Databases that tell the Truth: Authentic Data Publication*

Michael Gertz[1] April Kwong[1] Charles U. Martel[1] Glen Nuckolls[2]
Premkumar T. Devanbu[1] Stuart S. Stubblebine[3]
[1] University of California at Davis, CA
[2] University of Texas at Austin, TX
[3] Stubblebine Research Labs, Madison, NJ

Abstract

The publication of high-value and mission critical data on the Internet plays an important role in the government, industry, and health-care sectors. However, owners of such data are often not able or willing to serve millions of query requests per day and furthermore satisfy clients' data requirements regarding the integrity, availability, and authenticity of the data they manage in their databases.

In this article, we give an overview of our work on authentic publication schemes in which a data owner employs a (possibly untrusted) data publisher to answer queries from clients on behalf of the owner. In addition to query answers, publishers provide clients with verification objects a client uses to verify whether the answer is the same as the owner would have provided. We consider two popular types of database systems, those managing relational data and those managing XML data in the form of XML repositories.

1 Introduction

In today's information-centric society, the reliable dissemination of information over the Internet plays a crucial role, in particular when the data is vital to high-value decisions in financial, health-care, and government sectors. Owners of such data must provide mechanisms that ensure integrity, authenticity, and non-repudiation requirements of data consumers. However, providing such a protection over public networks is an expensive matter. In addition to maintaining a secure computing and database infrastructure that ensures the availability of data and query services, owners have to digitally sign results of queries submitted by clients to ensure non-repudiation of the results. Data owners, however, may not be willing or able to maintain an infrastructure that protects signing keys against attacks and can furthermore manage millions of queries a day.

A possible solution to this problem is that the data owner uses third-party data publishers that provide for a secure and efficient computing infrastructure. A data owner periodically distributes the data to publishers, which then answer queries from clients on behalf of the data owner. Although this approach is much more scalable, a client querying a data publisher might worry that the publishers engages in deception and does not give the same result to a query as the data owner would have given. The client would also have to trust the key management of the publisher. In general, clients simply might not trust data publishers.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

*Funding for this work was provided by the NSF ITR award 0085961.

In this paper, we present an approach called *authentic publication scheme* where a data owner employs one or more *untrusted data publishers*. In this approach, which is illustrated in Figure 1, the data owner employs digests that are bottom-up hashes computed recursively over tree-type structures representing the entire set of objects in the owner's database. These digests are distributed to all clients interested in the owner's data. This is done in a secure fashion by using, for example, signing keys. The owner then distributes his data to data publishers and finally goes off-line. When a clients now issues a query against a data publisher, he receives a query result and a *verification object (VO)*. This VO is generated by the publisher and is based on the bottom-up hash scheme previously used by the data owner. The client then uses the VO, previously distributed digests, and the query result to verify that the result is correct, i.e., the same result would have been computed by the owner. The VO in combination with the digest realize a hard to forge proof the client uses to verify the answer. If the publisher provides the client with an incorrect query result or forged verification object, the client is able to detect such cases of possible deception.

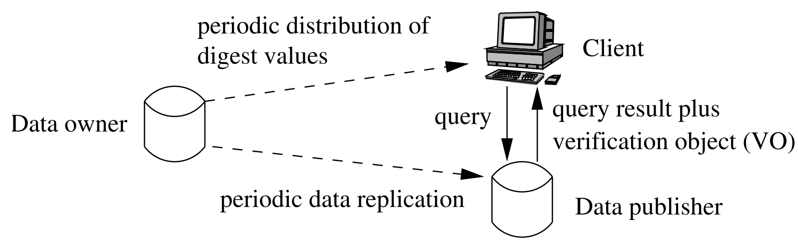

Figure 1: Authentic Publication Scheme

Our approach is founded on cryptographic assumptions regarding the security of hash-functions and public key systems. Compared to related approaches, there are several advantages of our authentic publication scheme. First, a client only has to trust the digests distributed by the data owner; he does not have to trust the publisher. Second, data owners only have to periodically distribute the digest of their data to clients and their data to data publishers, and remain off-line otherwise. Third, verification objects are nearly linear in the size of the answer to a query.

In the following section, we present the authentic publication scheme in the context of relational databases and describe how data owner and data publishers compute digests and verification objects for different types of queries, respectively. In Section 3, we illustrate how the authentic publication scheme is used in the context of databases that manage XML data. We conclude in Section 4 with a summary and outline of future work.

2 The Relational Case

In the following, we present the authentic publication scheme for the case where data owner and publisher employ a relational database. We first present the concept of Merkle-Hash Trees owner and publisher use to compute digest values and verification objects for query results, respectively. We then illustrate how results for different types of queries against a relational database can be verified by clients. Details of this approach can be found in [3, 5].

2.1 Merkle-Hash Trees

The authentic publication scheme is based on digests (distributed by data owners) and verification objects (computed by data publishers) that are determined from tree-type structures over sets of data. For this, we adopt a Merkle-Hash Tree approach [10]. Assume a relation R with attributes $\mathcal{A} = \langle A_1, A_2, \ldots, A_n \rangle$. A Merkle-Hash

Tree over R, denoted $MHT(R, \mathcal{A})$, is a binary tree with $|R|$ leaf nodes and hash value $h(i)$ associated with each node i of the tree as follows.

1. First, for each tuple $t \in R$, a tuple hash $h_t = h(h(t.A_1)||h(t.A_2)|| \ldots ||h(t.A_n))$ is computed using a collision resistant hash function ($||$ denotes the concatenation operator). For a hash length of 128 bits, for example, the probability of random collisions of hash values is close to 2^{-128}.
2. Next, assume a total order on R, based on the primary key in $\mathcal{A}$ or any other sequence of attributes in $\mathcal{A}$ (which might include the primary key as last sorting criterion). The hash value $h(i)$ for a node $i \in MHT(R, \mathcal{A})$ is computed as follows.
   (a) If i is a leaf node, then $h(i) = h_t(t_k), k = 1, \ldots, |R|$.
   (b) If i is an internal node, then $h(i) = h(h(l), h(r))$, where l and r are the left and right children of node i.

The root hash, denoted $h_{R,\mathcal{A}}$ is the *digest* of all values in the Merkle-Hash Tree $MHT(R, \mathcal{A})$. Figure 2 illustrates the computation of the digest for a sorted relation. An important property of this approach is that if the hash

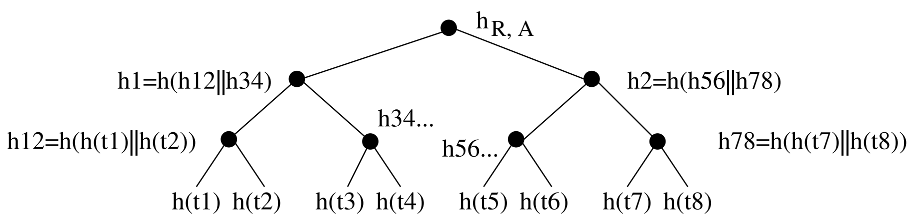

Figure 2: Computation of digest using Merkle-Hash Tree on a sorted relation with tuples $t1, \ldots, t8$

value for a node is known, then the hash values of its children (and values of tuples in the case of leaf nodes) cannot be forged. This property is based on cryptographic assumptions of collision-resistant hash functions and has initially been used by Naor and Nissim in their work on revocation and updates of certificates [13].

In our approach, the data owner computes digests for different relations and sorting criteria used in client applications, and securely distributes the digests to clients. The data publisher maintaining a copy of the owner's data uses the Merkle-Hash Tree scheme to compute verification objects for query results. That is, the hash function used by the owner for computing digests is known to clients and publisher. An important concept in computing verification objects is that of a *hash path*. A hash path describes nodes and node values in a Merkle-Hash Tree that are necessary and sufficient to recompute the root digest and thus to verify the existence (or absence) of tuples in a relation.

Definition 1: Let i be the leaf node corresponding to a tuple $t \in R$ in a Merkle-Hash Tree $MHT(R, \mathcal{A})$. The hash path for i, denoted $path(i)$, comprises all nodes necessary to compute the root digest $h_{R,\mathcal{A}}$. Such a hash path always has the length $\lceil \log_2(|R|) \rceil$ and comprises $2 * \lceil \log_2(|R|) \rceil - 1$ nodes where exactly two nodes are leaf nodes. Of these, only $\lceil \log_2(|R|) \rceil + 1$ are needed to recompute $h_{R,\mathcal{A}}$. Hash paths can also be provided for non-leaf nodes.

Example 1: Consider the tuple $t6$ in Figure 2. In order to recompute $h_{R,\mathcal{A}}$, the tuple $t5$ (or its hash value $h(t5)$) and the values $h78$ and $h1$ are necessary and sufficient. If a client knows the digest $h_{R,\mathcal{A}}$ and the tuple $t6$ is in R, then any incorrect value for $t6$ or nodes in $path(t6)$ should result in a hash value different from the digest. Now assume one wants to show that there is no tuple t' in R such that $t5 < t' < t6$ (given the criterion on which R is sorted in $MHT(R, \mathcal{A})$). The two tuples $t5$ and $t6$, and the node values $h78$ and $h2$ would suffice to prove this case. The two tuples $t5$ and $t6$ can be considered *witnesses* for the absence of t'. Any other combination of tuples or incorrect (forged) hash values for inner nodes would again result in a digest different from $h_{R,\mathcal{A}}$.

The nodes in $path(i)$ constitute a *verification object (VO)*, showing that tuple t_i is (not) in relation R with digest $h_{R,\mathcal{A}}$. Verification objects are computed by data publishers and passed on to clients in addition to the result of a query. Clients take the query result and the VO and recompute the digest as illustrated above (details of a respective protocol are given in [3]). If the recomputed digest is the same as the digest previously distributed by the data owner, the client can be assured that (1) no qualified tuple has been left out from the result, and (2) the result only contains qualified tuples. If the digests do not match, then either the query result is incorrect (i.e., different from what the data owner would have computed) or hash values have been forged.

The above approach can be easily extended to provide a proof for the existence of a sequence of tuples in a relation R. Recall that we always assume a total order among tuples representing the leaves in a Merkle-Hash Tree $MHT(R, \mathcal{A})$. To prove the existence of a certain sequence s of tuples, there are always two tuples that are not included in s, and they describe the greatest lower bound $glb(s)$ and lowest upper bound $lub(s)$ of s, respectively. If the tuple $glb(s)$ ($lub(s)$) does not exist in R, its value is determined based on the smallest (largest) tuple in s. Now let $cover(s)$ denote the inner node in $MHT(R, \mathcal{A})$ that is the root node of the minimal subtree in $MHT(R, \mathcal{A})$ having all tuples in s as leaf nodes. Obviously, for the leaf nodes $glb(s), lub(s)$ and the inner node $cover(s)$, there always exists a lowest common ancestor node, denoted $lca(s)$, which in the extreme case is the root of the Merkle-Hash Tree.

To prove the existence of s, the VO then basically comprises (1) the hash path for $lca(s)$, and (2) the hash paths for $glb(s)$ and $lub(s)$. Given the sequence s of tuples, this information is necessary and sufficient to show that s is in R. The same approach is used to prove that a sequence is empty (using $glb(s)$ and $lub(s)$ as witness tuples), thus showing that a tuple does not exist in R. In the following, we call VOs for cases where witness tuples $glb(s)$ and $lub(s)$ are used to "surround" (non-existing) sequences of tuples as *boundary case VOs.*

Example 2: Consider the sequence $s = \langle t6, t7 \rangle$ in Figure 2. We have $glb(s) = t5$ and $lub(s) = t8$ as "near miss" tuples, and $cover(s) = h2$. The node $h2$ also happens to be $lca(s)$. In this case, the VO for s includes hash paths for $glb(s)$ and $lub(s)$ to the node $h2$, and the hash path for $h2$. Using the values for $s6$ and $s7$, the digest of the Merkle-Hash Tree can be recomputed.

2.2 Verification of Answers to Queries

In the following, we illustrate the computation of verification objects (VOs) in the context of different types of queries formulated in relational algebra. The types of queries are known to the data owner and publisher. This is a reasonable assumption since queries are typically embedded in client applications and not ad hoc, and clients are interested in obtaining proofs for results to their queries. We assume that respective digests on Merkle-Hash Trees that support such query patterns have been distributed by the data owner to clients and that the data publishers knows about these Merkle-Hash Trees structures.

Selection. Suppose a client issues selection queries of the form $\sigma_{A_i \Theta C}(R)$ against the publisher's database, and a result set $q \subseteq R$ is returned by the publisher. Tuples in the Merkle-Hash Tree supporting the computation of VOs are sorted on A_i. The following cases are considered for Θ being the equality predicate: (1) if A_i is the primary key in R and $q = \{t\}, t \in R$, then there exists exactly one tuple satisfying the query. In this case, the VO is simply the hash path for t. (2) if A_i is the primary key in R and $q = \emptyset$, then the VO is based on the two witness tuples that would "surround" the non-existing result tuple. That is, the VO is a boundary case VO as illustrated at the end of Section 2.1. (3) A_i is not a primary key and $q \neq \emptyset$. In this case, the result set builds a contiguous sequence s of tuples in $MHT(R, A_i)$. Also in this case, a boundary case VO is constructed. In case A_i is not the primary key and $q = \emptyset$, the same approach as for case (2) is applied. For selection conditions based on a predicate $\Theta \in \{\neq, <, >\}$, results to a query typically comprise one or two sequences of tuples (leaf nodes in $MHT(R, \mathcal{A})$) that satisfy the condition. Thus, again boundary case VOs are used to show the correctness of respective answer sets.

In all the above cases, the size of the VO is linear in the size of the query result and $\log(|R|)$. The VO plus query answer are sufficient to show that the answer to a selection query provided by the publisher is exactly the same as the data owner would have computed.

Projection. Computing VOs for projection queries of the form $\pi_{\mathcal{A}'}(R)$, where $\mathcal{A}'$ is a proper subset of attributes in R, involves two components. Assume a Merkle-Hash Tree in which the tuples (leaf nodes) are sorted based on $\mathcal{A}'$. Each tuple t in an answer q to a projection query potentially results from a set of tuples $S(t) \subset R$, where tuples in $S(t)$ have identical values for the attributes $\mathcal{A}'$. To show that t is in the answer set, the hash path for a witness tuple from $S(t)$ is provided as part of the VO. To show that there are no missing tuples in between two tuples t and t' $(t, t' \in q)$, the VO also has to provide information that the sets $S(t)$ and $S(t')$ are consecutive sequences in $MHT(R, \mathcal{A}')$. This again is done using boundary case VOs.

Joins. We illustrate the case for the most common type of join, the natural join $R \bowtie S$. The Merkle-Hash Tree supporting the computation of respective VOs is constructed as follows. Assume attribute A occurring in both R and S is the join attribute. Using the full outer join $R ⟗ S$, three groups of leaf nodes are obtained for the Merkle-hash Tree: (1) tuples that result from the natural join $R \bowtie S$, (2) right null-padded tuples from R for which no matching tuples in S exist, and (3) left null-padded tuples from S for which no matching tuples in S exist. That is, the data publisher materializes potential results to join queries. Such an approach is reasonable since our scheme works under the assumption that data publishers can provide the computing and database infrastructure necessary to efficiently answer millions of queries a day. The data owner, on the other hand, has to materialize the respective Merkle-Has Tree only once to determine and distribute digests to clients.

VOs for answers to natural join queries obviously employ boundary case VOs, as described in Section 2.1. It should be noted that the materialization of $R ⟗ S$ has another advantage, namely that VOs for answers to queries that have conditions on the non-existence of matching tuple can be easily computed.

Complex Queries. We conclude this section with a brief outline of how queries are handled that contain nested operators, in particular those that resemble select-from-where queries used in SQL. The key idea, which is detailed in [3], is to use so-called *multi-dimensional range trees*, a data structure used in computational geometry [1] to deal with sets of points in a multi-dimensional space. In this approach, an initial Merkle-Hash Tree is computed, say for a complete and sorted relation. This tree is used to computer partial VOs for outer query components, e.g., a final selection of tuples that satisfy a (join) condition. Each inner node of that tree (a node covering a sequence of leaf node tuples) is linked to another Merkle-Hash Tree. This tree then supports the computation of VOs for subsequent query component on only the covered tuples, e.g., a join of only these tuples with tuples from another relation or another selection condition. The construction and maintenance of such "cascading Merkle-Hash Tree" relies on the assumption that the patterns of queries submitted by clients to the data publisher are known and that the publisher has sufficient resources to constructs such tree structures based on to the queries patterns and the data owners relations.

3 The XML Case

Recently there have been several advancements in using the eXtensible Markup Language (XML) for the exchange, management, and querying of information on the Internet. Several applications settings in industry and government already employ XML to exchange and manage document data in a flexible and standardized fashion. In these settings, XML Repositories (based on native XML or relational database architectures) are built that contain collections of XML documents and data. The repository data is made available to client applications, where clients request documents or data that satisfy specified conditions based on path or tree pattern queries.

Analogous to the publication of data residing in relational databases, the integrity, authenticity, and non-repudiation of XML repository data are important requirements clients would like to see satisfied by the repository's owner when querying the repository. To address this issue in cases where XML repository owners are unable or unwilling to provide clients with a respective computing infrastructure, we extend the authentic publication scheme introduced in the previous section to XML repositories. In the following, we first outline the XML data model and query approach underlying our scheme. We then illustrate how digests for an XML repository are constructed by the owner, and how verification objects for query results are computed by repository publishers. More details and different uses of this approach are given in [2, 4, 9].

3.1 XML Data Model and Path Queries

We assume a general XML data model in which XML data is represented as an ordered, node-labeled tree. Nodes in the tree are labeled with element names and leaf nodes carry text data. To motivate the basic idea of our approach, in the following, we do not consider attributes, comments, entities, and processing instructions. We assume that a complete XML repository is modeled as a single tree and that a schema in the form of a Document Type Definition (DTD).

We use simple path queries to query an XML repository. The language underlying these queries is a subset of XPath [15]. It supports the child axis ("/"), descendant axis ("//"), wildcard ("$*$"), and conditions on text values of selection nodes in a query. We currently do not consider tree pattern queries, i.e., queries that include branching ("[]"). This is not a limitation of the approach but is related to scalability issues since the number of possible path queries against an XML repository is much smaller than the number of possible tree pattern queries.

3.2 XML Repository Digests and Certified Query Results

In order for a data owner to employ our authentic publication scheme, a model is needed that allows the computation of digest values for XML repository data. The data publisher maintaining a copy of the repository then uses the same model to compute verification objects for answers to path queries posed by clients. There are several approaches to hash and sign XML data, such as DOMHASH [6] and the XML Digital Signature Recommendation [14], respectively. All these models provide a means to perform a recursive, bottom-up hashing over tree structured data. In particular, the XML Digital Signature scheme describes a single signature over an entire XML repository to validate the authenticity of any subtree in the repository. Although in principle this approach can be used by a publisher to prove to a client that every subtree returned as part of a query result is in fact in the XML repository, it does not provide a means to prove that all qualified subtrees are included in an answer. Our approach eliminates this limitation.

We start with a solution to compute digests and verification objects for an XML repository and query results, respectively. Again, our aim is to find a Merkle-Hash Tree-like structure on data objects that supports these types of computations. Suppose a DTD D is associated with the owner's XML repository. We assume that D is non-recursive (although this assumption can be weakened by considering a fixed recursion depth). Based on D, a so-called *XTrie* is computed. An XTrie is a rooted, node-labeled tree, and it enumerates all admissible rooted paths, i.e., sequences of element names, in a repository.

Given the owner's XML repository R conforming to the DTD D and a node i in the Xtrie, one can identify all subtrees in R whose rooted path corresponds to the path leading to node i. Let $s_1, s_2, \ldots, s_n$ denote the subtrees in R, in repository order, that are rooted at node i. Each subtree s_i is hashed using a secure hash-function (see, e.g., DOMHASH [6]); let $h(s_i)$ denote the computed hash value. The hash values $h(s_1), h(s_2), \ldots, h(s_n)$ now build the leaf nodes of a Merkle-Hash Tree introduced in Section 2.1. Thus, a digest for the n subtree structures is determined and associated with the node i in the Xtrie. This procedure is repeated for every node in the Xtrie. Thus, for each node in the Xtrie, a digest is computed based on subtree structures in the repository R that are rooted at that node.

Eventually, an Xtrie is obtained where each node carries a digest of all subtree structures in the XML repository that are rooted at that node. Such an XTrie, although not a binary tree (in the Xtrie, a node can have more or less than two children), can be Merkle-hashed bottom-up, resulting in a digest for all subtree structures in the repository. This digest is securely distributed to clients by the repository owner. The final step for the repository owner is to provide a publisher with an exact copy of the XML repository.

We now consider the processing of client queries and the computation of verification objects at the publisher site. We assume that clients know the Xtrie structure, for example, as schema for formulating queries (note that an Xtrie is less expressive than a DTD). A path query submitted to a publisher is processed against the Xtrie to find the root of subtrees whose path matches the path query. With each such path query, zero, one or more nodes in the Xtrie can be associated, depending on the pattern used in the path query. Assume one node in the Xtrie matches the query. As answer to the query, the publisher will provide the client with the sequence of subtrees from the XML repository whose root node matches the query in the Xtrie. Furthermore, the publisher computes a verification object that comprises the following components: (1) the rooted path of the node matching the query, and (2) the hash path for that node (see Definition 1). The client uses this information to recompute a digest. If it matches the digest previously distributed by the owner, the client can be assured that the publisher provided all matching subtrees that can be found in the XML repository and no subtree has been left out from the result.

The above scheme easily extends to cases where more than one node in the Xtrie matches the path query or no node matches the query. Note that the client, having information about the Xtrie, can easily verify how many matches there have to be and, thus, how many VOs the publisher has to provide. In both scenarios, boundary case VO, as illustrated in the previous section, are computed by the publisher. An interesting feature of the authentic publication scheme for XML repositories is that basically only one digest is necessary, namely the digest over the complete XML repository as described above. Recall that in the case of relational databases, digests are fairly specific to the types of queries issued by clients and therefore require more digests and query specific Merkle-Hash Tree structures for tuple data. The approach for computing a digest for an XML repository and verification objects associated with answers to path queries discussed above does not address path queries that contain condition on text values and text nodes (leaves) of an XML repository. A respective extension to the above approach as well as complexity results for the authentic publication of XML data are given in [3].

4 Conclusions and Future Work

Authentic publication schemes provide data owners with an effective means to employ possibly untrusted data publishers to answer queries from clients on behalf of the data owner. The schemes outlined in this article focus on two popular types of databases as they can be found in practice, relational databases and XML repositories. The proposed schemes require only a minor effort by the data owner for computing digests on the data objects managed in the database and periodically distributing these digests to clients interested in the owner's data. In particular, our schemes satisfy important security requirements, namely that a client can verify – based on verification objects associated with query answers – that the query result computed by a data publisher is the same as the owner would have provided. The techniques for computing digests and verification objects, of course, add to the complexity of evaluating queries. However, since we assume mostly static data, i.e., data that is only updated once week or month, this is not disadvantage, in particular since the owner (with constrained resources) has to compute digests only periodically.

More general approaches for authenticating data structures than those presented in this article have been proposed by Goodrich et al. [8] and Martel et al. [11]. Another extension allows multiple owners to jointly certify a digest for a data set that combines their individual data sets [12]. We are currently studying several alternatives for implementing our authentication schemes, with a primary focus on relational databases. Using the GiST package [7], we are evaluating different types of index structures that can be used for both answering queries and "encoding" Merkle-Hash Tree structures and hash values in particular. Our focus is also on developing lightweight software components clients can plug into their applications to automatically perform the verification of VOs against digests distributed by the data owner. The purpose of these components is to make the verification process for queries fully transparent to applications querying a publisher's database.

References

[1] de Berg, M., Schwarzkopf, O., van Kreveld, M., Overmars, M.: *Computational Geometry: Algorithms and Applications*. Springer, 2000.

[2] Devanbu, P., Gertz, M., Kwong, A., Martel, C., Stubblebine, S.: Flexible Authentication of XML Documents. To appear in *Journal of Computer Security*.

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

[4] Devanbu, P., Gertz, M., Kwong, A., Martel, C., Stubblebine, S.: Flexible Authentication of XML Documents. In *Proc. of the 8th International ACM Conference on Computer and Communications Security (CCS-8)*, ACM, 136–145, 2001.

[5] Devanbu, P., Gertz, M., Martel, S., Stubblebine S.: Authentic Third-party Data Publication. In *14th IFIP 11.3 Working Conference in Database Security (DBSec'00)*, Kluwer Academic Publishers, 101–112, 2000.

[6] Maruyama, H., Tamura, K., Uramoto, N.: Digest Values for DOM (DOMHASH). RFC2803, www.faqs.ord/rfcs/rfc2803.html, April 2000.

[7] The GiST Indexing Project. gist.cs.berkeley.edu.

[8] Goodrich, M.T., Tamassia, R., Triandopoulos, N., Cohen, R.: Authenticated Data Structures for Graphs and Geometric Searching. In *Topics in Cryptography CT-RSA 2003*, LNCS 2612, 295–213, 2003.

[9] Kwong, A., Gertz, M.: Authentic Publication of XML Document Data. In *Proc. of the 2nd International Conference on Web Information Systems Engineering (WISE)*, IEEE Computer Society, 331–340, 2001.

[10] Merkle, A.C.: A Certified Digital Signature. In *Advances in Cryptology – CRYPTO '89, 9th Annual International Cryptology Conference*, LNCS 435, Springer, 218-238, 1989.

[11] Martel, C., Nuckolls, G., Devanbu, P., Gertz, M., Kwong, A., Stubblebine, S.: A General Model for Authentic Data Publication. To appear in Algorithmica (Springer).

[12] Nuckolls, G., Martel, C., Stubblebine, S.: Certifying Data from Multiple Sources. In *Proceedings of the 17th IFIP WG 11.3 Working Conference on Database and Applications Security*, 2003.

[13] Naor, N., Nissim, K.: Certificate Revocation and Certificate Update. In *Proceedings of the 7th USENIX Security Symposium*, 1998.

[14] XML-Signature Syntax and Processing. W3C Recommendation, www.w3c.org/Signature, February 2002.

[15] XML Path Language (XPath) 2.0. W3C Working Draft, www.w3c.org/TR/xpath20/, Nov. 2003