DSAC: An Approach to Ensure Integrity of Outsourced Databases using Signature Aggregation and Chaining

Maithili Narasimha, Gene Tsudik
Computer Science Department
School of Information and Computer Science
University of California, Irvine
{mnarasim, gts}@ics.uci.edu

ABSTRACT

Database outsourcing is an important emerging trend which involves data owners delegating their data management needs to an external service provider. In this model, a service provider hosts clients' databases and offers mechanisms to create, store, update and access (query) outsourced databases. Since a service provider is almost never fully trusted, security and privacy of outsourced data are important concerns.

A core security requirement is the integrity and authenticity of outsourced databases. Whenever someone queries a hosted database, the results must be demonstrably authentic (with respect to the actual data owner) to ensure that the data has not been tampered with. Furthermore, the results must carry a proof of completeness which will allow the querier to verify that the server has not omitted any valid tuples that match the query predicate.

Notable prior research ([6, 12, 18]) focused on so-called *Authenticated Data Structures.* Another prior approach involved the use of special digital signature schemes. In this paper, we extend the state-of-the-art to provide both authenticity and completeness guarantees of query replies. Our work also analyzes the new approach for various base query types and compares the new approach with Authenticated Data Structures.[1]

1. INTRODUCTION

Database outsourcing [9] is a prominent example of the general commercial trend of outsourcing non-core competencies. In the Outsourced Database (ODB) Model, a third-party database service provider offers adequate software, hardware and network resources to host its clients' databases as well as mechanisms to efficiently create, update and access outsourced data.

The ODB model poses numerous research challenges which influence overall performance, usability and scalability. One of the biggest challenges is the security of hosted data. A *client* stores its data (which is usually a critical asset) at an external, and potentially untrusted, database service provider. It is thus important to secure outsourced data from potential attacks not only by malicious outsiders but also from the service provider itself.

The two pillars of data security are secrecy and integrity. (We use the term integrity in a broad sense, encompassing both *data integrity* and *authentication* with respect to the actual owner.) The central problem in the context of secrecy[7, 10] is allowing a client to efficiently query its own data – which is hosted by a third-party service provider – while revealing to the latter neither the actual query nor the data over which the query is executed. In contrast, this paper focuses on the integrity of query replies for queries posed for outsourced databases. Specifically, in this work, we ensure that query results returned by the server are: (i) correct - the tuples in the result set have not been tampered with, and (ii) complete - no valid tuples have been omitted from the result set.

Other relevant prior work [6, 18, 8] examined integrity issues in outsourced databases and suggested solutions using Authenticated Data Structures. Recently, [15] investigated the notion of signature aggregation which enables bandwidth- and computation-efficient integrity verification of query replies. However, signature aggregation mechanism ensures only *correctness* of query replies. In this paper, we extend the work of [15] by proposing new techniques to provide *completeness* guarantees. We provide a detailed study of the applicability of our techniques for various base type queries. Further, we compare our approach with solutions using Authenticated Data Structures.

Scope: In this paper, we provide efficient mechanisms to ensure the correctness and completeness of range selection queries, projections, joins and other set operation queries. Range selection queries involve testing equality and other logical comparison predicate clauses. In other words, we consider the standard SQL queries involving *SELECT* clauses which (usually) result in the selection of a set of records or fields that match a given predicate or a set thereof. We specifically do not address queries that involve data aggregation (exemplified by arithmetic operations, such as SUM or AVERAGE) which usually return a single value as the answer to the posed query.

Further, while we consider dynamic databases and address tuple inserts and delete operations, we assume that the fre-

[1]We also point out some possible security flaws in the approach suggested in the recent work of [18].

quency of querying $\gg$ frequency of updates. In other words, our main goal is to provide techniques to verify the correctness and completeness of query replies that are efficient in terms of computation and bandwidth for the queriers.

Organization: The rest of this paper is organized as follows: Section 2 motivates our work. Section 3 discusses Authenticated Data Structures approach, followed by Section 4 which describes signature aggregation and its applications to verifying correctness of query replies. This section also proposes the extensions to achieve completeness guarantees. Section 5 describes our approach by considering various query types. Section 6 presents the analysis of the proposed techniques and compares our scheme with prior art. The paper concludes in section 8.

2. MOTIVATION

In the Outsourced Database (ODB) model, a *Database Service Provider* (referred to as simply **server** from here on) has the infrastructure to host outsourced databases and provides efficient mechanisms for remote clients to create, store, update and query their databases. The ODB model naturally triggers some important security concerns. One immediate concern is the secrecy of outsourced data with respect to the server. The main challenge here is reconciling the requirement for secrecy with the need to outsource data. Some notable previous work [9, 7] addressed this by devising methods for running encrypted (or obfuscated) queries over (partially) encrypted databases.

This paper addresses the integrity of outsourced data in the ODB model. (We note that data secrecy in ODB is orthogonal to integrity.) Specifically, we focus on integrity-critical databases which are outsourced to untrusted servers and accessed over insecure public networks. We assume that servers can be malicious and/or incompetent and, thus, might be processing and storing hosted data incorrectly. Furthermore, since it is difficult, in general, to guarantee absolute security of large on-line systems, we assume that the server can be compromised by an attack, such as a worm/virus or a system break-in. Therefore, we need efficient mechanisms to reduce the level of trust placed in the server and provide integrity guarantees to the clients.

2.1 Desired Security Properties:

From a technical perspective, candidate solutions must include the following properties:

Proof of Correctness: When an ODB client queries outsourced data, it expects a set of tuples satisfying all of the query's predicates. It also needs assurance that the results have been originated by the actual data owner and have not been tampered with either by an outside attacker or by the server itself. Note that the size of the reply can vary, in principle, between zero and n, where n is the total number of tuples in the database. In other words, a query reply can potentially be any one of the 2^n tuple subsets. Proof of correctness facilitates secure and efficient authentication of all possible query replies.

Proof of Completeness: A related property is the *completeness* of the result set. *Completeness* implies that the querier can verify that the server returned **all** tuples matching query predicates. A server, which is either malicious or lazy, might not execute the query over the entire database and return only partial results. Proof of completeness facilitates efficient verification that the results set contains every tuple that matches the query predicate.

3. PRIOR WORK

We now summarize the general approach of using authenticated data structures to provide authentication of query replies and discuss two related bodies of work that use this approach, in the contexts of "Third-Party Publication" and "Edge Computing", respectively.

The basis for these two bodies of work is the seminal work by Merkle [14]. This work introduced a data structure called a "Merkle Hash Tree" (MHT) which is intended to authenticate a set of n values $x_1, x_2, ..., x_n$. MHT is constructed as a a binary tree where the leaves correspond to the hashes of the n values of the elements in the set. Thus, 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 SHA [16]. The values of non-leaf nodes correspond to the hash of the concatenation of its two children (maintaining their order). A node with children v_1 and v_2 will be assigned the value $h(v_1||v_2)$. The root of the tree is digitally signed using a public key signature scheme (e.g., RSA or DSA). An MHT can be used to prove the existence of an element in the set with the help of a *verification object*. A verification object is a set of $log(n)$ internal tree nodes which help the verifier re-compute the root of the MHT whose signature can then be verified. Although an MHT can be very large, one only needs the (signed) root and a short verification object in order to verify that a particular leaf element is part of the tree. For example, to verify that leaf node 5 is in the MHT of figure 1, the verification object (VO) contains node values $5, h_1, h_{34}$, as well as the signature of the root. The verifier computes: $h'_2 = h(5)$, $h'_{12} = h(h_1||h_2)$ and $h'_{1234} = h(h'_{12}||h_{34})$ and then verifies the root by checking its signature.

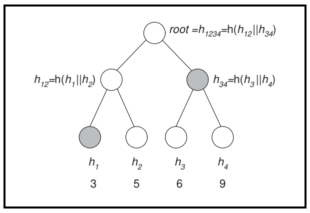


Figure 1: MHT Example: shades nodes represent the verification object for leaf value 5.

3.1 Authentic Third Party Data Publication

In [6] and several related publications, Devanbu, et al. focus on Third-Party Publication. We refer to this approach as the Authenticated Data Structures (or **AuthDS**) approach. In this setting, the (trusted) data owner produces integrity-critical content which is then published by third-party service providers in order to widely disseminate the content. The notable contributions of this work are as follows: (1) It demonstrates how to construct efficient and compact verification objects if a pre-computed authenticated data structure for that type of query exists. The terms *efficient and compact* generally mean logarithmic complexity in terms of the database size. (2) Instead of using standard MHTs (which are relatively inefficient binary search trees) as authenticated dictionaries, balanced and I/O efficient data structures, such as B-trees, are used.

Discussion: One limitation of the AuthDS approach is the need to pre-compute and store a potentially large number of authenticated data structures, in order to efficiently answer queries. Without pre-computed trees, the AuthDS approach cannot provide small verification objects. More importantly, without pre-computed trees for each sort-order, it becomes impossible to prove completeness of query replies. For example, to support simple range queries involving a single attribute, pre-computed hash trees for all possible sort orders of the relation must be pre-computed. This results in significant setup costs for the owner and high storage overhead for the server. Also, storing multiple trees for the same relation increases the cost of updates.

3.2 Authenticating Query Results in Edge Computing

Pang, et al. [18] focused on authentication in edge computing applications. (We refer to this work as the **VB-tree approach**). In it, a trusted central server outsources parts of the database to proxy servers situated at the edge of the network. The data structure used here is a VB-tree, which is basically a modified MHT and is built using a B-Tree where – instead of signing only the root – all leaf nodes as well as all internal nodes are also signed. As a result, verification objects are independent of the database size and hence, "potentially" much smaller. In comparison, the most efficient VO using the Devanbu, et al. approach [6] is logarithmic in the size of the entire database.

Discussion: The VB-tree approach only provides a proof of correctness. It does not address the completeness problem. Also, since a single VB-tree is used, there is no easy way to extend this scheme to provide completeness guarantees. The proposed scheme replaces a conventional cryptographic hash function used to compute the digests of individual values in a MHT with a computationally more expensive, homomorphic function which essentially computes a discrete exponentiation in a finite field. This function is insecure and can lead to forgery attacks as shown below:
The digest is computed as $h(x) = g^x \mod q$. The modulus q is chosen as $q = 2^r$ for some random r. This choice is insecure because computing discrete logs in multiplicative, algebraic groups (thus reversing the function h) is known to be hard if q is a large prime of at least 512 bits. If q, however, is a composite integer, then the problem of computing discrete logarithms is polynomially reducible to the combination of integer factorization of q and computing discrete logarithms in $\mathbb{Z}_p^*$ for each prime factor p of q. Now in the current context, since q is chosen as 2^r (with a small prime factor 2), $h()$ can be reversed efficiently which can lead to forgery attacks. We refer the interested readers to [13] for details on solving discrete-logarithm problems.

Also, the experimental analysis of [18] assumes that the size of a signed digest is 16 bytes. It demonstrates that, with this overhead, although the VB-tree fan-out is small, the overall approach is efficient in terms of storage and VO size. However, a 16-byte signed digest is **insecure**, since there is no cryptographically strong digital signature scheme that produces signatures of only 16 bytes in size. For example, RSA, which is the most well-known signature scheme, has a signature size of at least 128 bytes (1024 bits). [2] If we repeat the calculations with a digest size of 128 bytes and recompute storage overheads, the VB-tree approach becomes quite expensive in terms of both computation and storage.

Furthermore, VB-tree approach can be very expensive in terms of VO verification time for queriers, especially, for projection queries. This is because the verification object includes signed digests for all the attributes that are filtered out as well as all the tuples that do not belong to the query result set but do fall inside the enveloping tree [3] for a given query. In order to authenticate the query results, the scheme requires the querier to verify the signatures of all these filtered attributes and tuples that are not part of the actual result set. Clearly, receiving (recall that a signature is at least 128 bytes long) and verifying (a single RSA signature verification takes 0.16 msec on P3-977 MHz machine) all these signatures can be computationally very expensive for the querier.

Finally, VB-tree approach builds a single B-tree for each table (which is computed on the sorted order of the primary key of that table). If the query predicate requires searching on a non-key attribute, then the result set is no longer a set of contiguous tuples. This translates to an increase in the height of the enveloping tree (with worst case height being equal to the height of the VB-tree itself) which in turn results in extremely high bandwidth and computation overheads. Recall that the VO verification involves verifying the signatures of all the tuples that are not part of the actual result set but do fall inside the enveloping tree.

4. DIGITAL SIGNATURE AGGREGATION AND CHAINING (DSAC)

The main disadvantage of the aforementioned AuthDS approach is high overhead associated with building, storing and updating complex index structures. We now propose an alternative approach that is secure and efficient for most base-level queries, without requiring any complex data structures. We refer to our approach as the Digital Signature Aggregation and Chaining - **DSAC**.

A natural and naïve alternative to AuthDS is to use digital signatures at the granularity of individual tuples. The data owner signs each tuple before storing it in the outsourced database at the publisher/server's site. The server stores the tuple signature along with each tuple. In response to a query, the server simply sends the matching tuples and their signatures to prove integrity and authenticity of the result.

Although this naïve solution provides a proof of correctness, it has some drawbacks: first and foremost, the resultant VO (which comprises a set of signatures corresponding to each tuple in the result set) is neither bandwidth- nor computation-efficient for the querier. Considering that a query reply can potentially contain many thousands of tuples, sending/receiving and verifying individual tuple sig-

[2] A DSA signature is at least 40 bytes (320 bits) long, but verification of a DSA signature is more expensive computationally (It takes 0.16 msec to verify a RSA signature whereas it takes 8.52 msecs to verify a DSA signature on a P3-977 MHz machine).

[3] The enveloping tree is the smallest subtree within the VB-tree that envelops all the result tuples of the query

natures can be prohibitively expensive for the querier. Further, there is no easy way to provide a proof of completeness either.

In the remainder of this section, we develop modifications and enhancements that address the drawbacks of the naïve strategy described above.

REMARK: If the outsourced data is *static* or *archival* in nature, correctness and completeness can be provided easily, as described in Appendix A. However, in this paper, we focus on the more general (and challenging) case of dynamic databases.

4.1 Correctness

The ideal VO for providing correctness would involve minimal querier computation overhead and constant (in terms of integrity information) querier bandwidth overhead. The work in [15] proposed two signature schemes that enable such ideal (or near-ideal) solutions. These signature schemes allow us to aggregate multiple individual signatures into one *unified* signature, verifying which is *equivalent* to verifying ALL individual component signatures. The size of the aggregated signature equals that of a single plain digital signature (which is constant), irrespective of either the database size or the query reply size. In the ODB model, when the server receives a query, it executes the query to obtain the tuples matching the query predicate as well as their corresponding signatures. The server securely "combines" these individual signatures to obtain an aggregated signature and returns the result set comprising of the tuples as well as the aggregated signature. When the querier receives the set of tuples along with the aggregated signature it simply verifies the latter.

The first signature scheme proposed in [15] is the Condensed-RSA signature scheme. Condensed-RSA is based on the well-known RSA public key signature scheme. Condensed-RSA allows aggregation of a single signer's signatures which is possible due to the fact that RSA is *multiplicatively homomorphic*. The second is the Aggregated-BGLS scheme which allows signatures produced by multiple signers to be aggregated into a single quantity. The Aggregated-BGLS scheme is due to Boneh, et al.[5] and is *additively homomorphic*. The technical details of the two signature schemes (as well as a discussion of their ability to provide efficient proofs of correctness) can be found in Appendix B.

4.2 Completeness

Both signature schemes in [15] offer efficient proofs of correctness, however, they provide no completeness guarantees. In this section, we propose some extensions to achieve query completeness. To achieve this goal, we propose secure linking of tuple-level signatures to form a so-called *signature chain*. In order to construct the signature chains, we modify the tuple signature generation algorithm in the following way:

DEFINITION 1. *Signature of a tuple r is computed as:*

$$Sign(r) = h(h(r)||h(IPR_1(r))||\dots h(IPR_l(r)))_{SK}$$

where $h()$ is a cryptographic hash function such as SHA, $||$ denotes concatenation, IPR_i denotes the immediate predecessor tuple along dimension i, l is the number of searchable dimensions of that relation and SK is the private signing key of the data owner.

The immediate predecessors of a tuple are computed as follows: (1) Sort the tuples in increasing order along each searchable dimension (i.e., according to the attribute value for each searchable attribute); (2) The immediate predecessor of a given tuple along a given dimension is a tuple with the highest value for that attribute that is less than the value of the given tuple (highest lower bound) along that attribute.[4] Thus, each tuple has as many immediate predecessors as there are searchable attributes (dimensions). In other words, each tuple has l *Immediate Predecessors* where l is the number of searchable dimensions along which a query can be issued.

Thus in order to provide completeness, a tuple signature is computed by including the hashes of all immediate predecessor tuples, thereby explicitly chaining (linking) the signatures. We illustrate this with an example in figure 2. Suppose that there are three searchable dimensions. First, the tuples are sorted along each dimension. Consider tuple R_5. According to the figure, the immediate predecessors of R_5 along dimensions A_1, A_2 and A_3 are: R_6, R_2 and R_7, respectively. Now, compute the signature of R_5 as:[5]
$Sign(R_5) = h(h(R_5)||h(R_6)||h(R_2)||h(R_7))_{SK}$

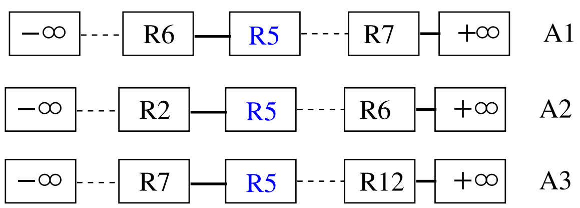


Figure 2: Signature Chain

With signatures chained in the above fashion, the server answers a range query by releasing all matching tuples, the *boundary tuples* which are just beyond the query range (to provide a proof of completeness) as well as the aggregated signature corresponding to the result set. The signature chain proves to the querier that the server has indeed returned all tuples in the query range. Specifically, the querier verifies that the values in the boundary tuples are just beyond the range posed in the query. At the same time, the querier verifies that there are no other tuples in between the boundary tuples and the immediately adjacent tuples which fall within the query range. This is because the boundary values are linked to the first and the last tuple. Therefore, the querier obtains a concise proof of completeness when the server releases the boundary tuples as defined above.

Empty Proofs: For range (or exact value) queries that result in no matches, the server composes an *Empty Proof* by returning only adjacent boundary tuples that subsume the non-existent value or range.

[4] If the attribute values of two tuples are the same, it is necessary to use an additional mechanism (for example: use the tuple id) to break the tie.

[5] The signature scheme here can be either condensed-RSA or aggregated-BGLS. Therefore, in an effort to keep the discussion general, we do not specify the details of the SIGN protocol

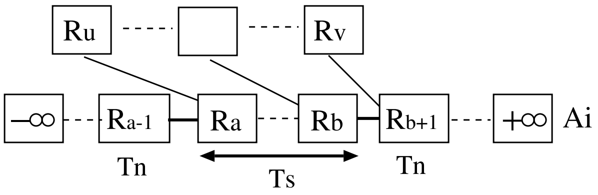
5. OPERATIONAL DETAILS

We now describe the overall procedure for computing authentic replies to queries that guarantee correctness and completeness using our approach outlined above. We first consider base level relational queries such as selections, projections, joins and set operations. We then describe how to handle dynamic outsourced databases that involve additional update operations, such as tuple inserts and deletes.

5.1 Selection:

A selection query $\sigma_C(R)$ is denoted as follows: $\sigma_C(R) = \{t | t \in R \text{ and } C(t)\}$ where R is a relation, C is a condition of the form $A_i \theta c$, A_i is an attribute of R, c is a constant value and $\theta \in \{=, \neq, <, \leq, >, \geq\}$

Given a selection query, the server computes a result set which is a set of contiguous (along that dimension) tuples. (It could also be an empty set.) Below, we outline our technique for composing a VO for selection queries for a relation R where the predicate condition C is on attribute A_i. Please note that although we use a single attribute in the selection predicate in the example below, it is straight-forward to use our scheme for selection query predicates involving multiple attributes.



The server composes the query reply as follows:

1. computes the tuple set T_s consisting of all the tuples that match the query posed. $T_s = \{R_a, \ldots, R_b\}$

2. computes the set T_n consisting of immediate predecessor and successor nodes of the first and last nodes respectively along the search dimension (i.e., the boundary tuples). $T_n = \{R_{a-1}, R_{b+1}\}$. These values are required to prove completeness. We note that the server needs to release only the relevant attributes' value in plain text and simply send the hashes of the remaining attributes. We assume that the relation R has r attributes $\{A_1, ..., A_r\}$ and C is a condition on attribute A_i. In this case, the server only needs to reveal $R_{a-1}.A_i$ and $R_{b+1}.A_i$ in plaintext and send the hashes $h(A_j)$ for the other $(r-1)$ attributes of R_{a-1} and R_{j+1}. Thus it is possible to prevent exposure of data (i.e., pertaining to the tuples that are beyond the left and right boundaries of the query result) to potentially unauthorized queriers.

3. obtains the corresponding signatures (either RSA or BGLS) $\{Sign(R_a), \ldots, Sign(R_{b+1})\}$ [6]

4. aggregates individual signatures to form a unified (aggregated) signature:
$\sigma = Aggregate(Sign(R_a), \ldots, Sign(R_{b+1}))$

5. for each tuple in T_s and tuple R_{b+1}, collects the hashes of immediate predecessor tuples along all other searchable dimensions $\{A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_l\}$, where l is the number of searchable attributes. (Recall that the signature of a tuple is computed on the contents of the tuple in conjunction with the contents of the immediate predecessor nodes along all the searchable dimensions.) Then for each tuple R_i, server computes 2 values:
$H_1(R_i) = h(IPR_1(R_i))|| \ldots h(IPR_{i-1}(R_i)))$ and
$H_2(R_i) = h(IPR_{i+1}(R_i))|| \ldots h(IPR_l(R_i)))$
Therefore,
$T_m = \{H_1(R_a), H_2(R_a), \ldots, H_1(R_{b+1}), H_2(R_{b+1})\}$
Specifically, the size of T_m is $((l-1)*(b+1-a)*|hash|)$ where $|hash|$ is the hash value of each of these tuples and is usually 160 bits long. Thus the VO comprises of $\{T_s, T_n, T_m, \sigma\}$

[6]Note that it is necessary to include $Sign(R_{b+1})$ to check for completeness. However, $Sign(R_{a-1})$ is not required since hash of R_{a-1} is included in $Sign(R_a)$.

5.2 Join:

A basic join operation $R \bowtie_C S$ involves two relations R and S where C is a condition of the form $A_i \theta A_j$, A_i and A_j are attributes of relation R and S respectively and $\theta \in \{=, \neq, <, \leq, >, \geq\}$. Both AuthDS and VB-tree approaches assume that all join queries are known *a priori* and require *additional* pre-computed B-trees to ensure authentication.

In the discussion that follows, we focus mainly on the equi-join operation, i.e., where θ is the equality predicate. Given a query of the type $R \bowtie_{A_r = A_s} S$, proving correctness is relatively simple using our approach. The server executes the join query and computes the list of tuples ($t \in R$ and $s \in S$) that match the equality predicate and obtains the corresponding signatures of t and s from R and S respectively. Server combines all individual signatures of tuples in the result set to compute the aggregated signature of the entire result set. Note that the aggregated signature is sufficient to prove correctness.

However, proving completeness of a join query is not straightforward. The querier needs to be assured that all tuples matching the equality predicate from R and S are present in the result set T_s. One way, albeit quite inefficient, to accomplish this is to pick the smaller relation (say S) and for each tuple s (or each contiguous set of tuples) in the set $S - T_s$, show an empty proof that s (more precisely $s.A_s$) does not exist in R. Recall that an empty proof involves releasing two adjacent tuples i, j of relation R such that $(i.A_r < s.A_s < j.A_r)$ and the signature of j. Note that if the server needs to show empty proofs for m tuples, server, instead of releasing m individual signatures, aggregates the m signatures into a single condensed/aggregated signature. Such a proof is clearly linear in the size of S. It remains an interesting open problem to modify the signature chaining mechanism to yield efficient completeness proofs which are linear in the size of the result set for arbitrary Join queries.

Using DSAC approach, it is possible to construct more efficient proofs of completeness if the join queries are known *a priori*. Then, while computing the signature of a tuple that is part of a join query result set, the hash of its immediate predecessor which is also in result set of the same join query is included in the tuple signature. This creates an explicit signature chain corresponding to the join query. Now, when a pre-computed $A \bowtie B$ query is executed, the server simply sends an aggregated signature that represents the signature chain of $A \bowtie B$. Note that, unlike the other two approaches, pre-computing a join query in our approach does not entail additional storage overhead.

5.3 Set Operations:

A set operation involves two relations U and V. Each relation is assumed to have its own signature chains.

- Union: $T_s = U \cup V$
  Providing proofs of correctness and completeness is straightforward: aggregate individual signatures for all tuples of U and all tuples of V to obtain a single signature for $U \cup V$; if U and V are intermediate results of a query evaluation or subsets of some other relations R and S, collect boundary tuples for U and V; finally construct the VO as described above for selection queries.

- Intersection: $T_s = U \cap V$
  To prove completeness and correctness, the server needs to convince the querier that each tuple in T_s is present in both U and V. Our approach is similar to that of AuthDS: the server picks the smaller of the two sets (say U) and for each element in $U - T_s$ the server sends back empty proof that that element (tuple) does not exist in V. This proof is linear in the size of U and shows that the result is correct and every element in $(U - (U \cap V))$ is not in V and hence complete.

5.4 Projections:

$\pi_L(R)$ is the projection of relation R onto the list L where L is typically a list of (some of the) attributes of R. $\pi_L(R) = \{< t.A_j, ..., t.A_k > | t \in R\}$ where A_i's are attributes of relation R. The result of the projection is a relation with a schema formed by the attributes on list L and the result is computed by examining every tuple in R.

In order to support projections, a tuple hash is computed as: $h(t) = h(h(t.A_1)||h(t.A_2)|| \ldots ||t.h(A_k))$. In other words, instead of hashing the entire tuple, we hash each attribute, concatenate the resulting hashes and hash them once again. Then, we compute a tuple signature of tuple as described in section 4.2. This way, the server needs to send only the hashes (instead of actual plaintext values) for each filtered attribute. Unfortunately, this basic solution is not very efficient in terms of bandwidth since it requires us to send individual hashes for each filtered attribute, for all tuples in the result set. (Note that it is necessary to send individual values to allow the querier to recompute the tuple signature since the tuple hash is computed by concatenating these individual hash values.) We propose further improvements to this basic solution to reduce the bandwidth overhead in the next section.

5.5 Database Updates:

Update operations involve the server and the data owner. An update operation typically requires recomputing some of the tuple signatures.

Insertion: We present the details of the multi-round protocol to insert a tuple when the data owner does not locally retain a copy of the database.

To insert a tuple r into table T (refer to figure 3), the owner sends the new tuple to the server. The server calculates the actual position of insertion along all l chains (where l is the number of searchable attributes) by examining the values of the individual attributes. The server computes the set of pairs of adjacent tuples $\{(R_{i_k}, R_{i_{k+1}})\}$ for inserting the new tuple, collects the signatures of all successor nodes $R_{i_{k+1}}$, aggregates these individual signatures to obtain σ and sends back these values (T_{ins}) to the data owner. T_h contains the additional hashes required to recompute the signatures of the successor nodes. [7]

Upon successful verification of σ, the owner computes the tuple signature for r by including the immediate predecessors' (i.e., all R_{i_k}) hash values and also updates the signature chains for the successor nodes (i.e., all $R_{i_{k+1}}$) by including r's hash value (along with the other appropriate hashes from T_h). The owner then sends back all $l+1$ new signatures T_{sig}. Thus inserting a new tuple into the database results in re-computing $l + 1$ signatures by the data owner.

Deletion: Performing a delete is similar to insert operation and is a multi-round protocol. Due to space restrictions, we only present a high-level description of the protocol. Data owner specifies the tuple(s) to be deleted based on some criterion. Server isolates parts of all the l signature chains that get affected by this operation and sends back sets of tuples that surround the tuple to be deleted back to the owner. The owner recomputes the signatures of the successor node of the node to be deleted, along each dimension, by replacing the hash of the node to be deleted with the hash of its predecessor along that dimension and returns the l new signatures back to the server.

6. ANALYSIS

We proposed a new **DSAC** scheme based on signature aggregation and chaining to provide authentication of query replies when the database is queried by remote clients over insecure public networks. In this section, we begin by analyzing the costs and overhead factors associated with our scheme and then compare its performance with AuthDS and VB-tree approaches.

We begin by summarizing the notation used in this section.

n	Total number of tuples in the relation
s	Number of tuples in the result set
t	Total number of attributes in the relation
l	Total number of searchable attributes; $1 \leq l \leq t$
$\|sign\|$	Size of a digital signature: 128 bytes for RSA, 64 bytes for BGLS
$\|hash\|$	Size of a hash. Default = 20 bytes

We now illustrate the bandwidth and computation advantages of DSAC over the naïve approach of sending and verifying individual tuple signatures. In our experiments, tuples are signed with the RSA signature scheme using a 1024-bit public modulus. The experiments were conducted on

[7]Note that each of the successor node $R_{i_{k+1}}$ has l "immediate predecessor nodes". When the predecessor along one dimension changes due to the new insertion, it becomes necessary to recompute the signatures of each of $R_{i_{k+1}}$. In order to do this, the hashes corresponding to the other $l - 1$ dimensions need to be sent back to the owner.

Owner | **Server**

$stmt$ = Insert tuple r into table T $\xrightarrow{stmt}$

$T_{ins} = \{(R_{i_k}, R_{i_{k+1}}) \forall i \in \{1, l\}, T_h, \sigma\}$

$R_{i_k}.A_i < r.A_i < R_{i_{k+1}}.A_i$

$\xleftarrow{T_{ins}}$

Verify σ,

$T_{sig} = Sign(r), Sign(R_{i_{k+1}}) \forall i \in \{1, l\}$

$\xrightarrow{T_{sig}}$

Update T with r and T_{sig}

Figure 3: Protocol to insert a new tuple into a table

a P3-977MHz Linux PC. We used the popular OpenSSL library[17] to implement all cryptographic functions. Figure 4(a) compares the time (in msec) for query verification using naïve approach with query verification with DSAC for varying size of the result set. We can see that signature aggregation greatly reduces the computational overhead required to verify the integrity of the result set.

Figure 4(b) contrasts measured bandwidth overhead for the naïve approach with that in DSAC. Recall that the naïve approach does not provide completeness guarantees. In DSAC, since the signatures are chained in order to provide completeness, we need to send additional hashes. Specifically, when the search predicate involves a particular attribute A_i, for each tuple in the result set, we need to send additional hashes corresponding to the immediate predecessor tuples along the remaining $(l-1)$ searchable attributes. We show the overhead for varying sizes of the result set (in records), for $l = 5$. It is easy to see that although DSAC incurs additional overhead to provide completeness, it still is much more bandwidth efficient than the naïve approach.

Next, we compare our scheme with AuthDS and VB-tree approaches.

Storage Costs: In AuthDS scheme, to obtain an efficient VO on the order of $O(log|n|)$ in size and, more importantly, to prove completeness of a range query, a separate B-tree for each search order is required. Therefore, for l searchable attributes, a total of l separate B-trees need to be pre-computed and stored at the server. Furthermore, to support other, more advanced queries, such as joins, the scheme requires separate data structures for each possible query. For example, to support $A \bowtie B$, the authors suggest pre-computing a tree corresponding to the cartesian product $A \times B$. Storing these trees can result in enormous storage overhead. Further, storing multiple trees for the same relation also increases the cost and complexity of the update operations since each update operation results in recomputing the tree hashes and the root signatures for all the trees and potentially some tree re-balancing operations.

In VB-tree scheme, each attribute value of a tuple is signed by the owner. Further, each tuple is also signed in its entirety. Finally, a single VB-tree is constructed per table where individual nodes of the tree are also signed. This incurs a substantial storage overhead of $O(n * t * |sign| + t * |sign|)$ in addition to the cost of storing the VB tree itself. Thus, VB-tree is significantly more expensive than DSAC in terms of storage. Furthermore, as with AuthDS approach, VB-tree requires separate pre-computed data structures in order to support Join queries.

In comparison, DSAC incurs fixed storage overhead of one signature per tuple irrespective of the number of searchable attributes or the number of queries to be supported. This low storage cost renders our scheme orders of magnitude more efficient as compared to both AuthDS and VB-tree approaches.

VO Size: In this analysis, we measure only the overhead and do not include the size of the actual result set. In AuthDS, the VO size for a selection/projection query can be expressed as: $VO_{size} = |s| \times \sum_i^k |hash| + (2\log|n| - 1) \times |hash| + |sign| + 2(|tuple|)$ where $\{A_i \ldots A_k\}$ are the filtered attributes of each tuple. $2(|tuple|)$ corresponds to 2 boundary tuples which are released to prove completeness and $|sign|$ corresponds to the size of the signature of the root. Note that $|s| \times \sum_i^k |hash|$ measures the hashes corresponding to filtered attributes and $(2\log|n| - 1) \times |hash|$ measures additional hashes that must be sent to re-compute the root of the B-tree.

In VB-tree, the VO size for a selection/projection query is: $VO_{size} = |s| \times \sum_i^k |sign| + (2\log|s| - 1) \times |sign|$ where $\log|s|$ is the height of the enveloping tree and $\{A_i \ldots A_k\}$ are the filtered attributes of each tuple. Note that this VO cost assumes that the search is being done on the primary key. In this case, a set of contiguous tuples is returned and the additional overhead is $O(log|s|)$ signed digests. However, if the search is on a non-primary key attribute, then the enveloping tree can become quite large (potentially, as large as the entire B-tree) and signed digests corresponding to all tuples that are not part of the result set need to be returned.

For the proposed DSAC approach, the VO size is expressed as: $VO_{size} = |sign| + |s| \times (\sum_i^k |hash| + \sum_1^{l-1} |hash|) + 2(|tuple|)$. We send back a condensed/aggregated signature to verify the correctness and completeness of the result set. Figure 4(c) shows the VO size overheads for the AuthDS, VB-tree and DSAC approaches. As can be seen from the figure, VB-tree approach incurs very high bandwidth overheads. DSAC approach is as efficient as the AuthDS approach while requiring the storage of a single signature per record.

Our scheme incurs an overhead of $(|s| \times \sum_1^{l-1} |hash|)$ for guaranteeing completeness. This is because we need to include the hashes of the immediate predecessor tuples along every searchable attribute while computing the signature of

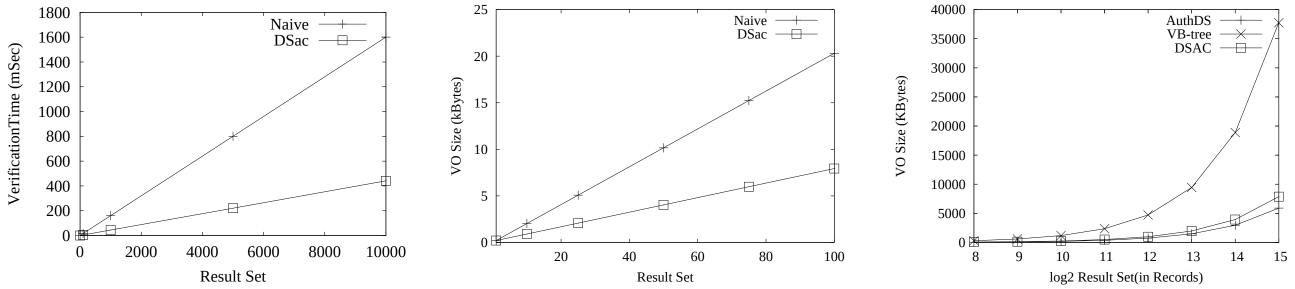


Figure 4: (a) Query Verification Costs compared to naïve, (b) Bandwidth Costs compared to naïve, (c) VO Size Costs compared to AuthDS and VB-tree

a tuple. One way to reduce this overhead would be to trade storage efficiency to gain bandwidth efficiency by using multiple signature chains. Furthermore, we can reduce the VO size while maintaining a single signature chain by utilizing secure hashing techniques described in [2, 3]. These techniques, which are proposed mainly in the context of Incremental Hashing, compute the hash of a message by breaking the message into smaller blocks and combining the hashes of individual blocks by using a "compression function". We skip the details but would like to mention that this family of hash functions can be adapted for use in our scheme in order to send back a single "compressed" hash for the tuple instead of sending one hash along each search dimension. Further, the same technique can also be used to reduce the bandwidth overhead associated with Projection queries. The detailed description of this technique is out of the scope of the current work.

Query Verification Costs: Query verification in both AuthDS and DSAC approaches involve computing simple hashes and combining them and verifying a single signature to verify the correctness and completeness of the result set. In comparison, VB-tree involves performing a number of signature verifications (since the scheme returns "signed" digests). Since signature verification is very expensive as compared to hashing, this scheme is computationally more expensive.

In summary, the proposed DSAC scheme is clearly more efficient in terms of computation, storage, bandwidth and also provides a richer set of features as compared to the VB-tree approach. When compared to AuthDS, DSAC is very efficient in terms of storage and is as efficient for VO size and verification costs. Both AuthDS and DSAC require expensive signature recomputations for tuple inserts and deletes. (For AuthDS, the roots of all the B-trees of that relation need to be recomputed by the owner whereas for DSAC, following an update operation, the signatures of l successor nodes need to be recomputed). However, as tuples are inserted and deleted over time, AuthDS involves additional intensive operations, such as re-balancing (one or more) b-trees in addition to re-calculating signatures for all roots.

7. FUTURE DIRECTIONS

Another desired property of ODB integrity is to ensure *freshness* of query replies. *Freshness* means the assurance that the query reply was generated with respect to the most recent snapshot of the database. (For static databases this requirement is trivially met.) Below, we outline one possible mechanism that aids in preventing the server from *replaying* old signature chains. To provide freshness, we can use a single Merkle Hash Tree – referred to as an FTree – for the entire relation. The root of the FTree is signed by the data owner and is assumed to be published and/or sent to all the queriers. When a query is posed, the server, in addition to the result set, provides the path from the result set to the root of the FTree and reveals the root signature. Upon receiving a query reply, the querier can verify freshness by recomputing the root of the FTree and validating it by verifying the owner's signature. The signature of the root is refreshed periodically (by the owner) in accordance with a system-wide freshness policy, thus ensuring that the data is fairly recent. As part of our future work, we plan to study this problem in depth. We also plan to conduct a detailed study of the applicability of our approach to other more advanced query types.

8. CONCLUSIONS

This work explored the problem of authenticity and integrity of query replies in outsourced databases. In particular, we developed a new approach (DSAC) based of signature aggregation and chaining which achieves authentication of query replies. The main contributions of this work are the proposed signature chaining mechanism which provides evidence of completeness of query result set and the analysis which sheds light on the applicability of our scheme for various query types in the relational model. We also compared our approach to the state-of-the-art in authenticated publishing which is based on authenticated data structures.

9. REFERENCES

[1] M. Bellare, J. Garay, and T. Rabin. Fast batch verification for modular exponentiation and digital signatures. In *Eurocrypt 1998*.

[2] M. Bellare, O. Goldreich, and S. Goldwasser. Incremental cryptography and application to virus protection. In *27th Annual Symposium of Theory of Computing*, 1995.

[3] M. Bellare and D. Micciancio. A new paradigm for collsion-free hashing: Incrementality at reduced cost. In *EUROCRYPT '1997*.

[4] M. Bellare and P. Rogaway. Random oracles are practical: a paradigm for designing efficient protocols. In *ACM Press*, pages 62–73, 1993.

[5] D. Boneh, C. Gentry, B. Lynn, and H. Shacham. Aggregate and Verifiably Encrypted Signatures from Bilinear Maps. In *EUROCRYPT '2003*, 2003.

[6] 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*, pages 101–112, 2000.

[7] H. Hacigümüş, B. Iyer, C. Li, and S. Mehrotra. Executing SQL over Encrypted Data in the Database-Service-Provider Model. In *SIGMOD*, 2002.

[8] H. Hacigümüş, B. Iyer, and S. Mehrotra. Encrypted Database Integrity in Database Service Provider Model. In *CSES'02 IFIP WCC*, 2002.

[9] H. Hacigümüş, B. Iyer, and S. Mehrotra. Providing Database as a Service. In *ICDE*, 2002.

[10] B. Hore, S. Mehrotra, and G. Tsudik. A Privacy-Preserving Index for Range Queries. In *VLDB*, 2004.

[11] A. Joux and K. Nguyen. Separating Decision Diffie-Hellman from Diffie-Hellman in cryptographic groups. In *Cryptology ePrint Archive*, number 2001/003, 2001.

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

[13] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of applied cryptography.* CRC Press, 1997. ISBN 0-8493-8523-7.

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

[15] E. Mykletun, M. Narasimha, and G. Tsudik. Authentication and Integrity in Outsourced Databases. In *Network and Distributed Systems Security*, 2004.

[16] National Institute of Standards and Technology (NIST). Secure Hash Standard. FIPS PUB 180-1, April 1995.

[17] OpenSSL Project, http://www.openssl.org.

[18] H. Pang and K-L Tan. Authenticating Query Results in Edge Computing. In *ICDE*, 2004.

[19] R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2), 1978.

APPENDIX

A. STATIC DATA

If the outsourced data is static or archival in nature, e.g., a census database, proofs of completeness can be provided quite easily as follows:

1. Sort all tuples in increasing order along each searchable dimension, i.e., according to the attribute value for each searchable attribute.
2. Compute a signature of each tuple by signing the "Running Hash" of all the tuples in the chain from the starting node to the current tuple as described below.

For example, assume that there is only one searchable dimension. The owner sorts tuples in ascending order along this dimension to obtain: $\{R_1, R_2, \ldots, R_n\}$. Owner then includes two boundary values: $(-\infty, +\infty)$ in the table and computes the signatures of R_1 through R_n as: $Sign(R_i) = h(R_i||h(R_{i-1}||\ldots||h(-\infty)\ldots))_{SK}$

At the end, it computes the signature of $+\infty$. The tuples and their signatures are stored at the server as before. Now, in order to prove both completeness and correctness of a range $\{R_i, R_j\}$, the server simply releases tuples $\{R_i, R_j\}$, running hash of R_{i-1}, and $Sign(R_j)$. Since the signatures are computed on running hashes, it can be easily seen that the reply set provides a concise proof of correctness and completeness. Note that, we do not require any signature aggregation in this scenario.

B. SIGNATURE AGGREGATION

B.1 Condensed-RSA

The RSA [19] signature scheme is multiplicatively homomorphic which makes it suitable for combining multiple signatures generated by a single signer into one *condensed* signature.[8] A valid condensed signature signifies to the verifier that each individual signature contained in the condensed signature is valid, i.e., generated by the purported signer. Aggregation of single-signer RSA signatures can be performed incrementally by anyone in possession of individual RSA signatures. By incrementally, we mean that the signatures can be combined in any order and the aggregation need not be carried out in a single operation.

RSA Signature Scheme: We first describe the setup of the standard RSA signature scheme. A party has a public key $pk = (n, e)$ and a secret key $sk = (n, d)$, where n is a k-bit modulus formed as a product of two $k/2$-bit primes p and q. Both public and private exponents $e, d \in Z_n^*$ and satisfy $ed \equiv 1 \mod \phi(n)$, where $\phi(n) = (p-1)(q-1)$. The minimum currently recommended k is 1024. The security of the RSA cryptosystem is based on the conjectured intractability of the large integer factorization problem.

In practice, an RSA signature is computed on the hash of an input message. Let $h()$ denote a cryptographically strong hash function (such as, SHA-1) which takes a variable length input m and produces a fixed-length output denoted as $h(m)$. A standard RSA signature on message m is computed as: $\sigma = h(m)^d \pmod n$. Verifying a signature involves checking that $\sigma^e \equiv h(m) \mod n$. Both signature generation and verification involve computing one modular exponentiation.

Condensed-RSA Signature Scheme: Given t different messages $\{m_1, ..., m_t\}$ and their corresponding signatures $\{\sigma_1, ..., \sigma_t\}$ generated by the same signer, a Condensed-RSA signature is computed as the product of all t individual signatures: $\sigma_{1,t} = \prod_{i=1}^{t} \sigma_i \pmod n$

The resulting aggregated (or condensed) signature $\sigma_{1,t}$ is of the same size as a single standard RSA signature. Verifying an aggregated signature requires the verifier to multiply the hashes of all t messages and checking that: $(\sigma_{1,t})^e \equiv \prod_{i=1}^{t} h(m_i) \pmod n$

Security of Condensed-RSA: [15] describes the security of Condensed-RSA by demonstrating that it is at least as secure as Batch verification of RSA [1]. Batch verification of RSA signatures was shown to be secure (in [1]) under the assumption that RSA is a collection of one-way functions. The proof assumes that the individual RSA signatures are generated using a full domain hash function (FDH) in place of a standard hash function (such as SHA-1) as described in [4].

B.2 BGLS

Boneh, et al. in [5] construct an interesting aggregated signature scheme that allows aggregation of signatures generated by multiple signers on different messages into one short signature based on elliptic curves and bilinear mappings. This scheme (BGLS) operates in a Gap Diffie-Hellman group (GDH) – a group where the Decisional Diffie-Hellman problem (DDH) is easy while the Computational Diffie-Hellman problem (CDH) is hard. The first instance of such a group was illustrated in [11]. Prior to describing the BGLS signature scheme, we briefly overview the necessary parameters [5].

- G_1 is a cyclic additive group with generator g_1

[8]We use the term *condensed* in the context of a single signer and *aggregated* in the context of multiple signers. Clearly, former is a special case of the latter.

- G_2 is a cyclic multiplicative group
- e is a computable bilinear map $e : G_1 \times G_1 \rightarrow G_2$ as described below

A bilinear mapping $e : G_1 \times G_1 \rightarrow G_2$, where $|G_1| = |G_2|$, satisfies the following two properties. (1) Bilinearity: $\forall P, Q \in G_1$ and $a, b \in \mathbb{Z}$, $e(aP, bQ) = e(P, Q)^{ab}$, (2) Non-degenerativity: $e(g_1, g_1) \neq 1$

BGLS Signature Scheme BGLS requires the use of a full-domain hash function $h() : \{0,1\}^* \rightarrow G_1$ that maps binary strings to non-zero points in G_1. Key generation involves picking a random $x \in Z_p$, and computing $v = xg_1$. The public key is $v \in G_1$ and the secret key is $x \in \mathbb{Z}_p$. Signing a message m involves computing $H = h(m)$, where $H \in G_1$ and $\sigma = xH$. The signature is σ. To verify a signature one needs to compute $H = h(m)$ and check that $e(\sigma, g_1) = e(H, v)$.

BGLS Aggregated Signature Scheme To aggregate t BGLS signatures, one computes the point-addition operation (on the elliptic curve) of the individual signatures as follows: $\sigma_{1,t} = \sum_{i=1}^{t} \sigma_i$, where σ_i corresponds to the signature of message m_i. The aggregated signature $\sigma_{1,t}$ is of the same size as a single BGLS signature, i.e., $|p|$ bits. Similar to Condensed-RSA, aggregation can be performed incrementally and by anyone. Verification of an aggregate BGLS signature $\sigma_{1,t}$ involves computing the point-addition of all hashes and verifying that: $e(\sigma_{1,t}, g_1) = \prod_{i=1}^{t} e(H_i, v_i)$

Due to the properties of the bilinear maps, we can expand the left hand side of the equation as follows:

$$e(\sigma_{1,t}, g_1) = e(\sum_{i=1}^{t} x_i H_i, g_1) = \prod_{i=1}^{t} e(H_i, x_i g_1) = \prod_{i=1}^{t} e(H_i, v_i)$$