Metadata Management in Outsourced Encrypted Databases

E. Damiani[1], S. De Capitani di Vimercati[1], S. Foresti[1],
S. Jajodia[2], S. Paraboschi[3], P. Samarati[1]

[1] University of Milan - 26013 Crema, Italy
{damiani,decapita,foresti,samarati}@dti.unimi.it
[2] George Mason University - Fairfax VA 22030-4444, USA
jajodia@gmu.edu
[3] University of Bergamo - 24044 Dalmine, Italy
parabosc@unibg.it

Abstract. Database outsourcing is becoming increasingly popular introducing a new paradigm, called *database-as-a-service*, where a client's database is stored at an external service provider. Outsourcing databases to external providers promises higher availability and more effective disaster protection than in-house operations. This scenario presents new research challenges on which the usability of the system is based. In particular, one important aspect is the *metadata* that must be provided to support the proper working of the system.
In this paper, we illustrate the metadata that are needed, at the client and server, to store and retrieve mapping information for processing a query issued by a client application to the server storing the outsourced database. We also present an approach to develop an efficient access control technique and the corresponding metadata needed for its enforcement.

1 Introduction

Nowadays databases hold a critical concentration of sensitive information and the volume of this information is increasing very quickly. Therefore, many organizations are adding data storage at a high rate. This data explosion is due in part to powerful database applications, deployed by organizations to capture and manage information. In such a scenario, *database outsourcing* is becoming increasingly popular. A client's database is stored at an external service provider that should provide mechanisms for clients to access the outsourced databases. The main advantage of outsourcing is related to the costs of in-house versus outsourced hosting: outsourcing provides *i)* significant cost savings and service benefits and *ii)* promises higher availability and more effective disaster protection than in-house operations. As a consequence of this trend toward outsourcing, highly sensitive data are now stored on systems run in locations that are not under the data owner's control. Therefore, data confidentiality and even integrity can be put at risk. These problems are traditionally addressed by means of encryption [10]. By encrypting the information, the client is guaranteed that it

alone can observe the data. The problem is then to perform a *selective retrieval* on encrypted information. Since confidentiality demands that data decryption must be possible only at the client side, techniques are needed enabling external servers to execute queries on encrypted data, otherwise the whole relations involved in the query would be sent to the client for query execution. A first proposal toward the solution of this problem was presented in [8, 9, 12, 13, 15] where the authors proposed storing, together with the encrypted database, additional indexing information. Such indexes can be used by the DBMS to select the data to be returned in response to a query. In [8, 9] the authors propose a hash-based method for database encryption suitable for selection queries. To execute interval-based queries, the B+-tree structures typically used inside DBMSs are adopted. Privacy homomorphism has been also proposed for allowing the execution of aggregation queries over encrypted data [14, 16]. In this case the server stores an encrypted table with an index for each aggregation attribute (i.e., an attribute on which the aggregate operator can be applied) obtained from the original attribute with privacy homomorphism. An operation on an aggregation attribute can then be evaluated by computing the aggregation at the server site and by decrypting the result at the client side. Other work on privacy homomorphism illustrates techniques for performing arithmetic operations (+, -, ×, /) on encrypted data and does not consider comparison operations [3, 11]. In [1] an order preserving encryption schema (OPES) is presented to support equality and range queries over encrypted data. This approach operates only on integer values and the results of a query posed on an attribute encrypted with OPES is complete and does not contain spurious tuples.

However, this scenario, called *database-as-a-service* (DAS), presents new additional research challenges on which the usability of the system is based. One challenge is to develop efficient access control techniques. As a matter of fact, all the existing proposals for designing and querying encrypted/indexing outsourced databases assume the client has complete access to the query result. However, this assumption does not fit real world applications, where different users may have different access privileges. As an example, consider a medical database that includes information about doctors and patients. Each user (doctor/patient) or group thereof should be granted selective access to only a specific subset of data. Enforcing selective access with the explicit definition of authorizations requires the data owner to intercept and process each query request (from the user to the server) and each reply (from the server to the user) to filter out data the requestor is not authorized to access. Such an approach may however cause bottleneck because it increases the processing and communication load at the data owner site. A promising direction to avoid such a bottleneck is represented by selectively encrypting data so that users (or groups thereof) can decrypt only the data they are authorized to access [7]. This solution requires defining and maintaining, both at the client and server, additional information at the level of *metadata* needed to enforce selective access.

In this paper, after a brief summary of our proposal for enforcing access control in the DAS scenario, we focus on the metadata that are needed to access the

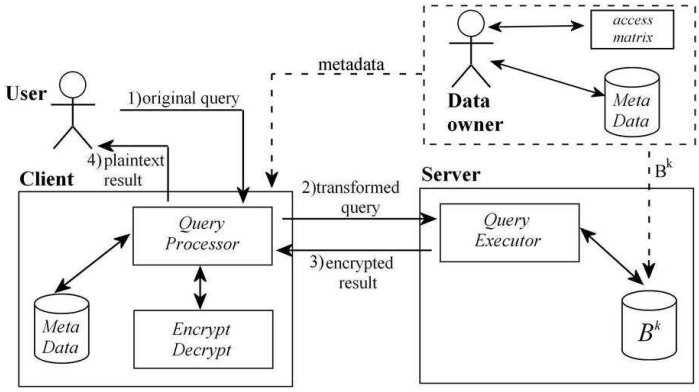

Fig. 1. DAS Scenario

outsourced database according to the policies defined by the data owner. In particular, we describe the metadata and compare different storage strategies each of which is characterized by a different usage in storage and bandwidth capacity. The remainder of this paper is organized as follows. Section 2 describes the DAS scenario and briefly illustrates a solution for enforcing access control through cryptography. Section 3 presents different strategies for storing and managing the metadata necessary to properly use the outsourced database. Section 4 illustrates how a query on a plaintext database is transformed into a query on the corresponding encrypted database. Finally, Section 5 concludes the paper.

2 DAS Scenario

We briefly introduce the considered DAS scenario, the encrypted database structure, and a solution for enforcing an access control policy on which the following metadata analysis is based.

2.1 Data Organization

The DAS scenario involves four entities (see Figure 1):

- *Data owner*: an organization that produces data to be made available for controlled external release;
- *User*: human entity that presents requests (queries) to the system;
- *Client*: front-end that transforms the user queries into queries on the encrypted data stored on the server;
- *Server*: an organization that receives the encrypted data from a data owner and makes them available for distribution to clients.

Clients and data owners are assumed to trust the server to faithfully maintain outsourced data. Specifically, the server is relied upon for the availability of

Patients

PatientId	Surname	Name	Disease	Doctor
125YP894	Carter	Andrew	Tonsillitis	Wayne
5896GT26	Rogers	Mark	Gastritis	Becker
654ED231	Wise	Paul	Hypertension	Wayne
442HN718	Brown	Luke	Hypertension	Lean
942MD745	Fisher	Robert	Tonsillitis	Becker
627IF416	Rogers	Alice	Arthritis	Wayne
058PI175	Brown	Mark	Hypertension	Lean
305EJ186	Rogers	Paul	Gastritis	Morris
276DL557	Fisher	Luke	Hypertension	Lean
364UK784	Rogers	Laura	Tonsillitis	Wayne

(a)

Patients^k

Counter	Etuple	IdKey	I_1	I_2	I_3	I_4	I_5
1	r*tso/yui+	AC	ω	γ	δ	π	η
2	hai4de-0q1	AB	ϑ	α	λ	π	μ
3	nag+q8*L	C	ω	γ	ϵ	ρ	η
4	K/ehim*13-	BCD	τ	β	δ	ρ	μ
5	3gia*ni+aL	BD	ω	β	λ	π	μ
6	F0/rab1DW*	BCD	ϑ	α	ϵ	ρ	η
7	Bid2*k1-l0	AB	ϑ	β	λ	ρ	μ
8	/bur2I/*-D	BC	τ	α	ϵ	π	η
9	O/c*yd-q2+	C	ω	β	δ	ρ	μ
10	bew0"!DE1a	ACD	ϑ	α	λ	π	η

(b)

Fig. 2. An example of plaintext relation(a) and encrypted relation (b)

outsourced databases. However, the server is assumed not to be trusted with the confidentiality of the actual database content. That is, we want to preserve the server from making unauthorized access to the data stored in the database. To this purpose, the data owner encrypts her data and gives the encrypted database to the sever.

Note that database encryption may be performed at different levels of granularity: relation level, attribute level, tuple level, and element level. Both relation level and attribute level imply the communication to the user of the whole relation involved in a query. On the other hand encrypting at element level would require an excessive workload for data owner and clients in encrypting/decrypting data. For balancing the client workload and query execution efficiency, we assume that the database is encrypted at tuple level.
The main effort of current research in this scenario is the design of a mechanism that makes it possible to directly query an encrypted database [12]. The existing proposals are based on the use of indexing information associated with each relation in the encrypted database [9, 15]. Such indexes can be used by the server to select the data to be returned in response to a query. More precisely, the server stores an encrypted table with an index for each attribute on which a query can include a condition. Different types of indexes can be defined, depending on the supported queries. For instance, hash-based methods are suitable for equality queries [15, 18] and B+-tree based methods support range queries [9]. For simplicity, in the remainder of this paper we assume that indexes have been created through a hash-based method and that there is an index for each attribute in each relation. Formally, each relation $\mathtt{r_i}$ over schema $\mathtt{R_i}(\mathtt{A_{i1}}, \mathtt{A_{i2}}, \ldots, \mathtt{A_{in}})$ in a plaintext database $\mathtt{B}$ is mapped onto a relation $\mathtt{r_i^k}$ over schema $\mathtt{R_i^k}(\underline{\mathtt{Counter}}, \mathtt{Etuple}, \mathtt{I_1}, \mathtt{I_2}, \ldots, \mathtt{I_n})$ in the encrypted database $\mathtt{B^k}$ where, `Counter` is the primary key; `Etuple` is an attribute for the encrypted tuple whose value is obtained using an encryption function E_k (k is the key); $\mathtt{I_i}$ is the index associated with the `i`-th attribute. For instance, given relation `Patients` in Figure 2(a), the cor-

responding encrypted relation is represented in Figure 2(b).[4] As it is visible from this table, the encrypted table has the same number of rows as the original one. The query processing is then performed as follows (see Figure 1): each query (1) is mapped onto a query on encrypted data (2) and is sent to the server that is in charge for executing it. The result of this query is a set of encrypted tuples (3), that are then processed by the client front-end to decrypt data and discard spurious tuples that may be part of the result. The final result (4) is then presented to the user. We will discuss the query processing in more details in Section 4.

2.2 Selective Access on Encrypted Databases

All existing proposals for designing and querying encrypted/indexing outsourced databases focus on the challenges posed by protecting data at the server side, and assume the client has complete access to the query result (e.g., [4, 6, 15, 21]). In other words, tuples are assumed to be encrypted using a single key; knowledge of the key grants complete access to the whole database. Clearly, such an assumption does not fit real world applications, which demand for selective access by different users, sets of users, or applications. Our solution exploits data encryption by including authorizations in the encrypted data themselves. While in principle it is advisable to leave authorization-based access control and cryptographic protection separate, in the DAS scenario such a combination can prove successful. The idea is then to use different encryption keys for different data. To access such encrypted data, users have to decrypt them, which could only be done by knowing the encryption algorithm and the specific decryption key being used. If the access to the decryption keys is limited to certain users of the system, different users are given different access rights. In classical terms, the access rights defined by the data owner can be represented by using an *access matrix* $\mathcal{A}$, where rows correspond to subjects, columns correspond to objects, and entry $\mathcal{A}[s, o]$ is set to 1 if s can read o; 0 otherwise. Given an access matrix $\mathcal{A}$, ACL_i denotes the vector corresponding to the i-th column (i.e., the access control list indicating the subjects that can read tuple t_i), and CAP_j denotes the vector corresponding to the j-th row (i.e., the capability list indicating the objects that user u_j can read). With a slight abuse of notation, in the following we will use ACL_i (CAP_j, resp.) to denote either the bit vector corresponding to a column (a row, resp.) or the set of users (tuples, resp.) whose entry in the access matrix is 1. Let us consider a situation with four users, namely `Alice`, `Bob`, `Carol`, and `David`, who need to read the tuples of relation `Patients`. Figure 3 illustrates an example of access matrix.

Different approaches can be taken to enforce the access rights reported in the access matrix. A trivial solution consists in encrypting each tuple with a different key and give users the keys for the tuples they can access. For instance, with respect to the access matrix in Figure 3, user `Carol` should receive the

[4] Here, the result of the hash function is represented as a Greek letter. Also, note that the meaning of attribute `IdKey` will be discussed in Section 3.

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
Alice	1	1	0	0	0	0	1	0	0	1
Bob	0	1	0	1	1	1	1	1	0	0
Carol	1	0	1	1	0	1	0	1	1	1
David	0	0	0	1	1	1	0	0	0	1

Fig. 3. An example of access matrix

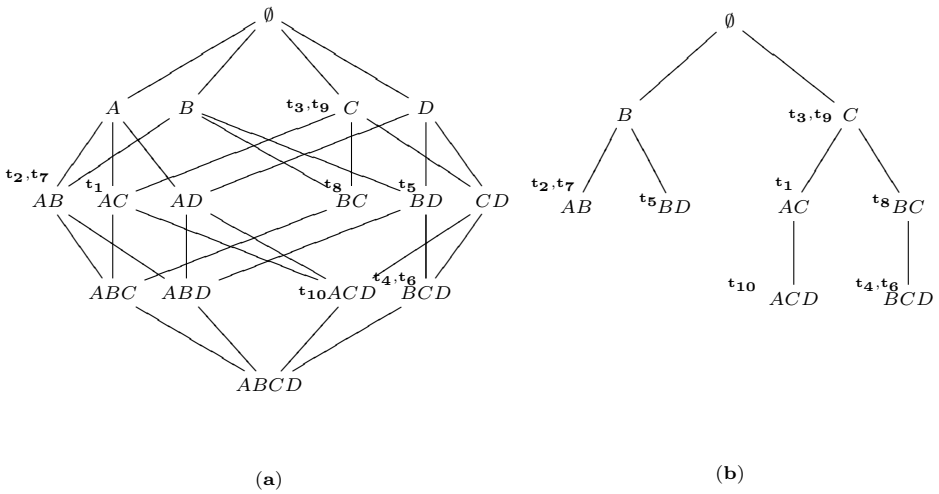

Fig. 4. An example of user hierarchy (a) and the corresponding tree (b)

keys used for encrypting tuples $\mathtt{t}_1$, $\mathtt{t}_3$, $\mathtt{t}_4$, $\mathtt{t}_6$, $\mathtt{t}_8$, $\mathtt{t}_9$, and $\mathtt{t}_{10}$. Obviously, this solution is not efficient and requires the management of too many keys. We propose an alternative solution that consists of collecting users into groups of privileges and encrypt each tuple (set thereof) with the key associated with the set of users who can access it. To this purpose, we base our approach on the definition and use of a *user hierarchy* whose elements are all possible sets of users and on which an order is defined corresponding to the subset relationship between them. Formally, a user hierarchy is defined as follows.

Definition 1 (User Hierarchy) *Given a set* $\mathcal{U}$ *of users, a* user hierarchy*, denoted* UH*, is a pair* $(\mathrm{P}(\mathcal{U}), \preceq)$*, where* $\mathrm{P}(\mathcal{U})$ *is the power set of* $\mathcal{U}$ *and* $\preceq$ *is a partial order on* $\mathrm{P}(\mathcal{U})$ *such that* $\forall \mathrm{X}, \mathrm{Y} \in \mathrm{P}(\mathcal{U})$, $\mathrm{X} \preceq \mathrm{Y}$ *iff* $\mathrm{Y} \subseteq \mathrm{X}$.

The subset relationship between sets of users implies a relationship on their rights. It is trivial to see that, given two sets of users X and Y, if Y is a subset of X (i.e., $\mathrm{X} \preceq \mathrm{Y}$), then users in Y can access all tuples that users in X can access and the vice versa is not true. With respect to our example in Figure 3, let BCD be the set of users `Bob`, `Carol`, `David`, and let BC be the set of users `Bob` and `Carol`. Users in BCD can access tuples $\mathtt{t}_4$ and $\mathtt{t}_6$ and users in BC can access tuples $\mathtt{t}_4$,$\mathtt{t}_6$, and $\mathtt{t}_8$.

By definition the user hierarchy includes all sets of users corresponding to ACL_i. A user hierarchy can be represented through a *directed acyclic graph* (DAG) having a node corresponding to each set of users and a path from node X to node Y if and only if $Y \preceq X$. Figure 4(a) illustrates the user hierarchy corresponding to the access matrix in Figure 3. Here, each node is labeled with the set of initial letters of the users' names belonging to the node, and each tuple $\mathtt{t_i}$ is depicted near node ACL_i. Our solution for the determination and assignment of keys exploits the user hierarchy together with a key generation and assignment schema based on the idea of key derivation. Sophisticated key derivation techniques that can be applied to DAGs have been extensively studied in the literature [2, 17, 19, 20]. Intuitively, these key generation schemes operate on the hierarchy computing the keys of lower-level nodes based on the keys of their predecessors. In other words, each node X of a hierarchy is associated with a key that can be used to derive the keys associated with all nodes Y, where $Y \preceq X$ and the opposite is not true. Therefore, using the user hierarchy for key assignment, each user u needs to know only the key associated with the node representing herself and each tuple t has to be encrypted with the key associated with the node representing its ACL. For instance, tuple $\mathtt{t_1}$ is encrypted with the key associated with node AC and `Carol` knows the key associated with node C (see Figure 4(a)). In this way, `Carol` can derive the key associated with node AC and can access tuple $\mathtt{t_1}$. Unfortunately, the key generation schemes working on DAGs are complex and they would require a lot of key storage (whose size grows exponentially with the nodes of the hierarchy). To avoid these problems, we decide to apply more simple techniques that work on trees [22]. We develop a greedy *transformation algorithm* from DAG to tree that converts a hierarchy in a corresponding tree [7]. This transformation is performed by first introducing in the tree the nodes corresponding to $ACLs$, and then by selecting, for each node, the "best parent node". This selection is performed by adopting a set of criteria that allow to reduce the number of keys in the system. For instance, a criterion requires the choice of the lowest candidate parent in the hierarchy, that is, the parent node corresponding to the biggest set of users. Another criterion states that it is better to choose as a parent, the node corresponding to an ACL. At the end of the transformation process, the algorithm removes from the structure obtained the nodes that are not necessary neither for encryption nor for key derivation. The resulting *user tree hierarchy* is defined as follows.

Definition 2 (User Tree Hierarchy) *Given a set $\mathcal{U}$ of users, a set $\mathcal{T}$ of tuples, and an access matrix $\mathcal{A}$, the* user tree hierarchy*, denoted* UTH*, is a pair* (N, $\preceq$)*, such that:*

- N $\subseteq$ P($\mathcal{U}$)*;*
- $\forall t \in \mathcal{T}$*,* $ACL_t \in N$*;*
- $\forall X, Y \in N$*,* X $\preceq$ Y *iff* Y $\subseteq$ X*;*
- $\forall X, Y, Z \in N$*,* X $\preceq$ Y *and* X $\preceq$ Z $\Rightarrow$ Z $\preceq$ Y *or* Y $\preceq$ Z*.*

The user tree hierarchy uses the same partial order relation as the one defined for the user hierarchy. The data owner has to communicate to each user $u \in \mathcal{U}$

the key associated with element $V \in N$ such that $u \in V$ and $u \notin W$, where W is the parent of V in the UTH. Note that to avoid accesses from untrusted users, the data owner has to check the users' identities before assigning them a key. Figure 4(b) illustrates the UTH corresponding to the user hierarchy in Figure 4(a). As an example, consider now user `Carol`: she knows key $\{k_C\}$ and can directly derive $\{k_{AC}, k_{BC}\}$ that in turns allows her to derive keys $\{k_{ACD}, k_{BCD}\}$. By using these keys, `Carol` can decrypt the set $\{\mathtt{t}_1, \mathtt{t}_3, \mathtt{t}_4, \mathtt{t}_6, \mathtt{t}_8, \mathtt{t}_9, \mathtt{t}_{10}\}$ of tuples corresponding to $CAP_{\mathtt{Carol}}$.

3 Metadata Management in the DAS Scenario

To properly access and manage the outsourced databases, the users, the data owners, and, possibly, the servers have to store some additional information that we call *metadata*. The client and server use these metadata to interpret and execute SQL statements, and to properly manage stored data. The distribution of metadata should follow two principles: *i)* users should know any additional metadata necessary to access the data for which they have a privilege, and *ii)* users should be able to efficiently search and query metadata by saving on bandwidth costs. To this purpose, metadata are stored in relational tables that can be accessed by SQL queries just like any other type of data. Metadata may be as simple as one keyword, or as complex as a derivation path for computing keys. There are three main types of metadata: *authorization metadata*, *descriptive metadata*, and *key management metadata*. Authorization metadata include information about the access control policy defined by the data owner (i.e., the access matrix). Basically, the authorization metadata contain the following tables (as usual, we underline the primary key of each relation).

- `TabUser(IdUser, Surname, Name)` maintains information about each user in the system. The schema of this table depends on the information needed by the data owner. For simplicity we assume that each user is identified by a unique identifier (attribute `IdUser`) and has a name (attribute `Name`) and surname (attribute `Surname`).
- `AccessMatrix(ERelation, Counter, IdUser)` maintains information about who (attribute `IdUser`) can access what (attributes `ERelation` and `Counter`).

These tables are very sensitive and therefore they have to be stored at the data owner's site. As an example, consider the access matrix in Figure 3: the corresponding authorization metadata are illustrated in Figure 5.

Descriptive metadata are data descriptors and are similar to the system catalogs automatically maintained by relational database systems. Basically, descriptive metadata describe the structure of the encrypted database. The main tables of the descriptive metadata are the following.

- `TabRelation(Relation, EncryptedRel)` maintains the correspondence between the name of a plaintext relation (attribute `Relation`) and the name of the corresponding encrypted relation (attribute `EncryptedRel`).

- `TabIndex(Relation, Attribute, Index, IdMethod)` maintains the correspondence between the name of an attribute (attribute `Attribute`) in a plaintext relation (attribute `Relation`) and the name of the corresponding index (attribute `Index`) together with the index method (attribute `IdMethod`).
- `TabMethod(IdMethod, Function, IdParameter, Value)` maintains information about the hash function (attribute `Function`) used with a specific index method (attribute `IdMethod`) together with the value (attribute `Value`) of the corresponding parameters (attribute `IdParameter`).
- `EncryptAlgo(Algorithm, IdParameter, Value)` maintains information about the encryption function (attribute `Algorithm`) used to encrypt data together with the value (attribute `Value`) of the corresponding parameters (attribute `IdParameter`).

The disclosure of these tables makes it possible for a malicious user to access the encrypted database. Therefore, the descriptive metadata should never be stored on the server. Note that each user only knows the portion of the descriptive metadata corresponding to the relations for which she has a read privilege in the access matrix; the data owner has instead a complete knowledge of these metadata. For instance, Figure 5 illustrates the descriptive metadata associated with `Carol` and the data owner. Here, we assume that the indexing method is a hash function implemented through the modular operator, that is, `I`=`A` mod M, where `I` is the index value corresponding to attribute `A` and M is a prime number stored in table `TabMethod`. We use this specific hash function because we need collisions.

Key management metadata include information about the key derivation method, the value of keys directly communicated by the data owner to users, and the key derivation paths. There are different strategies for storing these metadata: on clients, on server, or partially on clients and partially on server. While the client-side strategy saves network bandwidth but uses more client's memory, the server-side strategy requires more network bandwidth and saves client's storage capacity. We discuss these three strategies more in details in the following subsections.

3.1 Client-Side Key Metadata Storage

Key management metadata stored at each client include information about the portion of the user tree hierarchy associated with the corresponding user. Such a sub-hierarchy allows a user to derive the keys necessary for decrypting the data for which she has a read privilege. For instance, with respect to the user tree hierarchy in Figure 4(b), user `Carol` has to know the portion of the hierarchy rooted at node C. The relational tables stored at the client-side are therefore the following.

- `TabKey(IdKey, Value)` maintains the value of the keys (attribute `Value`) directly communicated to a user.

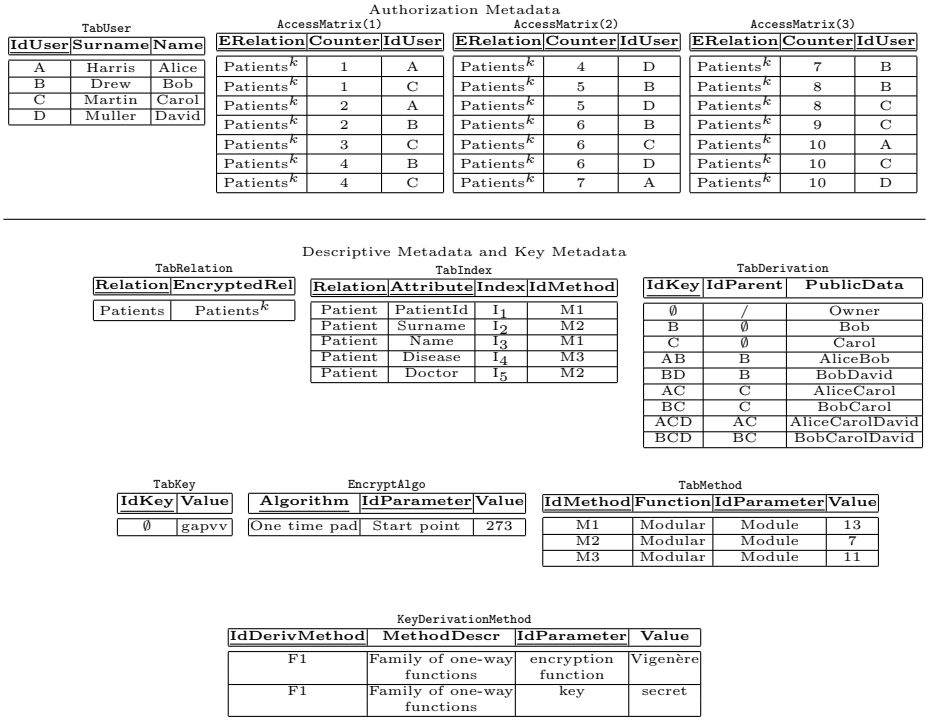

Fig. 5. Metadata associated with the data owner and `Carol`'s client

– `TabDerivation(IdKey, IdParent, PublicData)` maintains, for each key (attribute `IdKey`) in the considered sub-hierarchy, the identifier of its parent (attribute `IdParent`) and the public information (attribute `PublicData`)

necessary to derive the key; if a key corresponds to the root of the sub-hierarchy, attribute `IdParent` is conventionally set to `/`.
- `TabDecryption(EncryptedRelation, Counter, IdKey)` maintains, for each encrypted relation (attribute `EncryptedRelation`) and each tuple in the relation (attribute `Counter`), the identifier (attribute `IdKey`) of the decryption key associated with that tuple.
- `KeyDerivationMethod(IdDerivMethod, MethodDescr, IdParameter, Value)` maintains information about the key derivation method used for deriving the keys associated with the nodes in the user tree hierarchy.[5]

While users have to store the complete sub-hierarchy to which they can access, the data owner may decide to keep track of the information associated with each node of the hierarchy (i.e., the identifier and the public parameters used by the key derivation method) without storing the relationship parent-child. That is, the data owner can decide to store a simplified version of the `TabDerivation` table that includes only attributes `IdKey` and `PublicData`. Although this solution allows the data owner to save storage capacity, it requires to recompute the user tree hierarchy whenever the data owner needs to access the data. Moreover, if the access matrix changes (e.g., a user cannot access a tuple anymore) and the user tree hierarchy is updated without using the transformation algorithm, the new version of the hierarchy could be different from that obtained by applying the algorithm. In this case, also the data owner has to store the `TabDerivation` table as defined above.

3.2 Server-Side Key Metadata Storage

The client-side approach for storing the key management metadata has the great advantage that each user directly knows the information she needs to properly access the encrypted database. However, by analyzing these data more in details, it is easy to see that this approach requires a duplication of information: the association between a tuple t and the identifier of the key used to encrypt the tuple is duplicated for each user that can access t (table `TabDecryption`). The same applies for the key derivation paths: two users with non-disjoint user tree sub-hierarchies have a portion of the key derivation paths replicated in table `TabDerivation`.[6] The only sensitive information that should never be stored on the server is table `TabKey`. Therefore, to avoid data duplication and to allow the sharing of information among users, the user tree hierarchy and the association tuple-key identifier can be stored on the server. To this purpose, table `TabDerivation` containing the whole user tree hierarchy is maintained on the server and attribute `IdKey` defined in the relational schema of an encrypted relation (see Section 2) is used to maintain the association tuple-key identifier. To

[5] The schema of this relation may change depending on the key derivation method adopted.

[6] Note that the data inconsistency problem can be avoided by applying the traditional techniques developed in the distributed database area [5].

ensure metadata integrity, message authentication codes, that involve a secret key in the computation of the digest, are used. Obviously, the key used should be known by all users in the system. The main drawback of this solution, however, is that the user tree hierarchy traversal can only be performed by the client. This means that, to derive a key, the client has to perform a sequence of queries that retrieve tree nodes on a derivation path. Another minor drawback of this solution is that, due to the additional attribute `IdKey`, the result size returned to clients is greater than the result size obtained with the client-side strategy. However, the impact of attribute `IdKey` on the result size is minimal and therefore can be ignored.

3.3 Client-Side and Server-Side Combined Solution

A hybrid solution for storing the key management metadata can also be adopted thus combining the advantages of the two previous strategies. For instance, the association between tuple-key identifier can be stored on the server by using attribute `IdKey` as previously discussed, and the information used for the key derivation (i.e., the user tree hierarchy, the derivation method, and the public information associated with each element of the hierarchy) can be stored on the clients. In this way, we avoid a duplication of information and the key derivation process is more efficient because the client can execute it without querying the server.

The choice between a client-side, server-side, or a hybrid solution depends on the storage and bandwidth capacity available to clients. For instance, if the storage capacity is a more critical resource than the bandwidth capacity, a server-side solution is preferable. Otherwise, if the bandwidth capacity is a more critical resource than the storage capacity, a client-side or a hybrid solution is preferable. Note that when specific key derivation methods are used (e.g., the key derivation methods working on tree as in our approach), the size of the public data stored on clients is minimal and therefore the impact on the storage capacity is neglectable. For instance, the key derivation methods based on one-way hash functions [22] require, as public information, a unique name associated with each node of the hierarchy. The size of the public information is therefore of order $O(n \log n)$, where n is the number of elements in the user tree hierarchy. The key derivation methods working on DAGs and based on the modular exponentiation technique [2, 20] use as public data associated with an element n of the hierarchy, the product of the prime numbers associated with the nodes in the hierarchy that are not dominated by n. Therefore, in the worst case (i.e., for a leaf of the hierarchy) the size of the public information is of order $O(n(n-1)k) = O(n^2k)$, where n is the number of elements in the hierarchy and k is the number of bits for representing a prime number.

The computational cost of the derivation mechanism could be reduced if each client keeps a cache of the keys already computed. In this way, if the result of a query includes a tuple encrypted with such a key, it is not necessary recompute the decryption key. Obviously, this cache mechanism requires additional storage capacity on clients. It is also important to note that whenever there is a change

Original Clause (Q)	**Transformed Clause** (Q_s)
SELECT $\mathtt{A}_1, \ldots, \mathtt{A}_n$	SELECT `Counter`, `Etuple`, `IdKey`
FROM $\mathtt{R}_1, \ldots, \mathtt{R}_m$	FROM $\mathtt{R}_1^k, \ldots, \mathtt{R}_m^k$
WHERE $\mathtt{A}_j = \text{val}$	WHERE $\mathtt{I}_{\mathtt{A}_j} = f(\text{val})$
$\mathtt{A}_k = \mathtt{A}_w$	$\mathtt{I}_{\mathtt{A}_k} = \mathtt{I}_{\mathtt{A}_w}$

Fig. 6. Query transformation

in the access matrix, the cache should be cleared because the keys could be changed. Figure 5 illustrates the metadata associated with user `Carol` and the data owner by using a hybrid solution.

4 Query Processing

We now address the issue of evaluating client queries in the DAS scenario where a hybrid solution for storing the metadata is adopted. For simplicity, we assume that the encrypted database $\mathtt{B}^k$ consists of a single relation $\mathtt{Patients}^k$ (see Figure 2(b)), and that queries are selection-project expressions.[7] Based on the metadata stored, a query Q on a plaintext relation is split into a query Q_s on the corresponding encrypted relation that is executed on the server, and a client query Q_c for post-processing result of the server query. The transformation between query Q and query Q_s is performed as illustrated in Figure 6. As it is visible from this table, the list of attributes in the SELECT clause is replaced by attributes `Counter`, `Etuple`, and `IdKey`: due to the fact that relations are encrypted at the tuple level, a server can only return the whole encrypted tuple `Etuple`, and therefore the projection operation cannot be executed on the server. Attribute `IdKey` is necessary to identify the decrypting key. The list of relations in the FROM clause is replaced by the list of corresponding encrypted relations (table `TabRelation`) and conditions in the WHERE clause are transformed according to the index techniques. More precisely, each attribute $\mathtt{A}_j$ in the WHERE clause is replaced by the corresponding index (table `TabIndex`) and constant values are transformed by applying the appropriate index technique (table `TabMethod`).

As an example, suppose that `Carol` wants to find the name, surname, and doctor of patients who disease is "Tonsillitis". The SQL query is as follows.

$Q \equiv$ SELECT `PatientId`, `Surname`, `Name`, `Doctor`
FROM `Patients`
WHERE `Disease`= "Tonsillitis"

The query processor module retrieves from the metadata the name of the encrypted relation corresponding to `Patients`, the name of the index corresponding to attribute `Disease` and the hash function (with its parameters) used

[7] Note that more complex queries can also be supported (e.g., range queries can be supported by means of indexes based on B+-trees [9]). Details for this, however, are beyond the scope of this paper.

Counter	Etuple	IdKey
1	*r*tso/yui+*	*AC*
2	hai4de-0q1	AB
5	3gia*ni+aL	BD
8	*/bur21/*-D*	*BC*
10	*bew0"!DE1a*	*ACD*

(a)

PatientId	Surname	Name	Doctor
125YP894	Carter	Andrew	Wayne
364UK784	Rogers	Laura	Wayne

(b)

Fig. 7. Encrypted query result (a) and final result returned to `Carol` (b)

for creating the index. In this phase, it is also necessary to retrieve both the encryption function and the key derivation method, together with their parameters. To this purpose, the following SQL queries are executed (here, we use an embedded SQL syntax).

```
SELECT EncryptedRel INTO :R
FROM TabRelation
WHERE Relation = "Patients"
```

```
SELECT Index, IdMethod INTO :I, :M
FROM TabIndex
WHERE Relation = "Patients"
AND Attribute= "Disease"
```

```
SELECT Function INTO :h
FROM TabMethod
WHERE IdMethod = :M
```

```
SELECT Value INTO :P
FROM TabMethod
WHERE IdMethod = :M
```

```
SELECT Algorithm, Value INTO :E, :Pe
FROM EncryptAlgo
```

```
SELECT MethodDescr, Value INTO :DM, :Pm
FROM KeyDerivationMethod
```

By assuming that the value of variable `R` is $\texttt{Patients}^{\texttt{k}}$, the value of variable `I` is $\texttt{I}_4$, and the index value corresponding to "Tonsillitis" is π, the original plaintext query Q is translated as follows:[8]

$$Q_s \equiv \begin{array}{l}\text{SELECT } \texttt{Counter}, \texttt{Etuple}, \texttt{IdKey} \\ \text{FROM } \texttt{Patients}^{\texttt{k}} \\ \text{WHERE } \texttt{I}_4 = \text{“}\pi\text{”}\end{array}$$

Figure 7(a) illustrates the query result returned to the client. The client has to decrypt all tuples that `Carol` can access (a tuple is accessible by a user when the corresponding `IdKey` value appears in table `TabKey` or table `TabDerivation`) and to filter out those not matching the actual query predicates in Q. In our example, `Carol` can access tuples $\texttt{t}_1$, $\texttt{t}_8$, and $\texttt{t}_{10}$ and tuples $\texttt{t}_2$ and $\texttt{t}_5$ are discarded. The keys to be used for decrypting the tuples are computed as follows. First, for each tuple t of the query result, if table `TabKey` contains a tuple t' where $t'[\texttt{IdKey}] = t[\texttt{IdKey}]$, then the decryption key is already available. Otherwise, the decryption key is obtained by following the key derivation path stored in table `TabDerivation`.

[8] Note that Q_s is a dynamic embedded SQL statement that is built at run time and placed in a string host variable. For simplicity, we report the final format of the query executed at the server side.

Algorithm 1 (Key derivation path)

KeyDerivationPath(t[IdKey])
/* Initializes some variables */
i:=1; $Path[i]$:= t[IdKey]
While $Path[i] \neq$ / **do**
    $i := i + 1$
    SELECT IdParent INTO :$Path[i]$
    FROM TabDerivation
    WHERE IdKey=:$Path[i-1]$
return $Path$

Fig. 8. Algorithm for computing the key derivation path

Figure 8 illustrates the procedure for computing the key derivation path. Intuitively, for each tuple t, by starting from the leaf (i.e., t[IdKey]) of the path, table TabDerivation is queried to determine the parent of the current node of the path. The procedure terminates when root / is reached. Array $Path$ stores the key derivation path in reverse order. For instance, consider tuple $\mathtt{t}_{10}$: it is encrypted with key k_{ACD} that can be obtained by following the path $Path[3] = k_C \rightarrow Path[2] = k_{AC} \rightarrow Path[1] = k_{ACD}$. The decryption key is computed by applying the key derivation method DM along $Path$. Then, the client has to: (1) decrypt the tuples using function E, (2) apply the original condition to discard possibly spurious tuples that do not belong to the result set, and (3) execute the requested projections. Spurious tuples are discarded by applying the following query:

$Q_c \equiv$ SELECT PatientId, Surname, Name, Doctor
FROM Result
WHERE Disease= "Tonsillitis"

Figure 7(b) reports the final set of tuples that user Carol can read. Note that these tuples are a subset of the tuples for which Carol has the read privilege (see the access matrix in Figure 3).

As the server returns to the client also tuples that she cannot read, data may be subject to inference attacks. The inference problem in the DAS scenario has been considered in [6, 9], where the authors gave a quantitative model for evaluating the robustness of the indexes obtained by applying either the direct encryption or a hash-based method. In summary, they shown that to achieve a higher degree of protection against inference, it is convenient to use a hash function to encode indexes values.

5 Conclusions and Future Work

The management of metadata for accessing a remote encrypted database is of crucial importance in the database-as-a-service scenario. In this paper, we presented the metadata that provide abstract descriptions of the data structures

and data formats used in the underlying system. Issues to be investigated will include: *(i)* an effective implementation of the different solutions presented for metadata storage to better evaluate the trade off between storage and bandwidth consumption, and *(ii)* an evaluation of strategies addressing the dynamic updates of the access rights [7].

Acknowledgments

This work was supported in part by the European Union within the PRIME Project in the FP6/IST Programme under contract IST-2002-507591 and by the Italian MIUR within the KIWI and MAPS projects.

References

1. R. Agrawal, J. Kierman, R. Srikant, and Y. Xu. Order preserving encryption for numeric data. In *Proc. of ACM SIGMOND 2004*, Paris, France, June 2004.
2. S. Akl and P. Taylor. Cryptographic solution to a problem of access control in a hierarchy. *ACM Transactions on Computer System*, 1(3):239–248, August 1983.
3. C. Boyens and O. Gunter. Using online services in untrusted environments - a privacy-preserving architecture. In *Proc. of the 11th European Conference on Information Systems (ECIS '03)*, Naples, Italy, June 2003.
4. R. Brinkman, J. Doumen, and W. Jonker. Using secret sharing for searching in encrypted data. In *Proc. of the Secure Data Management Workshop*, Toronto, Canada, August 2004.
5. S. Ceri and G. Pelegatti. *Distributed Database Systems: Principles and Systems*. McGraw-Hill, 1984.
6. A. Ceselli, E. Damiani, S. De Capitani di Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati. Modeling and assessing inference exposure in encrypted databases. *ACM Transactions on Information and System Security (TISSEC)*, 8(1):119–152, February 2005.
7. E. Damiani, S. De Capitani di Vimercati, S. Foresti, S. Jajodia, and P. Samarati. Selective release of information in outsourced encrypted database. Technical report, University of Milan, 2005.
8. E. Damiani, S. De Capitani di Vimercati, M. Finetti, S. Paraboschi, P. Samarati, and S. Jajodia. Implementation of a storage mechanism for untrusted DBMSs. In *Proc. of the Second International IEEE Security in Storage Workshop*, Washington DC, USA, May 2003.
9. E. Damiani, S. De Capitani di Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati. Balancing confidentiality and efficiency in untrusted relational DBMSs. In *Proc. of the 10th ACM Conference on Computer and Communications Security*, Washington, DC, USA, October 27-31 2003.
10. G.I. Davida, D.L. Wells, and J.B. Kam. A database encryption system with subkeys. *ACM Transactions on Database Systems*, 6(2):312–328, June 1981.
11. J. Domingo-Ferrer and J. Herrera-Joanconmarti. A privacy homomorphism allowing field operations on encrypted data. *Jornades de Matematica Discreta i Algorismica*, March 1998.

12. H. Hacigümüs, B. Iyer, and S. Mehrotra. Providing database as a service. In *Proc. of 18th International Conference on Data Engineering*, San Jose, California, USA, February 2002.
13. H. Hacigümüs, B. Iyer, and S. Mehrotra. Ensuring integrity of encrypted databases in database as a service model. In *Proc. of the IFIP Conference on Data and Applications Security*, Estes Park Colorado, August 2003.
14. H. Hacigümüs, B. Iyer, and S. Mehrotra. Efficient execution of aggregation queries over encrypted relational databases. In *Proc. of the 9th International Conference on Database Systems for Advanced Applications*, Jeju Island, Korea, March 2004.
15. H. Hacigümüs, B. Iyer, S. Mehrotra, and C. Li. Executing SQL over encrypted data in the database-service-provider model. In *Proc. of the ACM SIGMOD'2002*, Madison, Wisconsin, USA, June 2002.
16. H. Hacigümüs and S. Mehrotra. Performance-conscious key management in encrypted databases. In *Proc. of the 18th Annual IFIP WG 11.3 Working Conference on Data and Applications Security*, Sitges, Catalonia, Spain, July 2004.
17. L. Harn and H. Lin. A cryptographic key generation scheme for multilevel data security. *Computers and Security*, 9(6):539–546, October 1990.
18. B. Hore, S. Mehrotra, and G. Tsudik. A privacy-preserving index for range queries. In *Proc. of the 30th VLDB Conference*, Toronto, Canada, 2004.
19. M. Hwang and W. Yang. Controlling access in large partially ordered hierarchies using cryptographic keys. *The Journal of Systems and Software*, 67(2):99–107, July 2003.
20. S. MacKinnon, P.Taylor, H. Meijer, and S.Akl. An optimal algorithm for assigning cryptographic keys to control access in a hierarchy. *IEEE Transactions on Computers*, 34(9):797–802, September 1985.
21. E. Mykletun, M. Narasimha, and G. Tsudik. Authentication and integrity in outsourced database. In *Proc. of the 11th Annual Network and Distributed System Security Symposium*, San Diego, California, USA, February 2004.
22. R.S. Sandhu. Cryptographic implementation of a tree hierarchy for access control. *Information Processing Letters*, 27(2):95–98, April 1988.