TRANSACTIONS ON DATA PRIVACY 9 (2016) 187–214

DBMask: Fine-Grained Access Control on Encrypted Relational Databases

Muhammad I Sarfraz*, **Mohamed Nabeel****, **Jianneng Cao*****, **Elisa Bertino***

*Purdue University, West Lafayette, IN, 47907, USA .

**Oracle, Redwood City, CA, 94065, USA.

***Institute for Infocomm Research, Singapore 13862.

E-mail: msarfraz@purdue.edu, nabeel.mohamed.nabeel@oracle.com, caojn@i2r.a-star.edu.sg, bertino@purdue.edu

Received 29 September 2015; received in revised form 7 October 2016; accepted 7 October 2016

Abstract. DBMask is a system that implements encrypted query processing with support for complex queries and fine grained access control with *create*, *update*, *delete* and cryptographically enforced *read* (CRUD) operations for data stored on an untrusted database server hosted in a public cloud. Past research efforts have not adequately addressed flexible access control on encrypted data at different granularity levels which is critical for data sharing among different users and applications. DBMask proposes a novel technique that separates fine grained access control from encrypted query processing when evaluating SQL queries on encrypted data and enforces fine grained access control at the granularity level of a column, row and cell based on an expressive attribute-based group key encryption scheme. DBMask does not require modifications to the database engine, and thus maximizes the reuse of the existing DBMS infrastructures. Our experiments evaluate the performance of an encrypted database, managed by DBMask, using queries from TPC-H benchmark in comparison to plaintext Postgres. We further evaluate the functionality of our prototype using a policy simulator and a multi-user web application. The results show that DBMask is efficient and scalable to large datasets.

Keywords. Encrypted query processing; attribute-based group key management; database-as-a-service

1 Introduction

The increasing demand for cost-effective and efficient data management has resulted in organizations adopting the "database as a service" paradigm. According to such paradigm, data are managed by a database management system (DBMS) hosted in a public cloud. 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. A market analysis firm projected an 86% annual increase in organizations adopting the paradigm of "database as a service" and an increase in revenue for "database as a service" providers from $150 million in 2012 to $1.8 billion by 2016. [1] However, data are the most valuable asset in an organization and inappropriate data disclosure puts the organization's business at risk.

[1] http://www.forbes.com/sites/oracle/2013/12/05/why-database-as-a-service-dbaas-will-be-the-breakaway-technology-of-2014/

A solution commonly adopted to protect data confidentiality relies on encryption. However, the use of encryption raises issues related to the efficient processing of queries on encrypted data. In order to address such issues, various techniques such as bucketization [15, 16] and secure indexing [11, 26] have been investigated. 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. An additional requirement is that fine-grained access control be cryptographically enforced. When the access control is not fully enforced using cryptography, the system is susceptible to bypasses and SQL injection attacks. For example, a malicious user may trick the database into returning more rows than a user has access to. The risk of such attacks can be reduced by utilizing cryptographic enforcement since malicious users are unable to decrypt the result. Therefore, a suitable mechanism for allowing the data owner to enforce fine-grained access control over data on untrusted cloud server is to encrypt data through certain cryptographic primitives but disclose decryption keys only to authorized users. While such a mechanism preserves confidentiality, it does not protect against active attacks such as updating or inserting garbled data or deleting records. Since unauthorized users do not have the keys to decrypt data, they cannot update or insert valid records, but can insert or update garbled data and delete records. In this paper, we propose DBMask, a novel approach to address fine-grained access control over encrypted data. DBMask 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. However, CryptDB suffers from limitations related to both encrypted query processing and fine grained access control.

With respect to limitations related to encrypted query processing, CryptDB uses onions of encryption to support multiple ciphertexts where each ciphertext corresponds to an onion layer. Each layer is applied to a specific query operation or purpose, and the encryption layers from the external layer to the most internal layer are increasingly weaker. Given a specific query, the layers of the onion are peeled off in order to support layer-specific query operations. 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. In DBMask, there are multiple ciphertexts where each ciphertext is only a single layer of encryption to support a comparison operation. The data are never decrypted to weaker encryptions inside the cloud server and therefore the security of the data does not weaken over time. Another 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. In contrast, DBMask architecture supports split execution of queries using a *filtering-refining* procedure where query contents that cannot be processed entirely on the server are processed at the proxy.

With respect to limitations related to fine grained access control, the row/cell level access control mechanism by CryptDB 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 bypasses and SQL injection attacks as mentioned above. In comparison, the access control mechanism in DBMask is cryptographically enforced. Additionally, the row/cell level specification of access control policies in CryptDB is difficult to manage. CryptDB uses 'speaks for' relationships that are specified at user level and each user is associated with many 'speaks for' relationships making the management of policies difficult. In DBMask, specification of access control policies is done using attribute based access control which is more expressive and easier to manage. Further, CryptDB fails to process queries over data items encrypted with different keys for different users based on an access control policy even though the users are authorized. DBMask on the other hand,

addresses this problem by separating access control from encrypted query processing. This is one of the novel contributions of our work. To this end, DBMask provides support for query processing while at the same time supporting expressive access control policies.

DBMask is a novel solution that addresses all the limitations of CryptDB. Our contributions include:

- An approach to support relational query operators by adding a comparison friendly encrypted column per column. Different algorithms such as order preserving encryption [7], symmetric key based searcheable encryption [24] 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 [23, 18, 17] to enforce access control policies with support for create, update, delete and cryptographically enforced read operations 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. Additionally, the enforcement of access control policies is supported both through an application and internally through a DBMS.

The paper is organized as follows. Section 2 discusses related work. Section 3 provides an overview of DBMask 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 server. Section 7 reports experimental results. Finally, Section 8 outlines the conclusions of the work.

2 Related Work

In theory, it is possible to utilize fully homomorphic encryption [14] to perform any arbitrary operation that a relational database requires. However, current implementations of fully homomorphic encryption are inefficient and are not suitable for practical applications [12]. With the increasing utilization of cloud computing services, recent research efforts [8] have developed privacy preserving access control systems by combining oblivious transfer and anonymous credentials. Such approaches have similarities to ours but also 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 limitation 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.

Attribute based encryption (ABE) approaches [6] are also related to our work in that they support expressive policies. However, 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. Techniques for efficient query processing over encrypted data have also been investigated. Hacigümüs et al. [15]

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. Wong et al. [27] propose a scheme supporting secure query processing by providing a set of elementary operators, like addition and multiplication, that work on encrypted data for relational tables. Asghar et al. [2] propose 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, uses specialized schemes to support relational operators, and enforces row/cell level access control by using an attribute based group key management (AB-GKM) scheme [18, 17], whereas the above approaches do not support fine-grained access control on relational data.

The idea of using specialized encryption techniques such as order preserving encryption [7] and additive homomorphic encryption [19] to perform different relational operations has been introduced in CryptDB [20]. The same idea has been extended to support complex analytical queries in MONOMI [25]. 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.

The solution discussed in this paper for supporting cryptographically enforced fine grained access control using AB-GKM scheme was first proposed in [21]. However, our previous solution has several limitations. It supports a simpler access control model with no support for enforcement of CRUD permissions. Moreover, it only supports simple queries. In the current paper, we extend DBMask to support CRUD permissions, complex queries and aggregates on the cloud server over encrypted data. We also discuss data upload and content management of access control data in detail. Additionally, the manual comparison between encrypted and trapdoor values for encrypted query processing is replaced with support for comparison using the internal equality mechanisms of a DBMS. Specifically, the access control enforcement mechanism is not only supported through an application, but also internally by a database. The performance and functionality of the extended approach are evaluated with additional experiments and a head-to-head comparison is constructed with state-of-the-art approaches in the related field of work.

3 Overview

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

3.1 System Architecture

Our system includes three entities: *data owner*, *data user*, and *cloud server*. Their interactions are shown in Figure 1. Users register their identity attributes with the data owner and the data owner generates secrets for the identity attributes and gives the encrypted secrets to users (step 1). The data owner then uses different secret keys to encrypt different portions of data, according to the access control policies. The encrypted data are uploaded to the cloud server (step 2). The data owner updates the user-secret pair database at the proxy and caches encrypted keys (step 3). A data user with authenticated identity attributes can verify itself to a proxy which is maintained by the

data owner. 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 (step 4). 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 cloud server (step 5). The encrypted query results are returned from the cloud server to the proxy (step 6), which decrypts the results using the secrets established at the time of verification and forwards them to the data user (step 7). If the query cannot be entirely executed on the cloud server, then the remaining portions of the query are executed at the proxy after decrypting the results. Notice that during the query processing stage, the cloud server learns neither the query being executed nor the result set of the query. The above brief mentions of each step are described in detail in Section 6.

The motivation and need behind a proxy, an intermediate entity between a user and the untrusted cloud server on the public cloud, is needed in order to abstract tasks such as query rewriting, splitting the query execution if the query cannot be completely executed on the untrusted cloud server, and decrypting query results. The proxy needs to be trusted since secret keys, the leakage of which would compromise the entire database, are needed in order to decrypt the results returned from the untrusted cloud server. The proxy is therefore maintained by the data owner which is fully trusted. The proxy being managed by the data owner still poses the problem of insider threats. The proxy is thus secured by insiders through the use of insider protection techniques [5].

The resources needed by the proxy are but not limited to (1) an in-memory database for processing the results of a query that cannot be entirely processed on the server; (2) storing metadata of encrypted tables; and (3) storing and updating the user-secret database to prevent unauthorized access to the proxy etc. In order for this model to be practical, a proxy must be small and require very little computational and storage resources. Then it makes it practical for a data owner to outsource data to the public cloud and maintain a small machine within the organization to implement the proxy. If by contrast, the proxy machine has to be very large, then the motivation behind outsourcing the data management to the public cloud is lost and a data owner may as well store data within the organization. The experiments in Section 7.1.1 show that the storage and computational resources needed for the proxy are small.

3.2 Adversary Model

We assume that the data owner is fully trusted. 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 attacker 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 the proxy is trusted. This is to avoid a "single point of failure". Otherwise, if the proxy were compromised at the preprocessing stage (i.e., the stage at which the keys to encrypt data are generated), the whole system would be compromised.

The cloud 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 cloud server is not 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). The cloud server itself may be compromised by external attackers. In such a case, data confidentiality is still preserved, since the attackers cannot decrypt the data. However, such a compromise does not protect against active attacks such as updating or inserting garbled data or deleting records. Suppose that an adversary launches

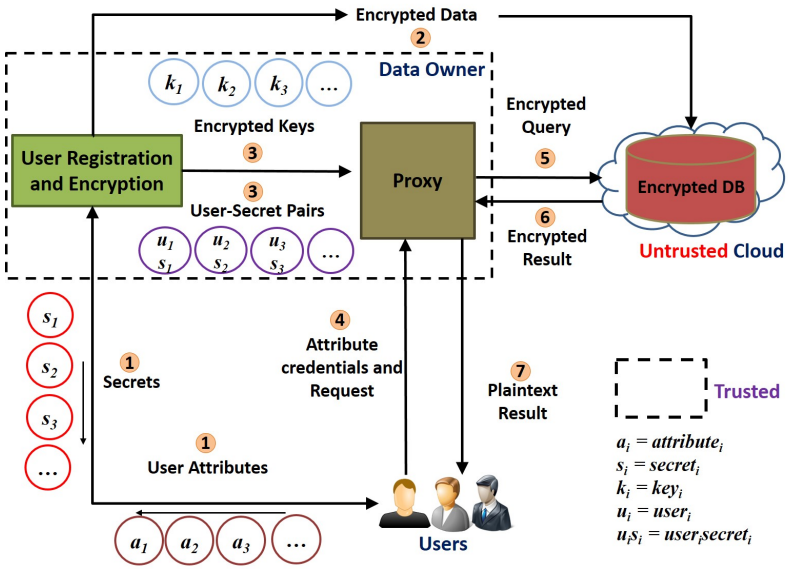


Figure 1: The system architecture

a bypass attack to gain direct access to the database. The adversary can access the data but cannot decrypt it; the adversary will not be able to do an update since the keys are not known; the adversary may insert records but cannot produce valid entries as the adversary does not know the keys; the adversary can however delete the records. The only way to prevent malicious updates, inserts or deletes would require modifications to the database engine. Our approach protects against malicious reads and invalid updates/inserts aiming at compromising confidentiality. An adversary can however delete records and we do not claim to protect against such attacks. In essence, encryption does not protect against active attacks such as updating or inserting garbled data or deleting records; a mechanism to protect against such active attacks will require modifications to the database engine.

Users are not trusted in our system and the proxy establishes trust via certified identity attributes issued by the data owner. DBMask 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 access control conditions in terms of identity attributes.
- A user whose identity attributes satisfy the ABAC policy associated with a data item is allowed to access the data item.

We propose our basic ABAC model and formally define the model as follows:

Definition 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. We refer to an expression over a set of attribute conditions as a Boolean Expression (BE).

Definition 2. Permission.
A permission is a cumulative permission represented by a bit mask composed of six bits where each bit represents a permission. The default permissions are `read` (bit 1), `write` (bit 2), `update` (bit 3), `delete` (bit 4), `create` (bit 5), `admin` (bit 6) corresponding to SELECT, INSERT, UPDATE, DELETE, CREATE and ALL SQL privileges respectively. A permission represented by 'D.WR' implies `read`, `write` and `delete` permissions are granted and `update` permission is denied.

Definition 3. ABAC Policy.
Let T be a table. An ABAC access control policy (ACP for short) defined over T is a tuple (s, o, p) where: o denotes a set of cells in T, s is a BE and p is a cumulative permission. Both s and p must be satisfied in order to access o.

Definition 4. 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 BE into disjunctive normal form (DNF). Note that this conversion can be done in polynomial time.
- For each distinct disjunctive clause, create a group.

Example 1: Consider the following two BEs defined over the attribute conditions C_1, C_2 and C_3: $\text{BE}_1 = C_1 \wedge (C_2 \vee C_3)$ and $\text{BE}_2 = C_2$. Then the corresponding policies in DNF are as follows: $\text{BE}_1 = (C_1 \wedge C_2) \vee (C_1 \wedge C_3)$ and $\text{BE}_2 = C_2$.

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 5. Group Poset.
A group poset is defined as the partially ordered set (poset) of groups where the binary relationship is $\subseteq$.

4.2 Policy Attachment and Application Enforced vs Database Enforced Access Control

Each column, row, or cell, depending on the desired level of access control, has an associated ACP. 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 6.5 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 ACP attached to T satisfied by the users attributes and restrict the query to only those columns, rows or cells depending on the granularity level.

DBMask supports two enforcement mechanisms namely Application-enforced Access Control (App-AC) and Database-enforced Access Control (DB-AC) for restricting the query to only those columns, rows and cells that are satisfied by an ACP for a particular user/group. In the case of App-AC, the restriction on a query is enforced by adding a predicate to the user query such that the predicate "encodes" the satisfied ACP as shown by example queries (Query 2, 3, 7 and 8) in Section 6. In the case of DB-AC, the restriction is enforced by using the native DBMS mechanism using `grant` and `revoke` for satisfied ACP on users/groups with database accounts as discussed and shown by example queries (Query 4, 5 and 9) in Section 6. The motivation behind providing support for both mechanisms is due to the fact that both are adopted by real world applications and both mechanisms have their advantages. For example, application level access control in comparison to database level access control does not require creating database accounts for users and implementing complicated access policy logic that can be difficult to express and update. On the other hand, application level access control in comparison to database level access control is prone to programming errors and malicious attacks that may violate principle of least privilege. The support of both mechanisms using one underlying access control model and key management scheme demonstrates the expressiveness and flexibility of DBMask.

4.3 Assigning Group Labels and Hierarchical Access Control

We label table cells by 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 determines the group(s) 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) [18, 17] 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 securely evaluating queries over encrypted data.

5.1 Key Management

Broadcast Group Key Management (BGKM) schemes [9, 4, 23] are a special type of GKM scheme whereby the rekey operation is performed with a single broadcast without requiring private communication channels. 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 [18, 17] provides all the benefits of BGKM schemes and also supports attribute based access control policies (ACPs).

The idea behind the AB-GKM scheme is as follows. A separate BGKM instance for each attribute condition is constructed. The BE 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 [22] 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 and refer the reader to [18] for details. 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 BE 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 BEs 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 server.

5.2 SQL-aware Comparison, Summation and Joins

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 [7] 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 comparison schemes and is constructed using 128-bit block 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. The blinding/unblinding mechanism is explained through example queries in Section 6.5. 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 [24, 10]. We refer to this approach as privacy preserving keyword comparison (PPKC). We adopt the keyword search technique proposed in [24] (PPKC-SEM) as it is better suited to relational data and its implementation is available. DBMask provides support for aggregates over encrypted data by implementing the Paillier cryptosystem [19]. We refer to the scheme for supporting summation over encrypted data as PPNC-HOM. PPNC-HOM can also be used for **UPDATE** increment operations and evaluating averages. In order to perform the **SUM** operation on the server, a user-defined function (UDF) is called that multiplies ciphertexts under PPNC-HOM encryption and returns the result which is decrypted at the proxy. PPNC-HOM has the strongest security guarantees among PPNC schemes. The ciphertext length for n-bit plaintext is n^2.

DBMask also provides support for 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 the PPNC or the 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 stores the blinded trapdoor values corresponding to the comparison friendly PPNC or PPKC

encrypted values. JOIN-DET stores the unblinded trapdoor values corresponding to the comparison friendly PPNC or PPKC encrypted values. JOIN-SEM provides better security guarantees than JOIN-DET.

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**, and **GenTrapdoor**, which we use for comparison in our cloud server 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 data owner produces an encrypted value e_x using **EncVal** algorithm that hides the actual value, but allows one to perform comparisons using the trapdoor value. Given an input (numerical or keyword) value t, the proxy produces an encrypted value e_t, called the trapdoor, using **GenTrapdoor** algorithm 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 values are compared to output 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.

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 server and users. Our system undergoes the following phases: system initialization, user registration, data encryption, data upload and content management 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 BEs 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 obtain their identity attributes certified by a trusted identity provider. These certified identity attributes are cryptographic commitments that hide the actual identity attribute value but

[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.

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.

6.3 Data Encryption

In our solution, each cell in an original table is expanded into multiple cells as shown in Figure 2. The first cell is encrypted for fine-grained access control, the second cell represents the assigned group labels, the third cell represents the permission mode rights on the cell (only in the case of App-AC mechanism), the fourth cell is encrypted for privacy-preserving matching, the fifth cell is encrypted using homomorphic encryption to support aggregates (this cell is optional), and the sixth cell stores blinded or unblinded trapdoor value for join operation. We denote the column resulting from the encryption for fine-grained access control as *data-col*, the one with associated group names as *label-col*, the one with associated permission rights as *mask-col*, the one resulting from the encryption for privacy-preserving matching as *match-col*, the one resulting from the homomorphic encryption for aggregates as *agg-col* and the one resulting from storing trapdoor values as *trap-col*.

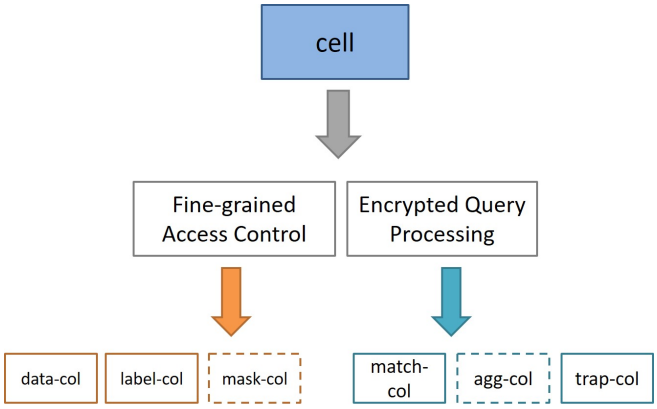


Figure 2: The expansion of each cell

Example 2: Consider Example 1 in Section 4 and suppose that C_1, C_2 and C_3 are conditions defined as follows. C_1 = "level $>$ 3", C_2 = "role = doctor" and C_3 = "role = nurse". Therefore, BE_1 is satisfied by all users whose level is greater than 3 and who are either doctors or nurses. BE_2 is satisfied by all doctors. Let ACP_1 = (G_1, G_3, 'SELECT * FROM patients WHERE age $<$ 40', 'DUWR') and ACP_2 = (G_2, 'SELECT age, diagnosis FROM patients WHERE age $<$ 40 AND diagnosis = 'Asthma', '.UWR'). Note that the table and column names in the Policy table (Table 1) stored on the untrusted server are anonymized and the values encrypted with data owner's group key.

We 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 the AB-GKM scheme [18, 17].

Table 1: Policy Table

Table	Columns	Condition	Groups	Mask
Patient	ID, Age, Diagnosis	Age < 40	G_1, G_3	'DUWR'
Patient	Age, Diagnosis	Age < 40 **AND** Diagnosis = 'Asthma'	G_2	'.UWR'

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 4 in Table 3 is encrypted with a master key k_{23} generated from the AB-GKM instance having k_2 and k_3 as input secrets with public information corresponding to this master key as PI_{23}. 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 2: PubInfo Table

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

Table 3: Encrypted Patient Table - Cell Level Access Control (App-AC)

ID-enc	ID-grp	ID-mask	ID-com	ID-trap	Age-enc	Age-grp	Age-mask	Age-com	...
$E_{k_3}(1)$	G_3	$DUWR$	$comp_n$ (1)	$comp'_n$ (1)	E_{k_3} (35)	G_3	$DUWR$	$comp_n$ (35)	...
$E_{k_3}(2)$	G_3	$DUWR$	$comp_n$ (2)	$comp'_n$ (2)	E_{k_3} (30)	G_3	$DUWR$	$comp_n$ (30)	...
$E_{k_0}(3)$	G_0	A.....	$comp_n$ (3)	$comp'_n$ (3)	E_{k_0} (40)	G_0	A.....	$comp_n$ (40)	...
$E_{k_3}(4)$	G_3	$DUWR$	$comp_n$ (4)	$comp'_n$ (4)	$E_{k_{23}}$ (38)	G_2,G_3	$DUWR$... UWR	$comp_n$ (38)	...

We now discuss the creation of **label-col**. Given a set of cells that satisfy ACPs, each cell is assigned one or more group labels. 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 1 in Table 3 is satisfied by both G_1 and G_3 but only assigned label of less privileged group, G_3. In terms of design for **label-col**, the groups are pushed down to row/cell in order to support cell level access controlas shown in Tables 3 and 4. While it is true that updating the group column(s) in the case of change to an ACPwould incur maintenance, it can be noted that the process of enforcing fine-grained access control becomes very simple where only a predicate needs to be added in the **WHERE** clause as explained by example queries in Section 6.5. Also note that multi-value group columns are normalized in the actual implementation. These details are omitted from Tables 3 and 4 for the sake of brevity and page limit.

Now, let us discuss the creation of **mask-col**. Given a set of cells that satisfy ACPs, each cell is assigned a bit mask that reflects the permissions granted to a group on that cell. If two or more groups

Table 4: Encrypted Patient Table - Row Level Access Control (DB-AC with PPNC-HOM)

ID-enc	ID-com	ID-agg	ID-trap	...	Diag-enc	Diag-com	Diag-trap	Groups
E_{k_3} (1)	$comp_n$ (1)	$comp''_n$ (1)	$comp'_n$ (1)	...	E_{k_3} (HIV)	$comp_k$ (HIV)	$comp'_k$ (HIV)	G_3
E_{k_3} (2)	$comp_n$ (2)	$comp''_n$ (2)	$comp'_n$ (2)	...	E_{k_3} (Cancer)	$comp_k$ (Cancer)	$comp'_k$ (Cancer)	G_3
E_{k_0} (3)	$comp_n$ (3)	$comp''_n$ (3)	$comp'_n$ (3)	...	E_{k_0} (Asthma)	$comp_k$ (Asthma)	$comp'_k$ (Asthma)	G_0
$E_{k_{23}}$ (4)	$comp_n$ (4)	$comp''_n$ (4)	$comp'_n$ (4)	...	$E_{k_{23}}$ (Asthma)	$comp_k$ (Asthma)	$comp'_k$ (Asthma)	G_2,G_3

have access to the same cell, then the permissions of the groups are concatenated and represented by a bit mask of the form $b_n...b_2b_1b_0$ where b_0 represents the permissions granted to G_0, b_1 represents the permissions granted to G_1 etc. For example, the Age value in row 4 in Table 3 is satisfied by both G_2 and G_3 and assigned the bit mask '..$DUWR$...UWR' where '...UWR' represents the permissions granted to G_2 and '..$DUWR$' represents the permissions granted to G_3. Note that the mask-col is only required in the case of App-AC mechanism and not in the case of DB-AC mechanism.

We now 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.

We now consider the creation of **agg-col**. Given a cell of numerical type, the PPNC-HOM.EncVal algorithm is used to encrypt the cell value. Note that the encryption of this column is optional.

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/PPNC.GenTrapDoor**blinding factor* or PPKC.GenTrapDoor/PPKC.GenTrapDoor**blinding factor* depending on the data type and the type of join scheme. The blinding factor is as explained in Section 5.2.

Tables 3 and 4 show the final tables using the App-AC and DB-AC mechanisms respectively with encrypted data-col's, match-col's and trap-col's. The $E_k(x)$ refers to the semantically secure encryption of the value x using the symmetric key k, $comp_n$ and $comp_k$ refer to PPNC.EncVal and PPKC.EncVal respectively, $comp'_n$ and $comp'_k$ refer to PPNC.GenTrapDoor and PPKC.GenTrapDoor respectively and $comp''_n$ refers to PPNC-HOM.EncVal.

6.4 Data Upload and Content Management

In order to allow access to authorized users based on ACPs, the data-col's must be encrypted using their respective group keys as explained above. In this section, we explain the step-wise mechanism for data upload and efficient content management of data resulting from access control. We will first explain content management of data resulting from access control. For each ACP defined by the data owner being stored or updated at the cloud server, the following steps are taken:

- A view reflecting the data-col's, label-col's and mask-col's of the set of cells satisfied by the ACP is created.
- If the DB-AC mechanism is used, the groups associated with the ACP are granted privileges (reflective of permission(s) in the ACP) on the view using `grant` SQL statement.

- A trigger is created and associated with the view that fires with INSERT and UPDATE events on the view. This trigger additionally calls a user-defined function (UDF) that takes as arguments the group(s) and the permission(s) associated with the ACP and updates the data-col value by encrypting it with a master key generated by the keys of group(s) taken as input argument, updates the label-col value to reflect the group(s) and updates the mask-col (if App-AC mechanism) with the permission(s).

Example 3: Consider the ACP_2 shown in Example 2. The mechanism explained above for content management for both App-AC and DB-AC mechanisms is described below.

App-AC:

CREATE VIEW $view\_patient\_G_2$ AS
SELECT Age-enc, Age-grp, Age-mask, Diag-enc, Diag-grp, Diag-mask
FROM Patient
WHERE Age-com $<$ 40 AND Diag-com = 'Asthma';

CREATE TRIGGER $trigger\_patient\_G_2$
AFTER INSERT, UPDATE ON $view\_patient\_G_2$
EXECUTE PROCEDURE UDF_UPDATE_DATA('G_2',
'UWR');

DB-AC:

CREATE VIEW $view\_patient\_G_2$ AS
SELECT Age-enc, Age-grp, Diag-enc, Diag-grp
FROM Patient
WHERE Age-com $<$ 40 AND Diag-com = 'Asthma';

GRANT UPDATE, INSERT, SELECT ON $view\_patient\_G_2$ TO G_2

CREATE TRIGGER $trigger\_patient\_G_2$
AFTER INSERT, UPDATE ON $view\_patient\_G_2$
EXECUTE PROCEDURE UDF_UPDATE_DATA('G_2');

We now explain the data upload. The data-col's corresponding to data being inserted are always initially encrypted with data owner's group key. The statement below describes a generalized SQL INSERT statement.

INSERT INTO *table* **VALUES** (E_{k_0}(data-col), G_0, 'A.....', $comp_{n|k}$(match-col), $comp'_{n|k}$(trap-col));

If the inserted data satisfies an ACP, then the trigger associated with the view reflecting the ACP is fired that updates the data-col, label-col and mask-col using the mechanism explained above. Similar is the case with SQL UPDATE where the data-col being updated is encrypted with data owner's group key. If the updated data satisfies an ACP, then the trigger associated with the view reflecting the ACP is fired and the columns in the view are updated.

6.5 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 10-16) by removing clauses, aggregate functions, and predicates with aggregate functions that cannot be computed on the server. The `PART` function (Lines 18-20) 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 server(Lines 22-24) 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 value in the `WHERE` clause is replaced with the trapdoor value computed by the proxy using PPNC.GenTrapdoor algorithm if the value is numeric or PPKC.GenTrapdoor algorithm if the value is a keyword. Additionally, the `REWRITE` function in the case of App-AC adds a predicate to the `WHERE` clause that determines the group(s) of the user requesting the query and wraps a UDF that evaluates the permission(s) using a bit mask offset for each group when the rewritten query is sent to the cloud server.

The cloud server 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 25). The proxy generates the necessary keys for decrypting the result set using the AB-GKM.KeyDer algorithm with the public information and the user secrets as well as the hierarchical key derivation (Line 26).

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 27-29). 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 using both the App-AC and DB-AC mechanisms. For each case, the queries reflect two scenarios. The first scenario which we refer to as DBMask-SEC provides maximum security and uses PPNC-SEM for numerical comparison, PPKC-SEM 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 for numerical comparison, PPKC-SEM for keyword search, JOIN-DET 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.

Query 1:

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

Query 2:

Algorithm 1 Pseudo-code for SECUREQUERYPLAN

Input: Q, abstract syntax tree (AST) for the query
M, metadata of the target table(s)
G, group(s) a user is member of
Output: P, a query plan for Q
for s in subqueries in Q **do**
  SECUREQUERYPLAN(Q,M,G)
end for
$FilterQ \leftarrow Q$
$ProxyFilters \leftarrow$ []
for f **in** $FilterQ$ **do**
  $f' \leftarrow$FILTER$(f, FilterQ)$ //query filtered at proxy
  for $f' \neq$ **Nil do**
    Remove f from $FilterQ$
    Add f' to $ProxyFilters$ //remaining portions of query to be evaluated at proxy
  end for
end for
for p in $ProxyFilters$ **do**
  if PART(p) in $FilterQ.relations$ **then**
    Add PART(p) to $FilterQ.projections$ //forming the query to be computed on the server
  end if
end for
for c **in** $s.where\_clause \parallel s.select\_clause$ **do**
  REWRITE(c, s, M, G)
end for
$P \leftarrow$ SERVERSQL$(FilterQ)$ // query computed on server
$P \leftarrow$ DECRYPT(P) // encrypted results decrypted at proxy
if ProxyFilters $\neq$ [] **then**
  $P \leftarrow$ PROXYSQL(P, ProxyFilters, Q) // remainder query executed at proxy
end if
return P

SELECT	*ID-enc*, *ID-grp*, *Age-enc*, *Age-grp*,
	Diag-enc, *Diag-grp*
FROM	*Patient*
WHERE	*Diag-com* **LIKE** PPKC.GenTrapdoor('Asthma'))
	AND *ID-grp* **LIKE** PPKC.GenTrapdoor('G_1')
	OR *ID-grp* **LIKE** PPKC.GenTrapdoor('G_3')
	AND *Age-grp* **LIKE** PPKC.GenTrapdoor('G_1')
	OR *Age-grp* **LIKE** PPKC.GenTrapdoor('G_3')
	AND *Diag-grp* **LIKE** PPKC.GenTrapdoor('G_1')
	OR *Diag-grp* **LIKE** PPKC.GenTrapdoor('G_3')
	AND UDF_Mask(*ID-mask*, [G_1, G_3], 'R')
	AND UDF_Mask(*Age-mask*, [G_1, G_3], 'R')
	AND UDF_Mask(*Diag-mask*, [G_1, G_3], 'R')

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 mechanism is App-AC and the underlying scenario is DBMask-SEC, as Query 3 if the mechanism is App-AC and underlying scenario is DBMask-PER, as Query 4 if the mechanism is DB-AC and the underlying scenario is DBMask-SEC and finally as Query 5 if the mechanism

is DB-AC and the underlying scenario is DBMask-PER. The rewritten queries are submitted to the cloud server. Notice that:

- the comparison on the *Age* column in the **WHERE** clause and the **ORDER BY** clause are removed from Query 2 and Query 4. The reason being that the cloud server does not have sufficient information to compare or order the query results. In the case of Query 3 and Query 5, since the Age-com column is encrypted using PPNC-OPE scheme, the comparison and the ordering can be done on the server.
- Query 4 and Query 5 are not encoded with group labels or permission(s) to restrict user access to column, rows and/or cells that are not satisfied by an ACP. The reason being that the enforcement is done internally by the DBMS as explained in Section 6.4.

Query 3:

SELECT *ID-enc*, *ID-grp*, *Age-enc*, *Age-grp*,
Diag-enc, *Diag-grp*
FROM *Patient*
WHERE *Age-com* $>$ PPNC.GenTrapdoor(35)
AND *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** '$\%G_3\%$'
AND UDF_Mask(*ID-mask*, [G_1, G_3], 'R')
AND UDF_Mask(*Age-mask*, [G_1, G_3], 'R')
AND UDF_Mask(*Diag-mask*, [G_1, G_3], 'R')
ORDER BY *Age-com* **ASC**

The cloud server returns the encrypted row 4 to the proxy. In order to decrypt the resultset (row 4), the proxy requires the keys k_3 to decrypt *ID-enc* column value, k_{23} to decrypt *Age-enc* column value and *Diag-enc* column value. The proxy derives the key k_3 using the AB-GKM scheme. To derive key k_{23}, the proxy uses k_3 and PI_{23} to derive k_{23}. The proxy then uses k_3 and k_{23} to decrypt the resultset and sends the resultset to the user. In the case of Query 2 and Query 4, the proxy orders the plaintext resultset using its in-memory database before sending the final resultset to the user.

Query 4:

SELECT *ID-enc*, *ID-grp*, *Age-enc*, *Age-grp*,
Diag-enc, *Diag-grp*
FROM *Patient*
WHERE *Diag-com* **LIKE** PPKC.GenTrapdoor('Asthma')

Query 5:

SELECT *ID-enc*, *ID-grp*, *Age-enc*, *Age-grp*,
Diag-enc, *Diag-grp*
FROM *Patient*
WHERE *Age-com* $>$ PPNC.GenTrapdoor(35) **AND**
Diag-com **LIKE** PPKC.GenTrapdoor('Asthma')
ORDER BY *Age-com* **ASC**

Query 6:
SELECT *p.Age*, *d.Description*
FROM *Patient p*, *Diagnosis d*
WHERE *p.ID* = *d.PatientID*

A user having the attribute "role = doctor" executes Query 6 through the proxy server. The proxy determines that the user is a member of the group G_2, re-writes the query and submits it to the cloud server. The rewritten query with the App-AC mechanism is Query 7 if the underlying scenario is DBMask-SEC and Query 8 if the underlying scenario is DBMask-PER. Note that Query 7 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 8, the joining columns are encrypted using a deterministic encryption scheme, namely JOIN-DET and hence the join is done using the equality mechanism of the DBMS. Note that the join columns do not need to be unblinded on the fly. The proxy receives and decrypts the encrypted result set using the mechanism explained above and sends the plaintext result back to the user.

Query 7:
SELECT *p.Age-enc*, *p.Age-grp*, *d.Description-enc*,
d.Description-grp
FROM *Patient p*, *Diagnosis d*
WHERE $UDF\_Compare$(*p.ID-trap* * *unblinding factor*, =,
unblinding factor * d.PatientID-trap) **AND**
p.Age-grp **LIKE** PPKC.GenTrapdoor('G_2') **AND**
d.Description-grp **LIKE** PPKC.GenTrapdoor('G_2')
AND UDF_Mask(*p.Age-mask*, [G_2], 'R')
AND UDF_Mask(*d.Description-mask*, [G_2], 'R')

Query 8:
SELECT *p.Age-enc*, *p.Age-grp*, *d.Description-enc*,
d.Description-grp
FROM *Patient p*, *Diagnosis d*
WHERE *p.ID-trap* = *d.PatientID-trap*
AND *p.Age-grp* **LIKE** '$\%G_2\%$'
AND *d.Description-grp* **LIKE** '$\%G_2\%$'
AND UDF_Mask(*p.Age-mask*, [G_2], 'R')
AND UDF_Mask(*d.Description-mask*, [G_2], 'R')

Query 9:
SELECT *ID-enc*, *Age-enc*, *Diag-enc*, *Groups*
FROM *Patient*
WHERE *Age-com* > PPNC.GenTrapdoor(35) **AND**
Diag-com = PPKC.GenTrapdoor('Asthma')
AND (*Groups* **LIKE** '$\%G_1\%$' **OR**
Groups **LIKE** '$\%G_3\%$')
ORDER BY *Age-com* **ASC**

Tables 4 shows the *Patient* table enforcing row level access control using the App-AC mechanism. Row level access control is a special case in our scheme where there is only a single additional *label-col* and *mask-col* (in the case of App-AC) in the table. Query 9 shows the transformation of Query 1 issued by the same user under DBMask-PER scenario using DB-AC mechanism. The cloud

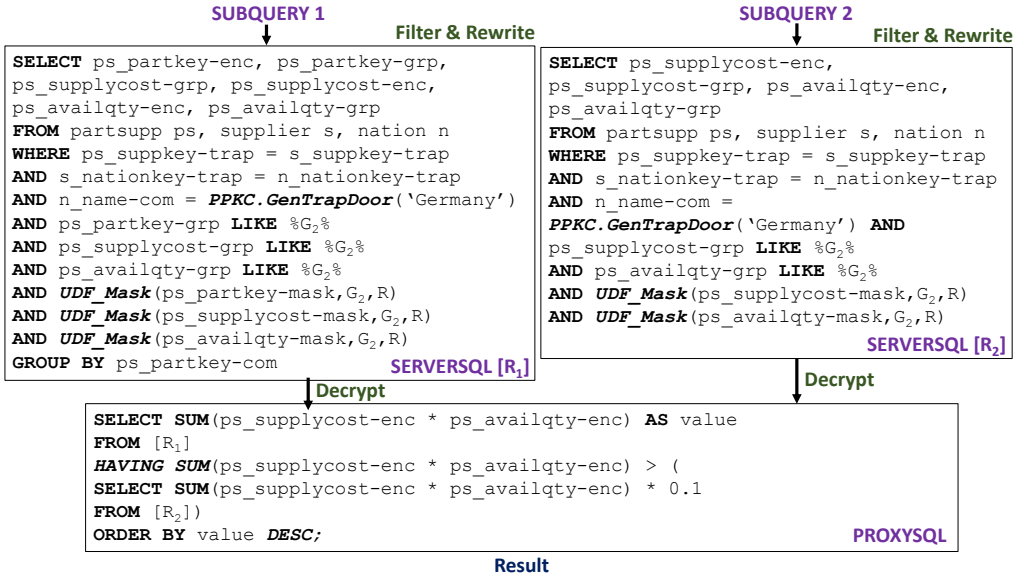


Figure 3: Query 10 execution plan (Query 11 from TPC-H)

server returns encrypted row 4 to the proxy. The decryption mechanism of the returned results is as explained above.

We now illustrate the processing of complex queries through the *filtering-refining* approach by calling the SECUREQUERYPLAN algorithm on subqueries of an original query recursively to output the final result. A user having the attributes "role = doctor" executes Query 10 with App-AC mechanism under DBMask-PER scenario (Query 11 from TPC-H benchmark) through the proxy server. The proxy determines that the user belongs to G_2. Figure 3 shows the execution plan of Query 10. Each subquery is filtered (e.g. SUM), rewritten (e.g. the joining columns are replaced with trap-col's), columns included in expressions that cannot be computed on the server are projected (e.g. ps_supplycost * ps_availqty) and then the expressions are computed on the server. The intermediate results are then decrypted and populated at the proxy where the original query is executed and the results are returned.

Query 10:

SELECT	*ps_partkey*, **SUM**(*ps_supplycost * ps_availqty)* AS value
FROM	*partsupp ps*, *supplier s*, *nation n*
WHERE	*ps_suppkey* = *s_suppkey* **AND** *s_nationkey* = *n_nationkey* **AND** *n_name* = 'Germany'
GROUP BY	*ps_partkey*
HAVING	**SUM**(*ps_supplycost * ps_availqty)* > (
SELECT	**SUM**(*ps_supplycost * ps_availqty)* * 0.1
FROM	*partsupp*, *supplier*, *nation*
WHERE	*ps_suppkey* = *s_suppkey* **AND** *s_nationkey* = *n_nationkey* **AND** *n_name* = 'Germany')
ORDER BY	*value* **ASC**

6.6 Handling Change in Dynamics

6.6.1 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 scheme, 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 databases even when the user dynamics change.

Example 4: 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 the 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 of which the new user is not a member.

6.6.2 ACP Dynamics

When an ACP is being updated or deleted, the data resulting from access control (data-col's, label-col's and mask-col's) need to be updated in order to prevent unauthorized access. The following steps are taken when updating or deleting an ACP:

- the columns in the view associated with the ACP are updated in the following manner:

  UPDATE `view` **SET** data-col = E_{k_0}(data-col), label-col = G_0, mask-col = 'A.....';

- if the DB-AC mechanism is used, the groups associated with the ACP are revoked privileges (reflective of permission(s) in the ACP) on the view using the `revoke` SQL statement.
- the view associated with the ACP and the trigger associated with the view are deleted.

In the case where an ACP is being updated, a new view reflecting the updated ACP is created and a trigger is associated with the view as explained in Section 6.4.

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 cloud 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 cloud server. The

cryptographic operations are supported by using the NTL library[3] while the access control policies expressed as boolean expressions are converted into DNF using the boolstuff library[4]. The 'data-col' in each table is constructed with a 128-bit block size AES in CBC mode together with a random initialization vector while the 'match-col' is encrypted with a 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.

7.1 Performance Evaluation

We compare the performance of our prototype by running a TPC-H query workload utilizing only a single group under row-level control with App-AC to evaluate only the encryption/decryption and comparison schemes. The TPC-H query workload is composed of complex analytical queries with support for sub-selects, aggregates over expressions, complex expressions in the `FROM`, `WHERE`, `GROUP BY`, and `HAVING` clauses. There are 22 queries in total in the workload. The performance of each query in the TPC-H workload is compared by running the queries on plaintext Postgres server against running the queries through MONOMI and the proxy of our prototype using the App-AC mechanism under the DBMask-PER scenario. We refer to the results from the case without the support of **SUM** operation on the cloud server as DBMask and for the case with support of **SUM** operation on the cloud server as DBMask-HOM. Additionally, the performance of select queries from TPC-H workload is evaluated when equality comparisons are performed using manually implemented UDFs as is the case in the earlier work of DBMask [21] in contrast to performing comparisons using the internal equality mechanism of a database, as is the case in the current work. We refer to the results from earlier work as DBMask-UDF and from current work as DBMask.

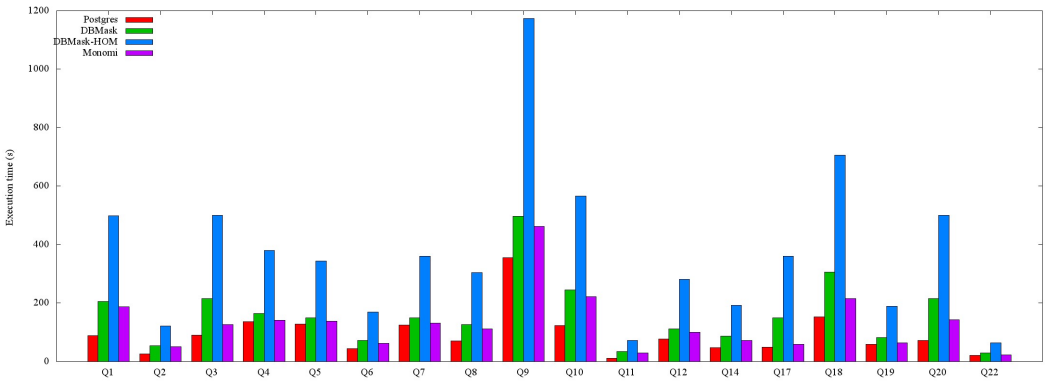


Figure 4: Execution time of TPC-H workload running under Postgres, DBMask, DBMask-HOM and MONOMI with scale factor 10

7.1.1 TPC-H

Figure 4 shows the execution time of all queries in TPC-H supported by both DBMask and MONOMI. Note that *Q13* and *Q16* are neither supported by DBMask or MONOMI as the underlying PPKC scheme does not provide support for keyword matching involving two or more patterns. MONOMI does not support views and hence cannot run *Q15*. Additionally, *Q21* due to correlating subqueries times out with both DBMask and MONOMI. For the remaining queries, the results show that in comparison to running an unencrypted trace of TPC-H workload on Postgres, there is an overall increase

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

in execution time by: 1.73x for DBMask, 2.34x for DBMask-HOM and 1.39x for MONOMI. The most significant contributing factor to lower performance of DBMask in comparison to MONOMI is due to larger table scans as DBMask stores multiple encryptions of each column in comparison to MONOMI which stores only a single encryption of each column with few additional materialized columns. The tables are larger for DBMask-HOM owing to large ciphertexts under homomorphic encryption. However, in order to have an objective comparison, one must consider the hidden costs and limitations associated with various optimization's by MONOMI. First, MONOMI during the setup phase takes as input a query workload to determine the encryption schemes of columns and to materialize any additional columns (e.g. storing encryption of ps_supplycost*ps_availqty *Q11*) related to query operators that cannot be computed on the cloud server. Aside from the cost associated with the training mechanism in the setup phase, any change in the query workload would require that the table to be changed is downloaded, decrypted and then re-inserted. In comparison, DBMask only requires the data owner to select the encryption schemes during the setup phase and a change in query workload would have no effect on the database design. Second, MONOMI uses a packing scheme that makes it possible to pack multiple values from a single row into one ciphertext for that row in order to reduce ciphertext expansion and space overhead. However, the packing mechanism significantly complicates the **SQL UPDATE** operation where only a subset of the values packed into one ciphertext need to be updated. DBMask would not significantly benefit from the packing scheme as it provides support for **SQL UPDATE** operations as shown with TPC-C workload in [21]. We consider an increase in the execution time by 1.73x in comparison to plaintext Postgres for complex queries by DBMask to be modest considering the gains in confidentiality. In terms of space utilization, MONOMI incurs a smaller overhead (1.7x) in comparison to DBMask (3.9x) and DBMask-HOM (5.7x). The space over head imposed by DBMask is primarily due to the storage of multiple encryptions of the same column so as to support a variety of SQL operations over encrypted data. In addition, metadata e.g. storing Public Information (PI) to generate keys, and access control policies and triggers for content management also contribute towards the additional space overhead imposed by DBMask. To assess the computational resources utilized by the proxy, we assess the processing time intervals for the queries in TPC-H workload with DBMask at the cloud server and at the proxy. The average processing time for a query on the cloud server is 107s in comparison to 51s on the proxy. It can be noted that the average time added by the proxy is less than one third of the total processing time. The breakdown of the time added by the proxy is as follows: 16% for encryption/decryption, 26% for query rewriting and 58% for in-memory processing. The proxy utilizes storage space of 318MB, contributed mainly by the in-memory database. It can be noted that the computational and storage requirements for the proxy are small.

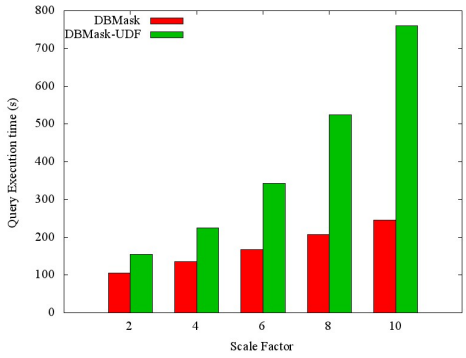


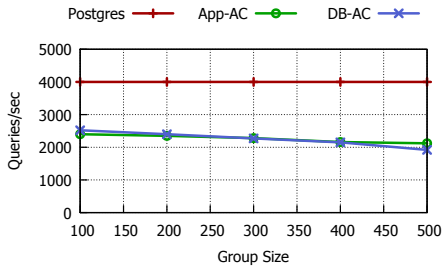
Figure 5: Execution time of *Q10* with varying scale factors

Since the performance of queries heavily depends on the number of rows and their content, it is interesting to see how the performance evolves given different scale factors. Figure 5 shows the execution time of *Q10* with varying scale factors. For DBMask as shown in Figure 5, execution time increases linearly with increasing table size. However, for DBMask-UDF, query execution time increases exponentially with increasing table size. The reason being that calling manually implemented UDFs for equality comparisons is much slower than the internal equality mechanisms for which a database is optimized and DBMask makes use of.

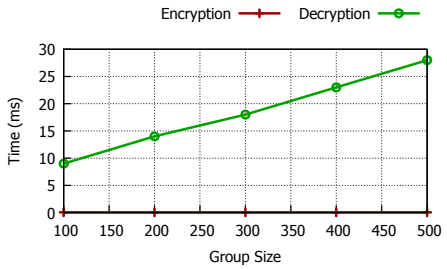
7.2 Functional Evaluation

We have also carried out experiments using a Policy Simulator (PS) on TPC-C query workload and a web based scientific application called Computational Research Infrastructure for Science (CRIS) in order to analyze the access control functionality of our prototype. The PS generates random ACP's using one attribute condition ($a_1 = x$ where a_1 is the attribute condition and x is a set of random values) and attaches the ACP's to the tables in TPC-C using row level access control under DBMask-PER scenario with both App-AC and DB-AC mechanism's. Additionally, for the DB-AC mechanism, the PS generates database accounts for all groups generated by random ACP's. We study the performance of the TPC-C workload by encoding the queries and evaluating the throughput and the average encryption/decryption time with varying group sizes. We also evaluate the overhead imposed when user dynamics and ACP dynamics change for the DB-AC mechanism with varying group sizes and table sizes.

CRIS [13] is a web based application providing 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 ACP based on six attribute conditions over user identity attributes that capture the access control requirements of this particular workspace in CRIS. The users 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 using App-AC mechanism, we run plaintext queries on Postgres and run the same queries on DBMask at different granularities of access control, namely column level, row level and cell level. The CRIS database has 87 tables with 298 columns in total.



(a) Throughput of TPC-C workload with varying number of groups under App-AC and DB-AC mechanism's

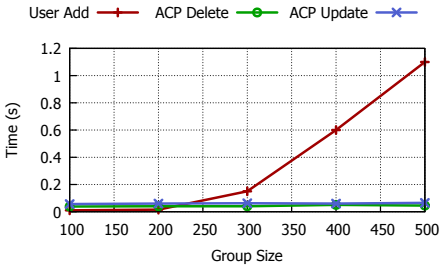


(b) Average encryption/decryption time with varying number of groups

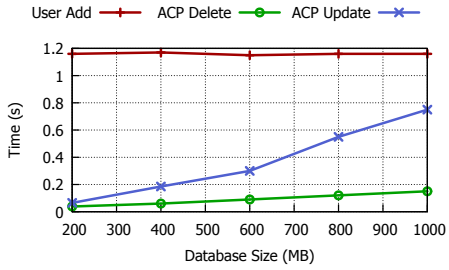
Figure 6: Effect on throughput and encryption/decryption with varying number of groups.

7.2.1 Policy Simulator

Figure 6a shows the effect on throughput under both the App-AC and DB-AC mechanism's. With increasing number of groups, the overall throughput is lower with App-AC by 40%-49% and by 37%-52% with DB-AC. The DB-AC mechanism performs better for smaller number of groups in comparison to App-AC as the DB-AC mechanism requires less disk space, performs fewer comparisons and uses the native mechanism of the DBMS for enforcement. However, App-AC scales better as the performance of DB-AC deteriorates with increasing number of groups. Fig 6b shows the average time to encrypt a query for an **INSERT** query operation and the average time to decrypt a row returned from a **SELECT** query operation with varying group sizes. The encryption mechanism is efficient and remains constant with increasing number of groups as the data is always initially inserted using the data owner's group key. In the case of decryption, the time to decrypt increases linearly with increasing number of groups as the key derivation process using AB-GKM scheme becomes more time consuming.



(a) Overhead imposed by change in user and ACP dynamics with varying number of groups



(b) Overhead imposed by change in user and ACP dynamics with varying database size under DB-AC mechanism

Figure 7: Overhead imposed by change in user and ACP dynamics

Figure 7a shows the overhead for change in user and ACP dynamics with increasing group size. For the case of adding/revoking users, the overhead is an aggregate of the time required to run the *ReKey* algorithm of AB-GKM and updating the rows of *Public Info* table on the cloud server. The overhead increases with increasing group size due increase in computational complexity of *ReKey* algorithm of AB-GKM. For the case of ACP dynamics, deleting a ACP incurs a lower overhead than an update since an update to a ACP requires deleting a ACP and then creating a new ACP. The overhead for change in ACP dynamics is an aggregate of the time required to update the data affected by the change in ACP revoking access to the affected group(s), deleting the view corresponding to the ACP and removing the trigger attached to this view. There is additional overhead for an update to ACP where a new view corresponding to the updated policy is created and a trigger is attached to this view. Increase in group size does not have an impact on the overhead imposed by change in ACP dynamics. Figure 7b shows the overhead for change in user and ACP dynamics with increasing database size with fixed group size. It can be noted that there is no impact on change in user dynamics with increase in table size. However, the overhead imposed by updating/deleting an ACP increases linearly with increase in table size. The reason being that the data affected by change to an ACP requires updating of large number of rows (for delete and update to ACP) and table scans for creating a new view (only for an update to ACP).

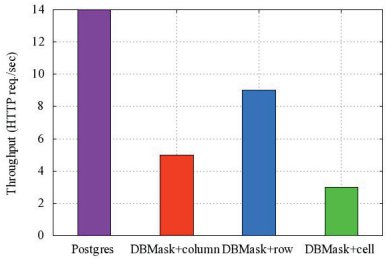


Figure 8: Throughput comparison of Sylvie's Workspace at row level access control

7.2.2 CRIS

Figure 8 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, with support for enforcing CRUD operations when evaluating SQL queries on encrypted relational data. Additionally, DBMask provides support for enforcement of access control both through an application and internally through a database using a single access control model and attribute based group key management scheme demonstrating the expressiveness and the flexibility of DBMask. DBMask introduces the idea of splitting fine-grained access control and predicate matching per each cell. The choice of predicate matching technique used is configurable so that different techniques can be plugged in depending on the security requirements. Our experimental results show that DBMask is efficient and overhead due to encryption and access control is low.

As future work, we plan to extend DBMask to study query workload patterns and relational operations to further optimize the support for encrypted query processing. We also plan to investigate the feasibility of implementing the current architecture into the hybrid cloud model where certain resources are managed in a private cloud and others externally on public cloud.

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.

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.

[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] E. Bertino. Data protection from insider threats. *Synthesis Lectures on Data Management*, 4(4):1–91, 2012.

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

[7] 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.

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

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

[10] 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.

[11] 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.

[12] 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.

[13] E. Dragut, P. Baker, J. Xu, M. Sarfraz, E. Bertino, A. Madhkour, R. Agarwal, A. Mahmood, and S. Han. Cris: Computational research infrastructure for science. In *IRI 2013*, pages 301–308.

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

[15] 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.

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

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

[18] M. Nabeel and E. Bertino. Attribute based group key management. *Transactions on Data Privacy*, 7(3):309–336, 2014.

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

[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] M. I. Sarfraz, M. Nabeel, J. Cao, and E. Bertino. Dbmask: Fine-grained access control on encrypted relational databases. In *CODASPY 2015*, pages 1–11.

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

[23] N. Shang, M. Nabeel, F. Paci, and E. Bertino. A privacy-preserving approach to policy-based content

dissemination. In *ICDE 2010*, pages 944–955.

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

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

[26] 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.

[27] W. K. Wong, B. Kao, D. W. L. Cheung, R. Li, and S. M. Yiu. Secure query processing with data interoperability in a cloud database environment. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1395–1406. ACM, 2014.

[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.