Private Data Indexes for Selective Access to Outsourced Data

Sabrina De Capitani di Vimercati
Università degli Studi di Milano
26013 Crema - Italy
sabrina.decapitani@unimi.it

Sara Foresti
Università degli Studi di Milano
26013 Crema - Italy
sara.foresti@unimi.it

Sushil Jajodia
George Mason University
Fairfax, VA 22030-4444
jajodia@gmu.edu

Stefano Paraboschi
Università di Bergamo
24044 Dalmine - Italy
parabosc@unibg.it

Pierangela Samarati
Università degli Studi di Milano
26013 Crema - Italy
pierangela.samarati@unimi.it

ABSTRACT

Cloud storage services have recently emerged as a successful approach for making resources conveniently available to large communities of users. Several techniques have been investigated for enabling such services, including encryption for ensuring data protection, as well as indexing for enabling efficient query execution on encrypted data. When data are to be made available selectively, the combined use of the two techniques must be handled with care, since indexes can put the confidentiality protection guaranteed by encryption at risk.

In this paper, we investigate this issue and propose an indexing technique for supporting efficient access to encrypted data while preventing possible disclosure of data to users not authorized to access them. Intuitively, our indexing technique accounts for authorizations when producing indexes so to ensure that different occurrences of the same plaintext value, but accessible by different sets of users, be not recognizable from their indexes. We show that our solution exhibits a limited performance overhead in query evaluation, while preventing leakage of information.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Access controls*; H.2.4 [**Database Management**]: Systems—*Relational databases*; H.2.4 [**Database Management**]: Systems—*Query processing*; H.2.7 [**Database Management**]: Database Administration—*Security, integrity, and protection*; H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing—*Indexing methods*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms

Security, Design, Management

Keywords

Privacy, data indexes, access control, data outsourcing

1. INTRODUCTION

Users as well as governmental, public, and private institutions are more and more relying on the use of service providers for the remote storage and dissemination of data. As a matter of fact, interest on cloud technology is considerably increasing, and network storage services represent one of its most successful applications, conveniently allowing data owners to make their data and resources available to a large community of users without the need of building a complex infrastructure and dealing with the many organizational and security issues associated with managing an own server connected to the outside world.

Wide scale adoption and acceptance of network storage services can effectively take place only if the data owner has some assurance that, while externally stored, data are properly protected. The basic approach typically adopted for ensuring protection to externally stored data relies on the use of encryption, which provides both integrity (allowing detection of possible data tampering) as well as confidentiality (since the data will be intelligible only to users who know the decryption key) [14, 15, 17, 20, 25]. Data confidentiality may need to be guaranteed not only from external users but also from the storage server itself that, while usually relied upon for data availability and services, may not be allowed to know the content of the data stored. The assurance that data content be hidden from the server is typically desired by the data owner and might be demanded by law (e.g., in the case of sensitive financial [19] or health care information [16]). Since maintaining confidentiality with respect to the storing server implies that the server cannot decrypt data for executing queries, outsourced encrypted data are typically associated with indexing information that can be exploited by the server to execute queries. Several approaches have been proposed by the research and development communities for indexing encrypted data so to efficiently query them (e.g., [1, 7, 15, 24]).

WPES'11, October 17, 2011, Chicago, Illinois, USA.
Copyright 2011 ACM 978-1-4503-1002-4/11/10 ...$10.00.

Shops

	Id	City	Year	Sales
t_1	001	NY	2010	600
t_2	002	Rome	2010	700
t_3	003	Rome	2011	600
t_4	004	NY	2011	700
t_5	005	Oslo	2011	700

(a)

	acl
t_1	A
t_2	A,B
t_3	B
t_4	A,C
t_5	C

(b)

Figure 1: An example of plaintext relation (a) and its access control policy (b)

An orthogonal aspect that needs to be addressed in data outsourcing relates to the need of the data owner to provide different data views to different users. This selective data access can be conveniently realized simply by assuming the use of different encryption keys for different portions of the data, producing for each user a view containing only the data that she can decrypt. Exploiting key derivation techniques and user group hierarchies, selective encryption can be realized while avoiding the need for the users to store different keys and for the data owner to produce different encrypted versions of the resources [8].

While indexing can be realized limiting the potential exposure of confidential information due to indexes (e.g., [1, 5, 7, 15, 24]) and selective encryption naturally extends existing encryption approaches, the combination of the two solutions in a safe way is still an open problem. In fact, the cloud storage server is typically assumed to offer a pure storage service and is not required to guarantee a strict separation among the portions of information available to the different users. As a matter of fact, selective encryption ensures obedience to the authorization policy without need for the server to enforce any control or to authenticate users. Hence, users could potentially have visibility of indexes even of tuples they are not allowed to access. Such visibility, together with the ability to view data for which they are authorized, opens the door to possible inferences by users.

In this paper, we address the problem of providing selective access to encrypted data and exploiting indexes for query execution, while ensuring that indexes do not allow users to draw inferences on tuples they are not authorized to access. Our work aims at closing the gap between indexing for efficient querying on one side and selective encryption for authorization enforcement on the other side, thus allowing for their joint exploitation.

For concreteness and simplicity, in line with existing approaches, in this paper data are organized in a relational table and indexes are defined on attributes. We note however that our proposal can be adapted to scenarios considering generic resources.

The contributions of this paper can be summarized as follows. We characterize the exposure of confidential information due to indexes, when access to the data is regulated by an access control policy (Section 3). We introduce an index function that computes index values that depend not only on the plaintext values they represents, but also on the users who can access them (Section 4). We then illustrate how to define this indexing technique to guarantee protection against inferences and support for efficient query evaluation (Section 5). We describe how queries formulated on the original (plaintext) relation are translated into equivalent queries on the corresponding encrypted relation using our indexes (Section 6). We also present the result of an experimental evaluation confirming the efficiency and effectiveness of our solution (Section 7). Finally, we discuss related works (Section 8) and give our conclusions (Section 9).

Shops^e

tid	etuple	$\mathtt{I}_c$	$\mathtt{I}_y$	$\mathtt{I}_s$
1	α	ι(NY)	ι(2010)	ι(600)
2	β	ι(Rome)	ι(2010)	ι(700)
3	γ	ι(Rome)	ι(2011)	ι(600)
4	δ	ι(NY)	ι(2011)	ι(700)
5	ϵ	ι(Oslo)	ι(2011)	ι(700)

Figure 2: An example of encrypted version of plaintext relation Shops in Figure 1(a)

2. BASIC CONCEPTS

We consider a cloud scenario where a *data owner* outsources her data to an external *cloud server*. The cloud server is considered *honest but curious*, that is, it is trusted with providing the required service but not authorized to read the content of the outsourced data. The cloud server is in charge of executing searches on the outsourced data on behalf of *authorized users*.

Outsourced data are modeled as a relational table r over schema $R(\mathtt{A}_1,\ldots,\mathtt{A}_n)$. Since the cloud server is not authorized to know the content of relation r, the data owner encrypts r before outsourcing it. Consistently with existing works (e.g., [5, 15, 24]), we assume symmetric encryption to be applied and r to be encrypted at the tuple level (i.e., each tuple is individually encrypted). Search operations (queries) on the encrypted relation are supported via a set of *indexes* build on the plaintext values of the attributes of the relation. The encrypted relation contains an index for each attribute on which conditions need to be evaluated and for which the advantage in query evaluation justifies the additional storage cost of the index (i.e., attribute selectivity is not low). An encrypted relation is formally defined as follows.

Definition 2.1 (Encrypted relation). *Let r be a relation over relation schema $R(\mathtt{A}_1,\ldots,\mathtt{A}_n)$. The* encrypted *version of r is a relation r^e over schema R^e(<u>tid</u>,etuple,$\mathtt{I}_1,\ldots,\mathtt{I}_l$), where $\forall t\in r$, $\exists t^e\in r^e$ such that t^e[etuple]$=E_k(t)$, with E a symmetric encryption function with key k, and $\mathtt{I}_i=\iota(t[\mathtt{A}_{j_i}])$, $i=1,\ldots,l$, with ι an index function.*

According to Definition 2.1, the encrypted version of relation r over schema $R(\mathtt{A}_1,\ldots,\mathtt{A}_n)$ is a relation r^e with attributes: *tid*, a numerical attribute added to the encrypted relation that acts as a primary key; *etuple*, an attribute that contains the ciphertext resulting from the encryption of a tuple; $\mathtt{I}_i$, i=1,…,l, an attribute that corresponds to the index over attribute $\mathtt{A}_{j_i}\in R$. With reference to plaintext relation Shops Figure 1(a), Figure 2 illustrates the corresponding encrypted relation Shops^e, where $\mathtt{I}_c$, $\mathtt{I}_y$, and $\mathtt{I}_s$ are indexes over attributes City, Year, and Sales, respectively. For the sake of readability of our examples, we use Greek letters to denote encrypted tuples and we report the tuples in the encrypted relation following the same order of the tuples in the plaintext relation.

Most of the indexing techniques proposed in the literature can be classified as follows.

- *Direct index* (e.g., [7]). The index is obtained by applying an encryption function to the plaintext values of the attribute. Index function ι then maps each plaintext value to a different index value and viceversa.

- *Flattened index* (e.g., [24]). The index is obtained by applying an encryption function to the plaintext values of the attribute and a post processing that flattens the distribution of index values. Index function ι then maps each plaintext value to a set of index values, but each index value corresponds to a unique plaintext value.

- *Bucket-based index* (e.g., [7, 15]). The index is obtained by distributing the tuples in buckets. The buckets can be explicitly defined by partitioning the domain of the attribute or by applying a secure hash function that distributes the plaintext values of the attribute in buckets containing the tuples generating a collision. Index function ι then maps different plaintext values to the same index value.

These indexing techniques support the evaluation at the cloud server of SELECT-FROM-WHERE SQL queries of the form "SELECT *Att* FROM *R* WHERE *Cond*", where $Att \subseteq R$ and *Cond* includes conditions of the form $\texttt{A}=v$, with $\texttt{A} \in R$ and v a constant value in the domain of `A`. Each query on the plaintext relation is translated into an equivalent query on the encrypted relation by applying the index function to the values appearing in the query. For instance, query "SELECT `City`, `Sales` FROM Shops WHERE `Year`=2010" on the plaintext relation in Figure 1(a) is translated into query "SELECT `etuple` FROM Shops^e WHERE $\texttt{I}_y = \iota(2010)$" on the encrypted relation in Figure 2.

3. PROBLEM DEFINITION

We consider a scenario where the data owner defines authorizations on the outsourced data to provide different views over the relation to different users of the system (we assume access by users to the tuples is read-only, while write operations are to be performed at the owner site, typically by the owner herself). We consider authorizations to be defined at the granularity of tuple, that is, each tuple within the relation can be read by a potentially different set of users. Given a tuple t, $acl(t)$ denotes the access control list associated with t, that is, the set of users who can access t. Figure 1(b) illustrates an example of access control policy for relation Shops in Figure 1(a). Authorizations holding on a relation can be realized when outsourcing the data by means of *selective encryption*, that is, by assuming the adoption of different keys for different tuples, depending on their acl, and ensuring that each user can decrypt all and only the tuples she is authorized to access. Selective encryption can be conveniently applied by releasing an individual key to each user, by encrypting each tuple with one key, and by exploiting key derivation techniques and a hierarchical organization of the keys based on the containment relationship between groups of users (i.e., acls) [8].

Shops

	Id	City	Year	Sales
t_1	▨	▨	▨	▨
t_2	002	Rome	2010	700
t_3	003	Rome	2011	600
t_4	▨	▨	▨	▨
t_5	▨	▨	▨	▨

Shops^e

tid	etuple	$\texttt{I}_c$	$\texttt{I}_y$	$\texttt{I}_s$
1	α	ι(NY)	ι(2010)	ι(600)
2	β	ι(Rome)	ι(2010)	ι(700)
3	γ	ι(Rome)	ι(2011)	ι(600)
4	δ	ι(NY)	ι(2011)	ι(700)
5	ϵ	ι(Oslo)	ι(2011)	ι(700)

(a)

Shops

	Id	City	Year	Sales
t_1	▨	▨	2010	600
t_2	002	Rome	2010	700
t_3	003	Rome	2011	600
t_4	▨	▨	2011	700
t_5	▨	▨	2011	700

Shops^e

tid	etuple	$\texttt{I}_c$	$\texttt{I}_y$	$\texttt{I}_s$
1	α	ι(NY)	ι(2010)	ι(600)
2	β	ι(Rome)	ι(2010)	ι(700)
3	γ	ι(Rome)	ι(2011)	ι(600)
4	δ	ι(NY)	ι(2011)	ι(700)
5	ϵ	ι(Oslo)	ι(2011)	ι(700)

(b)

Figure 3: Knowledge of user B before (a) and after (b) the inference

While selective encryption guarantees the correct enforcement of the access control policy (ensuring that each user can decrypt all and only tuples that she is authorized to access), the existence of indexes opens the door to possible leakages allowing users to infer the content of tuples they are not authorized to access. In fact, index values are closely related to the plaintext values they represent. Inferences can then happen when a user observes different occurrences of the same index value, some associated with tuples she is authorized to access and some associated with tuples she is not authorized to access, or when the user exploits knowledge of the index function to guess values that should be hidden to her.

To illustrate the problem let us first clarify the user knowledge on the system. The user knows: *i)* the index functions used to define the indexes in the encrypted relation (since they are available user-side for query generation); *ii)* the plaintext tuples that she is authorized to access (since she can retrieve them from the external server and decrypt their content); and *iii)* the encrypted relation potentially in its entirety (we assume that the server does not restrict access to the data based on the authorizations, and a query on the encrypted data can retrieve a superset of the tuples accessible by the user [10]). For instance, Figure 3(a) illustrates the knowledge of user B for our example, providing her view over the plaintext relation and over the encrypted relation. Here, grey cells in relation Shops denote the information that user B is not authorized to access.

Let us illustrate some knowledge that a user can infer over the content of tuples she is not authorized to read.

When direct indexes are used, the user knowledge exposes all the cells having the *same plaintext values* as the ones that the user knows, since each plaintext value is always represented by the same index value and viceversa. For instance, user B can access the second and third tuple in Shops^e and therefore she knows that index values ι(Rome), ι(2010), ι(2011), ι(700), and ι(600) correspond to plaintext values Rome, 2010, 2011, 700, and 600, respectively. As a consequence, user B infers that the first, fourth, and fifth tuples of relation Shops include the association ⟨2010,600⟩, ⟨2011,700⟩, and ⟨2011,700⟩ for the pair of attributes ⟨`Year`,`Sales`⟩, respectively. Also, B can infer that these tuples cannot assume value Rome for attribute `City`, since their value for $\texttt{I}_c$ is different from ι(Rome). Figure 3(b) illustrates the view of user B over relation Shops after exploiting her knowledge on the index-plaintext value correspondences. Here, light grey (orange) cells represent the index values in relation Shops^e and the information in relation Shops inferred by user B exploiting such index values. Grey cells in relation Shops denote the (unauthorized) information that user B cannot infer. In addition to these

Shops^e

tid	**etuple**	$\mathtt{I}_c$	$\mathtt{I}_y$	$\mathtt{I}_s$
1	α	ι_A(NY)	ι_A(2010)	ι_A(600)
2	β	ι_{AB}(Rome)	ι_{AB}(2010)	ι_{AB}(700)
3	γ	ι_B(Rome)	ι_B(2011)	ι_B(600)
4	δ	ι_{AC}(NY)	ι_{AC}(2011)	ι_{AC}(700)
5	ϵ	ι_C(Oslo)	ι_C(2011)	ι_C(700)

Figure 4: An example of encrypted relation with *acl*-based indexes

inferences that exploit different occurrences of the same index, a user could exploit her knowledge of the index function ι to determine the index $\iota(v)$ associated with each plaintext value v in the domain of each attribute $\mathtt{A}$ in R. This complete knowledge of the index-plaintext value correspondences would then permit the user to infer the whole content of the outsourced relation. Flattened indexes suffer from the same exposure to inference as direct indexes, since all the occurrences of an index value in the encrypted relation represent the same plaintext attribute value and users know the index function ι. When a user decrypts a tuple that she can access, the user can reconstruct the index-plaintext value correspondences for all the attribute values in the tuple. The user can then exploit her knowledge of the index function ι to determine the different index values to which the plaintext values that she knows have been mapped (to flatten the distribution of index values). Bucket-based indexes limit exposure to inference thanks to collisions as, when observing multiple occurrences of an index value, the user remains uncertain over which of the different colliding plaintext values corresponds to each occurrence. However, the user can establish with certainty that the index value does not correspond to certain plaintext values.

In the remainder of this paper, we illustrate an approach to compute indexes associated with data while ensuring protection from inferences such as the ones described above. To simplify the discussion, we assume that all the indexes associated with an encrypted relation are direct indexes. By ensuring protection in such a worst case scenario, our technique is clearly applicable when other indexing techniques are used.

4. CONFIDENTIAL INDEXING

Our solution for producing privacy-aware indexes builds on the observations made in the previous section characterizing vulnerabilities from inferences. In particular, to block such inferences, we require that different occurrences of a same plaintext value be mapped to different index values whenever such occurrences are subject to different authorizations. For instance, with reference to Figure 1, the index associated with value 2010 of attribute `Year` in the first tuple in relation Shops should be different from the index associated with the same value in the second tuple, since the two tuples have different acls. It is easy to see that guaranteeing such a property would block B's ability to infer the presence of 2010 in the first tuple.

A natural solution that first comes to mind, to guarantee that indexes be different for a same plaintext value if acls are different, is to produce index values directly depending on the acls (i.e., obtained by encrypting plaintext values with a key that only users in the acl know or can derive [2, 8]). Figure 4 illustrates an encrypted version of relation Shops in Figure 1(a), where indexes over attributes `City`, `Year`, and `Sales` have been computed based on the acls in Figure 1(b). Notation ι_U denotes an index function that only users in U can compute. While providing the required protection, this straightforward solution imposes a considerable burden at the client side. In fact, every condition of the form $\mathtt{A}$=v in a query submitted by user u would need to be translated into an equivalent condition of the form $\mathtt{I}_\mathtt{A}$ in V, where $\mathtt{I}_\mathtt{A}$ is the index over attribute $\mathtt{A}$ and V is the set of all possible index values produced by all index functions ι_U with $u \in U$. For instance, condition "`Year`=2010" in a query submitted by user B is translated as "$\mathtt{I}_y$ in $\{\iota_B(2010),\ \iota_{AB}(2010)\}$" if B knows the acls of the two tuples that she can access; as "$\mathtt{I}_y$ in $\{\iota_B(2010),\ \iota_{AB}(2010),\ \iota_{BC}(2010),\ \iota_{ABC}(2010)\}$", otherwise. Such index functions should be explicitly maintained and computed by the user, making query evaluation cumbersome at the client side.

Shops^e

tid	**etuple**	$\mathtt{I}_c$	$\mathtt{I}_y$	$\mathtt{I}_s$
1	α	ι_A(NY)	ι_A(2010)	ι_A(600)
2	β	ι_A(Rome)ι_B(Rome)	ι_A(2010)ι_B(2010)	ι_A(700)ι_B(700)
3	γ	ι_B(Rome)	ι_B(2011)	ι_B(600)
4	δ	ι_A(NY)ι_C(NY)	ι_A(2011)ι_C(2011)	ι_A(700)ι_C(700)
5	ϵ	ι_C(Oslo)	ι_C(2011)	ι_C(700)

Figure 5: An example of encrypted relation with user-based indexes

To minimize the burden at the client side, our approach makes indexing user-dependent while enabling automatic (salt-based) computation of the different index values to which a plaintext value can correspond. Each user u has an index function ι_u that depends on a private piece of information (e.g., an encryption key k_u) that she shares with the data owner. For each tuple t in r over schema R, indexed attribute $\mathtt{A}$ in R, and user u in $acl(t)$, the data owner computes an index value as $\iota_u(t[\mathtt{A}])$. (For the sake of exposition, we assume that a user applies the same index function over all attributes. Each user can easily adopt a different index function for each attribute.) Considering relation Shops and its access control policy in Figure 1, Figure 5 illustrates an encrypted version of the relation, where the indexes for attributes `City`, `Year`, and `Sales` have been computed adopting user-based indexing.

The adoption of user-based indexing does not close our story as simple user-based indexing would remain vulnerable to inferences. As per our starting observation, we must ensure that different occurrences of the same value be not inferrable by unauthorized users observing indexes. We note that a user can potentially draw inferences on tuples that she cannot access only when such tuples have index values in common with other tuples that the user can access. In other words, there can be inference whenever there are plaintext tuples that have the same value for at least one attribute and can be accessed by different, but overlapping, sets of users (i.e., the tuples have different but overlapping acls). We call plaintext tuples in such a relationship *conflicting tuples*, as formally stated in the following definition.

Definition 4.1 (Conflicting tuples). *Let r be a plaintext relation over relation schema $R(\mathtt{A}_1,\ldots,\mathtt{A}_n)$. Two tuples t_i,$t_j$$\in$$r$ are said to be in* conflict *over attribute $\mathtt{A}$, denoted $t_i$$\sim_\mathtt{A}$$t_j$, iff:* i) *$t_i[\mathtt{A}] = t_j[\mathtt{A}]$;* ii) *$acl(t_i) \neq acl(t_j)$, and $acl(t_i) \cap acl(t_j) \neq \emptyset$.*

Shops

	Id	City	Year	Sales
t_1	001	NY	2010	600
t_2	002	Rome	2010	700
t_3	003	Rome	2011	600
t_4	004	NY	2011	700
t_5	005	Oslo	2011	700

(a)

	acl
t_1	A
t_2	A,B
t_3	B
t_4	A,C
t_5	C

(b)

$t_1{\sim}_{\texttt{City}}t_4$
$t_2{\sim}_{\texttt{City}}t_3$
$t_1{\sim}_{\texttt{Year}}t_2$
$t_4{\sim}_{\texttt{Year}}t_5$
$t_2{\sim}_{\texttt{Sales}}t_4$
$t_4{\sim}_{\texttt{Sales}}t_5$

(c)

Figure 6: Plaintext relation Shops (a), its access control policy (b), and conflicting tuples (c)

Figure 6(c) summarizes the conflicting tuples in relation Shops in Figure 1(a) that, for the readers convenience, has been reproduced together with its access control policy in Figures 6(a)-(b). Simple user-based indexing would then allow visibility over the fact that: $t_1[\texttt{City}]{=}t_4[\texttt{City}]$, $t_2[\texttt{City}]{=}t_3[\texttt{City}]$, $t_1[\texttt{Year}]{=}t_2[\texttt{Year}]$, $t_4[\texttt{Year}]{=}t_5[\texttt{Year}]$, $t_2[\texttt{Sales}]{=}t_4[\texttt{Sales}]$, and $t_4[\texttt{Sales}]{=}t_5[\texttt{Sales}]$.

When two tuples are in conflict over an attribute, the attribute value in one of the tuples is exposed (through inference) to all users allowed to access only the other tuple. Formally, given two tuples t_i,$t_j{\in}r$, with $t_i{\sim}_{\texttt{A}}t_j$, tuple t_i *exposes* tuple t_j over attribute A to all users $u \in acl(t_i) \setminus acl(t_j)$. For instance, consider user B and the conflicting tuples $t_1{\sim}_{\texttt{Year}}t_2$ and $t_2{\sim}_{\texttt{Sales}}t_4$ in Figure 6(c). Tuple t_2 exposes to user B the value for attribute Year of tuple t_1 (2010) and the value for attribute Sales (700) of tuple t_4. Note that this last inference also allows B to infer the value for attribute Sales of tuple t_5 (by observing the common index present for C in tuples t_4 and t_5). Figure 7 illustrates the inferred view of user B over relation Shops after exploiting her knowledge on the index-plaintext value correspondences derived from the encrypted relation in Figure 5. Here, grey cells represent information that user B cannot access/infer, light grey cells represent information that user B has inferred from her knowledge, and white cells represent information that user B is authorized to access.

As discussed above, the inference channels previously described can be blocked by imposing conflicting tuples to be associated with different index values. An index function that provides this guarantee is called *safe* and is defined as follows.

Definition 4.2 (Safe index function). *Let r be a plaintext relation over relation schema $R(\texttt{A}_1,\ldots,\texttt{A}_n)$. An index function ι is* safe *iff $\forall t_i$,$t_j{\in}r$ s.t. $t_i{\sim}_{\texttt{A}}t_j$, with $\texttt{A}{\in}R$, and $\forall u{\in}acl(t_i){\cap}acl(t_j)$, $\iota_u(t_i[\texttt{A}]) \neq \iota_u(t_j[\texttt{A}])$.*

According to Definition 4.2, a safe index function ensures that conflicting tuples have different index values for all users that can access them. In this way, the index values cannot be exploited anymore for inference purposes.

In the following section, we will present an approach for defining a safe index function.

5. DEFINING A SAFE INDEX

To efficiently compute a safe index function, we start by classifying tuples based on conflicts and impose diversity of the indexes. To this end, we partition tuples in clusters such that tuples that are in conflict for at least one attribute belong to different clusters. A tuple partitioning satisfying such condition is said to be *safe*. We formally define a *safe partitioning* of tuples in r with respect to an arbitrary set $\mathcal{A}$ of attributes in R as follows.

Shops

	Id	City	Year	Sales
t_1	▨	▨	2010	▨
t_2	002	Rome	2010	700
t_3	003	Rome	2011	600
t_4	▨	▨	▨	700
t_5	▨	▨	▨	700

	acl
t_1	A
t_2	A,B
t_3	B
t_4	A,C
t_5	C

Figure 7: Inferred view of user B over relation Shops

Definition 5.1 (Safe partitioning). *Let r be a plaintext relation defined over relation schema $R(\texttt{A}_1,\ldots,\texttt{A}_n)$ and $\mathcal{A}$ be a set of attributes in R. A* safe partition *$\mathcal{C}_{\mathcal{A}}$ of r w.r.t. $\mathcal{A}$ is a set of classes of tuples $\{C_1,\ldots,C_p\}$ such that $\bigcup_{i=1}^{p} C_i = r$, $C_i \cap C_j = \emptyset$, with $i,j \in \{1,\ldots,p\}$ and $i \neq j$, and $\forall C{\in}\mathcal{C}_{\mathcal{A}}$, and $\forall t_i$,$t_j{\in}C$, and $\forall \texttt{A}{\in}\mathcal{A}$, $t_i{\not\sim}_{\texttt{A}}t_j$.*

A safe partitioning of r with respect to set $\mathcal{A}$ is composed of classes of tuples such that the tuples in a class are not in conflict over any attribute in $\mathcal{A}$. For instance, according to the example in Figure 6, a safe partition of Shops with respect to: attribute City is $\mathcal{C}_{\texttt{City}} = \{\{t_1,t_3\}, \{t_2,t_4,t_5\}\}$; attribute Year is $\mathcal{C}_{\texttt{Year}} = \{\{t_1,t_3,t_4\}, \{t_2,t_5\}\}$; attribute Sales is $\mathcal{C}_{\texttt{Sales}} = \{\{t_1,t_2,t_3,t_5\}, \{t_4\}\}$; set {City,Year,Sales} of attributes is $\mathcal{C}_{\texttt{City,Year,Sales}} = \{\{t_1,t_3,t_5\}, \{t_2\}, \{t_4\}\}$. Note that tuples that cannot belong to the same class in $\mathcal{C}_{\texttt{City}}$, $\mathcal{C}_{\texttt{Year}}$, or $\mathcal{C}_{\texttt{Sales}}$ cannot belong to the same class in $\mathcal{C}_{\texttt{City,Year,Sales}}$. For instance, t_1 and t_4 cannot belong to the same class in $\mathcal{C}_{\texttt{City}}$, since $t_1{\sim}_{\texttt{City}}t_4$. Then, t_1 and t_4 must belong to different classes in $\mathcal{C}_{\texttt{City,Year,Sales}}$.

Intuitively, a safe partitioning induces a safe indexing for attributes in $\mathcal{A}$ if different indexing computations (which we differentiate by applying salts) are used for tuples in different classes in $\mathcal{C}_{\mathcal{A}}$. While any safe partitioning allows us to define a safe index function, we are interested in computing a minimal safe partition of r with respect to the attribute(s) of interest. The reason for this is to minimize the different index functions (salts) that need to be supported, especially since salts lead to an increase in the size of the query generated by the user. The problem of computing a minimal safe partitioning can be formally defined as follows.

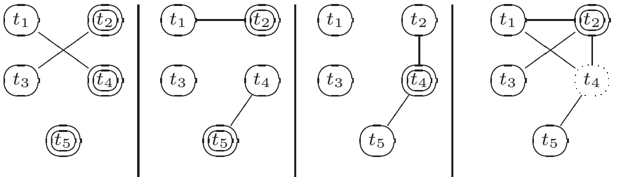
Problem 5.1 (Min-Partition). *Given a plaintext relation r defined over relation schema $R(\texttt{A}_1,\ldots,\texttt{A}_n)$ and a set $\mathcal{A}$ of attributes in R, determine a safe partition $\mathcal{C}_{\mathcal{A}}$ of r w.r.t. $\mathcal{A}$ (Definition 5.1) such that the number of classes in $\mathcal{C}_{\mathcal{A}}$ is minimized.*

The Min-Partition problem is NP-hard, as formally proved by the following theorem.

Theorem 5.1. *The* Min-Partition *problem is NP-hard.*

Proof. The proof is a reduction from the NP-hard problem of *minimum vertex coloring* [13], which can be formulated as follows: *given a graph $G(V,E)$, determine a minimum coloring of G, that is, assign to each vertex in V a color such that adjacent vertices have different colors, and the number of colors is minimized.*

The correspondence between the Min-Partition problem and the minimum vertex coloring problem can be defined as follows. The outsourced relation r is defined over schema



(a) $G_{\mathtt{City}}$ (b) $G_{\mathtt{Year}}$ (c) $G_{\mathtt{Sales}}$ (d) $G_{\mathtt{City,Year,Sales}}$

Figure 8: Conflict graphs with respect to City (a), Year (b), Sales (c), and {City,Year,Sales} (d)

$R(\mathtt{A})$. Any vertex v in G corresponds to a tuple $t{\in}r$. Any edge (v_i, v_j) in G corresponds to a conflict between t_i and t_j with respect to A, that is, $t_i[\mathtt{A}]{=}t_j[\mathtt{A}]$, $acl(t_i){\neq}acl(t_j)$, and $acl(t_i){\cap}acl(t_j){\neq}\emptyset$. The definition of a safe partition $\mathcal{C}_{\mathtt{A}}$ of r such that the number of classes in $\mathcal{C}_{\mathtt{A}}$ is minimized represents a solution to the minimum vertex coloring problem. Specifically, the color assigned to a vertex v in G corresponds to the class including the tuple t represented by vertex v. Therefore, any algorithm that solves the MIN-PARTITION problem can be used to solve the minimum vertex coloring problem. □

We solve the MIN-PARTITION problem by translating it into an instance of the minimum vertex coloring problem [13], and take advantage of the heuristics and approximation algorithms proposed for solving it, which efficiently compute a solution near to the optimum. We therefore define a *conflict graph* that models conflicts between tuples in r with respect to a set $\mathcal{A}$ of attributes as follows.

DEFINITION 5.2 (CONFLICT GRAPH). *Let r be a plaintext relation defined over relation schema $R(\mathtt{A}_1, \ldots, \mathtt{A}_n)$ and $\mathcal{A}$ be a set of attributes in R. The* conflict graph $G_{\mathcal{A}}$ *of r w.r.t. $\mathcal{A}$ is a non-directed graph $G_{\mathcal{A}}(V_{\mathcal{A}}, E_{\mathcal{A}})$ where $V_{\mathcal{A}}{=}r$ and $E_{\mathcal{A}}{=}\{(t_i, t_j) : t_i, t_j \in V_{\mathcal{A}}$, and $\exists\mathtt{A}{\in}\mathcal{A}$, $t_i{\sim}_{\mathtt{A}}t_j\}$.*

According to Definition 5.2 a conflict graph has a vertex for each tuple $t{\in}r$, and an edge between t_i and t_j iff there exists at least an attribute A in $\mathcal{A}$ such that $t_i{\sim}_{\mathtt{A}}t_j$. For instance, according to the example in Figure 6, Figures 8(a)-(c) illustrate the conflict graphs for attributes City, Year, and Sales, respectively. A safe partition $\mathcal{C}_{\mathcal{A}}$ minimizing the number of classes then corresponds to a minimum vertex coloring of conflict graph $G_{\mathcal{A}}$, where colors represent classes. The coloring of the conflict graphs in Figures 8(a)-(c), denoted by drawing vertices with different lines (i.e., solid, double, and dotted), correspond to safe partitions: $\mathcal{C}_{\mathtt{City}} = \{\{t_1,t_3\}, \{t_2,t_4,t_5\}\}$, $\mathcal{C}_{\mathtt{Year}} = \{\{t_1,t_3,t_4\}, \{t_2,t_5\}\}$, and $\mathcal{C}_{\mathtt{Sales}} = \{\{t_1,t_2,t_3,t_5\}, \{t_4\}\}$, respectively.

The conflict graph $G_{\mathcal{A}}(V_{\mathcal{A}}, E_{\mathcal{A}})$ of r with respect to a set $\mathcal{A}{=}\{\mathtt{A}_1, \ldots, \mathtt{A}_l\}$ of attributes corresponds to the composition of the conflict graphs $G_{\mathtt{A}_1}, \ldots, G_{\mathtt{A}_l}$ of r with respect to attributes $\mathtt{A}_i$ in $\mathcal{A}$, $i = 1, \ldots, l$. In fact, all these conflict graphs are defined over the same set of vertices, and edge $(t_i,t_j){\in}E_{\mathcal{A}}$ iff there exists at least an attribute A in $\mathcal{A}$ such that $t_i{\sim}_{\mathtt{A}}t_j$. For instance, according to the example in Figure 6, the conflict graph with respect to the set {City,Year,Sales} of attributes reported in Figure 8(d) is the composition of the conflict graphs in Figures 8(a)-(c). A coloring for $G_{\mathcal{A}}$ then represents a coloring also for each $G_{\mathtt{A}_i}$, with $\mathtt{A}_i{\in}\mathcal{A}$, $i = 1, \ldots, l$. However, a minimal coloring for $G_{\mathcal{A}}$ may not represent a minimal coloring for $G_{\mathtt{A}_i}$. For instance, the minimal coloring for the graph in Figure 8(d) requires 3 colors while the minimal colorings for the conflict graphs in Figures 8(a)-(c) require only 2 colors each.

	City			Year			Sales		
	A	B	C	A	B	C	A	B	C
t_1	s_A			s_A			s_A		
t_2	s'_A	s_B		s'_A	s_B		s_A	s_B	
t_3		s'_B			s'_B			s_B	
t_4	s'_A		s_C	s_A		s_C	s'_A		s_C
t_5			s_C			s'_C			s'_C

(a)

	City			Year			Sales		
	A	B	C	A	B	C	A	B	C
t_1	s_A			s_A			s_A		
t_2	s'_A	s_B		s'_A	s_B		s'_A	s_B	
t_3		s'_B			s'_B			s'_B	
t_4	s''_A		s_C	s''_A		s_C	s''_A		s_C
t_5			s'_C			s'_C			s'_C

(b)

Figure 9: Salt assignments for the attribute level (a) and the relation level (b) approaches

Once partition $\mathcal{C}_{\mathcal{A}}$ has been computed, the data owner defines a safe index function ι_u for each user u in the system in a way that the indexing computations for the tuples in different classes are differentiated by applying different *randomly generated salts* in the use of index function ι_u. More precisely, we introduce a *salt assignment function* for each attribute and user in $acl(t)$, driven by the partition $\mathcal{C}_{\mathcal{A}}$ of r, which assigns different salts to tuples in different classes . Formally, the salt assignment function of user u for attribute A is defined as follows.

DEFINITION 5.3 (SALT ASSIGNMENT FUNCTION). *Let r be a plaintext relation over relation schema $R(\mathtt{A}_1, \ldots, \mathtt{A}_n)$, A be an attribute in R, $\mathcal{C}_{\mathcal{A}}$ be a safe partition of r w.r.t. a set $\mathcal{A}$ of attributes in R such that $\mathtt{A}{\in}\mathcal{A}$ (Definition 5.1), $\mathcal{S}$ be a set of random salts, u be a user, and $T_u = \{t \in r : u \in acl(t)\}$. The* salt assignment function *$\sigma_{\mathtt{A},u} : T_u \rightarrow \mathcal{S}$ of u for attribute A implied by $\mathcal{C}_{\mathcal{A}}$ associates a salt $s{\in}\mathcal{S}$ with each $t{\in}T_u$ such that $\forall t_i,t_j{\in}T_u$, $\sigma_{\mathtt{A},u}(t_i){=}\sigma_{\mathtt{A},u}(t_j)$ iff t_i and t_j belong to the same class $C{\in}\mathcal{C}_{\mathcal{A}}$.*

The salt assignment function computes, for each attribute A and user u, the set of salts to be used in computing u's indexes for attribute A. Intuitively, each user will need, for each attribute, as many salts as the number of classes C in $\mathcal{C}_{\mathcal{A}}$ that contain tuples readable by the user. For instance, according to the partitioning induced by the coloring in Figure 8(a), the salt assignment function for attribute City will produce: two salts for user A (one for t_1, and a distinct one for t_2 and t_4), two salts for user B (one for t_2 and a distinct one for t_3) and one salt for user C (for t_4 and t_5). The salt assignment function for attributes Year and Sales is defined analogously. Figure 9(a) summarizes the salts assigned to the tuples in relation SHOPS in Figure 1(a) according to the partition induced by the colorings in Figures 8(a)-(c) for attributes City, Year, and Sales, respectively. The table has a row for each tuple in relation SHOPS, and three columns representing users A, B, and C for each indexed attribute City, Year, and Sales. A cell in the table contains $\sigma_{\mathtt{A},u}(t_i)$, with $\mathtt{A} \in$ {City,Year,Sales}, $u \in \{A, B, C\}$, and t_i the i-th row in the table corresponding to tuple t_i in SHOPS. Given a salt assignment function $\sigma_{\mathtt{A},u}$, the index value of user u for attribute A in tuple t is then defined as $\iota_u(t[\mathtt{A}], \sigma_{\mathtt{A},u}(t))$. Figure 10 illustrates the encrypted version of relation SHOPS in Figure 1(a), where indexes have been computed adopting the salt assignment functions reported in Figure 9(a). We note that the salts used by salt assignment function $\sigma_{\mathtt{A},u}$ are generated through a pseudo-random function, different for each u. The data owner has then to communicate to each user u only the pseudo-random function used by $\sigma_{\mathtt{A},u}$ and the number of salts needed for each attribute A.

SHOPS^e

tid	etuple	$\mathtt{I}_c$	$\mathtt{I}_y$	$\mathtt{I}_s$
1	α	$\iota_A(\text{NY},s_A)$	$\iota_A(2010,s_A)$	$\iota_A(600,s_A)$
2	β	$\iota_A(\text{Rome},s'_A)\iota_B(\text{Rome},s_B)$	$\iota_A(2010,s'_A)\iota_B(2010,s_B)$	$\iota_A(700,s_A)\iota_B(700,s_B)$
3	γ	$\iota_B(\text{Rome},s'_B)$	$\iota_B(2011,s'_B)$	$\iota_B(600,s_B)$
4	δ	$\iota_A(\text{NY},s'_A)\iota_C(\text{NY},s_C)$	$\iota_A(2011,s_A)\iota_C(2011,s_C)$	$\iota_A(700,s'_A)\iota_C(700,s_C)$
5	ϵ	$\iota_C(\text{Oslo},s_C)$	$\iota_C(2011,s'_C)$	$\iota_C(700,s'_C)$

Figure 10: An example of encrypted relation with user-based indexes and salts

The data owner can apply different strategies when defining safe indexes for r: *i)* she can define a different partition of r with respect to each attribute $\mathtt{A}$ (*attribute level approach*), separately solving the minimal coloring problem for each attribute, or *ii)* she can define a unique partition applied to all the indexed attributes in R (*relation level approach*), essentially identifying the minimal vertex coloring for the conflict graph defined with respect to the set of all indexed attributes. With reference to the example in Figure 6, Figures 9(a)-(b) illustrate the salt assignment functions obtained following the attribute level approach and the relation level approach, respectively. The salt assignment in Figure 9(b) has been defined, for all the indexed attributes in relation SHOPS, according to the partition induced by the coloring of the conflict graph in Figure 8(d), which produces three salts for user A (one for t_1, a distinct one for t_2, and another distinct one for t_4), two salts for user B (one for t_2 and a distinct one for t_3) and two salts for user C (one for t_4 and a distinct one for t_5). Since the coloring is unique for all the attributes, a single salt can be assigned to each user to protect all the indexes of a given tuple. Each user will then be communicated a single number of salts to be used when querying any attribute.

Both attribute level approach and relation level approach present advantages and disadvantages. In the attribute level approach, we expect to obtain a lower number of classes than the number of classes resulting in the relation level approach, as confirmed by the example. The number of salts for each u should then be lower, as well as the user overhead in query generation. The attribute level approach causes however a higher initialization overhead for the data owner, who needs to compute as many partitions of r as the number of attributes over which an index is defined. When this additional initialization cost is not justified by a considerable reduction in the number of salts used in the generation of queries (see Section 6), the data owner may prefer to apply a relation level approach (see Section 7).

6. QUERY EVALUATION

With our indexing technique, multiple index values (one for each authorized user) may be defined for the same index attribute of an encrypted relation. The representation of sets of values, while not directly supported in the relational data model, can be readily realized at the logical level by fragmenting the encrypted relation and reporting indexes in separate tables. The original encrypted relation can be simply obtained (by the server) by joining the different tables. The data owner therefore outsources an encrypted relation r^e_t and a set $r^e_{\mathtt{I}_1},\ldots,r^e_{\mathtt{I}_l}$ of index relations. Encrypted relation r^e_t is defined over schema R^e_t(<u>tid</u>, etuple), while each index relation $r^e_{\mathtt{I}}$ is defined over schema $R^e_{\mathtt{I}}$(<u>tid</u>, <u>$\mathtt{I}$</u>). Attribute *tid* in relation $R^e_{\mathtt{I}}$ references attribute *tid* in R^e_t and represents the identifier of the tuple to which the index value stored in $\mathtt{I}$ refers. As an example, Figure 11 illustrates the logical realization of the relation in Figure 10.

SHOPS^e_t

tid	etuple
1	α
2	β
3	γ
4	δ
5	ϵ

(a)

$R^e_{\mathtt{I}_c}$

tid	$\mathtt{I}_c$
1	$\iota_A(\text{NY},s_A)$
2	$\iota_A(\text{Rome},s'_A)$
2	$\iota_B(\text{Rome},s_B)$
3	$\iota_B(\text{Rome},s'_B)$
4	$\iota_A(\text{NY},s'_A)$
4	$\iota_C(\text{NY},s_C)$
5	$\iota_C(\text{Oslo},s_C)$

(b)

$R^e_{\mathtt{I}_y}$

tid	$\mathtt{I}_y$
1	$\iota_A(2010,s_A)$
2	$\iota_A(2010,s'_A)$
2	$\iota_B(2010,s_B)$
3	$\iota_B(2011,s'_B)$
4	$\iota_A(2011,s_A)$
4	$\iota_C(2011,s_C)$
5	$\iota_C(2011,s'_C)$

(c)

$R^e_{\mathtt{I}_s}$

tid	$\mathtt{I}_s$
1	$\iota_A(600,s_A)$
2	$\iota_A(700,s_A)$
2	$\iota_B(700,s_B)$
3	$\iota_B(600,s_B)$
4	$\iota_A(700,s'_A)$
4	$\iota_C(700,s_C)$
5	$\iota_C(700,s'_C)$

(d)

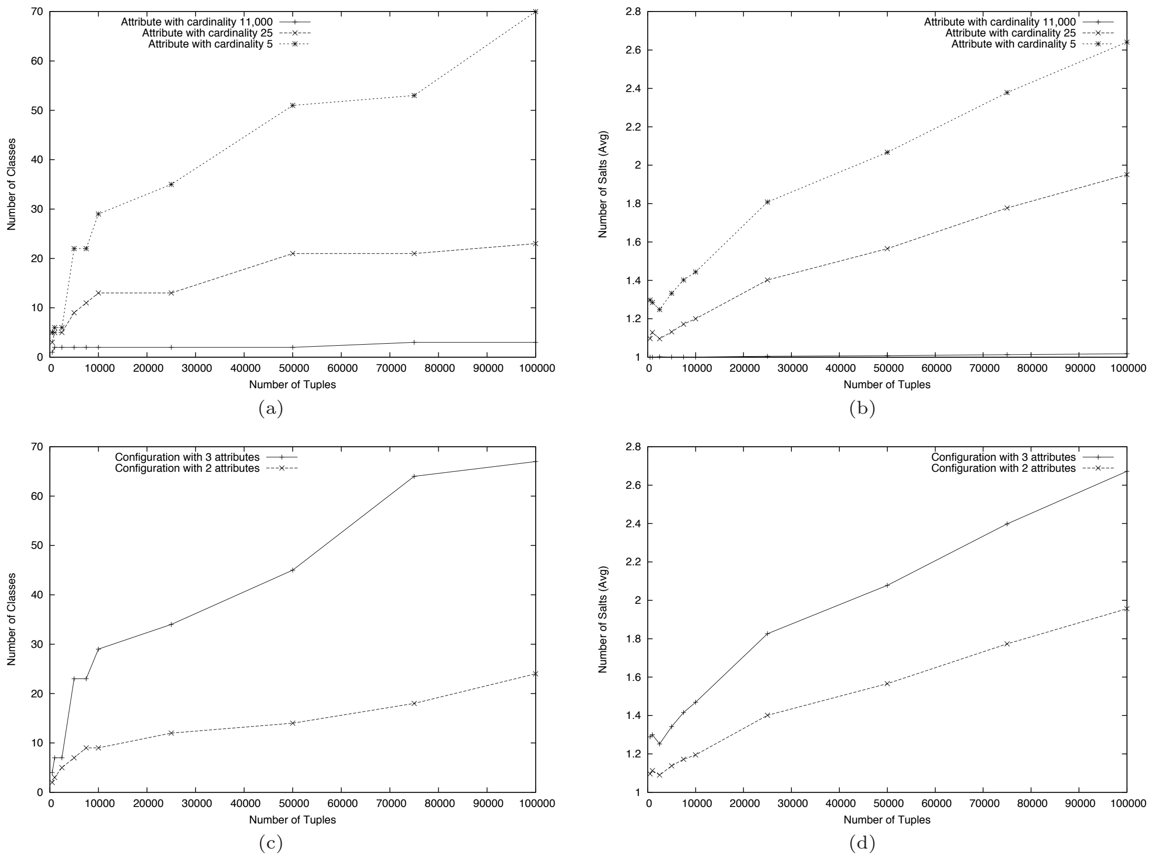
Figure 11: An example of encrypted (a) and index (b-d) relations for the encrypted relation in Figure 10

At the client side, each user u has knowledge of: *i)* the index function ι_u mapping plaintext values to indexes; *ii)* the maximum number of salts $n_{\mathtt{A},u}$ used for mapping values of attribute $\mathtt{A}$ that are accessible by u; and *iii)* the pseudorandom function used by the data owner to generate these salts. Each basic condition $\mathtt{A}=v$ in a query q submitted by user u is translated (client-side) into an equivalent condition $\mathtt{I}_\mathtt{A}$ IN V, where $\mathtt{I}_\mathtt{A}$ is the index over attribute $\mathtt{A}$ and V is the set of index values corresponding to v that user u can compute. The set V of values in the condition is obtained by applying index function ι_u to v combined with each of the $n_{\mathtt{A},u}$ salts generated by the user through the pseudo-random function shared with the data owner, that is, $\iota_u(v,s_i)$, $i = 1,\ldots,n_{\mathtt{A},u}$. For instance, consider the outsourced relations in Figure 11, representing plaintext relation SHOPS in Figure 1(a), and query q = "SELECT $\mathtt{City}$, $\mathtt{Sales}$ FROM SHOPS WHERE $\mathtt{Year}$=2010" submitted by user B. Since $n_{\mathtt{Year},B}$=2, user B computes the two index values, $\iota_B(2010,s_B)$ and $\iota_B(2010,s'_B)$, that possibly represent value 2010 in the encrypted table, and translates her query as follows: "SELECT $\mathtt{etuple}$ FROM SHOPS^e_t JOIN $R^e_{\mathtt{I}_y}$ ON SHOPS^e_t.$\mathtt{tid}$=$R^e_{\mathtt{I}_y}$.$\mathtt{tid}$ WHERE $\mathtt{I}_y$ IN $\{\iota_B(2010,s_B),\iota_B(2010,s'_B)\}$". When the requesting user receives the result of her query from the cloud server, she decrypts each tuple and possibly projects the attributes of interest ($\mathtt{City}$ and $\mathtt{Sales}$, in our example).

Queries submitted by users can be efficiently evaluated at the server-side. In fact, the cloud server can resort to traditional optimization techniques for the execution of the join operations between the encrypted (R^e_t) and index relations ($R^e_{\mathtt{I}}$), which therefore do not impact performance. Also, to guarantee an efficient retrieval of the tuples matching the user condition, the cloud server can define a relational physical index over attribute $\mathtt{I}$ for each index relation $R^e_{\mathtt{I}}$. The physical index could be based on a hash (preferably) or B-tree structure, depending on what is offered by the relational engine used on the server.

7. EXPERIMENTAL RESULTS

To evaluate the behavior of the indexing technique presented in previous sections, we implemented a prototype and executed a series of experiments. For the experiments we had to generate; *i)* the data (relation) to be outsourced and *ii)* the authorizations holding on the different tuples of the relation. As for the data, to guarantee the value distribution within the attributes to mimic a real database, we generated a relational table built following the TPC-H benchmark [23] specifications (TPC-H is commonly used in the database community to evaluate the performance of new



(a)

(b)

(c)

(d)

Figure 12: Number of classes in the partition (a,c) and average number of salts per user (b,d) in the attribute level (a,b) and relation level (c,d) approaches

solutions). We considered a relation of three attributes, including 5, 25, and 11,000 distinct values, respectively, and performed experiments with table cardinality varying between 500 and 100,000 tuples. As for the authorizations, since no standard approach exists in the security community to build the access privileges of a large scale community of users over a set of resources, we built a collection of access profiles by extracting the authorship information from the DBLP repository [22], which is representative of large social networking contexts. The idea is that each paper in DBLP, which we made correspond to one tuple in the relation generated as explained above,[1] must be accessible by all its authors. We implemented a C++ program that starts from a random author and considers all his/her publications and coauthors; then, one of the coauthors is randomly chosen and his/her publications and corresponding coauthors are iteratively retrieved. The program stops when the target number of publications has been retrieved (i.e., when all the tuples in the table are associated with an acl).

The parameter we were interested in evaluating for assessing the performance of the system is the number of salts per user. Indeed, user side computation is considered the most valuable resource and query translation requires an overhead that directly depends on the number of salts that the user needs to use in query translation (see Section 6). The time necessary to compute a correct coloring of the conflict graphs (see Section 5) does not represent a critical aspect for the proposed approach. In fact, this cost has to be paid only at set up time, when the relation is first outsourced to the cloud server. Analogously, the additional storage necessary for the index is not critical, since storage handled by the external server is inexpensive compared to other costs, like network transfer and client-side storage and computation.

We performed experiments evaluating the number of classes composing a safe partition of the outsourced relation, necessary to define a set of safe index functions, and the average number of salts that users need to translate their queries. Intuitively, the number of classes in the partition represents the upper bound to the number of salts required, while the average number of salts per user estimates the user overhead in query translation. In our experiments, we considered two different configurations, depending on whether the data owner defines a different partition for each attribute in the relation (*attribute level approach*), or a unique partition for all the indexed attributes in the relation (*relation level approach*).

Attribute level approach. Figure 12(a) reports the number of classes in the partition $\mathcal{C}_{\mathtt{A}}$ w.r.t. each of the attributes in the outsourced relation, when the number of tuples varies

[1] We did not use the data from the DBLP repository itself to generate our relation (opting instead for the generally used TPC-H benchmark), since it would have produced only attributes either with limited domain cardinalities or with distinct values for every tuple.

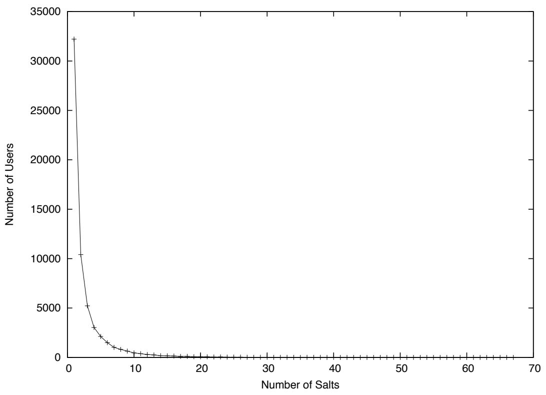


Figure 13: Number of salts assigned to users

from 500 to 100,000. The number of classes exhibits a range from 1 to 3 for the attribute with a domain of 11,000 distinct values, from 3 to 23 for the attribute with a domain of 25 distinct values, and from 5 to 70 for the attribute with a domain of 5 distinct values. Figure 12(b) illustrates, for the same configurations, the average number of salts that each user needs to use when translating her queries. The results confirm that the major contribution derives from the attribute with lowest domain cardinality. Also, it shows that the average number of salts per user is in the range 1.2-2.7, corresponding to a limited overhead in query execution.

Relation level approach. Figure 12(c) compares the number of classes in the partition $\mathcal{C}_\mathcal{A}$ of the outsourced relation when $\mathcal{A}$ includes: *i)* all the three attributes in the outsourced relation, and *ii)* two of the attributes in the outsourced relation (with domain of 11,000 and 25 values, that is, excluding the least selective). The number of classes in $\mathcal{C}_\mathcal{A}$ exhibits a range from 5 to 67 if $\mathcal{A}$ includes all the attributes in the relation, and from 2 to 24 if $\mathcal{A}$ includes only the two attributes stated above. The configuration with three attributes in $\mathcal{A}$ is characterized by a higher number of conflicts (i.e., edges in the conflict graph), mainly due to the attribute with the least cardinality. Figure 12(d) illustrates, for the same configurations, the average number of salts that each user needs to use in query translation. It is easy to see that, also when the data owner defines a unique partition $\mathcal{C}_\mathcal{A}$ for all the indexed attributes in the relation, the average number of salts per user is low, in the range 1.2-2.7 already observed for the attribute with the lowest domain cardinality. When using only two attributes, the average number of salts per user is even lower, in the range 1.1-2.0. The overhead in query execution remains limited.

Figure 13 depicts the distribution of salts for the different users, focusing on the most complex configuration, the one with 100,000 tuples and $\mathcal{A}$ including all the three attributes. An analysis of the data reported in the graph shows that less than 0.1% of the users need to use a number of salts greater than half the number of classes in the partition, while more than half the users just need one salt. It is clear that the number of classes corresponds to a relatively high number of salts for an extremely small fraction of users, so small that the impact on the overall performance is negligible.

The comparison between the results in Figure 12(b) and in Figure 12(d) allows us to better characterize the attribute vs the relation level approaches. The definition of a different partition for each attribute permits to reduce the overhead of the queries that specify restrictions on the most selective attributes, whereas the difference for less selective attributes is minimal. This approach requires a more extensive characterization of the schema, because a different value of number of salts has to be recorded for every attribute, whereas a single value has to be stored at the table level for each user if a unique partition is defined for all the indexed attributes. If the query profile exhibits a strong preference for queries over selective attributes and there is no worry about the slightly larger schema management costs, the definition of multiple partitions (one for each attribute) should be preferred. Otherwise, the definition of a unique partition should be preferred, due to its simplicity and limited overhead that is near to the optimum for the least selective attribute and still acceptable for the other attributes.

8. RELATED WORK

Several research efforts have been performed in the context of data outsourcing [20]. Most proposals have addressed the problem of efficiently performing queries directly on outsourced encrypted data, without decrypting sensitive information at the server side. Such proposals typically define indexes that are stored together with the encrypted data and are used by the storage server to select the data to be returned in response to a query [10]. A simple indexing technique consists in using as index the result of an encryption function (direct index) over plaintext values (e.g., [7]). While simple, this solution suffers from frequency-based attacks since the frequency distribution of index and plaintext values is exactly the same. In [7, 15] the authors propose bucket-based index techniques. The index values are computed by applying a hash function over the corresponding plaintext values [7] or by first partitioning the domain of an attribute into a set of non-overlapping subsets of contiguous values, which are usually of the same size, each of which is associated with a label [15]. The index value associated with a given plaintext value corresponds then with the label of the partition containing the plaintext value. These index techniques map multiple plaintext values to the same index value, thus generating collisions that prevent frequency-based attacks. Although effective for delegating to the cloud server the evaluation of equality conditions, direct and bucket-based indexing techniques do not support range queries. In [7], the authors present a $B+$-tree index that supports both equality and range queries (as well as GROUP BY and ORDER BY SQL clauses) at the server side. This approach is based on outsourcing an encrypted version of a $B+$-tree built over a plaintext attribute of the outsourced relation, which is iteratively visited by the user for query evaluation. An alternative indexing technique that supports equality and range conditions (as well as MAX, MIN, and COUNT aggregation functions) over encrypted data has been illustrated in [1] and is based on an order preserving encryption schema. In [24] the authors propose a flattened indexing method that exploits B-trees for supporting both equality and range queries, while reducing inference exposure thanks to an almost flat distribution of the frequencies of index values. All these solutions provide support for the evaluation of queries over encrypted data, but do not take access control restrictions into account. As a consequence, their observation may disclose confidential data.

In [21], the authors introduce the definition of B-tree indexes, one for each different access control list in the system. A B-tree index then supports access to all and only the tuples whose acl corresponds to the acl of the index. To guarantee confidentiality, the B-tree index of each acl is encrypted with a key that only the users in the acl know. This solution suffers from a high client side overhead in query evaluation, since the user submitting the query must visit all the B-trees associated with the acls to which she belongs.

Another related line of work addresses the problem of specifying and enforcing an access control policy on outsourced data, without the need for the data owner to filter query results. In [18] the authors present a framework for enforcing access control on published XML documents by using different cryptographic keys over different portions of the XML tree and by introducing special metadata nodes in the structure. The solution illustrated in [8], which can be adopted independently from the logical organization of the outsourced data, is based on a selective encryption strategy for enforcing the authorization policy specified by the data owner and relies on key derivation to guarantee that each user knows one secret key only and that each tuple is encrypted with one key only. This approach also permits to delegate to the cloud server the management of policy changes. The proposal introduced in [26] adopts attribute based encryption to provide system scalability. Each tuple is encrypted using the public key components of the attributes describing the context in which the tuple should be accessed. Each user is assigned a secret key that satisfies the logical expression over the system attributes representing the properties of the tuples she is authorized to access. All these proposals address the problem of enforcing authorizations and ensuring that users can decrypt only data that they are authorized to access. Our work is complementary to these proposals since it addresses the problem of ensuring that users cannot withdraw any inferences on the data they are not authorized to access (and hence not able to decrypt) by observing indexes associated with the data.

9. CONCLUSIONS

We have presented an approach for indexing encrypted data to be made accessible selectively. Our indexes are safe from inferences, meaning that they do not leak information on underlying data to users not authorized to access such data. Experimental results show that our solution enjoys limited overhead, ensuring light work on the client and network bandwidth at the cost of additional storage on the server. This choice fits well with cloud systems, where clients have limited resources, network communication costs are comparatively high, and storage costs are quite low. Our approach closes the gap between indexing techniques and selective encryption, therefore allowing their combined use and enabling efficient query execution while ensuring that indexes do not expose information to unauthorized users.

We note that while in the paper we presented our solution for the worst case scenario of direct indexes, our approach is readily applicable to other indexing techniques, such as flattened and bucket-based indexes, simply by applying it at the flat group or bucket granularity. Possible future work includes the consideration of protection against threats to index confidentiality due to the observation, by the server, of multiple queries and the possible collusion between server and users. A promising and directly applicable approach against this threat consists in enriching the technique presented in this paper with tuple clustering approaches, such as those illustrated in [9], associating indexes at the level of clusters in contrast to individual tuples.

10. ACKNOWLEDGEMENTS

This work was partially supported by the EC within the 7FP, under grant agreement 257129 (PoSecCo), by the Italian Ministry of Research within the PRIN 2008 project "PEPPER" (2008SY2PH4), and by the Università degli Studi di Milano within the project "PREVIOUS". The work of Sushil Jajodia was partially supported by the National Science Foundation under grants CCF-1037987 and CT-20013A.

11. REFERENCES

[1] R. Agrawal, J. Kierman, R. Srikant, and Y. Xu. Order preserving encryption for numeric data. In *Proc. of SIGMOD 2004*, Paris, France, June 2004.

[2] M. Atallah, M. Blanton, N. Fazio, and K. Frikken. Dynamic and efficient key management for access hierarchies. *ACM TISSEC*, 12(3):18:1–18:43, January 2009.

[3] A. Azzini, S. Marrara, R. Sassi, and F. Scotti. A fuzzy approach to multimodal biometric continuous authentication. *Fuzzy Optimization and Decision Making*, 7(3):243–256, September 2008.

[4] M. Barni et al. Privacy-preserving fingercode authentication. In *Proc. of MM&Sec 2010*, Rome, Italy, September 2010.

[5] 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 TISSEC*, 8(1):119–152, February 2005.

[6] S. Cimato, M. Gamassi, V. Piuri, R. Sassi, and F. Scotti. Privacy-aware biometrics: Design and implementation of a multimodal verification system. In *Proc. of ACSAC 2008*, Anaheim, CA, USA, December 2008.

[7] 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 CCS 2003*, Washington, DC, USA, October 2003.

[8] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Encryption policies for regulating access to outsourced data. *ACM TODS*, 35(2):12:1–12:46, April 2010.

[9] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Fragments and loose associations: Respecting privacy in data publishing. *Proc. of the VLDB Endowment*, 3(1):1370–1381, September 2010.

[10] S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, and P. Samarati. Privacy of outsourced data. In A. Acquisti, S. Gritzalis, C. Lambrinoudakis, and S. De Capitani di Vimercati, editors, *Digital Privacy: Theory, Technologies and Practices*. Auerbach Publications (Taylor and Francis Group), 2007.

[11] M. Gamassi, M. Lazzaroni, M. Misino, V. Piuri, D. Sana, and F. Scotti. Accuracy and performance of

biometric systems. In *Proc. of IMTC 2004*, Como, Italy, May 2004.

[12] M. Gamassi, V. Piuri, D. Sana, and F. Scotti. Robust fingerprint detection for access control. In *Proc. of RoboCare 2005*, Rome, Italy, May 2005.

[13] M. R. Garey and D. S. Johnson. *Computers and Intractability*. Freeman and Company, 1979.

[14] H. Hacigümüs, B. Iyer, and S. Mehrotra. Ensuring the integrity of encrypted databases in the database as a service model. In *Proc. of DBSec 2003*, Estes Park, CO, USA, August 2003.

[15] H. Hacigümüs, B. Iyer, S. Mehrotra, and C. Li. Executing SQL over encrypted data in the database-service-provider model. In *Proc. of SIGMOD 2002*, Madison, WI, USA, June 2002.

[16] Health Insurance Portability and Accountability Act. http://www.hhs.gov/ocr/privacy/.

[17] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structures for outsourced databases. In *Proc. of SIGMOD 2006*, Chicago, IL, USA, June 2006.

[18] G. Miklau and D. Suciu. Controlling access to published data using cryptography. In *Proc. of VLDB 2003*, Berlin, Germany, September 2003.

[19] Payment card industry (PCI) data security standard, September 2006. https://www.pcisecuritystandards.org/documents/pci_dss_v2.pdf.

[20] P. Samarati and S. De Capitani di Vimercati. Data protection in outsourcing scenarios: Issues and directions. In *Proc. of ASIACCS 2010*, Beijing, China, April 2010.

[21] E. Shmueli, R. Waisenberg, Y. Elovici, and E. Gudes. Designing secure indexes for encrypted databases. In *Proc. of DBSec 2005*, Storrs, CT, USA, August 2005.

[22] The DBLP computer science bibliography. http://dblp.uni-trier.de.

[23] The transaction processing performance council (TPC) benchmark H. http://www.tpc.org/tpch/.

[24] H. Wang and L. V. S. Lakshmanan. Efficient secure query evaluation over encrypted XML databases. In *Proc. of VLDB 2006*, Seoul, Korea, September 2006.

[25] M. Xie, H. Wang, J. Yin, and X. Meng. Integrity auditing of outsourced data. In *Proc. of VLDB 2007*, Vienna, Austria, September 2007.

[26] S. Yu, C. Wang, K. Ren, and W. Lou. Achieving secure, scalable, and fine-grained data access control in cloud computing. In *Proc. of INFOCOM 2010*, San Diego, CA, USA, March 2010.