DBMask: Fine-Grained Access Control on Encrypted Relational Databases

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

[§]Purdue University, USA
[1]msarfraz,[4]bertino@purdue.edu

[*]Oracle, USA
[2]nabeel.mohamed.nabeel@oracle.com

[#]The Institute for Infocomm Research, Singapore
[3]caojn@i2r.a-star.edu.sg

ABSTRACT

For efficient data management and economic benefits, organizations are increasingly moving towards the paradigm of “database as a service" by which 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 have not adequately addressed flexible cryptographically enforced access control on encrypted data at different granularity levels which is critical for data sharing among different users and applications. In this paper, we propose DBMask, a novel solution that supports fine-grained cryptographically enforced access control, including column, row and cell level access control, when evaluating SQL queries on encrypted data. Our solution does not require modifications to the database engine, and thus maximizes the reuse of the existing DBMS infrastructures. Our experiments evaluate the performance and the functionality of an encrypted database and results show that our solution is efficient and scalable to large datasets.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: *Access controls, Cryptographic controls*

Keywords

Encrypted Query Processing; Attribute-based Group Key Management; Database-as-a-Service

CODASPY'15, March 02 - 04 2015, San Antonio, TX, USA.
Copyright © 2015 ACM 978-1-4503-3191-3/15/03:.$15.00.
http://dx.doi.org/10.1145/2699026.2699101.

1. 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 interested in the paradigm of “database as a service". However in order to protect data from inappropriate disclosure either by the cloud or external attackers, the data are usually 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 [13, 14] and secure indexing [10, 27]. 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 [20], 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 the users and the cloud server. Authorized users log into 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 is 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 returns it to the user.

CryptDB [20] however suffers from the following limitations. The first 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 a classic deterministic encryption for equality selection, and the outer most one is a 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 query operations is at the cost of multiple decryptions of entire columns. 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). The second limitation is that the the row/cell level access control mechanism is not cryptographically enforced. Instead, CryptDB always encrypts a column using a single key and utilizes a proxy based reference monitor to enforce row level access control. When the access control is not fully enforced using cryptography, the system is susceptible to bypass and SQL injection attacks. For example, a malicious user may trick the database to return more rows than they have access to. The risk of such attacks can be reduced by utilizing cryptographic enforcement since malicious users are unable to decrypt the result. The third limitation is that the row/cell level access control policies are difficult to manage and less expressive compared to attribute based access control policies. CryptDB uses 'speaks for' relationships to specify row level access control policies. Since the policies in such a scheme are specified at user level, each user is associated with many 'speaks for' relationships and the management of policies becomes difficult. Further, CryptDB fails to process queries on the cloud server over data items encrypted with different keys for different users based on access control policy since the ciphertexts are encrypted with different keys even though the users are authorized. The forth limitation is that CryptDB is unable to execute queries that cannot be processed entirely on the server. For example, it does not support queries requiring both comparison and computation on the same column.

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

- A novel approach to support relational query operators by adding a comparison friendly encrypted column per column. Different algorithms such as order preserving encryption [6], symmetric key based searcheable encryption [25] and so on can be plugged in, depending on column data types and the security requirements.

- An approach based on an expressive attribute-based group key management scheme [24, 17, 16] to cryptographically enforce access control policies on outsourced databases at the granularity level of a table, a column, a row as well as a cell. Under our approach, 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.

- A new architecture to execute encrypted queries over an encrypted databased hosted on an untrusted cloud as a service.

The paper is organized as follows. Section 2 discusses related work. Section 3 provides an overview of our solution and the adversarial model. Section 4 introduces a fine-grained access control model and discusses its enforcement. Section 5 describes the key cryptographic constructs for SQL query operators and briefly analyzes the security of each construct. Section 6 describes the evaluation of encrypted queries over an encrypted database in the cloud. Section 7 reports experimental results. Finally, Section 8 outlines conclusions and future work.

2. RELATED WORK

In this section, we compare related work with our approach. 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 [12] to perform any arbitrary operation that a relational database requires. However, current implementations of fully homomorphic encryption are very inefficient and are not suitable for practical applications [11]. With the increasing utilization of cloud computing services, recent research efforts [7] have developed 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 access control policies involving disjunctions. By contrast, in our approach the encryption is independent of the policies.

While attribute based encryption (ABE) based approaches [5] support expressive policies, they cannot handle revocations efficiently. Yu et al. [28] proposed an approach based on ABE utilizing PRE (Proxy Re-Encryption) to address the revocation problem of ABE. While such approach 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. Samanthula et al. [21] propose an approach that relies on homomorphic encryption and PRE to handle user revocations without requiring re-encryption of relational data and also supports expressive policies. However, such approach incurs a heavy cost on the end user as all processing is done at the client side since the server does not perform secure query processing. Techniques for efficient query processing over encrypted data have also been investigated. Hacigümüs et al. [13] proposed a pioneering approach that makes it possible to perform 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. Such approach uses a 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. Asghar et al. [2] proposed a multi-user scheme that supports searches of keywords or conjuctions of keywords on untrusted servers while enforcing authorizations at table/column level. While DBMask utilizes split processing between the cloud server and the proxy for complex queries, DBMask is different in that it performs an exact query execution whenever possible and it enforces row/cell level access control utilizing the attribute based group key management (AB-GKM) scheme [17, 16], whereas the above approaches do not support fine-grained access control of the relational data.

The idea of using specialized encryption techniques such as order preserving encryption [6] and additive homomorphic encryption [18] to perform different relational operations has been introduced in CryptDB [20]. The same idea is extended to support more complex analytical queries in MONOMI [26]. As mentioned in Section 1, while these techniques lay the foundation for systematic query processing and access control, they suffer from several limitations. Similar approaches that utilize trusted hardware instead of an external proxy have also been proposed [3, 1]. Such approaches improve performance as sensitive queries are processed inside the trusted module on decrypted data. However, they require special expensive hardware modules as well as modifications to the existing database query processor.

3. OVERVIEW

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

3.1 System architecture

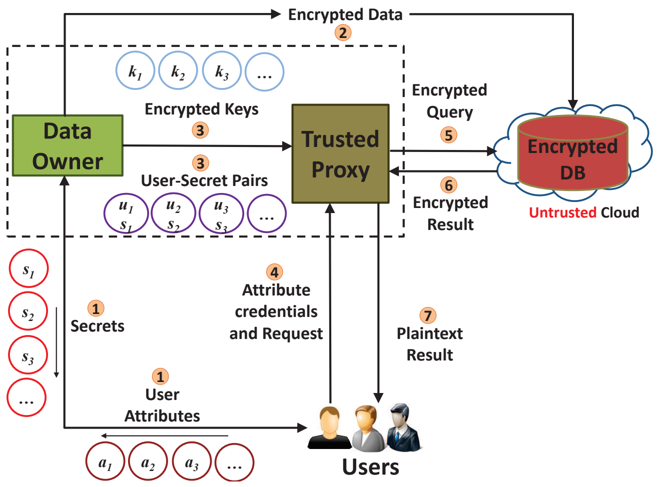

Figure 1: The system architecture

Our system includes four entities: *data owner*, *data user*, *proxy*, and *data server*. Their interactions are illustrated in the system architecture in Figure 1. The data owner uses different secret keys to encrypt different portions of data, according to the access control policies. The encrypted data are uploaded to the data server. A data user with authenticated identity 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. The encrypted query results are returned from the 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.

3.2 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. However, all the three remaining entities might be compromised. For each of them, we discuss the possible violations of data confidentiality, and how to address 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 but instead is *passive*. The server will never be given the key by which the ciphertext can be decrypted to obtain the plaintext. Still, to support query processing, we develop techniques which allow the server to efficiently evaluate SQL queries on encrypted data (see Section 6). In addition, the server itself may be compromised by external attackers. In such a case, the data confidentiality is still preserved, since the attackers cannot decrypt the data. The 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 5) 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, the data owner does not outsource the data encryption operation to the proxy, although it is trusted. This is to avoid a “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.

Users are not trusted in our system and the proxy establishes trust via certified identity attributes issued by the data owner. 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 in case a user is compromised. Instead, our key management scheme assures that the proxy is able to derive the secret keys only by collaboration with the authorized data user. 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 from posing as the authorized user any longer. Such a strategy makes it possible for the key k and the data to remain unchanged.

4. ACCESS CONTROL MODEL

In this section, we describe our access control model in detail.

4.1 Attribute based access control

We utilize the attribute based access control (ABAC) 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 is allowed to access the data item.

Access can be controlled at different granularity levels such as column level, row level, and cell level. Each column, row, or cell, depending on the desired level of access control, has an associated ABAC policy. In the case of column and cell level access control, the policy attachment is performed by adding an additional column for each column in the table and in the case of row level access control, the policy attachment is performed by adding a single additional column in the table (Please refer to Section 6.3 and 6.4 for explanation on policy attachment at different granularities of access control). Upon receiving an SQL query from a user for table T, the proxy needs to determine the ABAC policies attached to T satisfied by the users attributes and restrict the query to only those columns, rows or cells depending on the granularity level by adding a predicate to the user query. Such a predicate "encodes" the satisfied ABAC policies as shown by example queries in Section 6. We focus on cell level access control to propose our basic ABAC model. In the basic model, each cell in a database table is attached an ABAC policy. We formally define our model as follows:

DEFINITION 4.1. **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 4.2. **ABAC Policy**.
Let T *be a table. An ABAC access control policy (*ACP *for short) defined over* T *is a tuple* (s, o) *where:* o *denotes a set of cells in* T *and* s *is a Boolean expression over a set of attribute conditions that must be satisfied in order to access* o.

DEFINITION 4.3. **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 role based access control (RBAC), but in our approach, the assignment is performed automatically based on identity attributes. Given the set of ABAC policies specified by the data owner, the following steps are taken to identify groups:

- Convert each ABAC ACP into disjunctive normal form (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 defined over the attribute conditions C_1, C_2 and C_3: ACP$_1$ = $C_1 \wedge (C_2 \vee C_3)$ and ACP$_2$ = C_2. Then the corresponding policies in DNF are as follows: ACP$_1$ = $(C_1 \wedge C_2) \vee (C_1 \wedge C_3)$ and ACP$_2$ = C_2.

In this example, there are three groups G_1, G_2, G_3 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 4.4. **Group Poset**.
A group poset is defined as the partially ordered set (poset) of groups where the binary relationship is $\subseteq$.

4.2 Assigning group labels and hierarchical access control

We label the cells using descriptive group names where each cell may have multiple groups associated with it. If there is an ordering relationship between two groups associated with a cell, we discard the more privileged group and assign only the less privileged group. When the proxy decides the group(s) that a 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 indirectly attaches an ABAC policy to a cell as described at the beginning of this section.

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. We address this drawback while utilizing the benefits of hierarchical model by proposing a hybrid approach combining broadcast and hierarchical key management. We utilize a recently proposed expressive scheme called AB-GKM (attribute based GKM) [17, 16] as the broadcast GKM scheme which is described in Section 5.1. 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. An 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.

5. CRYPTOGRAPHIC CONSTRUCTS

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

5.1 Key management

Broadcast Group Key Management (BGKM) schemes [8, 4, 24] 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. The recently proposed attribute based GKM (AB-GKM) scheme [17, 16] 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 the oblivious commitment based envelope (OCBE) protocols [15] which are based on Pedersen commitments [19] [1] and zero knowledge proof of knowledge

[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.

techniques [22]. 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.

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 $\mathcal{T}$. 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 [23] 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**. The **Setup** algorithm takes the security parameter ℓ, the maximum group size N, and the number of attribute conditions N_a as input, and initializes the system. The **SecGen** algorithm gives a user_j, $1 \leq j \leq N$, a set of secrets for each commitment $\mathsf{com}_i \in \gamma$, $1 \leq i \leq m$. The **KeyGen** 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 $\mathcal{T}$. Given the set of identity attributes β, the set of public information tuples **PI**, and the access tree $\mathcal{T}$, the **KeyDer** algorithm outputs the symmetric K only if the identity attributes in β satisfy the access structure $\mathcal{T}$. 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. If an adversary 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.

5.2 SQL-aware comparison

DBMask provides support for both numerical and keyword comparison and is designed so that any comparison friendly numerical or keyword encryption scheme can be utilized to perform relational operations over encrypted data. In the case of numerical matching, we use two variants of AES and Boldyreva et al.'s [6] schemes to support privacy preserving comparison without requiring the decryption of numerical values. We refer to these approaches as privacy preserving numerical comparison (PPNC). For the purpose of reference to individual schemes, we refer to them as PPNC-SEM, PPNC-DET and PPNC-OPE. PPNC-SEM provides the maximum security guarantee among the PPNC schemes and is constructed using AES together with a blinding factor where a simple blinding factor would be,

$$g^r \text{ where } g \text{ is a generator and } r \in \mathbb{Z}_p$$

Since the goal is to reveal equality, the values are unblinded at the time of comparison by providing the unblinding factor (g^{-r})to the data server where the server unblinds on the fly. PPNC-DET has a weaker security guarantee than PPNC-SEM as it is deterministic encryption and is constructed using 128-bit block AES. PPNC-OPE is Boldyreva et al.'s scheme and has the weakest security guarantee among the PPNC schemes as it is an order-preserving encryption and the encrypted values reveal order of the plaintext. Like numerical comparison, one can utilize any encrypted keyword comparison technique [25, 9]. We refer to this approach as privacy preserving keyword comparison (PPKC). We adopt the keyword search technique proposed in [25] (PPKC-SEM) as it is better suited to relational data and its implementation is available. We provide an abstract description of how these comparison schemes are used in DBMask.

The above approaches can be summarized into four algorithms namely, **Setup**, **EncVal**, **GenTrapdoor** and **Compare**, which we use for comparison in our cloud based database system. The **Setup** algorithm takes as input a set of parameters P and initializes the underlying encryption scheme required for computations. Given a numerical or a keyword value x, the **EncVal** algorithm produces an encrypted value e_x that hides the actual value, but allows one to perform comparisons using the trapdoor value. Given an input (numerical or keyword) value t, the **GenTrapdoor** algorithm produces an encrypted value e_t, called the trapdoor, that is used with its corresponding encrypted value to perform comparisons. Given an encrypted value e_x for x and a trapdoor value e_t for t, the **Compare** algorithm compares the encrypted and trapdoor value and outputs the result.

Brief security analysis: An adversary, who has compromised the cloud server, cannot infer the plaintext values of the encrypted values except what is inherently revealed by the underlying encryption schemes since the private key is not stored at the cloud server. If an adversary has compromised both the proxy and the cloud server, the adversary cannot directly infer the plaintext values using the private information stored at the proxy since the private information used at the data owner to generate the encrypted value and the private information used at the proxy to generate trapdoors is different and it is cryptographically hard to derive one from the other. The adversary may however do a brute force attack by repeatedly executing 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.

5.3 Computing joins

In order to perform equality join, the joining columns have to be encrypted with the same key so the server can see matching values between two columns. Although the columns using either PPNC or PPKC scheme are encrypted with the same key in our approach, they cannot be matched since values are semantically secure. One way to support join would be to update one of the columns in the join operation at runtime to reflect its trapdoor values and then perform the join operation, but this has high computational and bandwidth complexity as it requires either downloading the encrypted values to the proxy, creating trapdoors and uploading to the database server or keeping a mapping between the encrypted values and trapdoors at the proxy and uploading it to database server prior to join operation. In order to efficiently perform join, we use two schemes namely JOIN-SEM and JOIN-DET with varying security guarantees. JOIN-SEM introduces a new column and stores the blinded trapdoor values corresponding to the comparison friendly PPNC or PPKC encrypted values in this column. The blinding mechanism of trapdoor values and the process of equality matching is as explained earlier in Section 5.2. JOIN-DET encrypts every column able to participate in join with a deterministic encryption scheme prior to uploading to the server and a mapping between joining columns and their respective tables is kept at the proxy. JOIN-SEM provides better security guarantees that JOIN-DET.

Brief security analysis: An adversary, who has compromised either the cloud server or the proxy cannot infer the plaintext values of the encrypted values as the the private key is not stored at the cloud server or the proxy but may repeatedly execute comparison operations to infer the plaintext values of the encrypted values in the cloud server.

6. 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 3, our system consists of four entities: data owner, proxy, cloud and users. Our system undergoes the following phases: system initialization, user registration, data encryption and upload, and data querying and retrieval. We now explain each phase in detail.

6.1 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 groups users satisfying the same disjunctive clauses. As mentioned in Section 4, these groups are used to construct the Group poset to perform hierarchical key derivation along with the AB-GKM based key management.

6.2 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 is (are) deleted. The user-secret database is also stored at the proxy with the secrets encrypted using a password only each user possesses. Each user has a different password encrypting her own secrets. 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.

6.3 Data encryption and upload

In our solution, each cell in an original table is expanded into either three if JOIN-DET is used as the scheme to perform join operation or four if JOIN-SEM is used to perform join. The first cell is encrypted for fine-grained access control, the second cell is encrypted for privacy-preserving matching ,the third cell represents the assigned group labels and the fourth cell if exists stores blinded trapdoor value for join operation. We denote the column resulting from the encryption for fine-grained access control as *data-col*, the one resulting from the encryption for privacy-preserving matching as *match-col*, the one with associated group names as *label-col* and the one resulting from storing blinded trapdoor as *trap-col*.

Example: Consider the example shown in Section 4 and 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.

Table 1: Assignment Table

Table	Columns	Condition	Groups
Patient	ID, Age, Diagnosis	Age < 40	G_1
Patient	Age	Age > 30	G_2
Patient	Age, Diagnosis	Age < 40 **AND** Diagnosis = 'Asthma'	G_3

Let us first discuss the creation of label-col. Suppose that the above ACPs are applied over a set of cells that satisfy the conditions shown in Table 1 and each cell is assigned one or more group labels as shown in Table 2. If two groups are connected in the group poset, only the label of less privileged group is assigned to the cell e.g. the Age value in row 4 is satisfied by both G_1 and G_3 but only assigned label of less privileged group, G_3. Note that the table and column names in the assignment table stored on the untrusted server are anonymized and the values encrypted.

Table 2: Patient Table

ID	ID-grp	Age	Age-grp	Diag	Diag-grp
1	G_1	35	G_1,G_2	HIV	G_1
2	G_1	30	G_1	Cancer	G_1
3	G_1	40	G_2	Asthma	G_1
4	G_1	38	G_2,G_3	Asthma	G_3

Now, let us 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 the AB-GKM scheme [17, 16]. The set of groups associated with a cell decide the key under which the cell is encrypted. For each 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) in the case where a cell is associated with multiple groups, 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 e.g. the Age value in row 1 is encrypted with a master key k_{12} generated from the AB-GKM instance having k_1 and k_2 as input secrets with public information corresponding to this master key as PI_{12}. As a consequence, if a user belongs to any of the groups assigned to the cell, the user can access the cell 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*.

Table 3: PubInfo Table

Groups	G_1	G_2	G_3	G_1, G_2	G_2, G_3
PI	PI_1	PI_2	PI_3	PI_{12}	PI_{23}

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, the 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. Table 4 shows the final table with both encrypted data-col's where $E_k(x)$ refers to the semantically secure encryption of the value x using the symmetric key k and comparison friendly match-col's, where $comp_n$ and $comp_k$ refer to PPNC.EncVal and PPKC.EncVal respectively.

Now, let us consider the creation of the trap-col. Given a cell in the original table, its value in the trap-col is generated by the PPNC.GenTrapDoor**blinding factor* or PPKC.GenTrapDoor**blinding factor* depending on the data type. The blinding factor is as explained in Section 5.2.

6.4 Data querying and retrieval

Processing a query over encrypted data is a *filtering-refining* procedure. The general algorithm for processing queries on encrypted data is shown in Algorithm 1 and the details are as follows. An authorized user sends a plaintext SQL query to the proxy, as if the outsourced database were unencrypted. In other words, encryption and decryption of the data in the database is transparent to users. The proxy parses the query and generates an abstract syntax tree of the query.

The query is first filtered (Lines 7-13) by removing clauses, aggregate functions, and predicates with aggregate functions that cannot be computed on the server. The `PART` function (Lines 15-17) then adds the columns referenced by filtered clauses or aggregate functions to the projections of the filtered query. The query is then rewritten for the cloud (Lines 19-21) by the `REWRITE` function by which each column to be included in the query result (i.e., column following the `SELECT` keyword in the query) is replaced by its corresponding "data-col" and each predicate in the `WHERE` clause is replaced with a user defined function (UDF). For each numerical matching predicate, the UDF includes the trapdoor value computed by the proxy using PPNC.GenTrapdoor algorithm and invokes the PPNC.Compare algorithm. Similarly, for each keyword matching predicate, the UDF includes the trapdoor value computed by the proxy using PPKC.GenTrapdoor algorithm and invokes the PPKC.Compare algorithm. The `REWRITE` function then adds a predicate to the `WHERE` clause that determines the group(s) of the user requesting the query before the rewritten query is sent to the cloud server.

Algorithm 1 Pseudo-code for SECUREQUERYPLAN

```
1:  Input:   $Q$, abstract syntax tree (AST) for the query
2:           $M$, metadata of the target table(s)
3:           $G$, group(s) a user is member of
4:  Output:  $P$, a query plan for $Q$
5:  $FilterQ \leftarrow Q$
6:  $ProxyFilters \leftarrow []$
7:  for $f$ in $FilterQ$ do
8:      $f' \leftarrow$FILTER$(f, FilterQ)$
9:      for $f' \neq$ Nil do
10:         Remove $f$ from $FilterQ$
11:         Add $f'$ to $ProxyFilters$
12:     end for
13: end for
14: for $p$ in $ProxyFilters$ do
15:     if PART($p$) in $FilterQ.relations$ then
16:         Add PART($p$) to $FilterQ.projections$
17:     end if
18: end for
19: for $c$ in $s.where\_clause \mid\mid s.select\_clause$ do
20:     REWRITE$(c, s, M, G)$
21: end for
22: $P \leftarrow$ SERVERSQL($FilterQ$)
23: $P \leftarrow$ DECRYPT ($P$)
24: if ProxyFilters $\neq$ [] then
25:     $P \leftarrow$ PROXYSQL($P$, ProxyFilters, $Q$)
26: end if
27: return $P$
```

The cloud executes the rewritten encrypted query over the encrypted database and filters the tuples that do not satisfy the predicates in the query before sending back the encrypted result set to the proxy (Line 22). The proxy generates the necessary keys for decrypting 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 (Line 23).

If the proxy has removed some clauses and/or aggregate functions (e.g., `SUM`) from the original query in the query filtering step, 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 by running the original query (Line 24-26). If no term from the query is removed, the decrypted result set is the final result and the proxy sends the final plaintext result back to the user.

We now illustrate query processing in DBMask through example queries. The queries reflect two scenarios. The first scenario which we refer to as DBMask-SEC provides maximum security and uses PPNC-SEM scheme for numerical comparison, PPKC-SEM scheme for keyword search, JOIN-SEM for computing joins and the label columns reflecting group information are encrypted. The second scenario which we refer to as DBMask-PER provides best performance and uses PPNC-OPE schem for numerical comparison, PPKC-SEM scheme for keyword search, JOIN-DET scheme for computing joins and the label columns are in plaintext.

A user having the attributes "role = doctor" and "level = 4" executes Query 1 through the proxy server.

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

Table 4: Encrypted Patient Table - Cell Level Access Control with JOIN-SEM

ID-enc	ID-com	ID-trap	ID-grp	Age-enc	Age-com	Age-trap	Age-grp	Diag-enc	Diag-com	Diag-trap	Diag-grp
$E_{k_1}(1)$	$comp_n(1)$	$comp'_n(1)$	G_1	$E_{k_{12}}(35)$	$comp_n(35)$	$comp'_n(35)$	G_1,G_2	E_{k_1}(HIV)	$comp_k$(HIV)	$comp'_k$(H..)	G_1
$E_{k_1}(2)$	$comp_n(2)$	$comp'_n(2)$	G_1	$E_{k_1}(30)$	$comp_n(30)$	$comp'_n(30)$	G_1	E_{k_1}(Cancer)	$comp_k$(Cancer)	$comp'_k$(Ca..)	G_1
$E_{k_1}(3)$	$comp_n(3)$	$comp'_n(2)$	G_1	$E_{k_2}(40)$	$comp_n(40)$	$comp'_n(40)$	G_2	E_{k_1}(Asthma)	$comp_k$(Asthma)	$comp'_k$(As..)	G_1
$E_{k_1}(4)$	$comp_n(4)$	$comp_n(4)$	G_1	$E_{k_{23}}(38)$	$comp_n(38)$	$comp'_n(38)$	G_2,G_3	E_{k_3}(Asthma)	$comp_k$(Asthma)	$comp'_k$(As..)	G_3

Table 5: Encrypted Patient Table - Row Level Access Control with JOIN-DET

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}(Asthma)	$comp_k$(Asthma)	G_1

Query 1:
SELECT *ID*, *Age*, *Diag*
FROM *Patient*
WHERE *Age* $>$ 35 **AND** *Diag* **LIKE** 'Asthma'
ORDER BY *Age* **ASC**

The proxy determines that the user is a member of groups G_1 and G_3. It thus re-writes the query as Query 2 if the underlying scenario is DBMask-SEC or as Query 3 if the underlying scenario is DBMask-PER and submits to the cloud server. Notice that the 'ORDER BY' clause is removed from Query 2. The reason being that the cloud server does not have sufficient information to order the query results.

Query 2:
SELECT *ID-enc*, *ID-grp*, *Age-enc*, *Age-grp*, *Diag-enc*, *Diag-grp*
FROM *Patient*
WHERE UDF_Compare_Num(*unblinding factor* * *Age-com*, PPNC.GenTrapdoor(35), '$>$') **AND**
UDF_Compare_Str(*Diag-com*, PPKC.GenTrapdoor('Asthma')
AND UDF_Compare_Str(*ID-grp*, PPKC.GenTrapdoor('G_1')) **OR**
UDF_Compare_Str(*ID-grp*, PPKC.GenTrapdoor('G_3'))
AND UDF_Compare_Str(*Age-grp*, PPKC.GenTrapdoor('G_1')) **OR**
UDF_Compare_Str(*Age-grp*, PPKC.GenTrapdoor('G_3'))
AND UDF_Compare_Str(*Diag-grp*, PPKC.GenTrapdoor('G_1')) **OR**
UDF_Compare_Str(*Diag-grp*, PPKC.GenTrapdoor('G_3'))

UDF_Compare_Num is a user defined function that invokes the PPNC.Compare algorithm and UDF_Compare_Str is a user defined function that invokes the PPKC.Compare algorithm. The cloud server returns the encrypted row 4 to the proxy. In order to decrypt the resultset, the proxy requires the keys k_1 to decrypt *ID-enc* column value, k_{23} to decrypt *Age-enc* column value and k_3 to decrypt *Diag-enc* column value. The proxy derives the key k_1 for the higher privileged group G_1 using the AB-GKM scheme. In order to generate k_3, instead of executing another AB-GKM key derivation algorithm, the proxy utilizes the hierarchical key derivation to derive k_3 from k_1. To derive key k_{23}, the proxy uses k_3 and PI_{23} to derive k_{23}. The proxy then uses k_1, k_3 and k_{23} to decrypt the resultset and sends the resultset to the user. In the case of Query 2, the proxy orders the plaintext resultset using its in-memory database before sending the final resultset to the user.

Query 3:
SELECT *ID-enc*, *ID-grp*, *Age-enc*, *Age-grp*, *Diag-enc*, *Diag-grp*
FROM *Patient*
WHERE UDF_Compare_Num(*Age-com*, PPNC.GenTrapdoor(35), '$>$') **AND**
UDF_Compare_Str(*Diag-com*, PPKC.GenTrapdoor('Asthma')
AND (*ID-grp* **LIKE** '%G_1%' **OR** *ID-grp* **LIKE** '%G_3%')
AND (*Age-grp* **LIKE** '%G_1%' **OR** *Age-grp* **LIKE** '%G_3%')
AND (*Diag-grp* **LIKE** '%G_1%' **OR** *Diag-grp* **LIKE** '%G_3%')
ORDER BY *Age-com* **ASC**

A user having the attribute "role = doctor" executes Query 4 through the proxy server. The proxy determines that the user is a member of the group G_3, re-writes the query and submits it to the cloud server. The rewritten query is Query 5 if the underlying scenario is DBMask-SEC and Query 6 if the underlying scenario is DBMask-PER. Note that Query 5 performs join by unblinding the trapdoor column on the fly. The blinding and/or unblinding mechanism of trapdoor values and the process of equality matching is as explained earlier in Section 5.2. In Query 6, the joining columns use the equality mechanism of the DBMS and not a manually implemented UDF since the columns are encrypted using a deterministic encryption scheme. The proxy receives and decrypts the encrypted result set using the mechanism explained above and sends the plaintext result back to the user.

Query 4:
SELECT *p.Age*, *d.Description*
FROM *Patient p*, *Diagnosis d*
WHERE *p.ID* $=$ *d.PatientID*

Query 5:
SELECT *p.Age-enc*, *p.Age-grp*, *d.Description-enc*, *d.Description-grp*
FROM *Patient p*, *Diagnosis d*
WHERE UDF_Compare_Num(*p.ID-trap*, *unblinding factor* * d.PatientID-trap,'$=$')
AND UDF_Compare_Str(*p.Age-grp*, PPKC.GenTrapdoor('G_3'))
AND UDF_Compare_Str(*d.Description-grp*, PPKC.GenTrapdoor('G_3'))

Query 6:

SELECT *p.Age-enc*, *p.Age-grp*, *d.Description-enc*, *d.Description-grp*
FROM *Patient p*, *Diagnosis d*
WHERE *p.ID-com* = *d.PatientID-com*
AND (*p.Age-grp* **LIKE** '%G_3%')
AND (*d.Description-grp* **LIKE** '%G_3%')

Table 5 shows the *Patient* table enforcing row level access control. Row level access control is a special case in our scheme where there is only a single additional *label-col* in the table to assign group labels i.e. *Groups*. Query 7 shows the transformation of Query 1 issued by the same user under DBMask-PER scenario. The cloud server returns encrypted rows 3 and 4 to the proxy. The decryption mechanism of the returned results is as explained above.

Query 7:

SELECT *ID-enc*, *Age-enc*, *Diag-enc*, *Groups*
FROM *Patient*
WHERE UDF_Compare_Num(*Age-com*, PPNC.GenTrapdoor(35), '>') **AND** UDF_Compare_Str(*Diag-com*, PPKC.GenTrapdoor('Asthma') **AND** (*Groups* **LIKE** '%G_1%' **OR** *Groups* **LIKE** '%G_3%')
ORDER BY *Age-com* **ASC**

6.5 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 transparently 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 requires both the proxy server and the cloud server to update the data. Notice that unlike 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 which the new user is not a member.

7. 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 as all functionality on the server side is implemented using UDF's. We use the memory storage engine of MySQL as the in-memory database at the proxy to store the contents of a query when the execution of a query cannot be completed entirely on the server. The cryptographic operations are supported by using the NTL library[4] while the access control policies expressed as boolean expressions are converted into DNF using the boolstuff library[5]. The 'data-col' in each table is constructed with 128-bit block size AES in CBC mode while the 'match-col' is encrypted with 128 bit key using the underlying PPNC or PPKC schemes. The experimental setup is run on 3.40 GHz Intel i7-3770 8 core processors with 8 GB of RAM in Ubuntu 12.04 environment. We compare the performance of our prototype by running a TPC-C query workload utilizing only a single group under row-level control to evaluate only the encryption/decryption and comparison schemes. The experiment on a web based scientific application called Computational Research Infrastructure for Science (CRIS) is done to analyze the access control functionality of our prototype. The experimental results below show a low runtime overhead with a 33% 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 it has access to and there is no unauthorized access with a modest overhead on performance.

7.1 TPC-C

The TPC-C workload queries consist of comparison predicates (=, < , >) ,other predicates such as `DISTINCT` and `COUNT`, aggregates such as `SUM` and `MAX` and sort operation `ORDER BY`. In total, the workload contains 53% Select, 4% Delete, 30% Update and 13% Insert statements. The performance of TPC-C query workload with transactions enabled is compared by running the workload on an unmodified Postgres server against running the workload through the proxy of CryptDB and the proxy of our prototype DBMask under two different scenarios as explained earlier in Section 6.4, namely DBMask-SEC and DBMask-PER. 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.

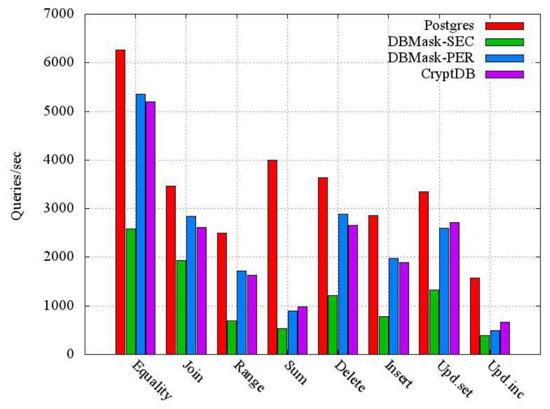

Figure 2: Throughput from TPC-C workload running under Postgres, DBMask, and CryptDB

Figure 2 shows the server throughput. The results show that in comparison to running an unencrypted trace of TPC-C workload on Postgres, there is an overall loss of throughput by: 34% for CryptDB, 33% for DBMask with underlying scenario as DBMask-PER and 68% for DBMask with underlying scenario as DBMask-SEC. DBMask outperforms CryptDB in most cases. CryptDB per-

[4]http://www.shoup.net/ntl/
[5]http://sarrazip.com/dev/boolstuff.html

forms best for Upd.inc operations where data is incremented before being updated and Sum operations since such operations cannot be performed on the server by DBMask. The data to be updated is first fetched from the server, updated at the proxy and sent back to the server. The same applies to Sum operations. In general, DBMask provides support for processing a greater range of queries by supporting split execution between proxy and server. We consider a throughput of 33% for encrypted query processing to be modest considering the gains in confidentiality and privacy. DBMask-PER provides better performance over DBMask-SEC as DBMask-PER provides indexing support for equality operations whereas DBMask-SEC does not, computes range operations on the server whereas DBMask-SEC first filters the results at the server before performing the range operation at the proxy, and uses the DBMS equality mechanism for join operations and matching group labels whereas DBMask-SEC uses user-defined functions for equality matching.

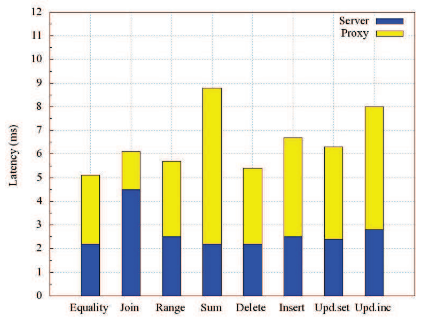

Figure 3: Server and proxy latency

To assess latency, we measure the processing time of the same type of queries used above by studying the intervals at each stage of processing, namely at server and proxy. Fig 3 shows server and proxy latency for several query operations with DBMask-PER as the underlying scenario. We observe that there is an overall increase by 44% on the server side. The proxy adds on average 4.3 ms to the interval of which 24% is utilized in encryption/decryption and the most (67%) is spent in query rewriting, parsing and processing. A Select Sum statement returned a latency of 0.91 ms on an unmodified Postgres database and a latency of 8.8 ms with DBMask primarily due to further processing of the query at the proxy.

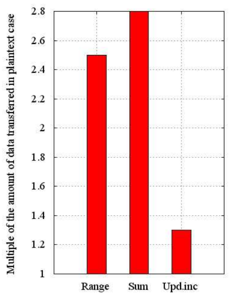

Figure 4: Server data transferred between proxy and server

To assess the bandwidth utilization incurred by split execution of queries between proxy and server by DBMask, we evaluate the average data transferred from the server to the proxy. Fig 4 shows average data transferred for those queries that cannot be processed entirely on the server by DBMask. In the worst case, the data transferred is 2.8x in comparison to data transferred from server when plaintext queries are executed entirely on the server. This does not significantly increase the bandwidth requirements between the proxy and the server.

Table 6: Server space for TPC-C Workload

System	Size (GB)	Relative to plaintext
Plaintext	1.2	-
CryptDB	5.4	4.5x
DBMask-PER	3.8	3.2x

Table 6 shows the amount of disk space used on the server by plaintext database, DBMask and CryptDB. DBMask increases the amount of data stored by 3.2 times and hence would not result in significant increase to storage cost. This is also relatively small in comparison to the space overhead imposed by CryptDB (4.5x).

7.2 CRIS

CRIS is a web based application supporting an easy to use system for managing and sharing scientific data. 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. To test the functionality of DBMask, 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. We define four ACPbased on six attribute conditions over user identity attributes that capture the access control requirements of this particular workspace in CRIS. The users based on the mechanism explained in Section 3 are arranged into groups and each group is assigned a randomly chosen secret using AB-GKM. To evaluate the performance overhead imposed by DBMask and study the influence of access control, we run plaintext queries on Postgres and compare it to running queries on DBMask at different granularities of access control namely column level, row level and cell level. CRIS database has 87 tables with 298 columns in total.

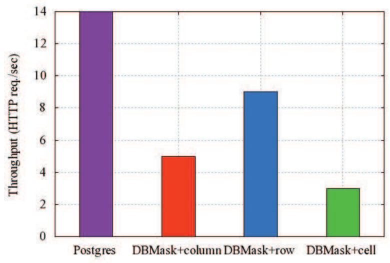

Figure 5: Throughput comparison of Sylvie's Workspace at row level access control

Figure 5 shows the effect on throughput by running CRIS on Postgres in comparison to DBMask with the underlying scenario DBMask-PER. Each HTTP request by a logged in user consists of multiple queries in order to allow a user to create, read, update and/or delete a project(s), experiment(s) or job(s). The results show that there is a loss of throughput by 64% for column level, 36% for row level and 79% for cell level access control with DBMask and a logged in user is only able to access objects it is permitted. We consider this to be a reasonable overhead considering the gains in confidentiality and privacy. The finer granularity of row level over column level results in better performance as row level consumes less disk space and is able to take advantage of indexing to speed table scans. Our scheme is best suited for row level access control.

8. CONCLUSION

In this paper, we proposed DBMask, a novel solution that supports cryptographically enforced fine-grained access control, including row level and cell level access control, when evaluating SQL queries on encrypted relational data. Similar to CryptDB [20] and MONOMI [26], DBMask does not require modification to the database engine, and thus maximizes the reuse of existing DBMS infrastructures. However, unlike CryptDB and MONOMI, the level of security provided by the encryption techniques in DBMask does not change with time as DBMask does not perform any intermediate decryptions in the cloud database. DBMask introduces the idea of splitting fine-grained access control and predicate matching per each cell. Hence, DBMask can perform access control and predicate matching at the time of query processing by simply adding predicates to the query being executed. The choice of predicate matching technique used is configurable so that different techniques can be plugged in depending on the requirements. Unlike existing systems, DBMask can efficiently handle large databases even when user dynamics change. Our experimental results show that our solution is efficient and overhead due to encryption and access control is low.

As future work, we plan to extend DBMask to expand the supported relational operations as well as further optimize the supported relational operations. We also plan to investigate the feasibility of moving some of the proxy functionality into the cloud as a trusted component where the cloud service provider has access to neither the encryption keys nor the internal processing.

9. ACKNOWLEDGMENT

The work reported in this paper has been partially supported by the Purdue Cyber Center and by the National Science Foundation under grant CNS-1111512.

10. REFERENCES

[1] A. Arasu, S. Blanas, K. Eguro, M. Joglekar, R. Kaushik, D. Kossmann, R. Ramamurthy, P. Upadhyaya, and R. Venkatesan. Secure database-as-a-service with cipherbase. In *SIGMOD 2013*, pages 1033–1036, New York, NY, USA. ACM.

[2] M. R. Asghar, G. Russello, B. Crispo, and M. Ion. Supporting complex queries and access policies for multi-user encrypted databases. In *CCSW 2013*, pages 77–88. ACM.

[3] S. Bajaj and R. Sion. Trusteddb: A trusted hardware based database with privacy and data confidentiality. In *SIGMOD 2011*, pages 205–216, New York, NY, USA. ACM.

[4] S. Berkovits. How to broadcast a secret. In *EUROCRYPT 1991*, pages 535–541.

[5] J. Bethencourt, A. Sahai, and B. Waters. Ciphertext-policy attribute-based encryption. In *SP 2007*, pages 321–334, Washington, DC, USA. IEEE Computer Society.

[6] A. Boldyreva, N. Chenette, and A. O'Neill. Order-preserving encryption revisited: Improved security analysis and alternative solutions. In *CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 578–595. Springer Berlin Heidelberg, 2011.

[7] J. Camenisch, M. Dubovitskaya, and G. Neven. Oblivious transfer with access control. In *CCS 2009*, pages 131–140, New York, NY, USA. ACM.

[8] G. Chiou and W. Chen. Secure broadcasting using the secure lock. *IEEE TSE*, 15(8):929–934, Aug 1989.

[9] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. In *CCS 2006*, pages 79–88, New York, NY, USA. ACM.

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

[11] M. Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan. Fully homomorphic encryption over the integers. In H. Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 24–43. Springer Berlin Heidelberg.

[12] C. Gentry. Fully homomorphic encryption using ideal lattices. In *STOC 2009*, pages 169–178, New York, NY, USA. ACM.

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

[14] B. Hore, S. Mehrotra, and G. Tsudik. A privacy-preserving index for range queries. In *VLDB 2004*, pages 720–731.

[15] J. Li and N. Li. OACerts: Oblivious attribute certificates. *IEEE TDSC*, 3(4):340–352, 2006.

[16] M. Nabeel and E. Bertino. Poster. towards attribute based group key management. In *CCS 2011*, pages 821–824.

[17] M. Nabeel and E. Bertino. Attribute based group key management. *To Appear in Transactions on Data Privacy*, 2014.

[18] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT 1999*, pages 223–238.

[19] T. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *CRYPTO 1992*, pages 129–140, London, UK. Springer-Verlag.

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

[21] B. K. Samanthula, G. Howser, Y. Elmehdwi, and S. Madria. An efficient and secure data sharing framework using homomorphic encryption in the cloud. In *CLOUD-I 2012*, pages 8–16. ACM, 2012.

[22] C. Schnorr. Efficient identification and signatures for smart cards. In *CRYPTO 1989*, pages 239–252, New York, NY, USA. Springer-Verlag New York, Inc.

[23] A. Shamir. How to share a secret. *The Communication of ACM*, 22:612–613, November 1979.

[24] N. Shang, M. Nabeel, F. Paci, and E. Bertino. A privacy-preserving approach to policy-based content dissemination. In *ICDE 2010*, pages 944–955.

[25] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *SP 2000*, pages 44–55.

[26] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing analytical queries over encrypted data. In *PVLDB 2013*, pages 289–300. VLDB Endowment.

[27] S. Wang, D. Agrawal, and A. El Abbadi. A comprehensive framework for secure query processing on relational data in the cloud. In *SDM 2011*, pages 52–69.

[28] S. Yu, C. Wang, K. Ren, and W. Lou. Attribute based data sharing with attribute revocation. In *ASIACCS 2010*, pages 261–270, New York, NY, USA. ACM.