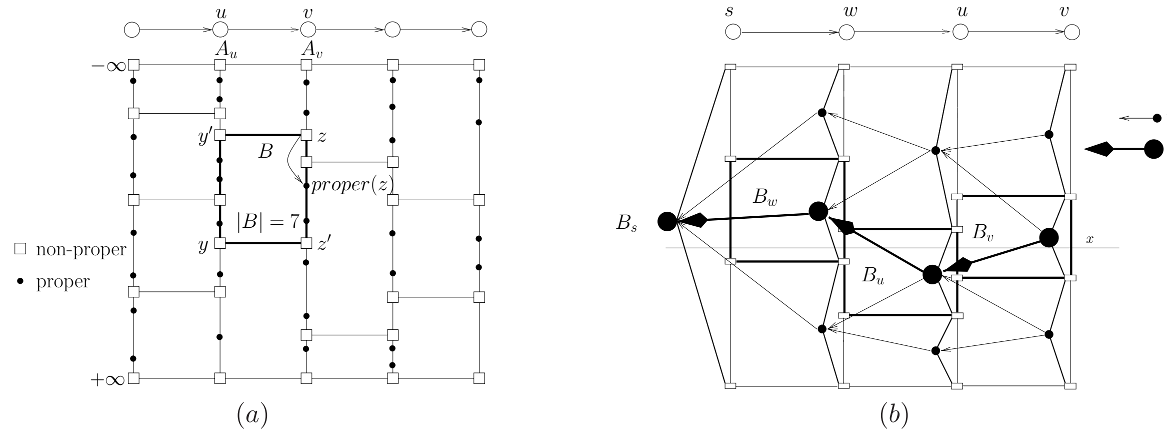

Figure 2: (a) The fractional cascading data structure over a path. Squares and dots represent non-proper and proper elements respectively. Edge (u, v) has three blocks. (b) Inter-block hashing: DAG H defines the second level hashing. For any query element x, $\mathcal{Q}$ traversal and target blocks define a tree T.

Each pair of neighboring bridges (y, z), (y', z') of edge (u, v) defines a *block* B which contains all elements of A_u and A_v lying between the two bridges. If $y' \leq y$, then bridges (y, z) and (y', z') are respectively called the *higher* and the *lower* bridge of B. The size $|B|$ of a block B is the number of the elements (both proper and non-proper) that it contains. Block sizes constitute a crucial parameter for the performance of the data structure. If n is the total size of the original catalogs, i.e., the total number of proper elements, then, as shown in [6], the total number of non-proper elements is $O(n)$ only if block sizes are proportional to the bounded degree d of G. Thus, in fractional cascading blocks are chosen to have bounded size both from above and below: for each block B, $\alpha \leq |B| \leq \beta$, where α and β are some constants proportional to d. Each non-proper element $z \in A_v$ is associated (by storing a reference) to its next proper element *proper*(z) in A_v, i.e. its successor in the original catalog C_v. Figure 2 describes the data structure built over a path of G.

Suppose that we know the successor, say l, of x at an augmented catalog A_u. In $O(1)$ time, we can locate x at C_u and, if $e = (u, v) \in E$, at A_v: starting from l, we traverse A_u moving to higher elements until a bridge, say (y, z), is reached that connects to A_v, we then follow that bridge and finally traverse A_v moving to smaller elements until x has been located. Bridge (y, z) is called the *entrance* bridge of catalog A_u.

Given the query value x and a query graph Q, we initially perform a binary search to locate x at the augmented catalog A_s, where s is the root of G, in time $O(\log n)$. Recall that the query graph Q always contains the root s. Starting from node s, we traverse Q and visit all of its nodes once. Given Q, such a traversal of Q can be performed by considering any topological order of Q. A move from a node u to an adjacent one v, corresponds to the procedure described above: having located x in C_u, we locate x in C_v in constant time. By traversing the query graph Q in this a way, we solve the iterative search problem in $O(kd + \log n)$, where k is the number of vertices of Q and n is the total number of proper elements in the catalogs of G.

5.2 Hashing over the data structure

Let h is a commutative cryptographic collision-resistant hash function. We assume that a set of rules have been defined, so that h can operate on elements of catalogs, nodes of graph G and previously computed hash values. The hashing scheme can be viewed as a two level hashing structure, built using the path hash accumulator scheme: *intra-block* hashing is performed within each block defined in the data structure and *inter-block* hashing of performed through all blocks of the data structure. In the sequel, we describe each hashing structure.

Intra-block hashing: Consider any edge (u, v) of G, i.e., u is one of the parents of v. Also, consider any two neighboring bridges (y', z') and (y, z) that define block B. Assume that $z, z' \in A_v$. We define P to be the sequence of elements of B that exist in A_v plus the non-proper elements of the corresponding bridges that lie in A_v. That is, $P = \{p_1, p_2, ..., p_t\}$, a sequence in increasing order, where, if $z' \leq z$, $p_1 = z'$ and $p_t = z$. We refer to P as the *hash side* of B. Using the path hash accumulator scheme, we compute the digest $D(P)$ of sequence P. For each element p_i, we set $\mathcal{N}(p_i) = \{\text{proper}(p_i), v\}$ and in that way the path hash accumulator can support authenticated membership queries and authenticated path property queries. Here one such property of P is the corresponding node v.

We iterate the process for all blocks defined in the data structure: for each block B in the data structure having a hash side P in A_v, H_B is the hash of v and the digest of the $D(P)$. We also define B_s to be a fictitious block, the augmented catalog A_s. The hash side of B_s is all the block itself and in such a way the hash value H_{B_s} is well defined and can be computed. All the path hash accumulator schemes used define the first level hashing structure.

Inter-block hashing: The second level hashing structure is defined through a directed acyclic graph $\mathcal{H}$ defined over blocks. That is, nodes of $\mathcal{H}$ are blocks of the data structure. Suppose that w is a parent of u and u is a parent of v in G. If B is a block of an edge (u, v), then we add to the set of edges of $\mathcal{H}$ all the directed edges (B, B'), where B' is a block of edge (w, u) that shares elements from A_u with B. Additionally, if v is a child of the root s in G, then for all blocks B in edge (s, v) we add to the set of edges of $\mathcal{H}$ all the directed edges (B, B_s). The construction of $\mathcal{H}$ is now complete. B_s is the unique root of $\mathcal{H}$. Figure 2(b) shows the graph $\mathcal{H}$ that corresponds to a path.

Each block (node) B of $\mathcal{H}$ is associated with a label $L(B)$. If B is a leaf (sink) in $\mathcal{H}$ then $L(B) = H_B$. If B is the parent of blocks $B_1, B_2, \ldots, B_t$ in $\mathcal{H}$, listed in arbitrary order, then $L(B)$ equals the path hash accumulation over $B_1, B_2, \ldots, B_t$ using $\mathcal{N}(B_i) = \{B_i, H_{B_i}\}$. This hashing over $\mathcal{H}$ corresponds to the second level hashing structure. Finally, we set $D(\mathcal{D})= L(B_s)$ to be the digest of the whole data structure $\mathcal{D}$, that is signed by the source.

5.3 Answer authentication information

Given the hash scheme that we have developed over the catalog graph G, a query graph Q and a query element x, we describe now what is the authentication information given to the user.

Let x be the query element and let v be any node of the query graph Q. Let s_v be the successor of x in C_v. In the location process, while locating x in the augented catalog A_v, we find two consecutive elements z and y of A_v, such that $z \leq x \leq y$. Elements y and z may be either proper or non-proper. They are both elements of a block B, such that the entrance bridge of A_v is the higher bridge of B. We have that y is the successor of x in A_v and that $s_v = y$, if y is proper, or $s_v = \text{proper}(y)$, if y is non-proper. We call y and B, respectively, the target element and the target block of A_v.

Two useful observations are that: (1) in the location process, the traversal of the query graph Q is chosen so that each node of Q is visited once and (2) any two blocks visited by the location process (target blocks) that correspond to incident edges in Q share elements of the common augmented catalog, and, thus, are adjacent in graph $\mathcal{H}$. It follows that all the target blocks define a subgraph T of $\mathcal{H}$. T consists of the all target blocks and the edges of $\mathcal{H}$ that connect neighboring target blocks (Figure 2(b)).

Lemma 7 *For any query graph Q, graph T is a tree.*

For any node v, let y_v be the target element of A_v and B_v the target block of A_v. The answer authentication information will consist of:

1. *Intra-block:* for each node v of Q, the target element y_v of A_v and a verification sequence p_v from y_v up to the path hash accumulation of the hash side of B_v, and
2. *Inter-block:* for every node (or target block) B_v of T that is not a leaf, the verification sequences from every child of B_v in T up to the pash hash accumulation $L(B_v)$.

Lemma 8 *If n is the total number of proper elements in the catalogs of G and d is the bounded degree of G, then for any query graph Q of k nodes, the size of the answer authentication information is $O(\log n + k \log d) = O(\log n + k)$.*

5.4 Verification of an answer

We assume that the answer given to the user is a set $A = \{(a_v, v) : v \text{ is node of } G\}$, where a_v is supposed to be the successor of x in C_v. The answer authentication information consists of two verification sequences for each node (target block) of T: one intra-block and one inter-block. These sequences form a hash tree in our two level hashing scheme. The verification process is basically defined by this hash tree. Intuitively, an intra-block verification sequence of a target block B_v provides a *local proof* that a_v is the successor of x in C_v, and then, all these local proofs are accumulated through inter-block verification sequences into the signed digest.

In particular, given the elements x, y, z, and a node v, if the predicates: 1) $z \leq x \leq y$, 2) y and z are consecutive elements in A_v, and 3) if $x \neq y$ and y is non-proper then proper(y) is the next proper element of y in A_v, hold simultaneously, then they constitute a proof that the successor of x in C_v is element proper(y). Such a proof must be given for every v of Q.

Given A, x and the answer authentication information, the user first checks to see if there is any inconsistence between values a_v and y_v for every v of G, i.e. if $a_v \neq y_v$, if y_v is proper, or if $a_v \neq$ proper(y_v) otherwise. If there is at least one inconsistence, the user rejects the answer. Otherwise, all that is needed is to verify the signed digest $D(\mathcal{D})$ of the data structure. Observe, that the user possesses all the data needed for the computation of the signed digest.

Lemma 9 *If n is the total number of proper elements in the catalogs of G, then for any query graph Q of k nodes, the answer verification time is $O(\log n + k \log d) = O(\log n + k)$, where d is the bounded degree of G.*

If the digest is verified, then based on the collision-resistance property of the hash function h, the user has a proof that the answer is correct: for each v of G, the user can verify all the three conditions previously discussed. A faulty answer can lead to a forged proof only if some collisions of h have been found. Thus, the security of our scheme is reduced to the collision-resistance property of h.

Lemma 10 *For any catalog graph G of k nodes and of total size n, both intra-block and inter-block hashing schemes can be computed in $O(n)$ time using $O(n)$ storage.*

Theorem 11 *Given a catalog graph G of bounded degree d and of total size n, the authenticated fractional cascading data structure $\mathcal{D}$ for G solves the authenticated iterative search problem for G with the following performance : $\mathcal{D}$ can be constructed in $O(n)$ time and uses $O(n)$ storage; given an element x and a graph query Q with k vertices, x can be located in every catalog of Q in $O(\log n + k)$ time; and the answer authentication information has size $O(\log n + k)$; the answer verification time is $O(\log n + k)$.*

5.5 Applications

Our authenticated fractional cascading scheme can be used to design authenticated data structures for various fundamental two-dimensional geometric search problems, where iterative search is implicitly performed (see [7]). In all of these problems, the underlying catalog graph has degree bounded by a small constant. In the following, n denotes the problem size.

Theorem 12 *There is an authenticated data structure for answering line intersection queries on a polygon that can be constructed in $O(n \log n)$ time and uses $O(n \log n)$ storage. Denoting with k the output size, queries are answered in $O(\log n + k)$ time; the answer authentication information has size $O((k+1)\log \frac{n}{k+1})$; and the answer verification time is $O((k+1)\log \frac{n}{k+1})$.*

Theorem 13 *There are authenticated data structures for answering ray shooting and point location queries that can be constructed in $O(\log n)$ time and use $O(n \log n)$ storage. Queries are answered in $O(\log n)$ time; the answer authentication information has size $O(\log n)$; and the answer verification time is $O(\log n)$.*

Theorem 14 *There are authenticated data structures for answering orthogonal range search, orthogonal point enclosure and orthogonal intersection queries that can be constructed in $O(n \log n)$ time and use $O(n \log n)$ storage. Denoting with k the output size, queries are answered in $O(\log n + k)$ time; the answer authentication information has size $O(\log n + k)$; and the answer verification time is $O(\log n + k)$.*

All these results have applications to the authentication of geographic information systems.

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

[3] J. D. Bright and G. Sullivan. Checking mergeable priority queues. In *Digest of the 24th Symposium on Fault-Tolerant Computing*, pages 144–153. IEEE Computer Society Press, 1994.

[4] J. D. Bright and G. Sullivan. On-line error monitoring for several data structures. In *Digest of the 25th Symposium on Fault-Tolerant Computing*, pages 392–401. IEEE Computer Society Press, 1995.

[5] J. D. Bright, G. Sullivan, and G. M. Masson. Checking the integrity of trees. In *Digest of the 25th Symposium on Fault-Tolerant Computing*, pages 402–411. IEEE Computer Society Press, 1995.

[6] B. Chazelle and L. J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1(3):133–162, 1986.

[7] B. Chazelle and L. J. Guibas. Fractional cascading: II. Applications. *Algorithmica*, 1:163–191, 1986.

[8] R. F. Cohen and R. Tamassia. Combine and conquer. *Algorithmica*, 18:342–362, 1997.

[9] 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.

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

[11] O. Devillers, G. Liotta, F. P. Preparata, and R. Tamassia. Checking the convexity of polytopes and the planarity of subdivisions. *Comput. Geom. Theory Appl.*, 11:187–208, 1998.

[12] G. Di Battista and R. Tamassia. On-line maintenance of triconnected components with SPQR-trees. *Algorithmica*, 15:302–318, 1996.

[13] D. Eppstein, G. F. Italiano, R. Tamassia, R. E. Tarjan, J. Westbrook, and M. Yung. Maintenance of a minimum spanning forest in a dynamic plane graph. *J. Algorithms*, 13(1):33–54, 1992.

[14] U. Finkler and K. Mehlhorn. Checking priority queues. In *Proc. 10th ACM-SIAM Symp. on Discrete Algorithms*, pages S901–S902, 1999.

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

[16] M. T. Goodrich, R. Tamassia, and J. Hasic. An efficient dynamic and distributed cryptographic accumulator. Technical report, Center for Geometric Computing, Brown University, 2001. http://www.cs.brown.edu/cgc/stms/papers/accumulators.pdf.

[17] 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.

[18] V. King. A simpler minimum spanning tree verification algorithm. In *Workshop on Algorithms and Data Structures*, pages 440–448, 1995.

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

[20] G. Liotta. Low degree algorithms for computing and checking Gabriel graphs. Technical Report CS-96-28, Center for Geometric Computing, Dept. Computer Science, Brown Univ., 1996.

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

[22] K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, UK, 2000.

[23] K. Mehlhorn, S. Näher, M. Seel, R. Seidel, T. Schilz, S. Schirra, and C. Uhrig. Checking geometric programs or verification of geometric structures. *Comput. Geom. Theory Appl.*, 12(1–2):85–103, 1999.

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

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

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

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

[28] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–381, 1983.

[29] G. F. Sullivan and G. M. Masson. Certification trails for data structures. In *Digest of the 21st Symposium on Fault-Tolerant Computing*, pages 240–247. IEEE Computer Society Press, 1991.

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

[31] J. Westbrook and R. E. Tarjan. Maintaining bridge-connected and biconnected components on-line. *Algorithmica*, 7:433–464, 1992.

Appendix

Proof of Lemma 1 Consider the query property($\bar{p}(v,u)$) on p. Moreover, consider the set $\mathcal{A}(\bar{p})$ of *allocation nodes* in $T(p)$ of subpath $\bar{p} = \bar{p}(v,u)$. For a tree node w, $w \in\mathcal{A}(\bar{p})$ if the leaves of the subtree defined by w are all nodes of $\bar{p}$ but the same is not the case for w's parent, if any. Observe that each $w \in\mathcal{A}(\bar{p})$ corresponds to a subpath of path $\bar{p}$. Since $T(p)$ is a balanced binary tree, there are $O(\log n)$ allocation nodes for subpath $\bar{p}$ that can be found in $O(\log n)$ time by tracing the leaf-to-root tree paths in $T(p)$ from v and u up to r. Since, the path property $\mathcal{P}$ satisfies the concatenation criterion, we have that the path property $\mathcal{P}(\bar{p})$ can be computed by using the tree structure and by applying $O(\log n)$ times the concatenation function $\mathcal{F}$.

Clearly, the answer given to the user is the property $\mathcal{P}(\bar{p})$. For any node u of $T(p)$, let $(u_1, \ldots, u_k)$ be the node-to-root tree path connecting u with the root r of $T(p)$. We define the *verification sequence* of u to be the sequence $\mathcal{V}(u) = (s_1, s_2, ..., s_k)$, where s_j, $1 \leq j \leq k$, is the label of the sibling of node u_j. The answer authentication information consists of three parts: (1) For each allocation node w of subpath $\bar{p} = \bar{p}(v,u)$, the property $\mathcal{P}(w)$; these properties are given as a sequence $(\alpha_1, ..., \alpha_m)$, such that the set of leaf nodes of any allocation nodes α_i and α_{i+1}, $1 \leq i \leq m-1$, forms a subpath of $\bar{p}$; (2) For each $w \in\mathcal{A}(\bar{p})$ the labels of its children, if they exit; and the label of the siblings of the left most and right most allocation nodes of $\bar{p} = \bar{p}(v,u)$; (3) If z is the least common ancestor of v and u, the verification sequence of z.

Given the answer authentication information, the user is able to proof the validity of the answer $\mathcal{P}(\bar{p})$. In particular, the user first uses sequence $(\alpha_1, ..., \alpha_m)$ to recompute $\mathcal{P}(\bar{p})$, by repeatedly applications of the concatenation function $\mathcal{F}$. If $\mathcal{P}(\bar{p})$ is not confirmed, the user rejects the answer. Otherwise, the user completes the verification process by recomputed the path hash accumulation. Since, $head(p) \in \mathcal{P}(p)$ and $tail(p) \in \mathcal{P}(p)$, we achieve the desired security result: in order an incorrect answer given to the user to be accepted as correct, at least one collision on the cryptographic hash function h must be computed.

For any property(v) query, we proceed as above; just observe that property(v) corresponds to a property(v,v). For any locate(p,σ,q) query, we locate the target node v by performing a top-down search into $T(p)$ starting from the root: at a node u with children w_1 and w_2, the path selection function σ is used to select either the path that corresponds to w_1 or the path that correspond to w_2. Then, the answer is the located node v and the proof is the proof that corresponds to a property(v,v) operation. For any subpath(p,α,q) query, a similar top-down tree search is performed using the path advance function α to first compute the target subpath; the proof is constructed used by considering the corresponding allocation nodes. □

Proof of Theorem 2 Let size(v) denote the number of nodes in the subtree defined by v and let u the parent node of v. Edge $e = (u, v)$ is called *heavy* if size(u) > size(v)/2. The edge labeling of a dynamic tree T of n nodes with root w and n nodes, such an edge is labeled solid only if it is heavy, has the following important property: for any node u of T there are at most $\log n$ dashed edges on the path from u to w.

We use such an edge labeling, for the partition of T into solid paths. Consider the collection of paths Π that correspond to the final tree $\mathcal{F}$ (after the edge paths and the root path have been added). Each path p in Π is implemented as a biased binary tree $T(p)$, where node weights are defined using function size(). We consider the weight $w(v)$ of node v to be either a node or a path property (depending on if v is leaf node in $T(p)$ or not). If p is a path having no child path (μ_p is a leaf in $\mathcal{F}$), then $w(v) =$ size(v). Otherwise (μ_p is not a leaf in $\mathcal{F}$), then $w(v) = w(u_1) + w(u_2)$, if v is internal node in $T(p)$ and u_1,u_2 children of v. Otherwise, v is a node of path p. If $v = head(p)$ and p is solid, then $w(v) = w(u_1) + w(u_2)$, where u_1,u_2 are the children of v in T (through dashed edges). If $v \neq head(p)$ and p is solid, then $w(v) = w(u) + 1$, where u is the unique dashed child of v and $w(u) = 0$, if no such child exists. If p is dashed, $w(v) = w(u) + 1$, where u is the node connected with v with the corresponding dashed edge. If p is root path, again $w(v) = w(u) + 1$, where u is the root of the corresponding tree root.

Using the above biasing, it can be shown that any leaf-to-root path in $\mathcal{F}$, when performed *through* the individual biased trees $T(p)$s, has length $O(\log n)$. The proof is based in the analysis in [28]. Observe, that all operations correspond to accessing (and modifying) paths of this kind. In particular:

Updates Operations *link*() and *cut*() can be implemented in $O(\log n)$ time by modifying only $O(\log n)$ path hash accumulators and by examining, modifying and restructuring only $O(\log n)$ nodes in total. Restructuring means connecting a node to new children. Observe that the path property P satisfies the concatenation criterion. Our scheme works by, every time a node v is restructured, recalculating $L(v)$, which can be done in $O(1)$ time, since the values of the children and neighbors of u are known. Consequently, our update operations can be performed in $O(\log n)$ time.

Queries The first step in processing query $\mathsf{property}(u, v)$ is to determine if nodes u and v are in forest F. We can use the authenticated data structure of [15] that supports containment queries in $O(\log n)$ time with responses of length $O(\log n)$. Given that nodes v and u are in $\mathcal{F}$, the path property query is performed by accessing three *multipaths*, i.e., three paths that is the concatenation of subpaths of paths in Π according the hierarchy induced by $\mathcal{F}$. Consider the least common ancestor μ_l of μ_v and μ_u in tree $\mathcal{F}$ (μ_l may overlap with μ_v and/or μ_u and l is the actual least common ancestor considering the individual trees). Let $\Pi(v)$, $\Pi(u)$ be the multipaths from μ_v, μ_u to μ_l and $\Pi(l)$ be the multipath from μ_l to the root of the tree implementing $\pi(\omega)$. Following $\Pi(v)$ and $\Pi(u)$, we compute the answer $A(v, u)$ considering the allocation nodes of each path hash accumulator visited. Similarly, we provide property *subproofs* $proof(\Pi(v))$, $proof(\Pi(u))$ for each path hash accumulator that is visited. Finally, following $\Pi(l)$ up to the signed root, we compute the *multipath verification sequence* $\mathcal{V}(l)$ that allows the user to recalculate the signed hash value given $A(v, u)$, $proof(\Pi(v))$ and $proof(\Pi(u))$. By the biased scheme used over $\mathcal{F}$, the set of allocation nodes is $O(\log n)$, thus, the size of the answer $A(v, u)$ and the proof $(\mathcal{V}(l)$,$proof(\Pi(v))$,$proof(\Pi(u)))$ is $O(\log n)$. The verification time is also $O(\log n)$.

Security The hashing scheme is based on the path hash accumulation scheme. By allowing neighboring (in $\mathcal{F}$) paths to share information (properties) we achieve the desired security results: any attack to our data structure is reduced to a collision on the cryptographic hash function h. □

Proof Sketch of Theorem 3 Both operations correspond to authenticating a path property. $\mathsf{areConnected}$ corresponds to the existence or not of some node of the root path $\pi(\omega)$ in the path in $\mathcal{F}$ connecting v and u (there is always such a path). This can be easily expressed by assigning some unique *id* value to every path in Π. The same idea is applied for $\mathsf{includesNodeType}$ operation. $\mathsf{pathLength}$ can be achieved by including the path attribute $P(v) = \#\{\text{leaves in subtree defined by } v\}$ in path property $\mathcal{P}(v)$. path queries are answered by first performing a $\mathsf{areConnected}$ query. If there is a path, it can be found by including all leaf nodes of the subtree $T(v)$ defined by node v in path property $\mathcal{P}(v)$ and answering a path property query. To this end, we need a slightly different definition for the path attribute. Observe that $\mathcal{P}(v) = O(size(T(v)))$ and, thus, the introduced complexity is $O(\log n + k)$, where k is the length of the path. □

Proof Sketch of Lemma 7 Consider the topological order used to define the traversal of the query graph G. This topological order defines a directed subtree T_Q of Q. There is a correspondence between edges of T_Q and target blocks, i.e. between edges of T_Q and nodes of T. □

Proof of Lemma 8 The hash side of B_s has size $|A_s| = O(n)$ and the hash side of any other target block has size at most $\beta = O(d)$. Thus, the intra-block answer authentication information consists of k verification sequences, $k-1$ of length $O(\log d)$ and one of length $O(\log n)$, and, thus, has size $O(\log n + k \log d)$ size. For the inter-block answer authentication information, recall that G and, thus, both Q and T_Q, have out-degree bounded by d and that every target block can share elements with at most $\beta = O(d)$ other target blocks. Thus, T has in-degree bounded by $O(d)$. That is, all, but $L(B_s)$, the second level path hash accumulations are built over sequences of length $O(d)$. $L(B_s)$ is built over at most dn blocks that share elements with A_s. Observe that there is a one-to-one correspondence between every inter-block verification sequence and an edge in T. It follows that the inter-part answer authentication information consists of $k-2$ verification sequences of size $\log d$ and one of size $\log n$, thus, has $O(\log n + k \log d)$ size. In total, since d is a constant, the answer authentication information is of size $O(\log n + k)$. □

Proof Sketch of Lemma 9 Recall that the verification time of a path hash accumulator is proportional to the size of the verification sequence. □

Proof Sketch of Lemma 10 G has bounded in-degree by $d = O(1)$ and every target block can share elements with at most $\beta = O(1)$ other blocks. Moreover, the path hash accumulation of a sequence of length m can be computed in $O(m)$ time and space. □