DBMask: Fine-Grained Access Control on Encrypted Relational Databases

Mohamed Nabeel #1, Muhammad I. Sarfraz *2, Jianneng Cao #3, Elisa Bertino #4

# *Dept. of Computer Science, Purdue University*
305 N. University Street, West Lafayette, IN, USA
[1] nabeel@cs.purdue.edu
[3] caojn@cs.purdue.edu
[4] bertino@cs.purdue.edu

* *Dept. of Electrical Engineering, Purdue University*
475 Northwestern Ave., West Lafayette, IN, USA
[2] msarfraz@purdue.edu

***Abstract*—For efficient data management and economic benefits, organizations are increasingly moving towards the paradigm of "database as a service" where their data are managed by a database management system (DBMS) hosted in a public cloud. However, data are the most valuable asset in an organization, and inappropriate data disclosure puts the organization's business at risk. Therefore, data are usually encrypted in order to preserve their confidentiality. Past research has extensively investigated query processing on encrypted data. However, a naive encryption scheme negates the benefits provided by the use of a DBMS. In particular, past research efforts do not have adequately addressed flexible access control on encrypted data at different granularity levels which is critical when data are shared among different users and applications. Previous access control approaches in the best case only support as minimum granularity level the table column, by which the authorization is associated with an entire column within a table. Other approaches only support access control granularity at the database level, meaning that authorizations are associated with the entire database, and thus either a user can access the entire database or cannot access any data item. In this paper, we propose DBMask, a novel solution that supports fine-grained access control, including row and cell level access control, when evaluating SQL queries on encrypted data. Our solution does not require modification to the database engine, and thus maximizes the reuse of the existing DBMS infrastructures. Our experimental results show that our solution is efficient and scalable to large datasets.**

I. Introduction

The advances of Internet technology and the increasing demand for cost-effective and efficient data management have prompted the emergence of cloud storage servers, such as Rackspace, Amazon EC2, and Microsoft Azure. These third party clouds provide reliable data storage and efficient query processing services able to scale to large data volumes. By outsourcing data to the cloud, organizations save the cost of building and maintaining a private database system and have to pay only for the services they actually use. Therefore, organizations are increasingly fueled to move to the paradigm of "database as a service". However in order to protect data from inappropriate disclosure either by the cloud or external attackers, in most cases data are encrypted before being outsourced to the cloud. The use of encryption raises issues related to the efficient processing of queries on encrypted data. In order to address such issues, past research has extensively investigated various techniques, such as bucketization [1], [2] and secure indexing [3], [4]. However, these techniques do not differentiate among authorized users of the data and thus do not support flexible access control with different units of access control granularity. This is inconsistent with the data sharing requirements of most real-world applications.

Our work is inspired by the CryptDB project [5], which is the first research effort that has systematically investigated access control for SQL queries on encrypted relational data. The CryptDB architecture assumes a proxy between users and the cloud server. Authorized users log in to the proxy by entering passwords, from which the proxy derives secret keys. Given a plaintext query submitted by a user, the proxy first checks if the query can be authorized according to the access control policies. If this is the case, the proxy encrypts the query (i.e., encrypts table/column names and the constants in the query) by the corresponding secret key derived from the user's password. The encrypted query is then forwarded to the cloud, which runs the query over encrypted data and returns the result to the proxy. The proxy then decrypts the query result and forwards it to the user.

CryptDB [5] suffers however from the following two limitations. The first is that the minimum granularity support by its access control mechanism is the column-level. Such a granularity level is too coarse to satisfy the requirements of some real applications. For example, an employee may be permitted to access only his/her own record in an outsourced employee table. With column-level granularity, the only way to allow an employee to see his/her data is to grant the employee access to every column in the employee table. As a consequence, the employee can access the whole table. Access control is thus not enforced as required by the policy. The second limitation is the onions of encryption. An onion is a multiple layers of encryptions. Each layer is applied for a specific query operation or purpose, and the encryption layers from the external layer to the most internal layer are increasingly weaker. Consider an onion for equality matching. In this case, depending on the expected queries, there would be three layers in CryptDB: the inner most layer is an adapted deterministic encryption for equality join, the middle one is classic deterministic encryption for equality selection, and the outer most one is random encryption to assure the maximum security. Given a query equality join, the proxy transmits the secret keys to the cloud server, so that the server can peel off the first two layers (i.e., random encryption and deterministic encryption) and run the join operation. Therefore, it is easy to see that the support of multiple query operations is at the cost of decryption. In addition, although onions offer multiple levels of security, the security level decreases over time when the outer layers are removed. Hence, the real security level an onion can guarantee is the protection offered by the inner most encryption. Furthermore, to support diverse operations, multiple onions need to be generated (e.g., an *order* onion is necessary if range queries are to be supported).

To address the limitations of CryptDB, in this paper we propose DBMask, a novel solution that supports fine-grained access control when evaluating SQL queries on encrypted relational databases. Our solution does not require modification to the database engine, and thus maximizes the reuse of the existing database management systems (DBMS) infrastructures. The contributions of DBMask include the following:

- We propose a fine-grained access control model for relational data. The granularity level can be a table, a column, a row as well as a cell.
- We enforce the access control policies on outsourced databases by an expressive attribute-based group key management scheme [6], [7]. Different portions of data are encrypted by different keys according to the access control policies, so that only authorized users receive the keys to decrypt the data they are authorized for access.
- Our approach uses the blinded attribute-value pair encryption technique [8], which securely encrypts each numerical value by only one layer of encryption but still supports most of the relational query processing operators.
- Besides operations on numerical values, secure keyword matching is also supported.

The paper is organized as follows. Section II provides an overview of our solution and the adversarial model. Section III presents a fine-grained access control model and discusses its enforcement. Section IV provides information about the key cryptographic constructs for SQL query operators and a brief security analysis of each construct. Section V describes the evaluation of encrypted queries over an encrypted database in the cloud. Section VI reports experimental results. Section VII surveys related work and compares with our system. Finally, Section VIII outlines conclusions and future work

II. Overview

In this section, we provide an overview of our system architecture and the adversarial model.

A. System architecture

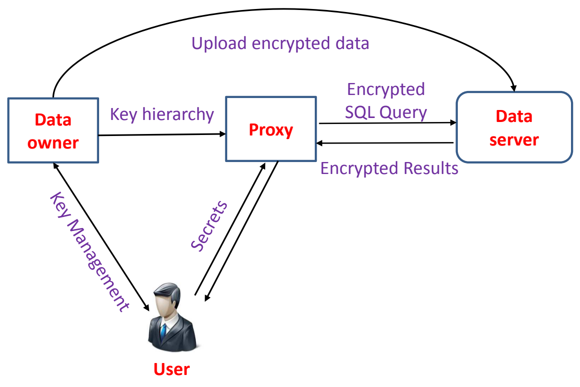

Fig. 1. The system architecture

Our system includes four entities: *data owner*, *data user*, *proxy*, and *data server*. Their interactions are illustrated in the system architecture in Figure 1. Data owner uses different secret keys to encrypt different portions of data, according to pre-defined access control policies. The secret keys are organized in a lattice for an efficient management. Data owner can also build secure indices over the encrypted data to improve the search performance. The encrypted data together with their secure indices are uploaded to the data server. A data user with authenticated attributes can verify itself to the proxy. The successful attribute based verification of the user to the proxy allows the proxy to either derive or obtain one or multiple secret keys required to encrypt the user query. Given a plaintext query submitted by the user, the proxy uses these keys to rewrite the query into an encrypted query, which can then be executed on the encrypted data in the data server. Encrypted query results are returned from data server to the proxy, which decrypts the results using the secrets established at the time of verification and forwards them to the data user. Notice that during the query processing stage, the data server learns neither the query being executed nor the result set of the query.

B. Adversary model

The data owner remains offline after it uploads the encrypted data to data server. We assume that the data owner is fully trusted and is not vulnerable to attacks. However, all the three remaining entities might be compromised. For each of them, we will discuss the possible violations of data confidentiality, and how to restrict them.

The data server is assumed to be honest-but-curious. It does not attempt to *actively* attack the encrypted data, e.g., by altering the query answers or changing the encrypted data. Instead, it is *passive*, and will try to learn the confidential data. To address this, all the data outsourced to the server by the data owner are encrypted. The server will never be given the key, by which ciphertext can be decrypted to obtain plaintext. Still, to support query processing, we develop techniques, which allow the server to efficiently evaluate SQL queries on encrypted data (see Section IV). In addition, the server itself may be compromised by external attackers. In such a case, the data confidentiality is still preserved, since attackers cannot access encrypted data. Attackers might also change the query answers and/or the encrypted data (e.g., by swapping the attribute values of any two tuples). However, such active attacks are out of the scope of this work.

The proxy is a trusted third party. All the secret keys, which are generated by the data owner and stored at the proxy, are encrypted. Our key management scheme (see Section IV) requires that these encrypted secret keys cannot be decrypted by the proxy alone. Instead, they can only be decrypted by the proxy with the help of authorized data users. An attack that has compromised the proxy can access the keys of logged-in users. Consequently, it can also access the data, authorized to those users. However, the secret keys of all the inactive users remain secure. In our model, data owner does not outsource the data encryption operation to the proxy, although it is trusted. This is to avoid "single point of failure". Otherwise, if the proxy is compromised at the pre-processing stage (i.e., the stage of generating the keys to encrypt data), then the whole system is compromised.

In the system, there can be many data users. In general, they are more vulnerable to attacks than the proxy and the data server, because they have less knowledge and fewer resources to provide high-level security, and because they are distributed and the possible number of them is big. Our system does not store at the user side any secret key, which can be used to decrypt the data. Otherwise, the problem of key distribution would be raised, and data would need to be re-encrypted after the users are compromised. Instead, our key management requires that only the collaboration between proxy and authorized data user allows the proxy to derive secret keys. Such a procedure can be seen as a function: $k \leftarrow f(As)$, where k is the secret key, As is the set of user's attributes, and f is a function representing the collaboration between the proxy and the user. Now suppose that the user is compromised (i.e., its attributes are disclosed). To prevent unauthorized data access, the data owner can update the attributes to As', so as to prevent attackers with As posing as authorized user any more. Such a strategy allows the key k and the data to remain unchanged.

III. Access Control Model

In this section, we describe our access control model in detail. We utilize ABAC (attribute based access control) model which has the following characteristics.

- Users have a set of identity attributes that describe properties of users. For example, organizational role(s), seniority, age and so on.
- Data is associated with ABAC policies that specify conditions over identity attributes.
- A user whose identity attributes satisfy the ABAC policy associated with a data item can access the data item.

While the access can be controlled at different granularity levels such as table level, row level, cell level, and so on, we focus on the row level access control to propose our ABAC basic model. In the basic model, each tuple (row) in a database table is attached an ABAC policy. The policy attachment is performed by adding an additional column to the table. Upon receiving an SQL query from a user for a table T, the proxy server needs to determine the ABAC policies attached to T satisfied by the user's attributes and restrict the query to only those rows by adding a predicate to the user query. Such predicate "encodes" the satisfied ABAC policies.

We formally define our model as follows:

Definition 3.1: **Attribute Condition**.
An attribute condition cond is an expression of the form: "name_A op l", where name_A is the name of an identity attribute A, op is a comparison operator such as $=$, $<$, $>$, $\leq$, $\geq$, $\neq$, and l is a value that can be assumed by attribute A.

Definition 3.2: **ABAC Policy**.
An ABAC policy ACP is a tuple (s, o) where: o denotes a set of rows in the table T and s is a Boolean expression over a set of attribute conditions that must be satisfied in order to access o.

We observe that grouping users based on the ABAC policies they satisfy enhances access control enforcement as it provides one level of indirection. Such a grouping of users allows to enforce access control policies on a set of users instead on individual users. Further, relationships between groups can be exploited to improve the management. Considering the fact that every ACP can be converted into disjunctive normal form (DNF), we define a *g*roup as follows for a collection of attribute conditions.

Definition 3.3: **Group**.
We define a group G as a set of users which satisfy a specific conjunction of attribute conditions in an ABAC policy.

The idea of groups is similar to user-role assignment in RBAC, but in our approach, the assignment is performed automatically based on identity attributes. Given the set of data owner defined ABAC policies, the following steps are followed to identify the groups:

- Convert each ABAC ACP into DNF. Note that this conversion can be done in polynomial time.
- For each distinct disjunctive clause, create a group.

Example:
Consider the following two ACPs with the attribute conditions C_1, C_2 and C_3.

- ACP$_1$ = $C_1 \wedge (C_2 \vee C_3)$
- ACP$_2$ = C_2

The ACPs in DNF form are as follows:

- ACP$_1$ = $(C_1 \wedge C_2) \vee (C_1 \wedge C_3)$
- ACP$_2$ = C_2

In this example, there are three groups G_1, G_2, G_3 for the set of users satisfying the attribute conditions $C_1 \wedge C_2$, $C_1 \wedge C_3$, and C_2 respectively.

We exploit the hierarchical relationship among groups in order to support hierarchical key derivation and improve the performance and efficiency of key management. We introduce the concept of *G*roup Poset as follows to achieve this objective.

Definition 3.4: **Group Poset**.
A group poset is defined as the partially ordered set (poset) of groups where the binary relationship is $\subseteq$.
Example:
Let G be all users. $G_1 \subseteq G_3$ and there is no ordering between G_1 and G_2. As Figure 2, the poset has the Hasse diagram $\phi \to G_1$, $\phi \to G_2$, $G_1 \to G_3$, $G_2 \to G$ and $G_3 \to G$. In the figure, the set of users satisfying the condition attached to the group G_i is denoted as U_i and U consists of all the users in the system.

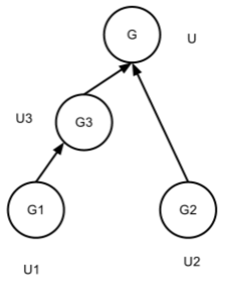

Fig. 2. An example group poset

A. Assigning Group Labels to Tuples and Users

Like the purpose-based access control model [9], we label the tuples using descriptive group names. Each tuple may have multiple groups associated with it. If there is an ordering relationship between two groups associated with a row, we discard the more privileged group and assign only the less privileged group. When the proxy decides the group(s) that user belongs to, it selects the most privileged groups. The idea is that a user in the more privileged group can become a member of the less privileged group by following the hierarchical relationship in the group poset. Note that the group label assignment to a tuple indirectly attach an ABAC policy to a row as described at the beginning of this section.
Example:
Assume that the attribute A_i satisfies the attribute condition C_i and there is a set of rows in the table T_1 that the users in the groups G_1 and G_3 satisfy. Further, assume that the user u_1 has the attributes A_1 and A_2.

Since $G_1 \to G_3$, the set of rows is labeled with the less restrictive group G_3. According to the selection criteria, the proxy chooses G_1 for u_1 based on its two attributes A_1 and A_2. When u_1 submits a query for the table T_1, the proxy selects all rows containing either G_1 or G_3 as u_1 is a member of both groups.

B. Broadcast and Hierarchical Key Management

In traditional hierarchical access control models [10], each node in a hierarchy is assigned a key and a user who has access to a key corresponding to a node in the hierarchy can access all the keys for all the descendant nodes using the public information available. Hierarchical key encryption techniques reduce the number of keys to be managed. However, a major drawback is that assigning keys to each node and giving them to users beforehand makes it difficult to handle dynamics of adding and revoking users. For example, when a user is revoked, one needs to update the keys given to other users through private communication channels. We propose to address this drawback while utilizing the benefits of hierarchical model by proposing a hybrid approach combining broadcast and hierarchical key management. A broadcast group key management (BGKM) allows one to efficiently handle group keys when user dynamics change. We utilize a recent expressive scheme called AB-GKM (attribute based GKM) as the broadcast GKM scheme which is described in Section IV-A. Instead of directly assigning keys to each node in the hierarchy, we assign a AB-GKM instance to each node and authorized users can derive the key using the key derivation algorithm of AB-GKM. A AB-GKM instance is attached to a node only if there is at least one user who cannot derive the key of the node by following the hierarchical relationship.
Example:
For the node for the group G_1 in the poset in Figure 2, an AB-GKM instance is created if $U_1 \neq \phi$. Similarly, for the node for the group G_2, an AB-GKM instance is created if $U_3 - U_1 \neq \phi$.

IV. Cryptographic Constructs

In this section, we describe the cryptographic constructs used in our approach for secure query evaluation over encrypted data.

A. Key management

Broadcast Group Key Management (BGKM) schemes [11], [12], [13], [6] are a special type of GKM scheme whereby the rekey operation is performed with a single broadcast without requiring private communication channels. Unlike conventional GKM schemes, BGKM schemes do not give subscribers private keys. Instead subscribers are given a secret which is combined with public information to obtain the actual private keys. Such schemes have the advantage of requiring a private communication only once for the initial secret sharing. The subsequent rekeying operations are performed using one broadcast message. Further, in such schemes achieving forward and backward security requires only to change the public information and does not affect the secrets given to existing subscribers. However, BGKM schemes do not support group membership policies over a set of attributes. In their basic form, they can only support 1-*out-of-*n threshold policies by which a group member possessing 1 attribute out of the possible n attributes is able to derive the group key. A recently proposed attribute based GKM (AB-GKM) scheme [7] provides all the benefits of BGKM schemes and also supports attribute based access control policies (ACPs).

Users are required to show their identity attributes to the data owner to obtain secrets using the AB-GKM scheme. In order to hide the identity attributes from the data owner while allowing only valid users to obtain secrets, we utilize oblivious commitment based envelope (OCBE) protocols [14] which is based on Pedersen commitments [15] [1] and zero knowledge proof of knowledge techniques [16]. We omit the technical details of the OCBE protocols due to the page limit. The OCBE protocols between the data owner and users provide the following guarantees in DBMask.

- The data owner does not learn the identity attributes of users as their identities are hidden inside Pedersen commitments.
- A user can obtain a valid secret for an identity attribute from the data owner only if the identity attribute is not fake. The data owner sends the secrets to the user in an encrypted message and the user can decrypt the message only if the user has a valid identity attribute.

We denote the set of all attribute conditions as $\mathcal{ACB}$ and the set of all ACPs as $\mathcal{ACPB}$. Example 4.1 shows an example ACP.

Example 4.1: The ACP ((“type $=$ regular” $\wedge$ “region $=$ Indiana”) $\vee$ “type $=$ premium”, $\{$new movie$\}$) states that a user, either having a premium subscription or having a regular subscription in Indiana region, has access to new movies.

The idea behind the AB-GKM scheme is as follows. A separate BGKM instance for each attribute condition is constructed. The ACP is embedded in an access structure $\mathcal{T}$. $\mathcal{T}$ is a tree with the internal nodes representing threshold gates and the leaves representing BGKM instances for the attributes. $\mathcal{T}$ can represent any monotonic policy. The goal of the access tree is to allow deriving the group key for only the subscribers whose attributes satisfy the access structure $\mathcal{T}$. Figure 3 shows the access tree for the ACP presented in Example 4.1.

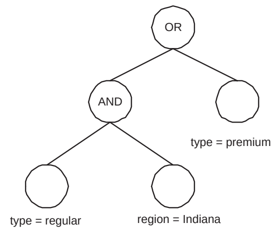

Fig. 3. An example access tree

Each threshold gate in the tree is described by its child nodes and a threshold value. The threshold value t_x of a node x specifies the number of child nodes that should be satisfied in order to satisfy the node. Each threshold gate is modeled as a Shamir secret sharing polynomial [17] whose degree equals to one less than the threshold value. The root of the tree contains the group key and all the intermediate values are derived in a top-down fashion. A subscriber who satisfies the access tree derives the group key in a bottom-up fashion.

We only provide the abstract algorithms of the AB-GKM scheme. The AB-GKM scheme consists of five algorithms: **Setup**, **SecGen**, **KeyGen**, **KeyDer** and **ReKey**. An abstract description of these algorithms are given below.

- **Setup(ℓ, N, N_a)**:
  It takes the security parameter ℓ, the maximum group size N, and the number of attribute conditions N_a as input, and initializes the system.
- **SecGen(γ)**:
  The secret generation algorithm gives a user$_j$, $1 \leq j \leq N$ a set of secrets for each commitment $\mathsf{com}_i \in \gamma$, $1 \leq i \leq m$. OCBE protocols are used to assure the privacy of subscribers from the group controller. At the end of the execution of this algorithm, the group controller does not learn the attributes of subscribers and the subscribers can recover the secret(s) only if they have valid credentials.
- **KeyGen(ACP)**:
  The key generation algorithm takes the access control policy ACP as the input and outputs a symmetric key K, a set of public information tuples **PI**, and an access tree $\mathcal{T}$.
- **KeyDer(β, PI, $\mathcal{T}$)**:
  Given the set of identity attributes β, the set of public information tuples **PI**, and the access tree $\mathcal{T}$, the key derivation algorithm outputs the symmetric K only if the identity attributes in β satisfy the access structure $\mathcal{T}$.
- **ReKey(ACP)**:
  The rekey algorithm is similar to the **KeyGen** algorithm. It is executed whenever the dynamics in the system change, that is, whenever subscribers join and leave or ACPs change.

Brief security analysis. An adversary, who has compromised the cloud server, cannot infer the keys used to encrypt the data from the public information stored in the cloud server as the AB-GKM scheme is key hiding even against computationally unbounded adversaries. An adversary, who has compromised the proxy, the AB-GKM secrets of the users who are currently online are compromised as the proxy derives these secrets using users’ passwords and encrypted secrets. Since the data owner performs the setup and key generation operations of the AB-GKM scheme, such an attack does not allow the attacker to infer the secret information stored at the data owner. If such an attack is detected, the proxy can invalidate the existing secrets of the online users and request the data owner to generate new set of secrets using the AB-GKM scheme for the users without changing the underlying keys used to encrypt/decrypt the data. Since the secrets at the time of compromise and after regeneration are different, it is cryptographically hard for the adversary to derive the underlying encryption/decryption keys from the invalid secrets. Notice that, unlike a traditional key management scheme, since the underlying encryption/decryption keys are not required to be changed, such a compromise does not require to re-encrypt the data stored in the cloud.

[1]Pedersen commitment is a cryptographic commitment allows a user to commit to a value while keeping it hidden and preserving the user’s ability to reveal the committed value later.

B. Numerical matching

Nabeel et. al. [8] recently proposed a privacy preserving comparison of encrypted numerical values that does not require decrypting the numerical values. We refer to this approach as PPNC (privacy preserving numerical comparison) in the remainder of the paper. The approach can be summarized into the four algorithms, **Setup**, **EncVal**, **GenTrapdoor** and **Compare**, which we use for numerical comparison in our cloud based database system.

- **Setup**(t, l):
  The security parameter t decides the bit length of the prime numbers of the Paillier cryptosystem. The system parameter l is the upper bound on the number of bits required to represent any data values published, and we refer to it as *domain size*. For example, if an attribute can take values from 0 up to 500 ($< 2^9$), l should be at least 9 bits long. Using these parameters, it initializes the underlying Paillier cryptosystem and random values required for the computations.
- **EncVal**(x):
  Given an input value x, it produces a semantically secure encrypted value e_x that hides the actual value, but allows to perform comparisons using the trapdoor values which is described below.
- **GenTrapdoor**(t):
  Given an input value t, it produces a semantically secure encrypted value e_t, called the trapdoor, that is used with its corresponding encrypted value to perform oblivious comparisons.
- **Compare**(e_x, e_t, op):
  Given an encrypted value e_x for x and a trapdoor value e_t for t, it obliviously compares and outputs the result of $xopt$. It should be noted that this algorithm is executed without decrypting e_x and e_t. Hence, at the end of the evaluation, the actual x and t remains oblivious to the algorithm. Further, the output only reveals a randomized difference of the two values and, thus, does not leak even the actual difference.

Brief security analysis. An adversary, who has compromised the cloud server, cannot infer the plaintext values of the encrypted values as the the private key is not stored at the cloud server and the encrypted values are semantically secure. An adversary, who has compromised both the proxy and the cloud server, cannot directly infer the plaintext values using the private key information stored at the proxy since the private key used at the data owner to generate the encrypted value and the private key used at the proxy to generate the trapdoor are different and it is cryptographically hard to derive one key from the other. The adversary may however repeatedly execute comparison operations to infer the plaintext values of the encrypted values in the cloud server. Detection and prevention of such an attack is beyond the scope of this paper.

C. Keyword search

Besides the numerical matching, DBMask also allows a user to check whether an encrypted string contains certain words. We adopt the keyword search technique proposed in [18] to support such a secure operation. Given a string, we extract keywords from it, and encrypt them. Then, given a keyword, a trapdoor is generated. The trapdoor is then sequentially compared with the encrypted keywords. If there is a match, then the corresponding string contains the given keyword with a probability close to 1 (i.e., the false rate is negligible). We refer to this approach as PPKC (privacy preserving keyword comparison) in the remainder of the paper.The details of the technique are summarized as follows.

- **Setup**(G, f, $\mathcal{P}$, k', k'', E):
  1) $G : \mathcal{K}_G \rightarrow \mathcal{X}^\ell$ is a pseudorandom generator, where $\mathcal{X} = \{0,1\}^{n-m}$. The data owner generates a stream cipher $S_1, S_2, \ldots, S_\ell$, where S_i has $n-m$ bits. Note that the stream cipher is kept confidential from both the proxy and the data user. 2) $f : \mathcal{K}_F \times \{0,1\}^* \rightarrow \mathcal{K}_F$ is a pseudorandom function. 3) $F : \mathcal{K}_F \times \mathcal{X} \rightarrow \mathcal{Y}$ is a pseudorandom function, where $\mathcal{Y} = \{0,1\}^m$. 4) $\mathcal{P} : \mathcal{K}_\mathcal{P} \times \mathcal{Z} \rightarrow \mathcal{Z}$ is a pseudorandom permutation, where $\mathcal{Z} = \{0,1\}^n$. 5) k' and k'' are two keys, which are shared between the data owner and the proxy. 6) E is a symmetric encryption function.
- **EncVal**(W_i):
  Given a keyword W_i, the data owner computes $X_i = \mathcal{P}(k'', W_i)$. The data owner then generates $k_i = f(k', X_i)$. Suppose that S_i is the stream cipher for W_i. The data owner calculates $F(k_i, S_i)$, and W_i's ciphertext $C_i = \mathcal{P}(k'', W_i) \oplus T_i$, where $T_i = S_i || F(k_i, S_i)$. Let str be a string with keywords $W_1, W_2, \ldots, W_j$. The data owner sends to the server $(E(str), C_1, C_2, \ldots, C_j)$, where $E(str)$ is the encryption of the string.
- **GenTrapdoor**(W):
  Given a keyword W, the proxy generates its trapdoor as (X, k), where $X = \mathcal{P}(k'', W)$ and $k = f(k', X)$.
- **Search**(C_i, X):
  The user sends the trapdoor (X, k) to the server. For each ciphertext C_i, the server checks if $C_i \oplus X$ has the form of $S_i || F(k, S_i)$, where S_i is the first $n-m$ bits of $C_i \oplus X$. If the form holds, then C_i is an encryption of keyword W.

Extensions. The above searchable encryption scheme is applicable only to exact keyword matching. However, it can be easily extended to support the following two conditions: 1) Whether a string contains both keyword W and keyword W'. 2) Suppose that the keywords to be encrypted are ordered according to their locations in the string. Then, it is possible to check whether two keywords W and W' in the string are close to each other (e.g., the number of keywords between them is smaller than a threshold).

Brief security analysis. An adversary, who has compromised the cloud server, cannot infer the plaintext keywords from the ciphertexts. See [18] for detailed proofs. An adversary may break both the proxy and the server. In such a case, direct inference of plaintext from the ciphertext is still impossible, because the encrypted keyword $\mathcal{P}(k'', W_i)$ is randomized by $T_i = S_i || F(k_i, S_i)$, where S_i is kept confidential and k_i is unknown. However, a brutal force attack is still possible. The adversary can first build a dictionary of keywords and then generates the trapdoor for each keyword in the dictionary. Finally, given the ciphertext of a keyword, the adversary compares it with each generated trapdoor (as in *sequential search*) until finding the matching one. The keyword corresponding to this matching trapdoor is the desired plaintext.

V. Secure Query Evaluation Over Encrypted Data

In this section, we provide a detailed description of our privacy preserving query processing scheme for encrypted databases in a public cloud. As mentioned in Section II our systems consists of four entities: data owner, proxy, cloud and users. Our system undergoes the following phases: System initialization, user registration, data encryption and uploading, and data querying and retrieval. We now explain each phase in detail.

A. System initialization

The data owner runs the Setup algorithm of the underlying cryptographic constructs, that is, AB-GKM.Setup, PPNC.Setup and PPKC.Setup [2]. The data owner makes available the public security parameters to the proxy so that the proxy can generate trapdoors during data querying and retrieval phase.

The data owner also converts the ACPs into DNF and users satisfying each disjunctive clause form a group. As mentioned in Section III, these groups are used to construct the Group poset to perform hierarchical key derivation along with the AB-GKM based key management.

B. User registration

Users first get their identity attributes certified by a trusted identity provider. These certified identity attributes are cryptographic commitments that hide the actual identity attribute value but still bind the value to users. Users register their certified identity attributes with the data owner using the OCBE protocol. The data owner executes the AB-GKM.SecGen algorithm to generate secrets for the identity attributes and gives the encrypted secrets to users. Users can decrypt and obtain the secrets only if they presented valid certified identity attributes. The data owner maintains a database of user-secret values. When a user or an identity attribute is revoked, the corresponding association(s) from the user-secret database are deleted. The user-secret database is also stored at the proxy with the secrets encrypted using a password only each user possesses. Every time the user-secret database changes, the data owner synchronizes its changes with the proxy.

[2]We use the dot notation to refer to an algorithm of a specific cryptographic construct. For example, AB-GKM.Setup refers to the Setup algorithm of AB-GKM scheme.

C. Data encryption and uploading

In our solution each cell in an original table is encrypted twice as illustrated in Figure 4: the first encryption is for fine-grained access control, and the second is for privacy-preserving matching. Correspondingly, each column in the original table is expanded to two. We denote the column resulting from the encryption for fine-grained access control as *data-col*, and the one resulting for the encryption for privacy-preserving matching as *match-col*.

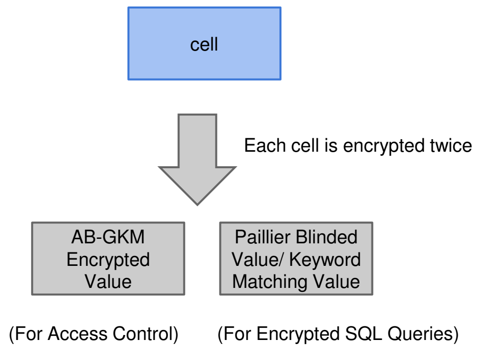
cell
Each cell is encrypted twice
AB-GKM Encrypted Value
Paillier Blinded Value/ Keyword Matching Value
(For Access Control)
(For Encrypted SQL Queries)

Fig. 4. The two encryptions of each cell

Let us first discuss the creation of data-col. Given a cell in the original table, its encryption in the corresponding data-col is generated by a secret key derived from AB-GKM scheme [7] as follows. Consider the row containing the cell in the original table. Based on the ACPs, each row is assigned one or more group labels. The set of groups decides the key, under which the cell in the row is encrypted. If two groups are connected in the group poset, only the label of less privileged group is assigned to the row. The intuition behind is that users in the more privileged group can reach the less privileged group by following the hierarchical relation in the group poset. After removing the labels of groups with higher privileges, a row can still be associated with multiple groups. For each remaining group G_i, a group secret key K_i is generated by executing the AB-GKM.KeyGen algorithm. In order to avoid multiple encryptions (i.e., one group secret key for one encryption), the AB-GKM.KeyGen algorithm is again executed to generate a master group key K using the group keys K_i's as secret attributes to the algorithm. As a consequence, if a user belongs to any of the groups assigned to the row, the user can access the row by executing the AB-GKM.KeyDer algorithm twice. The first execution generates the group key and second derives the master key. Public information to derive the key is stored in a separate table called *PubInfo*.
Example:
Following the example in Figure 2, suppose that C_1, C_2 and C_3 are conditions defined as as follows. C_1 = "level $>$ 3", C_2 = "role = doctor" and C_3 = "role = nurse". Therefore, ACP$_1$

is satisfied by all users whose level is greater than 3 and who are either doctors or nurses. ACP$_2$ is satisfied by all doctors.

Assume that the above ACPs are applied to the *Patient* table and group labels are assigned as shown in the following table.

TABLE I
PATIENT TABLE

ID	Age	Diagnosis	Groups
1	35	HIV	G_1
2	30	Cancer	G_1, G_2
3	40	Asthma	G_2, G_3
4	38	Gonorrhea	G_1

In Table I (Patient table), each group G_i is assigned a unique k_i. Rows 1 and 4 are encrypted using key k_1. Since rows 2 and 3 have multiple groups, in order to avoid multiple encryptions/decryptions, a master key is assigned using AB-GKM by considering the group keys as input secrets to the AB-GKM.KeyGen algorithm. Row 2 is encrypted using key k_{12} generated from the AB-GKM instance having k_1 and k_2 as input secrets. We denote the public information corresponding to this master key as PI_{12}. Similarly, row 3 is encrypted with key k_{23}. The public information is stored in Table II.

TABLE II
PUBINFO TABLE

Groups	PI
G_1	PI_1
G_2	PI_2
G_3	PI_3
G_1, G_2	PI_{12}
G_2, G_3	PI_{13}

Table III shows the broadcast encrypted values where $E_k(x)$ refers to the semantically secure encryption of the value x using the symmetric key k.

TABLE III
ENCRYPTED PATIENT TABLE

ID-enc	Age-enc	Diag-enc	Groups
$E_{k_1}(1)$	$E_{k_1}(35)$	E_{k_1}(HIV)	G_1
$E_{k_{12}}(2)$	$E_{k_{12}}(30)$	$E_{k_{12}}$(Cancer)	G_1, G_2
$E_{k_{23}}(3)$	$E_{k_{23}}(40)$	$E_{k_{23}}$(Asthma)	G_2, G_3
$E_{k_1}(4)$	$E_{k_1}(38)$	E_{k_1}(Gonorrhea)	G_1

Now, let us consider the creation of match-col. Given a cell in the original table, its encryption in the match-col is generated as follows. Our scheme supports both numerical matching and keyword search for strings. If the cell is of numerical type, PPNC.EncVal algorithm is used to encrypt the cell value. If the cell is of type string, the PPKC.EncVal algorithm is used to perform the encryption. Once each cell is encrypted twice, the encrypted table is uploaded to the cloud.

Table IV shows the final table with both encrypted data-col's and comparison friendly match-col's, where $comp_n$ and $comp_k$ refer to PPNC.EncVal and PPKC.EncVal respectively.

D. Data querying and retrieval

Processing of a query over encrypted data is a *filtering-refining* procedure. Given an encrypted query forwarded to the server by the proxy, the server evaluates the predicates of the query on the encrypted data. Encrypted tuples that satisfy the predicates are returned to the proxy, while others are pruned. The proxy decrypts the data, refines the query result, and sends the plaintext result to the user. The details are as follows.

- An authorized user sends a plaintext SQL query to the proxy, as if the outsourced database is unencrypted. In other words, encryption and decryption of the data in the database is transparent to users.
- The proxy parses the query and generates the abstract syntax tree of the query in order to rewrite the query for the cloud. It first removes ORDER BY clause, GROUP BY clause, HAVING clause, aggregate functions, and predicates with aggregate functions from the query. Then, for each column to be included in the query result (i.e., column following the SELECT keyword in the query), it replaces it by its corresponding "data-col". After that, in each predicate for numerical matching, the proxy computes the trapdoor value using PPNC.GenTrapdoor algorithm and replaces the predicate with a user defined function (UDF) which invokes the PPNC.Compare algorithm. Similarly, for each keyword matching predicate, the proxy computes the trapdoor value using PPKC.GenTrapdoor algorithm and replaces the predicate with a user defined function which invokes the PPKC.Compare algorithm. The rewritten query is then sent to the cloud server.
- The cloud executes the rewritten query over the encrypted database. It filters tuples that do not satisfy the predicates in the query, and sends back the encrypted result set to the proxy.
- The proxy generates necessary keys to decrypt the result set using the AB-GKM.KeyDer algorithm with the public information[3] and the user secrets as well as the hierarchical key derivation. If the proxy has removed some clauses (e.g., ORDER BY) and/or aggregate functions (e.g., SUM) from the original query in query rewriting, it populates an in-memory database with the decrypted result set and refines the query result according to the constraints in the clauses and/or aggregate functions. If no term from the query is removed, decrypted result set is the final result. The proxy sends the final plaintext result back to the user.

For queries that cannot be processed in a single round, multiple queries are executed at the cloud server in order to obtain the final answer for user queries. Intermediate results obtained from the cloud server in such multi-round queries are loaded into an in-memory database at the proxy and executed to generate results for the subsequent queries to the cloud server. When a query is split into multiple sub-queries, the query planner at the proxy server generates the sub-queries

[3]The public information (i.e., PI) is stored at the cloud server and retrieved together with the query.

TABLE IV
TWICE ENCRYPTED PATIENT TABLE

ID-enc	ID-com	Age-enc	Age-com	Diag-enc	Diag-com	Groups
$E_{k_1}(1)$	$comp_n(1)$	$E_{k_1}(35)$	$comp_n(35)$	E_{k_1}(HIV)	$comp_k$(HIV)	G_1
$E_{k_{12}}(2)$	$comp_n(2)$	$E_{k_{12}}(30)$	$comp_n(30)$	$E_{k_{12}}$(Cancer)	$comp_k$(Cancer)	G_1, G_2
$E_{k_{23}}(3)$	$comp_n(3)$	$E_{k_{23}}(40)$	$comp_n(40)$	$E_{k_{23}}$(Asthma)	$comp_k$(Asthma)	G_2, G_3
$E_{k_1}(4)$	$comp_n(4)$	$E_{k_1}(38)$	$comp_n(38)$	E_{k_1}(Gonorrhea)	$comp_k$(Gonorrhea)	G_1

such that the cloud server performs the maximum possible computation and the utilization of the bandwidth between the proxy and the cloud server is minimized.

Now we illustrate query processing DBMask through example queries.

A user having the attribute "role = doctor" executes Query 1 through the proxy server.

Query 1:
SELECT *ID, Age, Diag*
FROM *Patient*

The proxy determines that the user is a member of the group G_3, re-writes the query as Query 2 and submits to the cloud server.

Query 2:
SELECT *ID-enc, Age-enc, Diag-enc*
FROM *Patient*
WHERE *Groups* **LIKE** '%G_3%'

The proxy receives the encrypted row 3 of the Patient table from the cloud server. The proxy uses the secrets of the user and PI_3 to derive k_3 using AB-GKM.KeyDer algorithm. Then it uses k_3 and PI_{23} to derive k_{23}. The proxy uses this key to decrypt the encrypted result set and send the plaintext result back to the user.

A user having the attributes "role = doctor" and "level = 4' executes Query 3 through the proxy server.

Query 3:
SELECT *ID, Age, Diag*
FROM *Patient*
WHERE *Age* $>$ 35
ORDER BY *Age* **ASC**

The proxy determines that the user is a member of the groups G_1 and G_3, re-writes the query as Query 4 and submits to the cloud server.

Query 4:
SELECT *ID-enc, Age-enc, Diag-enc*
FROM *Patient*
WHERE UDF_Compare_Num(*Age-com*, PPNC.GenTrapdoor(35), '>')
AND (*Groups* **LIKE** '%G_1%' **OR** *Groups* **LIKE** '%G_3%')

UDF_Compare_Num is a user defined function that simply invokes the PPNC.Compare algorithm internally. Notice the 'ORDER BY' clause is removed from the query as the cloud server does not have sufficient information to order the query results. The cloud server returns the encrypted rows 3 and 4 to the proxy. In order to decrypt the resultset, the proxy derives the key k_1 for the higher privileged group G_1 using the AB-GKM scheme. In order to generate k_2, instead of executing another AB-GKM key derivation algorithm, it utilizes the hierarchical key derivation to derive k_2 from k_1. The derivation of the key k_{23} is similar to the previous example. The proxy uses k_1 and k_{23} to decrypt the resultset and orders the plaintext resultset using its in-memory database before sending the final resultset to the user.

The same user executes Query 5 through the proxy server.

Query 5:
SELECT *ID, Age, Diag*
FROM *Patient*
WHERE *Age* $>$ 35 **AND** *Diag* **LIKE** 'Asthma'
ORDER BY *Age* **ASC**

The query is very similar to Query 3, except for the additional predicate "*Diag* **LIKE** 'Asthma"'. Therefore, the query is re-written similar to Query 4 by including the additional predicate "UDF_Compare_Str(*Diag-com*, PPKC.GenTrapdoor('Asthma'))" where UDF_Compare_Str is a user defined function that simply invokes the PPKC.Search algorithm internally. We omit the details of the execution as it is very similar to the previous query.

E. Handling user dynamics

When users are added or revoked, or attributes of existing users change, the user dynamics of the system change. This requires changing the underlying constructs. Since DBMask utilizes AB-GKM, these changes are performed transparent to other users in the system. When a new identity attribute for a user is added to the system, the data owner simply adds the corresponding secret to the user-secret database. Similarly, when an existing attribute for a user is revoked from the system, the data owner simply removes the corresponding secret from the user-secret database. In either scenario, the data owner recomputes the affected public information tuples and inform both the proxy server and the cloud server to update the data. Notice that unlike the traditional symmetric key based systems, DBMask does not need to re-key existing users and they can continue to use their existing secrets. Since no re-keying is performed, the encrypted data in the database remains the same even after such changes. Therefore, DBMask can handle very large datasets even when the user dynamics change.
Example:
Assume that a user having the attribute "role = doctor" is added to the system. This affects only the group G_2. The data owner executes AB-GKM.Re-Key operation with the same

symmetric key k_2 as the group key to generate the new public information PI'_2. The proxy and the cloud server are updated with the new secret and the new public information respectively. Notice that this change affects neither the secrets issued to other users nor the public information related to other groups in which the new user is not a member.

VI. Experiments

This section evaluates the performance overhead and the functionality of our prototype implementation. We implemented DBMask in C++ on top of Postgres 9.1 while not modifying the internals of the database itself. This is achieved by including UDFs in the query containing encrypted fields during the rewriting process of a query at the proxy. We use the memory storage engine of MySQL as the in-memory database at the proxy to store contents of a query when the execution of a query requires more than one round of communication between the proxy and the server. The cryptographic operations are supported by using NTL and OpenSSL libraries. The experimental setup is run on 3.40 GHz Intel i7-3770 processor with 8 GB of RAM in Ubuntu 12.04 environment. We analyze the performance of our prototype by running a TPC-C query workload and the functionality of our prototype by using a web based scientific application called Computational Research Infrastructure for Science (CRIS). The performance of TPC-C query workload when running on an unmodified Postgres server is compared to running the workload through the proxy of our prototype. The experimental results below show a low runtime overhead with a 29-31% loss in throughput in comparison to unmodified Postgres. The access control functionality deployed by our prototype on CRIS ensures that a logged in user is only able to retrieve data that the user is authorized to access.

A. TPC-C

TABLE V
TPC-C Workload

Application	# Tables	# Columns	# Queries
TPC-C	9	92	3300

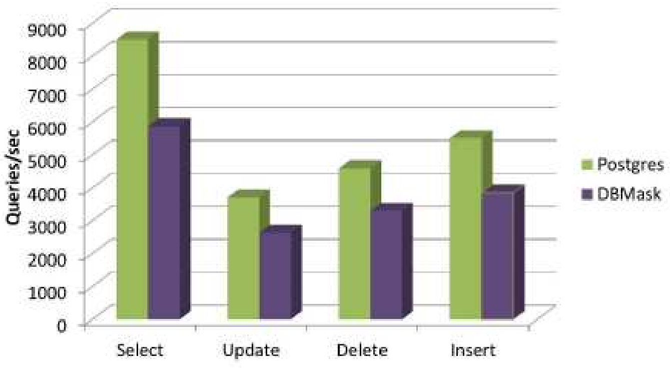

Fig. 5. Throughput from TPC-C Workload

The TPC-C workload queries contain comparison predicates (=, $<$, $>$) and other predicates such as DISTINCT and COUNT as well as aggregates such as SUM and MAX. In total, the workload contains 1900 SELECT, 100 DELETE, 900 UPDATE and 400 Insert statements with 100 Selects using SUM predicate. Our prototype currently supports operations on integer and text data types and do not support certain fields such as those that perform date manipulation. We study the performance of the TPC-C workload by evaluating two different metrics: the server throughput for different sql queries and the interval between issuing a query and receiving the results.

Figure 5 shows the server throughput. The results show that there is an overall loss of DBMask's throughput by 29-31% in comparison to running an unencrypted trace of TPC-C workload on Postgres. The most variation is noticed in SELECT statements where simple SELECT statements perform relatively better than aggregates such as SUM.The throughput of the remaining queries that make the greater part of TPC-C workload is relatively modest. We consider a lower throughput of 31% for encrypted query processing to be modest considering the gains in confidentiality and privacy.

To understand latency, we measure the processing time of the same type of queries used above by studying the intervals at each stage of processing, namely server and proxy. We observe that there is an increase by 30% on the server side. The proxy adds on average 3.1 ms to the interval of which 13% is utilized in encryption/decryption and the most (67%) is spent in query rewriting and processing. A simple SELECT statement with no predicates returned a latency of 0.42 ms on unmodified Postgres and a latency of 0.45 ms with DBMask. In the case of a SELECT statement with SUM predicate, unmodified Postgres returned a latency of 0.45 ms and DBMask returned 6.7 ms primarily due to further processing of the query at the proxy.

TABLE VI
Cryptographic Operations

Scheme	# Encrypt	Decrypt
AES-CBC	0.002 ms	0.002 ms
PPNC	0.0001 ms	0.00001 ms
PPKC	0.00256 ms	0.00276 ms

Table VII shows the average time required to encrypt/decrypt a 64-bit integer or a 15-byte text using the three different encryption schemes used in our approach. It can be observed that the overhead involved in cryptographic operations is small since the schemes are efficient. There is an increase in the amount of data that is stored (2.0x) in the DBMS since we store two columns for each original column in the table in order to facilitate fine grained access control and encrypted query processing.

B. CRIS

CRIS is a web based application with its primary tenets to provide an easy to use system in order to create, maintain and share scientific data. CRIS currently has a community of users in Agronomy, Biochemistry, Bioinformatics and Healthcare Engineering at Purdue University.[4] The data in the form of projects, experiments and jobs residing in CRIS is of sensitive nature and hence must be protected from unauthorized usage. CRIS has an access control mechanism where users are organized into groups and both users and groups have a set of permissions on data objects. To test the functionality of our prototype we select a workspace which acts as a container for all activities and data to be managed by a single group of scientists consisting of 19 users with 8 attributes each. The users based on common attributes are arranged into 4 groups and each group is assigned a randomly chosen secret using AB-GKM.SecGen algorithm. A selected set of sensitive columns are encrypted as some need not be encrypted since they are not sensitive and each row is assigned a label for access control.

In this experiment, we evaluate the effect on throughput by running CRIS on Postgres in comparison to DBMask while making sure unauthorized users are not allowed access to sensitive data . Each HTTP request from the application by a logged in user consists of multiple SELECT queries with no predicates. The results show that there is a 33% loss of throughput, but the confidentiality and privacy are preserved. A logged in user is only able to access the objects it has permission for. There is also an increase by around 22-28 ms when creating, updating or deleting a certain project, experiment or a job due to the overhead DBMask introduces.

TABLE VII
THROUGHPUT COMPARISON FOR CRIS

	Postgres	DBMask
Throughput(HTTP req./sec)	9	6

VII. RELATED WORK

In this section, we discuss research work related to our work and compare them our approach. DBMask is the first system to support row/cell level access control and query processing over encrypted data without decrypting on a relational database. Some of the techniques used in DBMask are built on prior work from the cryptographic community.

In theory, it is possible to utilize fully homomorphic encryption [19] to perform any arbitrary operation that a relational database demands. However, the current implementations of fully homomorphic encryption is very inefficient and are not suitable for any practical application [20].

Querying non-relational encrypted data has also been extensively researched. These techniques are used in the *outsourced storage model* where a user's data are stored on a third-party server and encrypted using the user's symmetric or public key. The user can use a query in the form of an encrypted token to retrieve relevant data from the server, whereas the server does not learn any more information about the query other than whether the returned data matches the search criteria. There have been efforts to support simple equality queries [18], [21] and more recently complex ones involving conjunctions and disjunctions of range queries [22]. While these approaches assist in constructing the predicate portion of SQL queries, they do not directly apply to query processing over encrypted relational data.

With the increasing utilization of cloud computing services, there has been some recent research efforts [23], [24] to construct privacy preserving access control systems by combining oblivious transfer and anonymous credentials. The goal of such work has similarities to ours but we identify the following limitations. Each transfer protocol allows one to access only one record from the database, whereas our approach does not have any limitation on the number of records that can be accessed at once since we separate the access control from query processing. Another drawback is that the size of the encrypted database is not constant with respect to the original database size. Redundant encryption of the same record is required to support ACPs involving disjunctions. However, our approach encrypts each data item only twice as we have made the encryption independent of ACPs. While attribute based encryption (ABE) based approaches [25] support expressive policies, they cannot handle revocations efficiently. Yu et al. [26] proposed an approach based on ABE utilizing PRE (Proxy Re-Encryption) to handle the revocation problem of ABE. While it solves the revocation problem to some extent, it does not preserve the privacy of the identity attributes as in our approach. Further, these approaches mostly focus on non-relational data such as documents, whereas DBMask is optimized for relational data.

Efficient query processing over encrypted data has been researched utilizing various techniques in the past. Hacigümüs et al. [1] proposed the pioneering work in this area by performing as much query processing as possible at the remote database server without decrypting the data and then performing the remaining query processing at the client site. Their idea is to utilize bucketization technique to execute approximate queries at the remote database server and then execute the original query over the approximate result set returned by the remote server. They later extend their work to support aggregate queries [27], [28]. While DBMask utilizes split processing between the cloud server and the proxy for complex queries, DBMask is different in that it performs exact query execution whenever possible and it enforces row/cell level access control utilizing the AB-GKM scheme, whereas their approach does not support fine-grained access control of the relational data.

The idea of utilizing specialized encryption techniques such as order preserving encryption [29], [30], additive homomorphic encryption [31], and so on to perform different relational operations is first introduced in CryptDB [5]. The same idea is extended to support more complex analytical queries in MONOMI [32]. As mentioned in Section I, while they lay the foundation for systematic query processing and access control, they suffer from two limitations. First the security of the encrypted data reduces to deterministic encryption over

[4]cris.cyber.purdue.edu

time as the outer layers of stronger encryptions are removed in order to execute certain queries. Second, these techniques do not support fine grained access control over the encrypted relational database. DBMask addresses these two limitations by separating access control from query processing. Further, in DBMask, data are never decrypted to weaker encryptions inside the cloud server and therefore the security of the data do not weaken over time.

VIII. Conclusion

In this paper, we proposed DBMask, a novel solution that supports fine-grained access control, including row and cell level access control, when evaluating SQL queries on encrypted relational data. Similar to CryptDB [5] and MONOMI [32], DBMask does not require modification to the database engine, and thus maximizes the reuse of the existing DBMS infrastructures. However, unlike CryptDB and MONOMI, the level of security provided by the encryption techniques in DBMask does not degrade with time as DBMask does not perform any intermediate decryptions in the cloud database. DBMask introduces the idea of two encryptions per each cell for fine-grained access control and predicate matching. Hence, DBMask can perform access control and predicate matching at the time of query processing by simply adding predicates to the query being executed. Unlike the existing systems, DBMask can efficient handle large database even when user dynamics change. Our experimental results show that our solution is efficient and overhead due to encryption and access control is low.

Currently, DBMask does not support all relational operations. In the future, we plan to extend DBMask to expand the supported relational operations as well as further optimize the supported relational operations.

References

[1] H. Hacigümüs, B. R. Iyer, C. Li, and S. Mehrotra, "Executing sql over encrypted data in the database-service-provider model," in *SIGMOD*, 2002, pp. 216–227.

[2] B. Hore, S. Mehrotra, and G. Tsudik, "A privacy-preserving index for range queries," in *VLDB*, 2004, pp. 720–731.

[3] E. Damiani, S. D. C. di Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati, "Balancing confidentiality and efficiency in untrusted relational dbmss," in *CCS*, 2003, pp. 93–102.

[4] S. Wang, D. Agrawal, and A. El Abbadi, "A comprehensive framework for secure query processing on relational data in the cloud," in *Secure Data Management*, 2011, pp. 52–69.

[5] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan, "Cryptdb: protecting confidentiality with encrypted query processing," in *SOSP*, 2011, pp. 85–100.

[6] N. Shang, M. Nabeel, F. Paci, and E. Bertino, "A privacy-preserving approach to policy-based content dissemination," in *ICDE 2010*.

[7] M. Nabeel and E. Bertino, "Towards attribute based group key management," in *CCS 2011*.

[8] M. Nabeel, N. Shang, and E. Bertino, "Efficient privacy preserving content based publish subscribe systems," in *SACMAT*, 2012, pp. 133–144.

[9] J.-W. Byun, E. Bertino, and N. Li, "Purpose based access control of complex data for privacy protection," in *Proceedings of the tenth ACM symposium on Access control models and technologies*, ser. SACMAT '05. New York, NY, USA: ACM, 2005, pp. 102–110.

[10] M. J. Atallah, M. Blanton, N. Fazio, and K. B. Frikken, "Dynamic and efficient key management for access hierarchies," *ACM Transaction on Information System Security*, vol. 12, no. 3, pp. 18:1–18:43, Jan. 2009.

[11] G. Chiou and W. Chen, "Secure broadcasting using the secure lock," *IEEE TSE*, vol. 15, no. 8, pp. 929–934, Aug 1989.

[12] S. Berkovits, "How to broadcast a secret," in *EUROCRYPT 1991*, pp. 535–541.

[13] X. Zou, Y. Dai, and E. Bertino, "A practical and flexible key management mechanism for trusted collaborative computing," *INFOCOM 2008*, pp. 538–546.

[14] J. Li and N. Li, "OACerts: Oblivious attribute certificates," *IEEE TDSC*, vol. 3, no. 4, pp. 340–352, 2006.

[15] T. Pedersen, "Non-interactive and information-theoretic secure verifiable secret sharing," in *CRYPTO '92*. London, UK: Springer-Verlag, 1992, pp. 129–140.

[16] C. Schnorr, "Efficient identification and signatures for smart cards," in *Proceedings of the 8th CRYPTO Conference on Advances in Cryptology*. New York, NY, USA: Springer-Verlag New York, Inc., 1989, pp. 239–252.

[17] A. Shamir, "How to share a secret," *The Communication of ACM*, vol. 22, pp. 612–613, November 1979.

[18] D. X. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *IEEE Symposium on Security and Privacy*, 2000, pp. 44–55.

[19] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the 41st annual ACM symposium on Theory of computing*, ser. STOC '09. New York, NY, USA: ACM, 2009, pp. 169–178.

[20] M. Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, "Fully homomorphic encryption over the integers," in *Advances in Cryptology EUROCRYPT 2010*, ser. Lecture Notes in Computer Science, H. Gilbert, Ed. Springer Berlin Heidelberg, 2010, vol. 6110, pp. 24–43.

[21] D. Boneh, G. Crescenzo, R. Ostrovsky, and G. Persiano, "Public-key encryption with keyword search," in *EUROCRYPT*, 2004.

[22] D. Boneh and B. Waters, "Conjunctive, subset, and range queries on encrypted data," *CRYPTO*, pp. 535–554, 2007.

[23] S. Coull, M. Green, and S. Hohenberger, "Controlling access to an oblivious database using stateful anonymous credentials," in *Irvine: Proceedings of the 12th International Conference on Practice and Theory in Public Key Cryptography*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 501–520.

[24] J. Camenisch, M. Dubovitskaya, and G. Neven, "Oblivious transfer with access control," in *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*. New York, NY, USA: ACM, 2009, pp. 131–140.

[25] J. Bethencourt, A. Sahai, and B. Waters, "Ciphertext-policy attribute-based encryption," in *SP '07: Proceedings of the 2007 IEEE Symposium on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 321–334.

[26] S. Yu, C. Wang, K. Ren, and W. Lou, "Attribute based data sharing with attribute revocation," in *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '10. New York, NY, USA: ACM, 2010, pp. 261–270.

[27] H. Hacigümüs, B. Iyer, and S. Mehrotra, "Efficient execution of aggregation queries over encrypted relational databases," in *Database Systems for Advanced Applications*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin / Heidelberg, 2004, vol. 2973, ch. 10, pp. 633–650.

[28] H. Hacigümüş, B. Iyer, and S. Mehrotra, "Query optimization in encrypted database systems," in *Database Systems for Advanced Applications*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, vol. 3453, pp. 43–55.

[29] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu, "Order preserving encryption for numeric data," in *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 2004, pp. 563–574.

[30] A. Boldyreva, N. Chenette, and A. O'Neill, "Order-preserving encryption revisited: Improved security analysis and alternative solutions," in *Advances in Cryptology CRYPTO 2011*, ser. Lecture Notes in Computer Science, P. Rogaway, Ed. Springer Berlin Heidelberg, 2011, vol. 6841, pp. 578–595.

[31] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *EUROCRYPT '99*, 1999, pp. 223–238.

[32] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich, "Processing analytical queries over encrypted data," in *Proceedings of the 39th international conference on Very Large Data Bases*, ser. PVLDB'13. VLDB Endowment, 2013, pp. 289–300.