Access Control and Query Verification for Untrusted Databases

Rohit Jain, Sunil Prabhakar

Access Control and Query Verification for Untrusted Databases

Rohit Jain and Sunil Prabhakar

Department of Computer Sciences, Purdue University
West Lafayette, IN, USA
{jain29,sunil}@cs.purdue.edu

Abstract. With the advent of Cloud Computing, data are increasingly being stored and processed by untrusted third-party servers on the Internet. Since the data owner lacks direct control over the hardware and the software running at the server, there is a need to ensure that the data are not read or modified by unauthorized entities. Even though a simple encryption of the data before transferring it to the server ensures that only authorized entities who have the private key can access the data, it has many drawbacks. Encryption alone does not ensure that the retrieved query results are trustworthy (*e.g.*, retrieved values are the latest values and not stale). A simple encryption can not enforce access control policies where each entity has access rights to only a certain part of the database. In this paper, we provide a solution to enforce access control policies while ensuring the trustworthiness of the data. Our solution ensures that a particular data item is read and modified by only those entities who have been authorized by the data owner to access that data item. It provides privacy against malicious entities that somehow get access to the data stored at the server. Our solutions allow easy change in access control policies under the lazy revocation model under which a user's access to a subset of the data can be revoked so that the user can not read any new values in that subset of the data. Our solution also provides correctness and completeness verification of query results in the presence of access control policies. We implement our solution in a prototype system built on top of Oracle with no modifications to the database internals. We also provide an empirical evaluation of the proposed solutions and establish their feasibility.

Keywords: Access Control, Cloud Computing, Query Verification, Private Outsourcing

1 Introduction

Access control mechanisms are an important part of a database system with which the data owner limits a user's access to a subset of the data. In a typical setting, the database server enforces access control policies by rewriting user queries to limit access to the authorized subset. When the data owner wants to revoke or grant a user, access to a certain part of the data, the data owner does that by informing the server. Traditionally, the server is assumed to be

trustworthy and the data owner assumes that the access control policies will be faithfully enforced by the server. However, this assumption is not reasonable when the database is hosted at a third-party server, *e.g.*, cloud, as the data owner lacks control over the hardware and software running at the server. Even when the server is trusted, there is a threat from a malicious insider or an intruder.

Another important problem that arises when the database systems are hosted at an untrusted server is to verify the trustworthiness of query execution. Much work has been done [1,2,3,4] towards verifying correctness and completeness of query results. However, most of these solutions do not work in the presence of access control rules, as they leak information that is outside the query range and outside the scope of the user's authorization.

In this paper, we provide solutions that ensure that a data item in the database is read and modified only by authorized users, and none other (including the server). The data encrypted by our solution is still queriable. Our solution provides mechanisms to verify the trustworthiness of query results in the presence of access control rules. For this, we extend our previous work [1] on ensuring the trustworthiness of data retrieved from an untrusted database that can be modified by multiple entities. The contributions of this work are:

- A novel mechanism to enforce access control rules without trusting the server
- Solutions that allow users to verify the correctness and completeness of query results in the presence of access control rules
- A demonstration of the feasibility of the solution through a *prototype in Oracle*, and its evaluation

The rest of this paper is organized as follows. Section 5 discusses some related work. Section 2 describes our model and presents some preliminary tools that are necessary for this work. Section 3 presents our solutions. A discussion of the implementation of the solution and an empirical evaluation is presented in Section 4. Finally, Section 6 concludes the paper.

2 Preliminaries

In this section, we start by explaining the different entities involved in our model. Then, we explain Merkle Hash Trees and Merkle B+ Tree which we use for building our solutions, and also discuss their use to verify the correctness and completeness of query results.

2.1 Model

There are three main entities involved: **Alice**, the database *owner*; **Bob**, the *(untrusted) database server* that will host the database; and **Carol**, the user(s) that

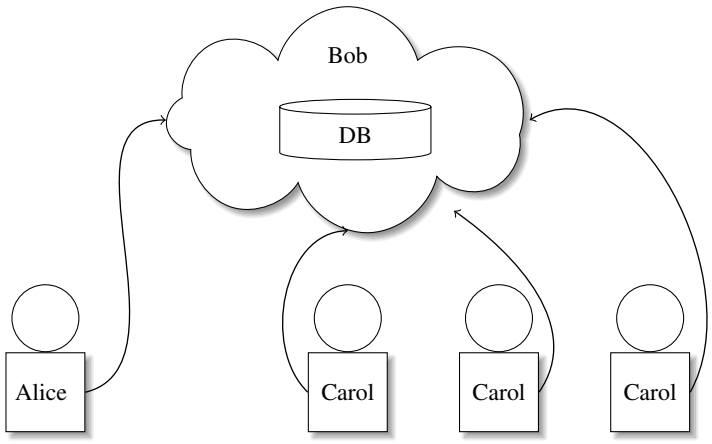


Fig. 1. The various entities involved: The database owner (Alice); The database server(Bob); and Authorized users (Carol).

will access this data (may include Alice) from the server. Users are authorized by Alice and can independently authenticate themselves with the server. A user can read or write data to the parts of the database she is authorized to. Figure 1 shows the various entities in this model.

Alice wants to ensure that the data are accessed by only those entities that were authorized by her. Alice and Carol want to ensure that the query results were indeed correct and complete in presence of access control policies.

An acceptable solution should allow Alice to grant or revoke access to a user at any point in time, without much work. The solution will disable Bob from being able to read the encrypted data. However, Alice and Carol should still be able to execute queries and run updates on the encrypted data.

Note that our assumptions about Bob are minimal. In most settings, the server is likely to be at least semi-honest – *i.e.*, it will not maliciously compromise data privacy by not following access control rules, or compromize data integrity by maliciously modifying the data or query results. However, due to poor implementation, failures, over commitment of resources, or other reasons, some loss of data or breach of privacy may occur. Given the lack of direct control over the server, Alice should not assume that Bob is infallible.

Lazy Revocation Model: As mentioned before, simple encryption can ensure that only authorized users can read or write the data. However, this introduces many problems. One such problem is related to dynamic access control rules. In a simple encryption method, when a user's access is revoked from a subset of the data, the data have to be re-encrypted. This can be a very costly process due to network usage and computation for encryption. To alleviate this burden, we consider the *Lazy Revocation Model*. Under this model, when a

Table 1. Symbol Table

Symbol	Description
t_i	the i^{th} tuple of a relation
$h(x)$	the value of a one way hash function over x
$\Phi(n)$	label of node n in MB-tree or MHT
H_i	label of the i^{th} node in the MB-tree
$a\|\|b$	concatenation of a and b
VO	a verification object
$Proof$	root label of the MB-tree
R	a set of ranges that partitions the data
r_i	$r_i \in R$
S_i, K_i	State and key for range r_i
$Enc_k(x)$	encryption of x using symmetric-key k
B_n	access control bitmap for node n

user is granted access to a subset of the database, the user can read or write to that subset. If the user's access is revoked from that subset, the data are not re-encrypted immediately. Instead, the new values in that subset are encrypted with a new version of the key so that the evicted user can no longer read the new values in that subset. Since the user had access to the old data before eviction, it can be assumed that the user had cached that data, hence it is not important to re-encrypt old values. We will consider the lazy revocation model for access control policies.

2.2 Correctness and Completeness

We begin by discussing the use of Merkle Hash Trees (MHT) to prove correctness. And then further discuss a variant, the MB-tree, which we use to prove completeness. We will use an MB-tree as a building block for our overall solution. Correctness requires that any data item in the query result are indeed part of the database and is not a fabricated value. An MHT can be used to establish the correctness of query results. An MHT is a binary tree with labeled nodes. We represent the label for node n as $\Phi(n)$. For an internal node, n, with children n_{left} and n_{right}, $\Phi(n)$ is defined as:

$$\Phi(n) = h(\Phi(n_{left})||\Phi(n_{right})) \tag{1}$$

where $||$ is concatenation and h is a one-way hash function. Table 1 explains the symbols used in this paper. Labels for leaf nodes are computed as the hash of the tuple value represented by that leaf. The root label is called 'Proof'.

Initially, an MHT is created on top of the database table. Alice stores only the root hash value ($Proof$) to authenticate future query results. To prove the correctness of a tuple, *i.e.*, to verify that a tuple existed in the database, Alice

Table 2. Sample Data Table

tupleID	A
1	23
2	29
3	35
4	48
5	59
6	63
7	65
8	70

Table 3. Bucketized Data Table

tupleID	A^*	Enc(A)
1	[20-30)	$Enc_{K_1}(23)$
2	[20-30)	$Enc_{K_1}(29)$
3	[30-40)	$Enc_{K_1}(35)$
4	[40-50)	$Enc_{K_2}(48)$
5	[50-60)	$Enc_{K_2}(59)$
6	[60-70)	$Enc_{K_2}(63)$
7	[60-70)	$Enc_{K_3}(65)$
8	[70-80)	$Enc_{K_3}(70)$

can ask Bob for some extra data (called *Verification Object (VO)*) from the MHT and recompute the root hash label. If the computed root hash label is the same as that she stored initially, she is convinced about the correctness of the tuple.

Completeness requires that all data items that should have been part of the query result are indeed present in the query result. Correctness and completeness combined establish the correntness of read-only queries. MHT can be extended to use B+ trees instead of a binary tree [2]. MB-trees can be used to verify both correctness and completeness. To prove completeness of a range query, Bob provides extra data with which Alice can verify that tuple values just preceeding, and just following (in sorted order) the query results were indeed outside the query range. Alice can also verify that no data is missing from the query result and the returned values are indeed part of the database. For more details, please refer to [1,2]. Figure 2 shows a sample MB-tree structure built on the attribute A of Table 2.

As an example, consider a query $\sigma_{30<A<60}$. The result for this query would include t_3, t_4, and t_5. To verify the correctness and completeness of the query results, the server sends *VO* to the user which includes the tuples just preceeding and just following the query ranges (*i.e.*, t_2 and t_6). The *VO* also includes any node labels that are required to compute the root label (*i.e.*, $h(t_1)$ and H_7). Using *VO*, the user can generate the $Proof$. If the computed proof matches with the proof value computed by Alice, the user is assured that the query results were correct and complete.

Access Control: In the presence of access control rules, traditionally, the query range is divided into multiple parts to ensure that each sub range is accessible to the user. In that case, each sub range can be verified individually. However, for verification, the server has to reveal the tuples bordering each sub-range. These bordering tuples may not be accessible to the user, leading to information leakage. In the next section, we will discuss our proposed solutions to enforce access control rules while still allowing query verification and privacy from the server or hackers.

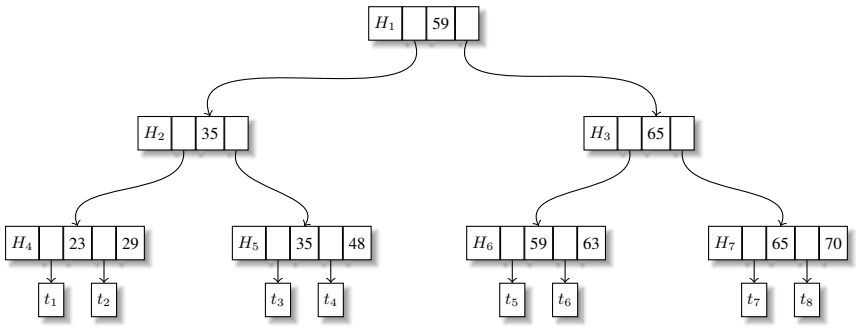


Fig. 2. An MB-tree on attribute A of Table 2

Updates: MHT or MB-tree work for static databases. When the data can be modified by users without prior knowledge of the data owner, as is the case in our model, MB-tree cannot be used directly. [1] proposes solutions with which authorized users can executed transactions at the server without being vetted by the data owner. These transactions can read or write data. This is done by engaging the server in a protocol that requires the server to declare the database state on which the transaction was executed and the database state that the transaction produced. The user can verify that the transaction read values from the declared consistent state to produce the next consistent state. However, this solution does not enforce access control rules. In the next section, we provide a solution that allows the user run and verify transactions in the presence of access control rules.

3 Access Control

As mentioned before, access control rules allow the data owner to restrict a user's access to a certain part of the database. The database owner may also want to hide the data from the server as well, while still allowing the users to read and query the data. The difficulty introduced by using access control rules is two fold. Firstly, verification algorithms have to be modified to assure the user that the partial database table visible to the user is indeed correct and complete. Secondly, the data have to be encrypted so the user sees only the allowed data, and, the data remain private from the server or an intruder. The server should still be able to run queries on this data. In this section, we provide our solutions to these problems.

In this paper, we consider fine-grained access control policies. We assume that the access control policies expose a user to a subset of each database table (this is the approach adopted by some commercial systems like Oracle VPD). In particular, we consider the following system for defining access control rules: $R = \{r_i | 0 \leq i \leq k\}$ is a set of ranges on an attribute that partitions the data into

k disjoint subsets. Each user is allowed access (read and write) to a part of the database table defined by a subset of R, *i.e.*, the user, Carol, can access tuples $\{t_i | t_i \in \cup r_{i_l}\}$, where $\{r_{i_l}\} \subset R$ is the set of ranges accessible to Carol.

3.1 Verification in Presence of Access Control

Given a range query, all tuples that satisfy the range query may not be accessible to the user. In such case, the verification using the regular MB-tree *VO* will not work. Also, verification of query results usually involves reading extra tuple values [2,4]. These tuples may not be accessible to the verifier due to the access control rules. In such case, the verifier will not be able to verify a query or a transaction. Suitable adjustments to the authentication data structures are required to enable the verification of a query in presence of access control rules.

To solve this problem, we modify the MB-tree as follows: Each node, n, is extended with an access control bitmap, B_n, in which the i^{th} bit is "on" if there is a tuple in the subtree that belongs to the range r_i. Node labels are computed using Equation 2.

$$\Phi(n) = h(B_{child\_1}||\Phi(n_{child\_1})|| \ldots ||B_{child\_k}||\Phi(n_{child\_k})) \tag{2}$$

The *VO* now contains the nearest tuple value just preceding and the nearest tuple value just following the query range such that these tuples are accessible to the user. *VO* also contains all the tree nodes required to prove the correctness of these tuples and to prove that the tuples that were left out of the query results were indeed inaccessible to the user.

As an example, consider the following access control ranges on attribute A:

r_1 : [0, 35]
r_2 : [36, 64]
r_3 : [65, 100]

Under these accesss control ranges, each access control bitmap will have three bits, one for each access control range. Figure 3 shows an augmented MB-tree, as described above, built on Table 2.

When a user who is authorized to access r_1 and r_3 executes a range query $\sigma_{25<A<50}$, tuple t_2 and t_3 will form the query result. Tuple t_4 will not be part of the query result as it is not accessible to the user. To verify the correctness and completeness of the query result, the user has to verify that the missing tuples were indeed inaccessible to her. The user will also have to verify that the nearest tuple just before the query range was indeed t_1 and the nearest tuple just following the query range (and also accessible to the user) is indeed t_7.

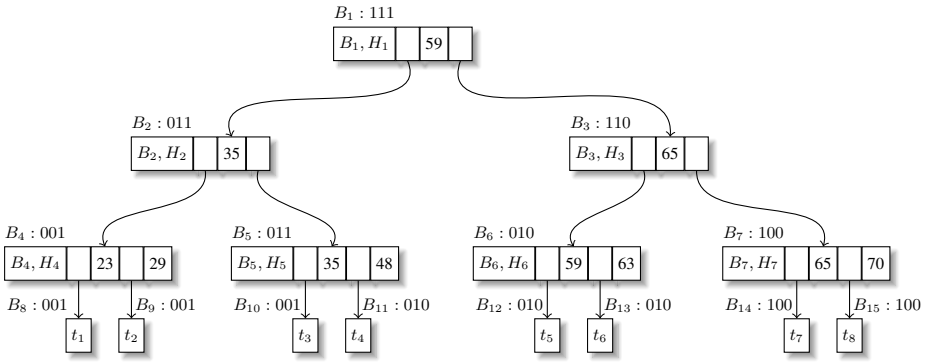


Fig. 3. Augmented MB-tree to allow Access Control

To prove the completeness of the query result, the *VO* of this query will include t_1 and t_7. To prove that the ommitted tuples (*i.e.*, t_4, t_5, and t_6) were indeed inaccessible to the user, the *VO* will also include the bitmaps B_6 and B_{11}. Using B_6, the user can be convinced that tuples t_5, and, t_6 were indeed inaccessible to her. Similarly, using B_{11} the user can be assured that Tuple t_4 was inaccessible to her. As in the case of regular MB-tree, the *VO* will include all other necessary labels required to calculate the root label.

3.2 Enforcing Privacy for Access Control

In this subsection, we present our solutions to encrypt the database, so that a user can read/write only the subset of the data that she has been authorized to access. The server (or any intruder) cannot read the data. As mentioned before, in this work we consider the *lazy revocation model*. Under this model, once a range, r_i, is removed from a user's accessible ranges, the future tuples in r_i are encrypted using a new key. All remaining and future users who can access r_i will be distributed the new key. Any pre-existing tuples in r_i are not necessarily re-encrypted with the new key. To decrypt the data in the range r_i, the user may need the current or previous keys of that range. Only the data owner decides which ranges are accessibe to the user.

The Key Regression scheme [5] provides a mechanism for versioning encryption keys for symmetric-key encryption. Given a version of the key, the user can compute all previous versions of the key. However, future versions of the key can not be derived using the current key. In the start, all data items in an access control range are encrypted using the first version of the key. Each user authorized to access the range is given that key. When a user is evicted from the range, the key is updated to a newer version. All future data items in the range are now encrypted using the new version of the key. Since the users cannot generate the new version of the key, the evicted user cannot read future tuples in the

range. A Key Regression scheme is defined using four algorithms. Algorithm *setup* is used by the data owner to setup the initial state. Algorithm *wind* is used to generate the next state. Algorithm *unwind* is used to derive the previous state, and *keyder* is used to generate the symmetric key for a given state. The tuples are encrypted using the symmetric key. We consider a particular key regression scheme that uses RSA to generate states.

Consider an RSA scheme with private key $< p, q, d >$, public key $< N, e >$, and security parameter k, such that p and q are two k-bit prime numbers, $N = pq$, and $ed = 1(mod\varphi(N)$ where $\varphi(N) = (p-1)(q-1)$. For each range r_i, a secrent random number $S_i \in Z_N^*$ is selected as the initial state.

Algorithm *wind*, *unwind* and *keyder* are defined in Algorithms 1, 2, and, 3 respectively.

Algorithm 1 wind (N, e, d, S_i)

$nextS_i = S_i^d \ (mod\ N)$
return $nextS_i$

Algorithm 2 unwind (N, e, S_i)

$prevS_i = S_i^e \ (mod\ N)$
return $prevS_i$

Algorithm 3 keyder(S_i)

$K_i = SHA1(S_i)$
return K_i

For each range, r_i, in range set R, the data owner generates a secret state $S_i \in Z_N^*$. The user stores the current states for each range it has access to. Using the current state of a range, the user can compute the corresponding symmetric-key to encrypt or decrypt the data in that range. Whenever, a range is added or removed from a user's accessible ranges, the state corresponding to that range is moved to the next state and all users who still have access to that range are informed about the new version of the state. If a tuple in a range is encrypted using a newer key, the user requests the new state from the data owner.

Due to encryption, the server cannot execute range queries. To be able to execute range queries on data, we use bucketization to divide the data into multiple buckets. Range queries are then suitably modified to search data among these bucket ranges.

Bucketization: Bucketization involves partitioning the attribute domain into multiple equi-width or equi-depth partitions. Attribute values are then converted from a specific value in the domain to the bucket labels. Table 3 is an example

of equi-width bucketization of Table 2 where each partition width is 10. Using equi-width bucketization reveals the density in each bucket. Equi-depth, on the other hand, requires frequent adjustments (which requires communication with the user) when database is updated frequently. [6,7] show that only limited information can be deduced due to bucketization.

As shown in Table 3, the attribute A^* represents the bucket labels after bucketization, and the encrypted tuple value is kept in a separate attribute. User queries are now executed on A^*. To verify the correctness and completeness of query results, our augmented MB-tree can be built on top of the bucketized field, A^*. The verification process will remain the same, except now tuples will be inserted in the tree according to A^*.

Thus, combining the solutions proposed in Subsections 3.1 and 3.2, the data owner and the users can be convinced that the data were not maliciously modified, and the data were accessed by the user that had appropriate authorizations.

4 Experiments

To demonstrate the feasibility and evaluate the efficiency of the proposed solutions, we implement our solutions on top of Oracle. The solutions are implemented in the form of database procedures using Pl/SQL and no internal modifications were done on the database. While we expect that the ability to modify the database internals or to exploit the index system will lead to a much more efficient implementation, our current goal is to establish the feasibility of our approach and to demonstrate the ease with which our solution can be adopted for any generic DBMS. Users are implemented using Python.

Setup: We create a synthetic database with one table *uTable* containing one million tuples of application data. *uTable* is composed of a table with two attributes ($TupleID$ and A). The table is populated with random values of A between -10^7 and 10^7. When tuples are encrypted, the ciphertext is stored in attribute $EncA$. Table 4 describes the different tables and indexes used in our prototype. An MB-tree is created on attribute A (integer). We consider three transactions implemented as stored procedures, namely *Insert*, *Delete*, and *Select*. *Insert* creates a new tuple with a given value of attribute A. *Delete* deletes the tuples which have a given value of attribute *A* and *Select* is a range query over attribute A. The experiments were run on an Intel Xeon 2.4GHz machine with 12GB RAM and a 7200RPM disk with a transfer rate of 3Gb/s, running Oracle 11g on Linux. We run Oracle with a strict serializable isolation level. We use a standard block size of 8KB.

[1] Used when supporting Access Controls

[2] Used when supporting Access Controls with Encryption

Table 4. Relations and Indexes in the database

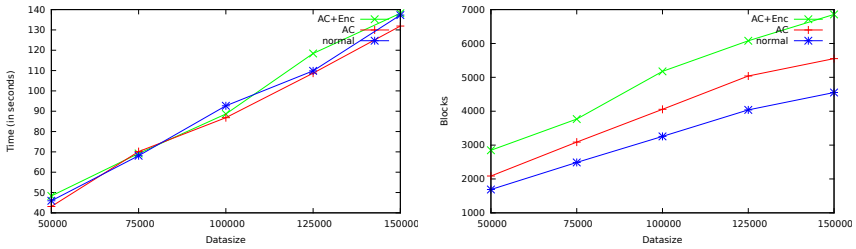
Table	Attributes	Indexes
uTable	TupleID, A, EncA[2], keyVersion[2]	A
uTableMBT	id, level, Label, keys, children, childLabels, key_min, key_max, accessBitmap[1], childAccessBitmap[1]	id, (key_min, key_max, level)
AccessControlRanges	id, key_min, key_max	
AccessControlRules	AcessControlRule_id, Use_id	

Implementation Details: The MB-tree has been implemented in the form of a database table – each node in the MB-tree is represented by a tuple in the MB-tree table (*uTableMBT*). Ideally, the MB-tree should be maintained as a B+ index trees of the database. However, that requires internal modifications to the index system of the database. We leave that for future work. Each MB-tree node, identified by a unique id, stores *uTable* tuples in the range $[key\_min, key\_max)$. $level$ denotes the height of the node from the leaf level, *i.e.*, leaf nodes have $level = 0$, and the root has the highest level. The $keys$ field stores the keys of the node, and the $children$ and $childLabels$ fields store the corresponding child ids and labels respectively. $Label$ stores the label of the node. When access control mechanisms are in place, two more attributes, $accessBitmap$ and $childAcessBitmaps$, are added to store the access control bitmap of the node and access control bitmaps of the child nodes respectively.

4.1 Results

We now present the results of our experiments. To provide a base case for comparison, we compare the performance of our solutions with a regular MB-tree based solution [2], where access control rules are not supported. This solution leaks information for transaction verification. Furthermore, this solution does not provide privacy against a malicious server. We analyze the costs of construction for the authentication data structures, execution of a transaction, and verification of a transaction.

The fanout for the authentication structure is chosen so as to ensure that each tree node is contained within a single disk block. In each experiment, time is measured in seconds, storage and IO is measured as the number of blocks read or written as reported by Oracle.The reported times and IO are the total time and IO for the entire workload. Each experiment was executed 3 times to reduce the error – average values are reported. In the plots, $Normal$ represents the solution from [2], AC represents our solution where access control bitmaps are added to the nodes to support access control rules, and $AC + Enc$ represents our solution that encrypts the tuple values and uses bucketization. AC and $AC + Enc$ both allow query verification in the presence of access control rules. $AC + Enc$ also



(a) Time vs data size (b) Blocks used vs data size

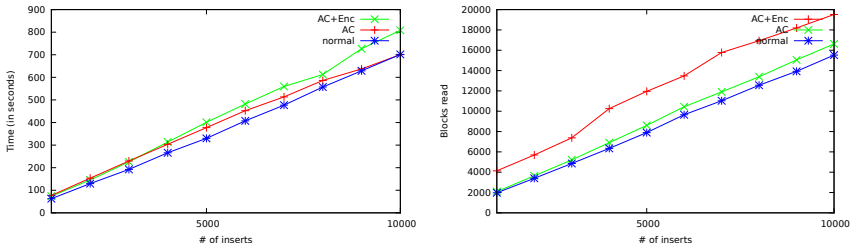
Fig. 4. Construction time and storage overhead

provides privacy against the server or an intruder. When bucketization is used, we divide the data into 1000 buckets. We use 200 access control ranges.

Construction Cost: First, we consider the overhead of constructing (bulk loading) the proposed data structures. To support access control rules, our solution requires augmenting MB-tree nodes with additional values that store the access control bitmaps. To provide privacy from the server, key regression is used that allows different versions of the encryption key. This requires storing additional attibutes to store the ciphertext and the key version. Figures 4(a) and 4(b) show the effect of data size on construction time and storage overhead, respectively. As expected, the storage cost is higher for our solutions. However, the construction time does not change significantly as the additional computation required for encryption is done by the user, keeping the computation cost for the server similar to just maintaining the MB-tree.

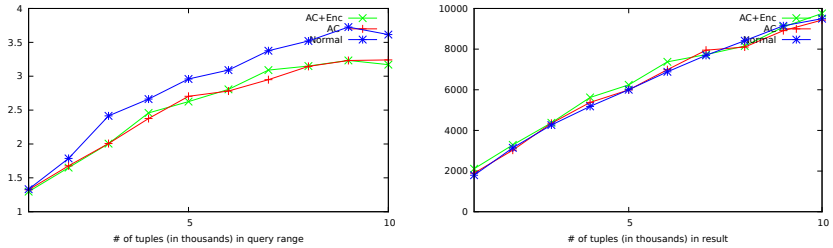
Insert Cost: We study the performance as the number of *Insert* transactions is increased. For this experiment no verification is performed. Figures 5(a) and 5(b) show the results. As expected, our solution incurs a higher overhead for IO as it requires keeping additional data. These costs increase linearly with the number of transactions. Surprisingly, this does not translate into a significant increase in the running time. This represents the computational overhead of hashing and concatenations which dominates the cost. Delete operation shows similar costs (not presented due to lack of space).

Search Cost: Search cost is influenced by both the size of the result (larger results will be more expensive to verify) and number of access control rules as that requires verifying that the tuples that were dropped from the query result were indeed not accessible to the user. To evaluate the performance of our solution for range queries (*Search*), we run 100 *Search* transactions for different ranges (thereby with different result set size) and verify all transactions. Figures 6(a) and 6(b) show the results. As the result set size increases, execution time and the amount of IO increase. For the regular MB-tree solution, the query range is divided into multiple sub-ranges based on access control rules. Each



(a) Time vs # of inserts (b) I/O vs # of inserts

Fig. 5. Insert time and I/O overhead



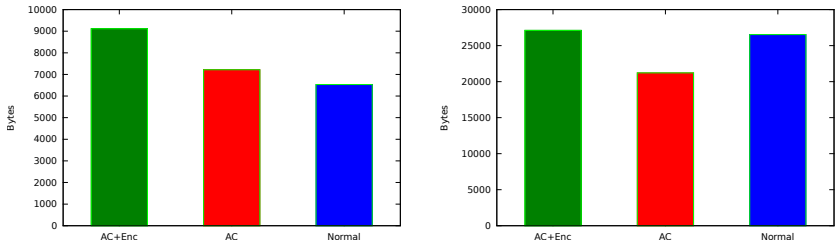
(a) Time vs # of tuples (b) IO vs # of tuples

Fig. 6. Cost of search+verification vs number of tuples in the result

sub-range that is accessible to the user is returned as query result. For verification, the server has to return the right and left most paths of each sub-range. However, in our solution, an access control bitmap is enough to verify that the sub-range is not accessible. This decreases the *VO* size and computation cost. As shown in the figure 6(a), our solution performs slightly better than MB-tree as our solution requires lesser *VO* size. As the result set size increases, the verification object size increases which results in an increase in verification time. The performance of our solution is comparable to that of an MB-tree alone.

Verification Cost: Our solution changes the *Verification Object* significant as our solution does not require bordering tuples outside the accessible range. However, since the node labels now include access control bitmaps, it increases the *VO* size. We now demonstrate the change in *VO* size in our solutions. To demonstrate the overhead of insert verification, we run 1000 *Insert* transactions and verify them. Average *VO* size is reported in figure 7(a). As expected, the *VO* size is higher for our solutions as it requires additional information, like access control bitmaps and key versions.

To demonstrate the overhead of search query verification on the system, we run 1000 *Search* transactions with varying ranges and varying access control ranges. The average *VO* size is reported in Figure 7(b). As discussed before, in a normal MB-tree, to support access control, a query range has to be divided



(a) VO Size for insert (b) VO Size for search

Fig. 7. Verification Object Size

into multiple sub-ranges so that the query accesses only the part of the data that are accessible to the user. For each sub-range, the *VO* includes the tuple just before and just following the sub-range. *VO* also includes all necessary nodes that are required to verify that the bordering tuples indeed existed the database. However, in our solution, this is not necessary. Each node contains information if the descendant tuples are accessible or not. Hence, *VO* does not always require the bordering tuples. Figure 7(b), that shows the effects of our solution on the *VO* size validates this. *VO* size for AC is smaller than the normal MB-tree. *VO* size for $AC + Enc$ is comparable to MB-tree. This is due to the ciphertexts.

Overall, we observe that our solutions are efficient and provide mechanisms for access control with reasonable overheads and perform better than current solutions in some cases.

5 Related Work

Much work has been done towards providing mechanisms to verify the correctness and completeness of query results from an untrusted database server [1,2,3,4,8,9]. While some of the earlier work only considered correctness of query results [8,10], later work consider both correctness and completeness [2,4]. Some of these work have also considered data updates from multiple sources [1,4]. [1] proposes a solution that uses Merkle B-Trees as authentication data structure, and allows multiple entities to independently run transactions on the untrusted database. Most of these works do not consider issues related to data privacy and access control. In these works, the user requires additional data items for verification, leading to information leakage.

Some work has been done towards providing verification for correctness and completeness in the presence of access control rules [11,12,13,14,15]. While [12] supports one-dimensional range queries and data updates, [11] supports multi-dimensional range queries and does not handle updates. Both these solutions do not provide privacy against the server. [13] provides a tree based

solution for verifying correctness of query results without information leakage. However, this solution does not provide mechanisms for verifying completeness. [14,15] focus on the access control problems with data authenticity for XML data. These solutions provide solutions for data privacy against users but not the server or an intruder. The server or any intruder would have full access to the data leading to breach of privacy.

[16] proposes solutions to provide privacy against the server. Data are encrypted before sending it to the server. The data are encrypted in such a way that user queries can still be executed on the encrypted data. However, this solution does not provide access control mechanisms. Much work has been done towards key management [17,18,19,5,20]. [5,20] consider the *lazy revocation model* under which following the revocation of user membership from a group, the content publisher encrypts future content in that group with a new cryptographic key and the new key is distributed to only current group members. The content publisher does not immediately re-encrypt all preexisting content since the evicted member could have already cached that content. [5] proposes a key derivation mechanism with which a user can derive old encryption keys using the current keys, however, it does not allow a user to derive future keys. When a user is evicted from the group, all future updates are encrypted using a newer version of the key. This saves a lot of computation and I/O cost whenever access control rules are changed. [17,18,19] propose key management solutions for access hierarchies. [17] proposes a solution to not only restrict a user's access to a subset of the data, but also restricts the user's access to a limited time.

In this paper, we propose solutions to solve both problems collectively – our solutions provide mechanisms to ensure trustworthiness of query results while ensuring that access control policies are enforced, and it also provides mechanisms for encrypting the data that ensures that a data item is accessed (read and/or write) by only those entities that were authorized to access it.

6 Conclusion

In this paper, we considered the problem of implementing access control policies on an untrusted database server, while ensuring that the query results are trustworthy. With our solution, the data owner can be assured that the data will be read by only those users that were authorized by her apriori. Furthermore, the data owner and the users can be assured of the trustworthiness of the query results without violating the access control policies. We demonstrate that the solutions can be implemented over an existing database system (Oracle) without making any changes to the internals of the DBMS. Our results show that the solutions do not incur heavy costs and are comparable to current solutions for

query verification (that do not support access control rules). We believe that the efficiency of the solutions can be further improved by modifying the internals and exploiting the index structures to get better disk performance. We plan to explore these issues in future work.

Acknowledgements: We thank Walid Aref for many discussions and valuable comments. The work in this paper is supported by National Science Foundation grants IIS-1017990 and IIS-09168724.

References

1. Jain, R., Prabhakar, S.: Trustworthy data from untrusted databases. In: ICDE. (2013)
2. Li, F., Hadjieftheriou, M., Kollios, G., Reyzin, L.: Dynamic authenticated index structures for outsourced databases. In: SIGMOD. (2006)
3. Mykletun, E., Narasimha, M., Tsudik, G.: Authentication and integrity in outsourced databases. In: NDSS. (2004)
4. Narasimha, M., Tsudik, G.: Authentication of outsourced databases using signature aggregation and chaining. In: DASFAA. (2006)
5. Fu, K., Kamara, S., Kohno, T.: Key regression: Enabling efficient key distribution for secure distributed storage. In: NDSS. (2006)
6. Ceselli, A., Damiani, E., Vimercati, S.D.C.D., Jajodia, S., Paraboschi, S., Samarati, P.: Modeling and assessing inference exposure in encrypted databases. TISSEC **8**(1) (February 2005)
7. Hore, B., Mehrotra, S., Tsudik, G.: A privacy-preserving index for range queries. In: VLDB. (2004)
8. Devanbu, P.T., Gertz, M., Martel, C., Stubblebine, S.G.: Authentic third-party data publication. In: DBSec. (2000)
9. Pang, H., Jain, A., Ramamritham, K., lee Tan, K.: Verifying completeness of relational query results in data publishing. In: SIGMOD. (2005)
10. Pang, H., Tan, K.: Authenticating query results in edge computing. In: ICDE. (2004)
11. Chen, H., Ma, X., Hsu, W., Li, N., Wang, Q.: Access control friendly query verification for outsourced data publishing. In: ESORICS. (2008)
12. Pang, H., Jain, A., Ramamritham, K., Tan, K.L.: Verifying completeness of relational query results in data publishing. In: SIGMOD. (2005)
13. Kundu, A., Bertino, E.: Structural signatures for tree data structures. In: VLDB. (2008)
14. Bertino, E., Carminati, B., Ferrari, E., Thuraisingham, B., Gupta, A.: Selective and authentic third-party distribution of xml documents. TKDE **16**(10) (October 2004)
15. Miklau, G., Suciu, D.: Controlling access to published data using cryptography. In: VLDB. (2003)
16. Hacigümüs, H., Iyer, B.R., Mehrotra, S.: Providing database as a service. In: ICDE. (2002)
17. Tzeng, W.G.: A time-bound cryptographic key assignment scheme for access control in a hierarchy. TKDE **14**(1) (January 2002)
18. Akl, S.G., Taylor, P.D.: Cryptographic solution to a problem of access control in a hierarchy. TOCS **1**(3) (August 1983)
19. Atallah, M.J., Frikken, K.B., Blanton, M.: Dynamic and efficient key management for access hierarchies. In: CCS. (2005)
20. Kallahalla, M., Riedel, E., Swaminathan, R., Wang, Q., Fu, K.: Plutus: Scalable secure file sharing on untrusted storage. In: FAST. (2003)