5

Authenticated Index Structures for Outsourced Databases

Feifei Li[1], Marios Hadjieftheriou[2], George Kollios[3], and Leonid Reyzin[3]

[1] Department of Computer Science
Florida State University
lifeifei@cs.fsu.edu
[2] AT&T Labs Inc.
marioh@research.att.com
[3] Computer Science Department
Boston University
gkollios@cs.bu.edu,reyzin@cs.bu.edu

Summary. In an outsourced database (ODB) system the database owner publishes data through a number of remote servers, with the goal of enabling clients at the edge of the network to access and query the data more efficiently. As servers might be untrusted or can be compromised, *query authentication* becomes an essential component of ODB systems. In this chapter we present three techniques to authenticate selection range queries and we analyze their performance over different cost metrics. In addition, we discuss extensions to other query types.

1 Introduction

Today, there is a large number of corporations that use electronic commerce as their primary means of conducting business. As the number of customers using the Internet for acquiring services increases, the demand for providing fast, reliable and secure transactions increases accordingly — most of the times beyond the capacity of individual businesses to provide the level of service required, given the overwhelming data management and information processing costs involved.

Increased demand has fueled a trend towards outsourcing data management and information processing needs to third-party service providers in order to mitigate the in-house cost of furnishing online services [1]. In this model the third-party service provider is responsible for offering the necessary resources and mechanisms for efficiently managing and accessing the outsourced data, by data owners and customers respectively. Clearly, data outsourcing intrinsically raises issues related with trust. Service providers cannot always be trusted (they might have malicious intend), might be compromised (by other parties with malicious intend) or run faulty software (unintentional

errors). Hence, this model raises important issues on how to guarantee quality of service in untrusted database management environments, which translates into providing verification proofs to both data owners and clients that the information they process is correct.

Three main entities exist in the ODB model as discussed so far: the data owner, the database service provider (a.k.a. server) and the client. In practice, there is a single or a few data owners, a few servers, and many clients. The data owners create their databases, along with the necessary index and authentication structures, and upload them to the servers. The clients issue queries about the owner's data through the servers, which use the authentication structures to provide provably correct answers. It is assumed that the data owners may update their databases periodically and, hence, authentication techniques should be able to support dynamic updates. In this setting, query authentication has three important dimensions: *correctness*, *completeness* and *freshness*. Correctness means that the client must be able to validate that the returned answers truly exist in the owner's database and have not been tampered with. Completeness means that no answers have been omitted from the result. Finally, freshness means that the results are based on the most current version of the database, that incorporates the latest owner updates. It should be stressed here that result freshness is an important dimension of query authentication that is directly related to incorporating dynamic updates into the ODB model.

There are a number of important costs pertaining to the aforementioned model, relating to the database construction, querying, and updating phases. In particular, in this chapter the following metrics are considered: 1. The computation overhead for the owner, 2. The owner-server communication cost, 3. The storage overhead for the server, 4. The computation overhead for the server, 5. The client-server communication cost, and 6. The computation cost for the client (for verification).

It should be pointed out that there are other important security issues in ODB systems that are orthogonal to the problems considered here. Examples include privacy-preservation issues [2, 3, 4], secure query execution [5], security in conjunction with access control requirements [6, 7, 8, 9] and query execution assurance [10]. Aslo, we concentrate on large databases that need to be stored on external memory. Therefore, we will not discuss main memory structures [11, 12, 13] or data stream authentication [14, 15].

2 Cryptographic Background

In this section we discuss some basic cryptographic tools. These tools are essential components of the authentication data structures that we discuss later.

2.1 Collision-resistant hash functions.

For our purposes, a hash function $\mathcal{H}$ is an efficiently computable function that takes a variable-length input x to a fixed-length output $y = \mathcal{H}(x)$. *Collision resistance* states that it is computationally infeasible to find two inputs $x_1 \neq x_2$ such that $\mathcal{H}(x_1) = \mathcal{H}(x_2)$. Collision-resistant hash functions can be built provably based on various cryptographic assumptions, such as hardness of discrete logarithms [16]. However, we concentrate on using heuristic hash functions that have the advantage of being very fast to evaluate. Specifically we focus on SHA-1 [17], which takes variable-length inputs to 160-bit (20-byte) outputs. SHA-1 is currently considered collision-resistant in practice, despite some recent successful attacks [18, 19]. We also note that any eventual replacement to SHA-1 developed by the cryptographic community can be used instead of SHA-1.

2.2 Public-key digital signature schemes.

A public-key digital signature scheme, formally defined in [20], is a tool for authenticating the integrity and ownership of the signed message. In such a scheme, the signer generates a pair of keys (SK, PK), keeps the secret key SK secret, and publishes the public key PK associated with her identity. Subsequently, for any message m that she sends, a signature s_m is produced by $s_m = \mathcal{S}(SK, m)$. The recipient of s_m and m can verify s_m via $\mathcal{V}(PK, m, s_m)$ that outputs "valid" or "invalid." A valid signature on a message assures the recipient that the owner of the secret key intended to authenticate the message, and that the message has not been changed. The most commonly used public digital signature scheme is RSA [21]. Existing solutions [9, 22, 23, 24] for the query authentication problem chose to use this scheme, hence we adopt the common 1024-bit (128-byte) RSA here. Its signing and verification cost is one hash computation and one modular exponentiation with 1024-bit modulus and exponent.

2.3 A Signature Aggregation Scheme.

In the case when t signatures $s_1, \ldots, s_t$ on t messages $m_1, \ldots, m_t$ signed by the same signer need to be verified all at once, certain signature schemes allow for more efficient communication and verification than t individual signatures. Namely, for RSA it is possible to combine the t signatures into a single aggregated signature $s_{1,t}$ that has the same size as an individual signature and that can be verified (almost) as fast as an individual signature. This technique is called Condensed-RSA [25]. The combining operation can be done by anyone, as it does not require knowledge of SK; moreover, the security of the combined signature is the same as the security of individual signatures. In particular, aggregation of t RSA signatures can be done at the cost of $t-1$ modular multiplications, and verification can be performed at the cost of $t-1$

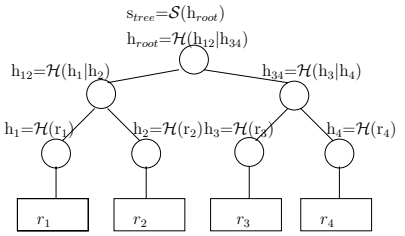

Fig. 1. Example of a Merkle hash tree.

multiplications, t hashing operations, and one modular exponentiation (thus, the computational gain is that $t-1$ modular exponentiations are replaced by modular multiplications). Note that aggregating signatures is possible only for some digital signature schemes.

2.4 The Merkle Hash Tree.

The straightforward solution for verifying a set of n values is to generate n digital signatures, one for each value. An improvement on this straightforward solution is the Merkle hash tree (see Figure 1), first proposed by [26]. It solves the simplest form of the query authentication problem for point queries and datasets that can fit in main memory. The Merkle hash tree is a binary tree, where each leaf contains the hash of a data value, and each internal node contains the hash of the concatenation of its two children. Verification of data values is based on the fact that the hash value of the root of the tree is authentically published (authenticity can be established by a digital signature). To prove the authenticity of any data value, all the prover has to do is to provide the verifier, in addition to the data value itself, with the values stored in the siblings of the path that leads from the root of the tree to that value. The verifier, by iteratively computing all the appropriate hashes up the tree, at the end can simply check if the hash she has computed for the root matches the authentically published value. The security of the Merkle hash tree is based on the collision-resistance of the hash function used: it is computationally infeasible for a malicious prover to fake a data value, since this would require finding a hash collision somewhere in the tree (because the root remains the same and the leaf is different—hence, there must be a collision somewhere in between). Thus, the authenticity of any one of n data values can be proven at the cost of providing and computing $\log_2 n$ hash values, which is generally much cheaper than storing and verifying one digital signature per data value. Furthermore, the relative position (leaf number) of any of the data values within the tree is authenticated along with the value itself. Finally, in [27] this idea is extended to dynamic environments, by dynamizing the binary search tree using 2-3 trees. Thus, insertions and deletions can be handled efficiently by the Merkle hash tree.

Table 1. Notation used.

Symbol	Description
r	A database record
k	A B^+-tree key
p	A B^+-tree pointer
h	A hash value
s	A signature
$\|x\|$	Size of object x
N_D	Total number of database records
N_R	Total number of query results
P	Page size
f_x	Node fanout of structure x
d_x	Height of structure x
$\mathcal{H}_l(x)$	A hash operation on input x of length l
$\mathcal{S}_l(x)$	A signing operation on input x of length l
$\mathcal{V}_l(x)$	A verifying operation on input x of length l
$\mathcal{C}_x$	Cost of operation x
$\mathcal{VO}$	The verification object

3 Authenticated Index Structures for Selection Queries

Existing solutions for the query authentication problem work as follows. The data owner creates a specialized authenticated data structure that captures the original database and uploads it at the servers together with the database itself. The structure is used by the servers to provide a verification object $\mathcal{VO}$, along with every query answer, which clients can use for authenticating the results. Verification usually occurs by means of using collision-resistant hash functions and digital signature schemes. Note that in any solution, some information that is known to be authentically published by the owner must be made available to the client directly; otherwise, from the client's point of view, the owner cannot be differentiated from any other potentially malicious entity. For example, this information could be the owner's public key of any public signature scheme. For any authentication method to be successful it must be computationally infeasible for a malicious server to produce an incorrect query answer along with a verification object that will be accepted by a client that holds the correct authentication information of the owner.

Next, we illustrate three approaches for query correctness and completeness for selection queries on a single attribute: a signature-based approach similar to the ones described in [9, 24], a Merkle-tree-like approach based on the ideas presented in [28], and an improved embedded tree approach [29]. We present them for the *static scenario* where no data updates occur between the owner and the servers on the outsourced database. We also present *analytical cost models* for all techniques, given a variety of performance metrics.

In particular, we provide models for the *storage*, *construction*, *query*, and *authentication* cost of each technique, taking into account the overhead of hashing, signing, verifying data, and performing expensive computations (like modular multiplications of large numbers). The analysis considers range queries on a specific database attribute A indexed by a B^+-tree [30]. The size of the structure is important first for quantifying the storage overhead on the servers, and second for possibly quantifying the owner/server communication cost. The construction cost is useful for quantifying the overhead incurred by the database owner for outsourcing the data. The query cost quantifies the incurred server cost for answering client queries, and hence the potential query throughput. The authentication cost quantifies the server/client communication cost and, in addition, the client side incurred cost for verifying the query results. The notation used is summarized in Table 1. In the rest, for ease of exposition, it is assumed that all structures are bulk-loaded in a bottom-up fashion and that all index nodes are completely full. Extensions for supporting multiple selection attributes are discussed in Section 6.

Aggregated Signatures with B$^+$-trees

The first authenticated data structure for static environments is a direct extension of aggregated signatures and ideas that appeared in [24, 9]. To guarantee correctness and completeness the following technique can be used: First, the owner individually hashes and signs all consecutive pairs of tuples in the database, assuming some sorted order on a given attribute A. For example, given two consecutive tuples r_i, r_j the owner transmits to the servers the pair (r_i, s_i), where $s_i = \mathcal{S}(r_i|r_j)$ ('|' denotes some canonical pairing of strings that can be uniquely parsed back into its two components; e.g., simple string concatenation if the lengths are fixed). The first and last tuples can be paired with special marker records. Chaining tuples in this way will enable the clients to verify that no in-between tuples have been dropped from the results or modified in any way. An example of this scheme is shown in Figure 2.

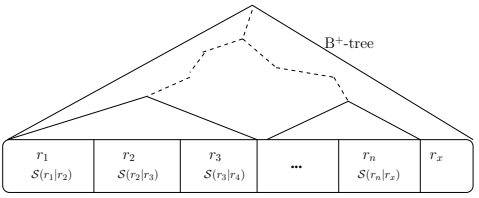

Fig. 2. The signature-based approach.

In order to speed up query execution on the server side a B^+-tree is constructed on top of attribute A. To answer a query the server constructs a $\mathcal{VO}$ that contains one pair $r_q|s_q$ per query result. In addition, one tuple to the left of the lower-bound of the query results and one to the right of the upper-bound

is returned, in order for the client to be able to guarantee that no boundary results have been dropped. Notice that since our completeness requirements are less stringent than those of [9] (where they assume that database access permissions restrict which tuples the database can expose to the user), for fairness we have simplified the query algorithm substantially here.

There are two obvious and serious drawbacks associated with this approach. First, the extremely large $\mathcal{VO}$ size that contains a linear number of signatures w.r.t. N_R (the total number of query results), taking into account that signature sizes are very large. Second, the high verification cost for the clients. Authentication requires N_R verification operations which, as mentioned earlier, are very expensive. To solve this problem one can use the aggregated signature scheme discussed in Section 2.3. Instead of sending one signature per query result the server can send one *combined signature* s^{π} for all results, and the client can use an aggregate verification instead of individual verifications.

By using aggregated RSA signatures, the client can authenticate the results by hashing consecutive pairs of tuples in the result-set, and calculating the product $m^{\pi} = \prod_{\forall q} h_q \pmod{n}$ (where n is the RSA modulus from the public key of the owner). It is important to notice that both s^{π} and m^{π} require a linear number of modular multiplications (w.r.t. N_R). The cost models of the aggregated signature scheme for the metrics considered are as follows:

Node fanout:

The node fanout of the B^+-tree structure is:

$$f_a = \frac{P - |p|}{|k| + |p|} + 1\,. \tag{1}$$

where P is the disk page size, $|k|$ and $|p|$ are the sizes of a B^+-tree key and pointer respectively.

Storage cost:

The total size of the authenticated structure (excluding the database itself) is equal to the size of the B^+-tree plus the size of the signatures. For a total of N_D tuples the height of the tree is equal to $d_a = \log_{f_a} N_D$, consisting of $N_I = \frac{f_a^{d_a}-1}{f_a-1}$ $(= \sum_{i=0}^{d_a-1} f_a^i)$ nodes in total. Hence, the total storage cost is equal to:

$$\mathcal{C}_s^a = P \cdot \frac{f_a^{d_a} - 1}{f_a - 1} + N_D \cdot |s|. \tag{2}$$

The storage cost also reflects the initial communication cost between the owner and servers. Notice that the owner does not have to upload the B^+-tree to the servers, since the latter can rebuild it by themselves, which will reduce the owner/server communication cost but increase the computation cost at the servers. Nevertheless, the cost of sending the signatures cannot be avoided.

Construction cost:

The cost incurred by the owner for constructing the structure has three components: the signature computation cost, bulk-loading the B^+-tree, and the I/O cost for storing the structure. Since the signing operation is very expensive, it dominates the overall cost. Bulk-loading the B^+-tree in main memory is much less expensive and its cost can be omitted. Hence:

$$\mathcal{C}_c^a = N_D \cdot (\mathcal{C}_{\mathcal{H}_{|r|}} + \mathcal{C}_{\mathcal{S}_{2|h|}}) + \frac{\mathcal{C}_s^a}{P} \cdot \mathcal{C}_{IO}. \tag{3}$$

$\mathcal{VO}$ construction cost:

The cost of constructing the $\mathcal{VO}$ for a range query depends on the total disk I/O for traversing the B^+-tree and retrieving all necessary record/signature pairs, as well as on the computation cost of s^π. Assuming that the total number of leaf pages accessed is $N_Q = \frac{N_R}{f_a}$, the $\mathcal{VO}$ construction cost is:

$$\mathcal{C}_q^a = (N_Q + d_a - 1 + \frac{N_R \cdot |r|}{P} + \frac{N_R \cdot |s|}{P}) \cdot \mathcal{C}_{IO} + \mathcal{C}_{s^\pi}, \tag{4}$$

where the last term is the modular multiplication cost for computing the aggregated signature, which is linear to N_R. The I/O overhead for retrieving the signatures is also large.

Authentication cost:

The size of the $\mathcal{VO}$ is equal to the result-set size plus the size of one signature:

$$|\mathcal{VO}|^a = N_R \cdot |r| + |s|. \tag{5}$$

The cost of verifying the query results is dominated by the hash function computations and modular multiplications at the client:

$$\mathcal{C}_v^a = N_R \cdot \mathcal{C}_{\mathcal{H}_{|r|}} + \mathcal{C}_{m^\pi} + \mathcal{C}_{\mathcal{V}_{|n|}}, \tag{6}$$

where the modular multiplication cost for computing the aggregated hash value is linear to the result-set size N_R, and the size of the final product has length in the order of $|n|$ (the RSA modulus). The final term is the cost of verifying the product using s^π and the owner's public key.

It becomes obvious now that one advantage of the aggregated signature scheme is that it features small $\mathcal{VO}$ sizes and hence small client/server communication cost. On the other hand it has the following serious drawbacks: 1. Large storage overhead on the servers, dominated by the large signature sizes, 2. Large communication overhead between the owners and the servers that cannot be reduced, 3. A very high initial construction cost, dominated by the cost of computing the signatures, 4. Added I/O cost for retrieving signatures, linear to N_R, 5. An added modular multiplication cost, linear to the result-set

size, for constructing the $\mathcal{VO}$ and authenticating the results, 6. The requirement for a public key signature scheme that supports aggregated signatures. For the rest of the chapter, this approach is denoted as Aggregated Signatures with B^+-trees (ASB-tree). The ASB-tree has been generalized to work with multi-dimensional selection queries in [24, 31].

The Merkle B-tree

Motivated by the drawbacks of the ASB-tree, we present a different approach for building authenticated structures that is based on the general ideas of [28] (which utilize the Merkle hash tree) applied in our case on a B^+-tree structure. We term this structure the Merkle B-tree (MB-tree).

As already explained in Section 2.4, the Merkle hash tree uses a hierarchical hashing scheme in the form of a binary tree to achieve query authentication. Clearly, one can use a similar hashing scheme with trees of *higher fanout and with different organization algorithms*, like the B^+-tree, to achieve the same goal. An MB-tree works like a B^+-tree and also consists of ordinary B^+-tree nodes that are extended with one hash value associated with every pointer entry. The hash values associated with entries on leaf nodes are computed on the database records themselves. The hash values associated with index node entries are computed on the concatenation of the hash values of their children. For example, an MB-tree is illustrated in Figure 3. A leaf node entry is associated with a hash value $h = \mathcal{H}(r_i)$, while an index node entry with $h = \mathcal{H}(h_1|\cdots|h_{f_m})$, where $h_1, \ldots, h_{f_m}$ are the hash values of the node's children, assuming fanout f_m per node. After computing all hash values, the owner has to sign the hash of the root using its secret key SK.

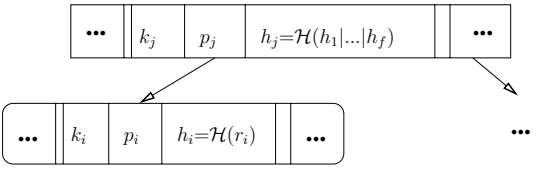

Fig. 3. An MB-tree node.

To answer a range query the server builds a $\mathcal{VO}$ by initiating two top-down B^+-tree like traversals, one to find the left-most and one the right-most query result. At the leaf level, the data contained in the nodes between the two discovered boundary leaves are returned, as in the normal B^+-tree. The server also needs to include in the $\mathcal{VO}$ the hash values of the entries contained in each index node that is visited by the lower and upper boundary traversals of the tree, except the hashes to the right (left) of the pointers that are traversed during the lower (upper) boundary traversals. At the leaf level, the server inserts only the answers to the query, along with the hash

values of the residual entries to the left and to the right parts of the boundary leaves. The result is also increased with one tuple to the left and one to the right of the lower-bound and upper-bound of the query result respectively, for completeness verification. Finally, the signed root of the tree is inserted as well. An example query traversal is shown in Figure 4.

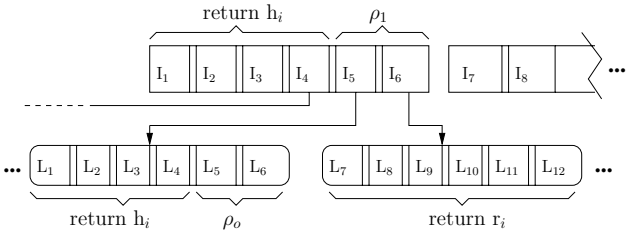

Fig. 4. A query traversal on an MB-tree. At every level the hashes of the residual entries on the left and right boundary nodes need to be returned.

The client can iteratively compute all the hashes of the sub-tree corresponding to the query result, all the way up to the root using the $\mathcal{VO}$. The hashes of the query results are computed first and grouped into their corresponding leaf nodes[4], and the process continues iteratively, until all the hashes of the query sub-tree have been computed. After the hash value of the root has been computed, the client can verify the correctness of the computation using the owner's public key PK and the signed hash of the root. It is easy to see that since the client is forced to recompute the whole query sub-tree, both correctness and completeness is guaranteed. It is interesting to note here that one could avoid building the whole query sub-tree during verification by individually signing all database tuples as well as each node of the B^+-tree. This approach, called VB-tree, was proposed in [22] but it is subsumed by the ASB-tree. Another approach that does not need to build the whole tree appeared in [32]. The analytical cost models of the MB-tree are as follows:

Node fanout:

The node fanout in this case is:

$$f_m = \frac{P - |p| - |h|}{|k| + |p| + |h|} + 1. \tag{7}$$

Notice that the maximum node fanout of the MB-tree is considerably smaller than that of the ASB-tree, since the nodes here are extended with one hash value per entry. This adversely affects the total height of the MB-tree.

[4] Extra node boundary information can be inserted in the $\mathcal{VO}$ for this purpose with a very small overhead.

Storage cost:

The total size is equal to:

$$\mathcal{C}_s^m = P \cdot \frac{f_m^{d_m} - 1}{f_m - 1} + |s|. \tag{8}$$

An important advantage of the MB-tree is that the storage cost does not necessarily reflect the owner/server communication cost. The owner, after computing the final signature of the root, does not have to transmit all hash values to the server, but only the database tuples. The server can recompute the hash values incrementally by recreating the MB-tree. Since hash computations are cheap, for a small increase in the server's computation cost this technique will reduce the owner/sever communication cost drastically.

Construction cost:

The construction cost for building an MB-tree depends on the hash function computations and the total I/Os. Since the tree is bulk-loaded, building the leaf level requires N_D hash computations of input length $|r|$. In addition, for every tree node one hash of input length $f_m \cdot |h|$ is computed. Since there are a total of $N_I = \frac{f_m^{d_m}-1}{f_m-1}$ nodes on average (given height $d_m = \log_{f_m} N_D$), the total number of hash function computations, and hence the total cost for constructing the tree is given by:

$$\mathcal{C}_c^m = N_D \cdot \mathcal{C}_{\mathcal{H}_{|r|}} + N_I \cdot \mathcal{C}_{\mathcal{H}_{f_m|h|}} + \mathcal{C}_{\mathcal{S}_{|h|}} + \frac{\mathcal{C}_s^m}{P} \cdot \mathcal{C}_{IO}. \tag{9}$$

$\mathcal{VO}$ construction cost:

The $\mathcal{VO}$ construction cost is dominated by the total disk I/O. Let the total number of leaf pages accessed be equal to $N_Q = \frac{N_R}{f_m}$, $d_m = \log_{f_m} N_D$ and $d_q = \log_{f_m} N_R$ be the height of the MB-tree and the query sub-tree respectively. In the general case the index traversal cost is:

$$\mathcal{C}_q^m = [(d_m - d_q + 1) + 2(d_q - 2) + N_Q + \frac{N_R \cdot |r|}{P}] \cdot \mathcal{C}_{IO}, \tag{10}$$

taking into account the fact that the query traversal at some point splits into two paths. It is assumed here that the query range spans at least two leaf nodes. The first term corresponds to the hashes inserted for the common path of the two traversals from the root of the tree to the root of the query sub-tree. The second term corresponds to the cost of the two boundary traversals after the split. The last two terms correspond to the cost of the leaf level traversal of the tree and accessing the database records.

Authentication cost:

Assuming that ρ_0 is the total number of query results contained in the left boundary leaf node of the query sub-tree, σ_0 on the right boundary leaf node, and ρ_i, σ_i the total number of entries of the left and right boundary nodes on level $i, 1 \leq i \leq d_q$, that point towards leaves that contain query results (see Figure 4), the size of the $\mathcal{VO}$ is:

$$\begin{aligned} &|\mathcal{VO}|^m = \\ &(2f_m - \rho_0 - \sigma_0)|h| + N_R \cdot |r| + |s| + \\ &(d_m - d_q) \cdot (f_m - 1)|h| + \\ &\sum_{i=1}^{d_q - 2} (2f_m - \rho_i - \sigma_i)|h| + \\ &(f_m - \rho_{d_q - 1} - \sigma_{d_q - 1})|h|. \end{aligned} \tag{11}$$

This cost does not include the extra boundary information needed by the client in order to group hashes correctly, but this overhead is very small (one byte per node in the $\mathcal{VO}$) especially when compared with the hash value size. Consequently, the verification cost on the client is:

$$\begin{aligned} \mathcal{C}_v^m = N_R \cdot \mathcal{C}_{\mathcal{H}_{|r|}} + \sum_{i=0}^{d_q - 1} f_m^i \cdot \mathcal{C}_{\mathcal{H}_{f_m|h|}} + \\ (d_m - d_q) \cdot \mathcal{C}_{\mathcal{H}_{f_m|h|}} + \mathcal{C}_{\mathcal{V}_{|h|}}. \end{aligned} \tag{12}$$

Given that the computation cost of hashing versus signing is orders of magnitude smaller, the initial construction cost of the MB-tree is expected to be orders of magnitude less expensive than that of the ASB-tree. Given that the size of hash values is much smaller than that of signatures and that the fanout of the MB-tree will be smaller than that of the ASB-tree, it is not easy to quantify the exact difference in the storage cost of these techniques, but it is expected that the structures will have comparable storage cost, with the MB-tree being smaller. The $\mathcal{VO}$ construction cost of the MB-tree will be much smaller than that of the ASB-tree, since the ASB-tree requires many I/Os for retrieving signatures, and also some expensive modular multiplications. The MB-tree will have smaller verification cost as well since: 1. Hashing operations are orders of magnitude cheaper than modular multiplications, 2. The ASB-tree requires N_R modular multiplications for verification. The only drawback of the MB-tree is the large $\mathcal{VO}$ size, which increases the client/server communication cost. Notice that the $\mathcal{VO}$ size of the MB-tree is bounded by $f_m \cdot \log_{f_m} N_D$. Since generally $f_m \gg \log_{f_m} N_D$, the $\mathcal{VO}$ size is essentially determined by f_m, resulting in large sizes.

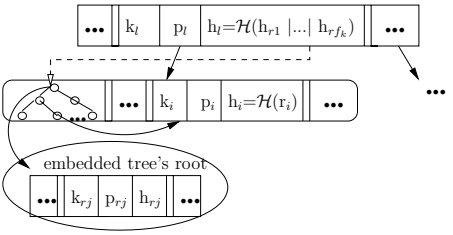

Fig. 5. An EMB-tree node.

The Embedded Merkle B-tree

In this section we present another data structure, the Embedded Merkle B-tree (EMB-tree), that provides a nice, adjustable trade-off between robust initial construction and storage cost versus improved $\mathcal{VO}$ construction and verification cost. The main idea is to have different fanouts for storage and authentication and yet combine them in the same data structure.

Every EMB-tree node consists of regular B^+-tree entries, augmented with an embedded MB-tree. Let f_e be the fanout of the EMB-tree. Then each node stores up to f_e triplets $k_i|p_i|h_i$, and an embedded MB-tree with fanout $f_k < f_e$. The leaf level of this embedded tree consists of the f_e entries of the node. The hash value at the root level of this embedded tree is stored as an h_i value in the parent of the node, thus authenticating this node to its parent. Essentially, we are collapsing an MB-tree with height $d_e \cdot d_k = \log_{f_k} N_D$ into a tree with height d_e that stores smaller MB-trees of height d_k within each node. Here, $d_e = \log_{f_e} N_D$ is the height of the EMB-tree and $d_k = \log_{f_k} f_e$ is the height of each small embedded MB-tree. An example EMB-tree node is shown in Figure 5.

For ease of exposition, in the rest of this discussion it will be assumed that f_e is a power of f_k such that the embedded trees when bulk-loaded are always full. The technical details if this is not the case can be worked out easily. The exact relation between f_e and f_k will be discussed shortly. After choosing f_k and f_e, bulk-loading the EMB-tree is straightforward: Simply group the N_D tuples in groups of size f_e to form the leaves and build their embedded trees on the fly. Continue iteratively in a bottom up fashion.

When querying the structure the server follows a path from the root to the leaves of the external tree as in the normal B^+-tree. For every node visited, the algorithm scans all $f_e - 1$ triplets $k_i|p_i|h_i$ on the data level of the embedded tree to find the key that needs to be followed to the next level. When the right key is found the server also initiates a point query on the embedded tree of the node using this key. The point query will return all the needed hash values for computing the concatenated hash of the node, exactly like for the MB-tree. Essentially, these hash values would be the equivalent of the $f_e - 1$ sibling hashes that would be returned per node if the embedded tree was not used. However, since now the hashes are arranged hierarchically in an f_k-way

tree, the total number of values inserted in the $\mathcal{VO}$ per node is reduced to $(f_k - 1)d_k$.

To authenticate the query results the client uses the normal MB-tree authentication algorithm to construct the hash value of the root node of each embedded tree (assuming that proper boundary information has been included in the $\mathcal{VO}$ for separating groups of hash values into different nodes) and then follows the same algorithm once more for computing the final hash value of the root of the EMB-tree.

The EMB-tree structure uses extra space for storing the index levels of the embedded trees. Hence, by construction it has increased height compared to the MB-tree due to smaller fanout f_e. A first, simple optimization for improving the fanout of the EMB-tree is to avoid storing the embedded trees altogether. Instead, each embedded tree can be instantiated by computing fewer than $f_e/(f_k - 1)$ hashes on the fly, only when a node is accessed during the querying phase. We call this the EMB^--tree. The EMB^--tree is logically the same as the EMB-tree, however its physical structure is equivalent to an MB-tree with the hash values computed differently. The querying algorithm of the EMB^--tree is slightly different than that of the EMB-tree in order to take into account the conceptually embedded trees. With this optimization the storage overhead is minimized and the height of the EMB^--tree becomes equal to the height of the equivalent MB-tree. The trade-off is an increased computation cost for constructing the $\mathcal{VO}$. However, this cost is minimal as the number of embedded trees that need to be reconstructed is bounded by the height of the EMB^--tree.

As a second optimization, one can create a slightly more complicated embedded tree to reduce the total size of the index levels and increase fanout f_e. We call this the EMB^*-tree. Essentially, instead of using a B^+-tree as the base structure for the embedded trees, one can use a multi-way search tree with fanout f_k while keeping the structure of the external EMB-tree intact. The embedded tree based on B^+-trees has a total of $N_i = \frac{f_k^{d_k} - 1}{f_k - 1}$ nodes while, for example, a B-tree based embedded tree (recall that a B-tree is equivalent to a balanced multi-way search tree) would contain $N_i = \frac{f_e - 1}{f_k - 1}$ nodes instead. A side effect of using multi-way search trees is that the cost for querying the embedded tree on average will decrease, since the search for a particular key might stop before reaching the leaf level. This will reduce the expected cost of $\mathcal{VO}$ construction substantially. Below we give the analytical cost models of the EMB-tree. The further technical details and the analytical cost models associated with the EMB^*-tree and EMB^--tree are similar to the EMB-tree case and can be worked out similarly.

Node fanout:

For the EMB-tree, the relationship between f_e and f_k is given by:

$$P \geq \frac{f_k^{\log_{f_k} f_e - 1} - 1}{f_k - 1}[f_k(|k| + |p| + |h|) - |k|] + [f_e(|k| + |p| + |h|) - |k|]. \quad (13)$$

First, a suitable f_k is chosen such that the requirements for authentication cost and storage overhead are met. Then, the maximum value for f_e satisfying (13) can be determined.

Storage cost:

The storage cost is equal to:

$$\mathcal{C}_s^e = P \cdot \frac{f_e^{d_e} - 1}{f_e - 1} + |s|. \quad (14)$$

Construction cost:

The total construction cost is the cost of constructing all the embedded trees plus the I/Os to write the tree back to disk. Given a total of $N_I = \frac{f_e^{d_e}-1}{f_e-1}$ nodes in the tree and $N_i = \frac{f_k^{d_k}-1}{f_k-1}$ nodes per embedded tree, the cost is:

$$\mathcal{C}_c^e = N_D \cdot \mathcal{C}_{\mathcal{H}_{|r|}} + N_I \cdot N_i \cdot \mathcal{C}_{\mathcal{H}_{f_k|h|}} + \mathcal{C}_{\mathcal{S}_{|h|}} + \frac{\mathcal{C}_s^e}{P} \cdot \mathcal{C}_{IO}. \quad (15)$$

It should be mentioned here that the cost for constructing the EMB^--tree is exactly the same, since in order to find the hash values for the index entries of the trees one needs to instantiate all embedded trees. The cost of the EMB^*-tree is somewhat smaller than (15), due to the smaller number of nodes in the embedded trees.

$\mathcal{VO}$ construction cost:

The $\mathcal{VO}$ construction cost is dominated by the total I/O for locating and reading all the nodes containing the query results. Similarly to the MB-tree case:

$$\mathcal{C}_q^e = [(d_e - d_q + 1) + 2(d_q - 2) + N_Q + \frac{N_R \cdot |r|}{P}] \cdot \mathcal{C}_{IO}, \quad (16)$$

where d_q is the height of the query sub-tree and $N_Q = \frac{N_R}{f_e}$ is the number of leaf pages to be accessed. Since the embedded trees are loaded with each node, the querying computation cost associated with finding the needed hash values is expected to be dominated by the cost of loading the node in memory, and hence it is omitted. It should be restated here that for the EMB^*-tree the expected $\mathcal{VO}$ construction cost will be smaller, since not all embedded tree searches will reach the leaf level of the structure.

Authentication cost:

The embedded trees work exactly like MB-trees for point queries. Hence, each embedded tree returns $(f_k - 1)d_k$ hashes. Similarly to the MB-tree the total size of the $\mathcal{VO}$ is:

$$|\mathcal{VO}|^e = N_R \cdot |r| + |s| + \sum_{0}^{d_q-2} 2|\mathcal{VO}|^m + |\mathcal{VO}|^m + \sum_{d_q}^{d_m-1} (f_k - 1)d_k|h|, \quad (17)$$

where $|\mathcal{VO}|^m$ is the cost of a range query on the embedded trees of the boundary nodes contained in the query sub-tree given by equation (11), with a query range that covers all pointers to children that cover the query result-set.

The verification cost is:

$$\mathcal{C}_v^e = N_R \cdot \mathcal{C}_{\mathcal{H}_{|r|}} + \sum_{i=0}^{d_q-1} f_e^i \cdot \mathcal{C}_k + (d_e - d_q) \cdot \mathcal{C}_k + \mathcal{C}_{\mathcal{V}_{|h|}}, \quad (18)$$

where $\mathcal{C}_k = N_i \cdot \mathcal{C}_{\mathcal{H}_{f_k|h|}}$ is the cost for constructing the concatenated hash of each node using the embedded tree.

For $f_k = 2$ the authentication cost becomes equal to a Merkle hash tree, which has the minimal $\mathcal{VO}$ size but higher verification time. For $f_k \geq f_e$ the embedded tree consists of only one node which can actually be discarded, hence the authentication cost becomes equal to that of an MB-tree, which has larger $\mathcal{VO}$ size but smaller verification cost. Notice that, as f_k becomes smaller, f_e becomes smaller as well. This has an impact on $\mathcal{VO}$ construction cost and size, since with smaller fanout the height of the EMB-tree increases. Nevertheless, since there is only a logarithmic dependence on f_e versus a linear dependence on f_k, it is expected that with smaller f_k the authentication related operations will become faster.

4 Authentication Index Structures in Dynamic Settings

In this section we analyze the performance of all approaches given dynamic updates between the owner and the servers. In particular we assume that either insertions or deletions can occur to the database, for simplicity. The performance of updates in the worst case can be considered as the cost of a deletion followed by an insertion. There are two contributing factors for the update cost: computation cost such as creating new signatures and computing hashes, and I/O cost.

Aggregated Signatures with B$^+$-trees

Suppose that a single database record r_i is inserted in or deleted from the database. Assuming that in the sorted order of attribute A the left neighbor

of r_i is r_{i-1} and the right neighbor is r_{i+1}, for an insertion the owner has to compute signatures $\mathcal{S}(r_{i-1}|r_i)$ and $\mathcal{S}(r_i|r_{i+1})$, and for a deletion $\mathcal{S}(r_{i-1}|r_{i+1})$. For k consecutive updates in the best case a total of $k+2$ signature computations are required for insertions and 1 for deletions if the deleted tuples are consecutive. In the worst case a total of $2k$ signature computations are needed for insertions and k for deletions, if no two tuples are consecutive. Given k updates, suppose the expected number of signatures to be computed is represented by $E\{k\}(k \leq E\{k\} \leq 2k)$. Then the additional I/O incurred is equal to $\frac{E\{k\}\cdot|s|}{P}$, excluding the I/Os incurred for updating the B^+-tree structure. Since the cost of signature computations is larger than even the I/O cost of random disk accesses, a large number of updates is expected to have a very expensive updating cost. The total update cost for the ASB-tree is:

$$\mathcal{C}_u^a = E\{k\} \cdot \mathcal{C}_s + \frac{E\{k\} \cdot |s|}{P} \cdot \mathcal{C}_{IO}. \tag{19}$$

The Merkle B-tree

The MB-tree can support efficient updates since only hash values are stored for the records in the tree and, first, hashing is orders of magnitude faster then signing, second, for each tuple only the path from the affected leaf to the root need to be updated. Hence, the cost of updating a single record is dominated by the cost of I/Os. Assuming that no reorganization to the tree occurs the cost of an insertion is $\mathcal{C}_u^m = \mathcal{H}_{|r|} + d_m(\mathcal{H}_{f_m|h|} + \mathcal{C}_{IO}) + \mathcal{S}_{|h|}$.

In realistic scenarios though one expects that a large number of updates will occur at the same time. In other cases the owner may decide to do a delayed batch processing of updates as soon as enough changes to the database have occurred. The naive approach for handling batch updates would be to do all updates to the MB-tree one by one and update the path from the leaves to the root once per update. Nevertheless, in case that a large number of updates affect a similar set of nodes (e.g., the same leaf) a per tuple updating policy performs an unnecessary number of hash function computations on the predecessor path. In such cases, the computation cost can be reduced significantly by recomputing the hashes of all affected nodes only once, after all the updates have been performed on the tree. A similar analysis holds for the incurred I/O as well.

Clearly, the total update cost for the per tuple update approach for k insertions is $k \cdot \mathcal{C}_u^m$ which is linear to the number of affected nodes $k \cdot d_m$. The expected cost of k updates using batch processing can be computed as follows. Given k updates to the MB-tree, assuming that all tuples are updated uniformly at random and using a standard balls and bins argument, the probability that leaf node X has been affected at least once is $P(X) = 1 - (1 - \frac{1}{f_m^{d_m-1}})^k$ and the expected number of leaf nodes that have been affected is $f_m^{d_m-1} \cdot P(X)$. Using the same argument, the expected number of nodes at level i (where $i = 1$ is the leaf level and $1 \leq i \leq d_m$) is

$f_m^{d_m-i} \cdot P_i(X)$, where $P_i(X) = [1 - (1 - \frac{1}{f_m^{d_m-i}})^k]$. Hence, for a batch of k updates the total expected number of nodes that will be affected is:

$$E\{X\} = \sum_{i=0}^{d_m-1} f_m^i [1 - (1 - \frac{1}{f_m^i})^k]. \tag{20}$$

Hence, the expected MB-tree update cost for batch updates is

$$\mathcal{C}_u^m = k \cdot \mathcal{H}_{|r|} + E\{X\} \cdot (\mathcal{H}_{f_m|h|} + \mathcal{C}_{IO}) + \mathcal{S}_{|h|}. \tag{21}$$

In order to understand better the relationship between the per-update approach and the batch-update, we can find the closed form for $E\{X\}$ as follows:

$$\begin{aligned}
&\textstyle\sum_{i=0}^{d_m-1} f_m^i (1 - (\frac{f_m^i - 1}{f_m^i})^k) \\
=\ &\textstyle\sum_{i=0}^{d_m-1} f_m^i (1 - (1 - \frac{1}{f_m^i})^k) \\
=\ &\textstyle\sum_{i=0}^{d_m-1} f_m^i [1 - \sum_{x=0}^{k} \binom{k}{x} (-\frac{1}{f_m^i})^x] \\
=\ &\textstyle\sum_{i=0}^{d_m-1} f_m^i - \sum_{i=0}^{d_m-1} \sum_{x=0}^{k} \binom{k}{x} (-1)^x (\frac{1}{f_m^i})^{x-1} \\
=\ &\textstyle k d_m - \sum_{x=2}^{k} \binom{k}{x} (-1)^x \sum_{i=0}^{d_m-1} (\frac{1}{f_m^i})^{x-1} \\
=\ &\textstyle k d_m - \sum_{x=2}^{k} \binom{k}{x} (-1)^x \frac{1 - (\frac{1}{f_m^{d_m}})^{x-1}}{1 - (\frac{1}{f_m})^{x-1}}
\end{aligned}$$

The second term quantifies the cost decrease afforded by the batch update operation, when compared to the per update cost.

For non-uniform updates to the database, the batch updating technique is expected to work well in practice given that in real settings updates exhibit a certain degree of locality. In such cases one can still derive a similar cost analysis by modelling the distribution of updates.

The Embedded MB-tree

The analysis for the EMB-tree is similar to the one for MB-trees. The update cost for per tuple updates is equal to $k \cdot \mathcal{C}_u^e$, where $\mathcal{C}_u^e = \mathcal{H}_{|r|} + d_e \log_{f_k} f_e \cdot (\mathcal{H}_{f_k|h|} + \mathcal{C}_{IO}) + \mathcal{S}_{|h|}$, once again assuming that no reorganizations to the tree occur. Similarly to the MB-tree case the expected cost for batch updates is equal to:

$$\mathcal{C}_u^e = k \cdot \mathcal{H}_{|r|} + E\{X\} \cdot \log_{f_k} f_e \cdot (\mathcal{H}_{f_k|h|} + \mathcal{C}_{IO}) + \mathcal{S}_{|h|}. \tag{22}$$

Discussion

For the ASB-tree, the communication cost for updates between owner and servers is bounded by $E\{K\}|s|$, and there is no possible way to reduce this cost as only the owner can compute signatures. However, for the hash based index structures, there are a number of options that can be used for transmitting

the updates to the server. The first option is for the owner to transmit only a delta table with the updated nodes of the MB-tree (or EMB-tree) plus the signed root. The second option is to transmit only the signed root and the updates themselves and let the servers redo the necessary computations on the tree. The first approach minimizes the computation cost on the servers but increases the communication cost, while the second approach has the opposite effect.

5 Query Freshness

The dynamic scenarios considered before reveal a third dimension of the query authentication problem, that of *query result freshness*. When the owner updates the database, a malicious or compromised server may still retain an older version of the data. Since the old version was authenticated by the owner already, the client will still accept any query results originating from an old version as authentic, unless the latter is informed by the owner that this is no longer the case. In fact, a malicious server may choose to answer queries using any previous version, and in some scenarios even a combination of older versions of the data. If the client wishes to be assured that queries are answered using the latest data updates, additional work is necessary.

This issue is similar to the problem of ensuring the freshness of signed documents, which has been studied extensively in the context of certificate validation and revocation. There are many approaches which we do not review here. The simplest is to publish a list of revoked signatures, one for every expired version of the database. More sophisticated ones are: 1. Including the time interval of validity as part of the signed root of the authenticated structures and reissuing the signature after the interval expires, 2. Using hash chains to confirm validity of signatures at frequent intervals [33].

Clearly, all signature freshness techniques impose a cost which is linear to the number of signatures used by any authentication structure. The advantage of the Merkle tree based methods is that they use one signature only — that of the root of the tree — which is sufficient for authenticating the whole database. Straightforwardly, database updates will also require re-issuing only the signature of the root.

6 Extensions

This section extends our discussion to other interesting topics that are related to the query authentication problem.

Multi-dimensional Selection and Aggregation Range Queries. The same ideas that we discussed before can be used for authenticating multi-dimensional range queries. In particular, any tree based multi-dimensional

index structure, like the R-tree, can be used to create verification objects for multi-dimensional data. The tree is extended with hash values that are computed using both the hash values of its children nodes in the tree and the multi-dimensional information that is used to navigate the tree. For the R-tree, this means that the hash value for a node N will contain all the hash values and the MBR's of the children nodes of N. Signature based approaches can be also used [34, 31]. Furthermore, aggregation queries can be authenticated using aggregation trees [35, 36]. The only difference is that the aggregate value of each subtree should be included in the computation of the hash values. That is, for each node N of an aggregation tree we add the aggregate value of the subtree that starts at N and we include this in the hash value of the node [37].

General Query Types. The authenticated structures presented before can support other query types as well. We briefly discuss here a possible extension of these techniques for join queries. Other query types that can be supported are projections and relational set operations.

Assume that we would like to provide authenticated results for join queries such as $R \bowtie_{A_i=A_j} S$, where $A_i \in R$ and $A_j \in S$ (R and S could be relations or result-sets from other queries), and authenticated structures for both A_i in R and A_j in S exist. The server can provide the proof for the join as follows: 1. Select the relation with the smaller size, say R, 2. Construct the VO for R (if R is an entire relation then the $\mathcal{VO}$ contains only the signature of the root node from the index of R), 3. Construct the $\mathcal{VO}$s for each of the following selection queries: for each record r_k in R, $q_k =$"SELECT * FROM S WHERE $r.A_j = r_k.A_i$". The client can easily verify the join results. First, it authenticates that the relation R is complete and correct. Then, using the $\mathcal{VO}$ for each query q_k, it makes sure that it is complete for every k (even when the result of q_k is empty). After this verification, the client can construct the results for the join query and be sure that they are complete and correct.

7 Conclusion

In this chapter we presented three approaches to authenticate range queries in ODBs. The first approach is based on signature chaining and aggregation, the second on combining a Merkle hash tree with a B+-tree and the third is an improved version of the hash tree approach. We discussed advantages and disadvantages of each approach and we gave an analytical cost model for each approach and different cost metrics. Finally, we discussed the performance of each method under a dynamic environment and we gave extensions of these techniques to other query types. A interesting future direction is to enhance the proposed methods to work efficiently for complex relational queries. Another direction is to investigate authentication techniques for other types of databases beyond relational databases.

References

1. Hacigumus, H., Iyer, B.R., Mehrotra, S.: Providing database as a service. In: Proc. of International Conference on Data Engineering (ICDE). (2002) 29–40
2. Hore, B., Mehrotra, S., Tsudik, G.: A privacy-preserving index for range queries. In: Proc. of Very Large Data Bases (VLDB). (2004) 720–731
3. Agrawal, R., Srikant, R.: Privacy-preserving data mining. In: Proc. of ACM Management of Data (SIGMOD). (2000) 439–450
4. Evfimievski, A., Gehrke, J., Srikant, R.: Limiting privacy breaches in privacy preserving data mining. In: Proc. of ACM Symposium on Principles of Database Systems (PODS). (2003) 211–222
5. Hacigumus, H., Iyer, B.R., Li, C., Mehrotra, S.: Executing SQL over encrypted data in the database service provider model. In: Proc. of ACM Management of Data (SIGMOD). (2002) 216–227
6. Miklau, G., Suciu, D.: Controlling access to published data using cryptography. In: Proc. of Very Large Data Bases (VLDB). (2003) 898–909
7. Rizvi, S., Mendelzon, A., Sudarshan, S., Roy, P.: Extending query rewriting techniques for fine-grained access control. In: Proc. of ACM Management of Data (SIGMOD). (2004) 551–562
8. Bouganim, L., Ngoc, F.D., Pucheral, P., Wu, L.: Chip-secured data access: Reconciling access rights with data encryption. In: Proc. of Very Large Data Bases (VLDB). (2003) 1133–1136
9. Pang, H., Jain, A., Ramamritham, K., Tan, K.L.: Verifying completeness of relational query results in data publishing. In: Proc. of ACM Management of Data (SIGMOD). (2005) 407–418
10. Sion, R.: Query execution assurance for outsourced databases. In: Proc. of Very Large Data Bases (VLDB). (2005) 601–612
11. Anagnostopoulos, A., Goodrich, M., Tamassia, R.: Persistent authenticated dictionaries and their applications. In: ISC. (2001) 379–393
12. Goodrich, M., Tamassia, R., Triandopoulos, N., Cohen, R.: Authenticated data structures for graph and geometric searching. In: CT-RSA. (2003) 295–313
13. Tamassia, R., Triandopoulos, N.: Computational bounds on hierarchical data processing with applications to information security. In: ICALP. (2005) 153–165
14. Li, F., Yi, K., Hadjieleftheriou, M., Kollios, G.: Proof-infused streams: Enabling authentication of sliding window queries on streams. In: Proc. of Very Large Data Bases (VLDB). (2007)
15. Papadopoulos, S., Yang, Y., Papadias, D.: CADS: Continuous authentication on data streams. In: Proc. of Very Large Data Bases (VLDB). (2007)
16. McCurley, K.: The discrete logarithm problem. In: Proc. of the Symposium in Applied Mathematics, American Mathematical Society (1990) 49–74
17. National Institute of Standards and Technology: FIPS PUB 180-1: Secure Hash Standard. National Institute of Standards and Technology (1995)
18. Wang, X., Yin, Y., Yu, H.: Finding collisions in the full sha-1. In: CRYPTO. (2005)
19. Wang, X., Yao, A., Yao, F.: New collision search for SHA-1 (2005) Presented at the rump session of Crypto 2005.
20. Goldwasser, S., Micali, S., Rivest, R.L.: A digital signature scheme secure against adaptive chosen-message attacks. SIAM Journal on Computing **17**(2) (1988) 96–99

21. Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM (CACM) **21**(2) (1978) 120–126
22. Pang, H., Tan, K.L.: Authenticating query results in edge computing. In: Proc. of International Conference on Data Engineering (ICDE). (2004) 560–571
23. Mykletun, E., Narasimha, M., Tsudik, G.: Authentication and integrity in outsourced databases. In: Symposium on Network and Distributed Systems Security (NDSS). (2004)
24. Narasimha, M., Tsudik, G.: Dsac: Integrity of outsourced databases with signature aggregation and chaining. In: Proc. of Conference on Information and Knowledge Management (CIKM). (2005) 235–236
25. Mykletun, E., Narasimha, M., Tsudik, G.: Signature bouquets: Immutability for aggregated/condensed signatures. In: European Symposium on Research in Computer Security (ESORICS). (2004) 160–176
26. Merkle, R.C.: A certified digital signature. In: Proc. of Advances in Cryptology (CRYPTO). (1989) 218–238
27. Naor, M., Nissim, K.: Certificate revocation and certificate update. In: Proceedings 7th USENIX Security Symposium (San Antonio, Texas). (1998)
28. Martel, C., Nuckolls, G., Devanbu, P., Gertz, M., Kwong, A., Stubblebine, S.: A general model for authenticated data structures. Algorithmica **39**(1) (2004) 21–41
29. Li, F., Hadjieleftheriou, M., Kollios, G., Reyzin, L.: Dynamic authenticated index structures for outsourced databases. In: Proc. of ACM Management of Data (SIGMOD). (2006)
30. Comer, D.: The ubiquitous B-tree. ACM Computing Surveys **11**(2) (1979) 121–137
31. Cheng, W., Pang, H., Tan, K.: Authenticating multi-dimensional query results in data publishing. In: DBSec. (2006)
32. Nuckolls, G.: Verified query results from hybrid authentication trees. In: DBSec. (2005) 84–98
33. Micali, S.: Efficient certificate revocation. Technical Report MIT/LCS/TM-542b, Massachusetts Institute of Technology, Cambridge, MA (1996)
34. Narasimha, M., Tsudik, G.: Authentication of outsourced databases using signature aggregation and chaining. In: DASFAA. (2006) 420–436
35. Lazaridis, I., Mehrotra, S.: Progressive approximate aggregate queries with a multi-resolution tree structure. In: Proc. of ACM Management of Data (SIGMOD). (2001) 401–412
36. Tao, Y., Papadias, D.: Range aggregate processing in spatial databases. IEEE Transactions on Knowledge and Data Engineering (TKDE) **16**(12) (2004) 1555–1570
37. Li, F., Hadjieleftheriou, M., Kollios, G., Reyzin, L.: Authenticated index sturctures for aggregation queries in outsourced databases. Technical report, CS Dept., Boston University (2006)