Providing Authentication and Integrity in Outsourced Databases using Merkle Hash Tree's

Einar Mykletun
University of California Irvine
mykletun@ics.uci.edu

Maithili Narasimha
University of California Irvine
mnarasim@ics.uci.edu

Gene Tsudik
University of California Irvine
gts@ics.uci.edu

1 Introduction

In this paper we intend to describe and summarize results relating to the use of authenticated data structures, specifically Merkle Hash Tree's, in the context of the Outsourced Database Model (ODB). In ODB, organizations outsource their data management needs to an external untrusted service provider. The service provider hosts clients' databases and offers seamless mechanisms to create, store, update and access (query) their databases. Due to the service provider being untrusted, it becomes imperative to provide means to ensure authenticity and integrity in the query replies returned by the provider to clients. It is therefore the goal of this paper to outline an applicable data structure that can support authenticated query replies from the service provider to the clients (who issue the queries). Merkle Hash Tree's (introduced in section 3) is such a data structure.

1.1 Outsourced Database Model (ODB)

The outsourced database model (ODB) is an example of the well-known Client-Server model. In ODB, the *Database Service Provider* (also referred to as the Server) has the infrastructure required to host outsourced databases and provides efficient mechanisms for clients to create, store, update and query the database. The owner of the data is identified as the entity who has to access to add, remove, and modify tuples in the database. The owner and the client may or may not be the same entity. Figure 1 depicts one possible ODB setting.

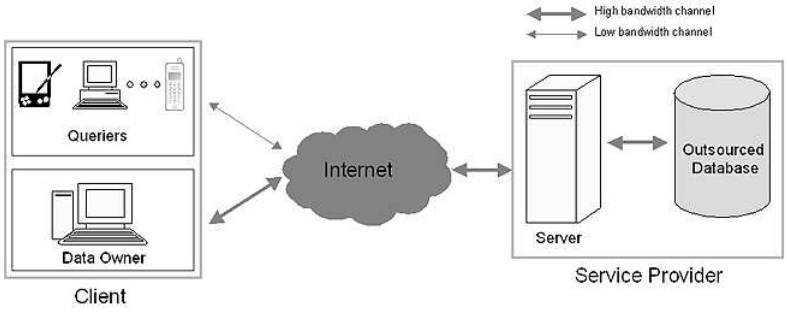

Figure 1. ODB Model

An ODB client is assumed to trust the server to faithfully maintain its data. Specifically, the server is relied upon for the replication, backup and availability of outsourced databases it stores. However, the server might not be trusted with the actual database contents and/or the integrity of these contents. There has been some work involved in maintaining the privacy of the outsourced data with respect to the server [1, 2]. However, the focus of this paper is on providing data authentication and integrity in the ODB model using Merkle Hash Trees.

In ODB, a client will issue a query to a server who will return the appropriate set of tuples (if any). The client then wishes to have a mechanism such as to verify the authenticity and integrity of this set, namely, whether the tuples originated from the data owner and if they have been modified in any way.

1.2 Organization of Paper

The remainder of this paper is organized as follows. Section 2 discusses work related to authenticated data structures. In section 3 we describe Merkle Hash Tree's and their application in the ODB Model. The use of B-trees as the underlying data structure for implementing Merkle Hash Tree's is discussed in section 4, before some performance results are given (section 5).

2 Related Work

A general description of authenticated data structures can be found in [3]. The authors define and create Search Directed Acyclic Graphs (Search DAG's) that model underlying data structures such as binary search trees, B-trees, skip lists [4], range trees etc. Using these Search DAG's, servers can answer queries by creating and returning verification objects. Equipped with verification objects, users are able to verify the authenticity of their received responses. Any data structure that can be modeled as a Search DAG can in turn be converted to an authenticated data structure. The use of binary trees as authenticated data structures has been studied in detail, specifically for the purpose of certificate revocation [5]. This work was later extended upon in [6] where it was proposed to use 2-3 tress instead of binary trees.

3 Using Merkle Tree's to provide authentication and integrity in ODB

We begin by describing how binary search trees can be used to construct authenticated dictionary structures and then explain how they can be applied to the ODB model.

3.1 Introduction to Merkle Hash Tree's

The following outlines a procedure for constructing an authenticated binary search tree based on a set of ordered values $x_1, x_2, ..., x_n$. Begin by building a tree in which the leaves correspond to the ordered elements in the set, and have each leaf node contain the hash value of its element. Therefore, a leaf associated with element x_i will contain the value $h(x_i)$, where $h()$ is a cryptographic one-way hash function, such as MD5 or SHA-1. Proceed up to the next level in the tree, creating internal nodes which correspond to the hash value of the concatenation of its children (maintaining their order). An internal node whose children are v_1 and v_2 will therefore have the value $h(v_1||v_2)$. Continue this process of building the higher levels in the tree until the root has been formed. Finally, the root node is digitally signed.

The above construction is due to Merkle [7] and is referred to as a Merkle Hash Tree (MHT). MHT's were initially used for the purpose of one-time signatures and authenticated public key distribution, namely providing authenticated responses as to the validity of a certificate. In a certificate revocation tree [5], the leaves correspond to ranges of unrevoked certificates. The tree is constructed and signed by the certification authority and then distributed to untrusted directory services. Entities wishing to verify the validity of a certificate can query such

a directory service and have confidence in the correctness of a response by using the returned data to verify the certificate authority's signature. The main advantage associated with the use of MHT's in revocation schemes it that of a short proof, in turn resulting in a low communication overhead between the clients and directory services. However, the costs associated with updating the tree are rather costly. MHT's have since been applied to outsourced applications [8], [9], among others.

3.2 Querying a Merkle Hash Tree

A MHT can be used to prove the existence of an element in the set stored among the leaf nodes, and we demonstrate this through an example involving an outsourced database. Remember that clients issue queries and receive as query replies a set of tuples, or parts of tuples (selected attributes), that they wish to authenticate. For notation purposes we will define n to be the number of values in the database and t to be the number of tuples in the result set returned to clients. We assume that a data owner has built an MHT for a relation, where the records have been sorted based on one of the relations's attribute values. The root has been signed by the owner and the MHT given to the untrusted outsourced database server. A client then wishes to query the server about the existence of a particular attribute value *v*. If *v* is present in the tree it will be represented by one of the leaves and the server will return the nodes on the *co-path* from the specific leaf node up to the root. The nodes on the co-path are defined as those needed to enable the client to recompute the root of the tree, such as to then verify its signature (which was generated by the data owner). In the event that the signature is valid, the client can be assured of that the response returned by the server is correct, and therefore that the attribute value indeed is present.

As an example we refer to figure 2. A client queries about attribute value 5, and the server responds with node values $3, 5, h_{34}$, of which 3 and h_{34} correspond to the co-path, as well as the root nodes signature. This permits the client to recompute the root in the following manner. It constructs $h'_1 = h(3)$ and $h'_2 = h(5)$ such as to create $h'_{12} = h(h'_1||h'_2)$. The root is finally computed and its signature is verified. Note that information pertaining to the tree structure needs to be transmitted to the client, such as to allow him to recreate the tree nodes correctly.

We have described the traversal of the MHT in a bottom-up approach. However, it is possible to allow for a top-down approach, which has the advantage of allowing the client to simulate a search in the tree and thereby avoid the need of transferring any structural information related to the reconstruction of the root. To accomplish this, some additional information needs to be present in each internal node, for example, the largest value in its left subtree, such as to help guide the search. The reader is encouraged to read [3] for a detailed description of a top-down approach.

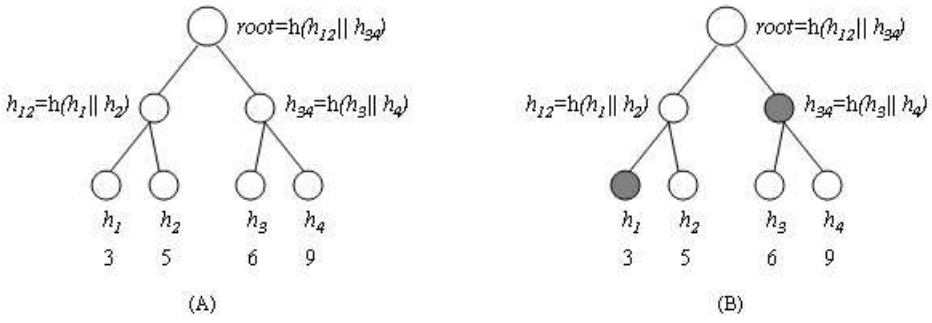

Figure 2. Example of a Merkle Tree (A) and the nodes included in the co-path (dark nodes) returned in a query response to attribute value 5 (B)

3.3 Merkle Hash Tree's in ODB

Although the above example served as a nice introduction to the protocol involved when querying a MHT, the operation of selecting a single record (a single leaf in the MHT tree) is not very common in databases. Typically, some sort of a range query is performed, involving the selection of consecutive leaves in the tree, and we now describe how such a query would be handled by an MHT. We can identify two boundary nodes pertaining to the range query, namely the left-most (minimum value) and right-most (maximum value) nodes contained in the set queried upon. A querier in possession of these two nodes, as well as all intermediary leaf nodes in between, is able to reconstruct a subtree. Therefore, none of the nodes in this subtree will be included in the set of co-path nodes returned by the server. In fact, only the nodes needed to recompute the path from the subtree's root up to the signed root, in addition to the co-path's of the two boundary nodes, need to be transmitted for verification purposes. If the height of the tree and subtree are h and h' respectively, then $h - h' + 2 * \log(n)$ nodes need to be returned.

3.4 Empty proofs

Another nice feature of MHT's is the ability to give empty proofs, namely, prove that a value is non-existent in the tree, i.e., that a record is non-existent in the database. This is accomplished by returning the co-paths for the two neighbor leaves covering the range of the selected record queried upon. As an example, assume a client queried about the attribute value 4 in figure 2. The server could then return co-path's for 3 and 5 which would inform the client of that no nodes representing values between 3 and 5 are present in the tree. Specifically, the co-path nodes returned by the server would be h_{12} and h_{34}.

3.5 Completeness of query replies

Combining the technique of providing proofs together with the MHT's range query abilities provides the client with query completeness. Query completeness refers to that all records in the database have been inspected by the server such as to determine if they match the query predicates. By providing query completeness, the clients are protected against a "lazy server" - a server that only performs an incomplete scan of the database, thereby potentially omitting valid records from the query reply.

Query completeness is achieved by including empty proofs for the leaf nodes neighboring the boundary leaves selected by the range query. Therefore, these "outside" nodes represent the database records which are the immediate neighbors of the border records selected by the query predicates. By including these (two) empty proofs and using the fact that the leaves are sorted according to the attribute queried upon, the client can be assured of that no records have been omitted from the query reply. Figure 3 shows an example of providing completeness in a query reply.

3.6 Constructing and Updating a MHT in ODB

One way to construct an MHT is for the owner to sort the attribute values, generate the tree, sign the root, and then send the attribute values together with the signature to the server. The server in turn sorts the attribute values, constructs the tree, and then verifies the signature of the root. Any later modifications to the tree can be carried out incrementally. Note that an incremental change can not be carried out by having the server simply modify the tree and send the new root to the owner as to request for an updated signature. If this were done, the owner would have no way of ensuring that the server is operating on the latest version of the tree[1]. To solve this problem, the

[1]Note that if the owner were to have a local copy of the MHT, then it could simulate the update and transmit the updated signature to the server. However, since we are operating with an outsourced application, we can assume that the server either does not have or wishes not to use its resources to maintain such a structure as a large MHT (on the order of the size of the database being outsourced)

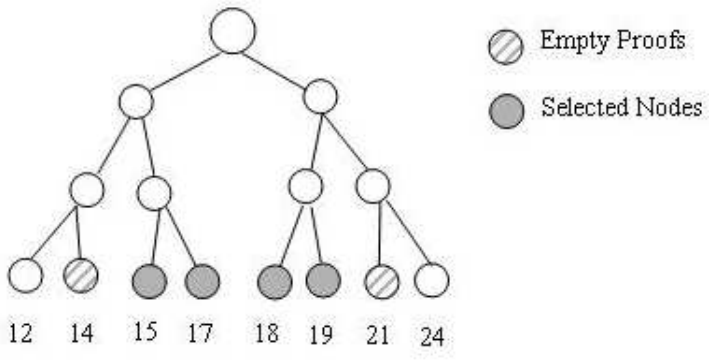

Figure 3. Providing completeness of query replies. Depicted is the result of a query *Select * where 15 $\leq$ attribute $\leq$ 20.* Empty proofs are returned for boundary nodes 14 and 21.

owner simply needs to keep state of the MHT, without storing the entire tree. In fact, it only needs to keep state of the latest generated root signature. When a modification to the MHT is needed, say a new value is to be inserted, the server returns to the owner the co-path of the leaf node where the new value is to be inserted. This allows the owner to recompute the nodes from this leaf node up to the root, and then generate a new signature. This signature is returned to the server which inserts the new leaf node and updates the MHT as necessary. Enabling the owner to recompute the current root allows him to verify that the latest tree is in fact being used. Note that the clients querying the server have no way of ensuring themselves of this, unless some form of time stamping is provided.

3.7 Performance of Merkle Tree's in ODB

We wish to measure the number of nodes sent from the server to the client (communication overhead) and the computation effort performed by the client to verify the query response. We will assume that the client has issued a range query and that the server will return (in addition to the actual tuples, which we do not count) the co-paths for the 2 boundary nodes, such as to allow for query completeness. This results in 2*$logn+b$ nodes being transmitted, where b represents the co-path from the selected subtree's root, to the tree's root. Upon receipt of these nodes, the client needs to reconstruct and verify the root, which will require 2*$logn + b$ hashes, and 1 signature verification.

3.8 Usefulness of Merkle Tree's in ODB

MHT's are best suited for range queries as only two full co-paths need to be returned from the server to the client (see section 3.3). The communication overhead increases for queries that select values not stored in consecutive leaf nodes. The worst case query, in terms of the bandwidth overhead associated with a MHT, is one in which every other record in the tree is selected. In general, selection queries that don't select ranges of records incur a large communication overhead. Another factor is the storage overhead associated with the MHT's at the server side. One tree needs to be built for each attribute to be queried upon. Each such tree is linear in the size of the table, i.e., the number of records in the table.

4 Using B-tree's to construct MHT's

Describing the concept of MHT's in terms of binary trees has the advantages of simplicity and clarity. However, the performance associated with the use of binary trees depends upon the size and dynamics of the underlying data

set. For data sets that fit into main memory, binary trees serve as an efficient data structure. However, once the data set is large enough to require secondary storage, binary trees incur large disk I/O overhead costs, especially when the data is dynamic (updates are frequent). B-tree's are the preferred data structure for these types of data sets. An example would be the searchable index structure used by databases which needs to be stored on disk due to its size. The advantage of using a B-tree is that very few disk accesses are needed during traversal of the tree. This is due to its large spanning factor which in turn reduces the height of the tree. Because the data sets in ODB that we wish to authenticate will typically require secondary storage, we will prefer to use B-tree's as the underlying data structure for MHT's.

4.1 B-trees in ODB

The protocol for constructing and using MHT's remains the same regardless of whether B-trees or binary trees are used as the underlying data structures. The leaf nodes are the hash values of the elements associated with them. Internal nodes are hashes of the concatenation of its children (up to B-1), and the root node is signed as usual. When providing the co-path from a leaf node to the root, it is necessary to include all siblings of the leaf node and all children of internal nodes. With a large spanning factor this can amount to a large amount of data that needs to be communicated from the server to the client. The number of nodes in the co-path for one leaf node would be $Blog_B\ n$, where n is the number of leaves in tree.

A solution which reduces this communication overhead is to replace each B-tree node by a BST of height $logB$ [3]. What this accomplishes is to reduce the size of the response from the server to the client. The number of nodes in a co-path would then reduce to $logB\ log_B\ n$, where $logB$ corresponds to the height of each BST representing the B-tree nodes, and $log_B\ n$ is the height of the B-tree.

5 Performance

We will summarize the performance of MHT's when implemented with both binary trees and with the improved B-trees. The two overhead factors we wish to reduce are those of querier computation and querier bandwidth. Querier computation refers to the amount of work required by the client to verify the integrity of the returned results. This will be measured in the number of hashes needed to be computed as well as digital signatures that have to be verified. Querier bandwidth is a measure of the amount of data needed to be transferred from the server to the client, in addition to the actual records, to enable the client to perform his integrity computations.

MHT Data Structure	Querier Computation	Querier Bandwidth
BST	2 $logn$ hashes + 1 sig	2 $logn$
B-tree	2 $logB\ log_B n$ hashes + 1 sig	2 $logB\ log_B n$

Table 1. Performance

The biggest difference between using B-tree's versus binary trees as the underlying data structure of MHT's, relates to the cost associated with tree updates. B-tree's were created as a secondary storage structure that would require few node modifications during updates. Binary trees on the other hand are best suited when the tree fits into main memory, such as to avoid expensive I/O costs (when the data set is stored on secondary storage). In addition to the I/O costs, the actual programming of the respective tree structures, including all operations, are considered to be simpler with a B-tree.

6 Conclusion

References

[1] H. Hacigümüş, B. Iyer, C. Li, and S. Mehrotra, "Executing sql over encrypted data in the database-service-provider model," in *ACM SIGMOD Conference on Management of Data*, pp. 216–227, ACM Press, June 2002.

[2] H. Hacigümüş, B. Iyer, and S. Mehrotra, "Providing database as a service," in *International Conference on Data Engineering*, March 2002.

[3] C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong, and S. G. Stubblebine, "A general model for authenticated data structures," 2001.

[4] M. T. Goodrich and R. Tamassia, "Efficient authenticated dictionaries with skip lists and commutative hashing," in *Technical Report, Johns Hopkins Information Security Institute*, 2000.

[5] P. C. Kocher, "On certificate revocation and validation," in *Proc. International Conference on Financial Cryptography*, vol. 1465, 1998.

[6] M. Naor and K. Nissim, "Certificate revocation and certificate update," in *Proceedings of the 7th USENIX Security Symposium (SECURITY -98)*, pp. 217–228, 1998.

[7] R. C. Merkle, "Protocols for public key cryptosystems," in *IEEE Symposium on Research in Security and Privacy*, 1980.

[8] P. Devanbu, M. Gertz, C. Martel, and S. G. Stubblebine, "Authentic third-party data publication," in *14th IFIP 11.3 Working Conference in Database Security*, pp. 101–112, 2000.

[9] S. Haber and W. S. Stornetta, "How to timestamp a digital document," in *Journal of Cryptology*, pp. 99–111, 1991.