 Electronic Notes in Theoretical Computer Science 168 (2007) 127–142 

[www.elsevier.com/locate/entcs](http://www.elsevier.com/locate/entcs)

Selective Data Encryption in Outsourced Dynamic Environments

# E. Damiani[a](#_bookmark0)*,*[1](#_bookmark2) S. De Capitani di Vimercati[a](#_bookmark0)*,*[2](#_bookmark2) S. Foresti[a](#_bookmark0)*,*[3](#_bookmark2)

S. Jajodia[b](#_bookmark1)*,*[4](#_bookmark2) S. Paraboschi[c](#_bookmark3)*,*[5](#_bookmark2) P. Samarati[a](#_bookmark0)*,*[6](#_bookmark2)

a *DTI - Universita` degli Studi di Milano, 26013 Crema - Italy*

b *George Mason University, Fairfax, VA 22030-4444*

c *DIGI - Universita` degli Studi di Bergamo, 24044 Dalmine - Italy*

Abstract

The amount of information held by organizations’ databases is increasing very quickly. A recently proposed solution to the problem of data management, which is becoming increasingly popular, is represented by database outsourcing. Several approaches have been presented to database outsourcing management, in- vestigating the application of data encryption together with indexing information to allow the execution of queries at the third party, without the need of decrypting the data. These proposals assume access control to be under the control of the data owner, who has to filter all the access requests to data.

In this paper, we put forward the idea of outsourcing also the access control enforcement at the third party. Our approach combines cryptography together with authorizations, thus enforcing access control via *selective encryption*. The paper describes authorizations management investigating their specification and representation as well as their enforcement in a dynamic scenario.

*Keywords:* encrypted databases, access control, key derivation, hierarchy, dynamic

# Introduction

Nowadays, databases hold a critical concentration of sensitive information and their volume is increasing very quickly. In such a scenario, *database outsourcing* is becom- ing 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

1 Email: [damiani@dti.unimi.it](mailto:damiani@dti.unimi.it)

2 Email: [decapita@dti.unimi.it](mailto:decapita@dti.unimi.it)

3 Corresponding author. Email: [foresti@dti.unimi.it](mailto:foresti@dti.unimi.it)

4 Email: [jajodia@gmu.edu](mailto:jajodia@gmu.edu)

5 Email: [parabosc@unibg.it](mailto:parabosc@unibg.it)

6 Email: [samarati@dti.unimi.it](mailto:samarati@dti.unimi.it)

1571-0661 © 2007 Elsevier B.V. Open access under [CC BY-NC-ND license.](http://creativecommons.org/licenses/by-nc-nd/3.0/)

doi:10.1016/j.entcs.2006.11.003

hosting (e.g., outsourcing provides significant cost savings and service benefits). Moreover, the data owner can concentrate her attention on her core business. As a consequence of this trend towards outsourcing, highly sensitive data are no more under the data owner’s control and their confidentiality and integrity may be put at risk. To preserve the confidentiality of the outsourced data, cryptographic tech- niques are usually adopted [[7](#_bookmark25)]. By encrypting the outsourced data, the client is guaranteed that she alone can access the data. However, the problem becomes how to guarantee a *selective retrieval* over encrypted information. To this purpose, different techniques have been proposed [[4](#_bookmark23),[6](#_bookmark26),[8](#_bookmark27),[9](#_bookmark28)] that associate indexes with the out- sourced data to enable the server to enforce queries without the need of accessing cleartext data.

Although the database outsourced scenario has been intensively studied in the last few years, the *access control* issue in such a scenario has never been considered. The existing access control mechanisms, designed for distributed applications, oper- ate on client-server architectures according to the basic assumption that the server is in charge of defining and enforcing access control. In our scenario this assump- tion is no more applicable, since the server does not know the access control policy defined by the data owner. Current proposals for querying encrypted outsourced databases assume that clients have complete access to the query result and therefore use a single key for encrypting the whole outsourced database. Access control can therefore be enforced only by involving the data owner, who has to filter out from the query result the tuples that a client cannot access. Obviously, this solution is too expensive and not applicable in a real-world scenario, which demands for selective access by different users or applications.

In this paper, we address the problem of enforcing access control by exploiting data encryption. The idea is then to use different encryption keys for different data as proposed, for example, for XML documents [[10](#_bookmark29)]. To access such encrypted data, users have to decrypt them by using the appropriate key. If different users know different keys, they have different access rights.

The remainder of this paper is organized as follows. Section [2](#_bookmark4) introduces the main concepts of access control and presents our selective encryption solution. Section [3](#_bookmark7) describes the algorithm we propose to efficiently enforce access control through selective encryption in the specific scenario. Section [4](#_bookmark12) illustrates how changes in the access control policies can be efficiently managed. Finally, Section [5](#_bookmark19) contains our conclusions.

# Scenario and Basic Definitions

Given a system with a set U of users and a set T of resources, we assume that the access control policies are represented via an access matrix A with |U | rows and |T | columns, where A[u,t] contains the operations that user u can perform on t. Since we consider the read operation only, each entry in the access matrix can simply assume two values: A[u,t]=1 if u can read t; 0 otherwise. Figure [1](#_bookmark5) represents an example of access matrix for a system with 6 tuples (t1.. . t6) and 4 users (*A*, *B* ,

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | t1 | t2 | t3 | t4 | t5 | t6 |
| A | 0 | 1 | 1 | 0 | 1 | 1 |
| B | 1 | 0 | 1 | 1 | 1 | 1 |
| C | 0 | 0 | 1 | 1 | 0 | 1 |
| D | 0 | 1 | 0 | 1 | 1 | 1 |

Fig. 1. An example of access matrix

*C* , and *D* ). Given an access matrix A, *acl*t denotes the access control list for tuple t, that is, the set of users that can access t; *cap*u denotes the capability list of user u, that is, the set of tuples she can access. For instance, with reference to Figure [1](#_bookmark5), *acl*t2 ={*A*,*D*} and *capB*={t1,t3,t4,t5,t6}.

In the considered scenario, the enforcement of access control policies cannot be delegated to the remote server, as it is not trusted for accessing neither database content nor access control policies. Consequently, the data owner has to be involved in the access control enforcement, unless the data themselves implement selective access. To this purpose, we propose to use *selective encryption* [[2](#_bookmark21),[10](#_bookmark29)] as a technique for enforcing selective access on encrypted data. Selective encryption consists of using different keys for encrypting data and communicating each user the correct *key ring* , such that she can access all and only the resources she is authorized to access.

Let K be the set of symmetric encryption keys used to protect data. We intro- duce two functions:

* + a *user key assignment φ* : U'→2K, which associates with each user u∈ U the set of keys k∈K in the user’s key ring;
  + a *resource key assignment λ* : K '→2T , which associates with each key k∈K the set of tuples t∈T encrypted with k.

Given a user key assignment function *φ*, a resource key assignment *λ*, and an access matrix A, the pair (*φ, λ*) is said to be *complete* with respect to A, denoted as (*φ, λ*) ⇒ A, if each user can decrypt all tuples she can access according to A. The pair (*φ, λ*) is said to be *sound* with respect to A, denoted as (*φ, λ*) ⇐ A, if no user can decrypt tuples that she cannot access according to A. The pair (*φ, λ*) is said to *correctly enforce* A, denoted as (*φ, λ*) ⇔ A, iff it is both sound and complete with respect to A.

A straightforward solution for adopting selective encryption in our scenario as- sociates a key with each tuple t and communicates to each user u the keys used to encrypt tuples in *cap*u . It is easy to see that this solution correctly enforces A, but it is too expensive to manage, due to the high number of keys each user has to keep. To overcome this problem, we propose to use a *key derivation method* that, given a key and a piece of information publicly available, derives another key in the system. Different key derivation methods have been proposed in the literature and they usually work on tree hierarchies [[11](#_bookmark30)] or DAGs [[1](#_bookmark20)]. In our scenario, it is natural to introduce the definition of a user hierarchy UH=(2|U|*,* ≤), where the domain of the hierarchy is the powerset of U and ≤ is the subset containment partial order relation. Graphically, the UH hierarchy can be represented as a graph, called *user*

*,,, , ,* ∅ **¸***¸¸¸¸¸¸¸¸*

*,,,*t *,*

*¸¸¸ ¸¸¸¸*

*,,,* 1 *¸¸*

*A ¸¸,,,*,, ¸*B* ,,,**¸¸¸¸***¸¸¸,,,,,, C,¸,¸,¸,¸, D ,,*

*,,,¸¸¸ ,,,¸,¸,¸¸¸¸,¸,¸,¸,¸,,*

*¸¸¸¸ ,,*

*,,,, ,,¸,,,, ,,,,,, ¸¸¸ ¸¸¸¸¸¸*

*¸¸,*

*AB,¸*

*AC ,¸¸*t2*¸*,, ¸*AD* ,,,*,¸*

*,, BC¸,, BD CD*

*,,¸¸¸¸*

*¸¸¸¸¸¸¸¸,¸,¸,,,*

*,,,,,¸¸*

*,,,,*

*,, ¸¸¸¸ ,,,,,¸,¸,,,¸¸¸ ¸¸,¸¸,,*

*¸¸,¸,*t*,*4

t3 ,*,,,*

t5*,,, ¸***¸** *,,*

,,¸ *ABC*,,*¸* ,, ¸*ABD¸* ,,,

*ACD*

,,¸ *BCD* ,,,

*¸¸¸¸¸¸¸¸¸ ,,,,,*

*¸¸¸ ¸*t6

*, ,,,*

,, ¸*ABCD* ,,,

Fig. 2. An example of UH

*graph*, with a vertex for each element in 2|U| and a path connecting *a* with *b* iff *b* ≤ *a*. A vertex v of the graph is therefore characterized by the set of users to which it corresponds. In the following, given a vertex v of the user graph, it will be used to denote the set of users that it represents. Figure [2](#_bookmark6) illustrates an example of UH hi- erarchy for U={*A*,*B*,*C*,*D*}, where the top element of the UH hierarchy is the vertex corresponding to the empty set. A value, called *level*, is associated with each vertex and corresponds to the cardinality of the set of users it represents. Also, each vertex v in the user graph is associated with a key kv and a path v1 → v2 → *...* → vn in the graph represents a *key derivation path*, meaning that starting from the key associated with vertex v1 and following the path, it is possible to derive the keys of all vertices vi*,* i = 2*,... , n,* in the path. For instance, path *AB* → *ABC* → *ABCD* in the UH hierarchy in Figure [2](#_bookmark6) means that key kABCD can be derived from key kABC that in turn can be derived from key kAB. According to the UH definition, *φ*(u)=kvi with vi={u} (i.e., the key ring of each user contains one key only); and t∈ *λ*(kv ) with *acl*t =v. Consequently, each user can derive all keys kv such that u∈v.

Here, it is important to highlight that key derivation methods working on trees are more convenient than key derivation methods working on DAGs because they better support dynamic scenarios where the access control policies may change. Moreover, DAG key derivation methods are based on complex mathematical theo- rems (e.g., modular exponentiation), which make the support of a dynamic scenario complex and not efficient. By contrast, methods proposed for tree hierarchies ex- ploit simple hash functions. To avoid the disadvantages due to DAG hierarchies, we transform the UH hierarchy in a tree hierarchy, denoted TUH, where, as in UH, t∈ *λ*(kv) with v=*acl*t . The main disadvantage of TUH with respect to UH is that now the key ring of each user u can now contain more than one key. For this reason, we elaborate an algorithm that takes the UH hierarchy as input and returns a TUH tree as output in such a way to minimize the whole number of keys in the system.

# A Transformation Algorithm

We describe a greedy algorithm that tries to solve the NP-hard problem of minimiz- ing the number of keys directly communicated to users (a proof sketch appears in the Appendix). Figure [3](#_bookmark9) illustrates the algorithm we have developed. Here, *parent*() and *children*() are two functions that take a vertex as input and return its parent

Algorithm 1 (TUH building) MAIN

/\* Initialization \*/

M := {∅} ∪ {*aclt*: *t* ∈ T }

NM := ∅

*E* := ∅

/\* Step 1: select Vertices \*/

/\* Choose non material vertices for TUH \*/

For *l* := |U| *.. .* 1 do

For *vi* ∈ {(M ∪ NM): | *vi* |= *l*} do For *l*' := *l. . .* 1 do

For *vj* ∈ {(M ∪ NM): | *vj* |= *l*'} do

*vk* := *vi* ∩ *vj*

If (*vk* ∈*/* M) then NM := NM ∪ {*vk*}

/\* Step 2: TUH Construction \*/

/\* Edges selection for TUH \*/

For *l* := |U| *.. .* 1 do

For *vi* ∈ {(M ∪ NM): | *vi* |= *l*} do parent(*vi*,*l*-1)

/\* Step 3: prune Tree \*/

/\* Delete non useful vertices from TUH \*/

For *l* := |U| *.. .* 1 do

For *vi* ∈ {NM : | *vi* |= *l*} do

*p* := *parent*(*vi* )

If |*children*(*vi*)|=1 then /\* *vi* is a non material vertex with one child \*/

*c* := *children*(*vi* )

*E* := *E* − {⟨*vi*,*c*⟩, ⟨*p*,*vi*⟩}

NM := NM − {*vi*}

parent(*c*, *l*)

else If |*children*(*vi*)|=0 then /\* *vi* is a non material leaf vertex \*/

*E* := *E* − {⟨*p*,*vi*⟩}; NM := NM − {*vi*}

/\* Step 4: key Assignment \*/

/\* Define each user’s key set\*/ For *i* =1 *.. .* |U| do *φ*(ui) := ∅ For *l*=1 *...* |U| do

For *v* ∈ {(M ∪ NM): | *v* |= *l*} do

For u ∈ {*v*−*parent*(*v*)} do *φ*(u) := *φ*(u) ∪ {*kv* }

PARENT(*v*, *l*)

*found parent* := *false*; *p* := ∅

While *found parent* =*false* do *N* := {*vi*∈(M∪ NM): |*vi*|=*l*} *found* := *false*

While (*N* /= ∅) ∧ (*found* =*false* ) do

Choose *vi*∈ *N* ; *N* := *N* − {*vi*}

If *vi*⊂*v* then

*found parent* := *true*

If *vi*∈M then

*p* := {*vi*}; *found* := *true*

else If |*children*(*vi*)|=1 then /\* *vi* is a non material candidate parent with 1 child \*/

*p* := {*vi*}

else If *p*/= ∅ then /\* Choose the candidate parent with more children \*/

If (|*children*(*p*)| /= 1)∧(|*children*(*p*)| *<* |*children*(*vi*)|) then *p*:= {*vi*}

else *p* := {*vi*} /\* *vi* is the first candidate parent \*/

*l* := *l* − 1

*E* := *E* ∪ ⟨*p*,*v*⟩

Fig. 3. Algorithm that builds TUH

and the set of its children, respectively.

The algorithm builds a key derivation tree TUH, which correctly enforces A and consists mainly of four steps briefly described in the following.

,, ¸∅ ,,,*¸¸*

,, ¸∅ ,,,

,, ¸∅ ,*,*,,

,, ¸*B* ,,,

*¸¸¸¸ ,,,*

*A* ,, ¸*B* ,,, *D*

*A* ,,¸ *B* ,,,*¸*

*¸*

*D*

*,,,*

*¸¸¸¸*

*¸¸¸¸¸¸¸¸ ,,*

*AB *,, ¸*AD* ,,,

*BC BD*

*¸¸*

*AB*

,,¸ *AD* ,,,

,, ¸*ABD* ,,,

,, ¸

*ABC*

*¸*

,,

*¸*

*¸¸¸ ¸¸¸¸¸*

*BC BD*

,, ¸*BCD* ,,,

*,,*

*BC*

,, ¸*AD* ,,,

,,¸ *ABD*, ,,

,,¸ *ABC*, ,,

,, ¸*ABCD* ,,,

,, ¸*ABC* ,,, ,, ¸*ABD*, ,, ,, ¸*BCD*, ,,

,, ¸*ABCD*, ,,

*¸¸¸¸*

,, ¸*ABCD* ,,,

,,¸ *BCD*, ,,

1. Step 1
2. Step 2
3. Steps 3 and 4

Fig. 4. TUH building example

Step 1: select vertices

Given the access matrix representing the policies to enforce, it is first of all nec- essary to select which vertices should be part of TUH. In particular, since each tuple t is encrypted with the key of the vertex representing *acl*t , all vertices correspond- ing to resource *acl*s have to be part of the tree. The set of these vertices, called *material*, is denoted by M. This set contains also the empty set vertex that will be the root vertex of the tree.

In addition to material vertices, also other vertices can be added to the structure, if they can be useful for reducing the number of keys directly assigned to users. The set of these vertices, called *non material*, is denoted by NM. It is easy to see that, the only useful non material vertices are those that can be assigned as direct ancestors of at least two vertices v1 and v2 in the tree. In this case, the key of a vertex v, parent of the two vertices v1 and v2 can be communicated to all users in v, instead of separately communicating both kv1 and kv2 . Thanks to this property, we can build NM by simply adding to it the vertices necessary to close M∪NM with respect to the intersection operator. As an example, consider the access matrix in Figure [1](#_bookmark5). The set of vertices selected by the algorithm is represented in Figure [4(a)](#_bookmark10), where material vertices are circled, and non material vertices are not circled.

Step 2: TUH construction

Once vertices have been chosen, it is necessary to connect them in a tree hier- archy, enforcing the policy in A while minimizing the size of the users’ key rings. To this purpose, the algorithm selects for each vertex in M∪NM but the empty set vertex, a parent through function parent. To minimize the number of keys in the system, function parent chooses a parent by applying the following criteria, which are listed in the same order as they are used.

1. Lower level vertices are preferred to higher level vertices.
2. In case of more candidate vertices at the same level, material vertices are preferred to non material vertices.
3. Among non material vertices, the vertices with exactly one child are preferred.
4. Among other non material vertices, the vertices with more children are pre- ferred.

For instance, with respect to the set of vertices in Figure [4(a)](#_bookmark10), the structure obtained through the second step of the algorithm is represented in Figure [4(b)](#_bookmark11). As an example of application of the criteria above, consider vertex *ABD*. Our algorithm chooses vertex *AD* as parent of *ABD*, instead of *AB* or *BD*, because it is material.

Step 3: prune tree

When all vertices in M∪NM have been correctly connected in a tree, the algo- rithm removes non material vertices that do not reduce the number of keys in the system, as they just make key derivation paths longer.

More precisely, a non material vertex can be removed if it has less than two children, because its key is used neither for tuple encryption nor for key reduction. The removal of a non material vertex with a child v, requires the assignment of an alternative parent to v through function parent, which evaluates candidate direct ancestors at higher levels than the removed vertex. The removal of a non material leaf has instead no consequences on the structure.

Figure [4(c)](#_bookmark11) represents the hierarchy obtained after pruning vertices *AB*, *A*, *D*, and *BD*. In this case, vertex *ABC*, child of *AB*, is connected to *BC* and *AD*, child of *A*, is connected to the root.

Step 4: key assignment

The last step is in charge of assigning a key to each vertex and preparing each user’s key ring. The key of a vertex v belongs to the key ring of u iff u∈v and u/∈*parent*(v). If u∈*parent*(v), u would be able to derive kv through the key of *parent*(v), which she knows either by derivation or by direct communication.

For instance, with respect to the tree in Figure [4(c)](#_bookmark11), *φ*(*A*) = {kABC*,* kAD} (from which user *A* can derive kABCD and kABD); *φ*(*B*) = {kB*,* kABD}; *φ*(*C*) = {kBC}; and *φ*(*D*)= {kAD*,* kBCD*,* kABCD}. The system has then eight different keys to manage.

The algorithm in Figure [3](#_bookmark9) builds a tree that correctly enforces the access control policy represented by the access matrix A (a proof sketch can be found in the Appendix). Its time complexity is *O*(|M ∪ NM|2), that is, polynomial in the number of vertices selected to build the tree hierarchy. Since in the worst case M ∪ NM coincides with the powerset of U (a proof sketch can be found in the Appendix), the time complexity is at most exponential in the number of users in the system. Note that the worst case time complexity depends on the number of users and not on the number of resources, and the first is usually lower than the second one. The space complexity of the solution found by the algorithm strictly depends on the number of vertices in TUH [[3](#_bookmark22)]. In the worst case, it is *O*(2|U|), if the tree has all the vertices in UH. The quality of the solutions computed through our algorithm has been experimentally proved in [[5](#_bookmark24)], where the algorithm has been applied to a simulated system with common characteristics to real life ones. The experiments evaluate both the average number of keys in users’ key rings and the number of material and non material vertices in TUH. Obviously, as the number of users and resources grows, also the number of keys increases, but it scales well with the system size.

# Key Management and Dynamic Access Control Poli- cies

The algorithm in Figure [3](#_bookmark9) builds TUH on the basis of the policy A defined at con- struction time. Therefore, changes in the access control policy, which are translated in changes on the access matrix, may require to change the TUH and the users’ key rings too.

To address dynamic changes of the access control policy, a straightforward solu- tion consists of rebuilding the tree hierarchy any time a change in the access matrix occurs. However, this method would be too expensive both in terms of data owner computation and in terms of system network usage; the data owner should recom- pute, and notify to the users, the encryption keys and then re-encrypt the tuples in the remote database. We therefore propose a method for adapting the TUH tree to a new access control configuration, trying to preserve the keys in the system. Obvi- ously, the tree obtained adapting the original TUH will not have the same structure as the tree we would obtain by executing the transformation algorithm over the new access control policy, but it has the great advantage of saving computational time and bandwidth occupation.

Insert/delete vertex

The main operations necessary to adapt TUH to changes in the access control policy are insertion and deletion of vertices; if a vertex moves from material to non material or viceversa, neither the tree structure nor the key derivation are affected. Due to the characteristics of the key derivation methods operating on trees, the only operations that do not require re-encryption are insertion and deletion of leaf vertices. Instead, when an internal vertex is removed (or inserted) the keys of its descendants have to be changed, because they depend on the key of the deleted (or inserted) vertex. This operation causes the redistribution of keys to users and the

re-encryption of the tuples encrypted through the keys that have been changed.

Let us now consider the case of an insertion of a new (material) vertex v that corresponds to a group of users that is not currently represented in the tree. Vertex v is inserted in TUH as a leaf and then function parent is used to choose an adequate parent for it. When v has been properly connected to the tree, its key kv is computed starting from the parent’s key. Key kv is then communicated to all users in v that cannot obtain it by derivation from other keys in their key rings. For instance, if we need to insert *AB* in the tree in Figure [4(c)](#_bookmark11), we insert the new vertex as child of *B* and kAB is then communicated to *A*.

Let us now consider the case of a deletion of a vertex in TUH. Whenever a leaf vertex becomes non material, it is convenient to remove it from TUH because it is no more useful and causes just a waste of space for its key storage. In this case, it is sufficient to remove the vertex from the hierarchy and to notify all users knowing the corresponding key that the vertex has been deleted. For consistency with the algorithm that builds TUH, also when a vertex with only a child becomes non material, it should be removed from the tree. However, this deletion is quite

,, ¸∅ ,*,*,,*,¸¸¸*

*,,*

, ,

,¸∅,,*,,*

,,¸ *AD* ,,,

,, ¸*ABD*, ,,

*,,,, ¸¸¸¸¸*

*, ,,,,*

,, ¸*B* ,,,

*¸*, ,

,¸*C*,,

,, ¸*B* ,,,

,,¸ *C* ,*,*,, *,*

,, ¸*AC* ,,,

*,, ,*

*, ,,*

*¸¸¸¸ ,,*

*BC*

,,¸ *BCD* ,,,

,, ¸*ABC* ,,,

,, ¸*ABCD* ,,,

,,¸ *AD* ,,,

,, ¸*ABD* ,,,

*,,,*

,, ¸*AC* ,,,

*,,,*

,,¸ *CD* ,,,

,, ¸*ABC* ,,,

,, ¸*ABCD* ,,,

*,,,*

,, ¸*CD* ,,,

*BC*

,,¸ *BCD* ,,,

(a) (b)

Fig. 5. An example of vertex insertion

expensive because all the direct and indirect descendants should change their keys. Consequently, non material vertices with only a child are not deleted. Due to the presence of non material internal vertices with just a child, whenever a leaf is deleted from TUH, also its parent is evaluated: if it becomes a non material leaf too, it is removed as well. This implies that the deletion operation is recursively applied along the path connecting the deleted leaf to the root. For instance, suppose that vertex *AD* in Figure [4(c)](#_bookmark11) becomes non material. Suppose now that also *ABD* becomes non material and therefore is removed. Due to this removal, *AD* becomes a non material leaf as well, and it is removed too. Note that due to vertices insertion and deletion, the tree structure degrades in a lower quality TUH, that is, users’ key rings grow more than necessary. For instance, consider the TUH in Figure [4(c)](#_bookmark11) and suppose to insert vertices *CD*, *AC*, and *C*. Figure [5(a)](#_bookmark13) represents the TUH obtained inserting these vertices according to the procedure above-mentioned, and Figure [5(b)](#_bookmark14) illustrates a better TUH, where *C* is inserted as an internal vertex, thus saving three keys. The causes of this increase in the key rings are mainly two:

1. a new vertex is inserted as a leaf, even if it could be the root of a subtree in

TUH;

1. non material vertices are removed but never inserted.

For this reason, the data owner should periodically rebuild the TUH hierarchy on the basis of the new access control policy, re-encrypt data, send the new encrypted database to the remote server, and communicate the new keys to the users. Since this operation is expensive, it is only executed when the system performances de- crease under a predefined threshold.

Insert/delete tuple

When the data owner inserts a new tuple t in the database, she has to specify its *acl*t because, on the basis of *acl*t , it is then possible to individuate the key that will be used to encrypt t (i.e., the key associated with the vertex corresponding to *acl*t ). Two cases can occur, depending on whether TUH contains such a vertex or not. In the first case, it is sufficient to encrypt t using kaclt , while in the second case it is first necessary to insert a new vertex representing *acl*t . Note that in both cases *acl*t becomes a material vertex.

For instance, with respect to the tree in Figure [4(c)](#_bookmark11), suppose that tuple t7 is inserted and that *acl*t7 ={*A, C, D*}. Since vertex *ACD* does not belong to the tree, it is inserted as a child of *AD*. Suppose now to insert t8 with *acl*t8 ={*B, C*}. In this case *BC* belongs to TUH and therefore it is sufficient to use its key to encrypt t8 and to move this vertex from NM to M.

When the data owner removes a tuple t from the remote database, she needs also to eventually update the TUH structure. If the key of the vertex corresponding to *acl*t is no more used for encryption purpose, the vertex becomes non material and is eventually removed from the hierarchy.

For instance, with respect to the tree in Figure [4(c)](#_bookmark11), suppose that tuple t2 with *acl*t2 ={*A, D*} is deleted. Since kAD is no more used for encryption, *AD* becomes non material but it is maintained in the tree because it has a child. Suppose now that tuple t6 with *acl*t6 ={*A, B, C, D*} is deleted. In this case, *ABCD* becomes non material and is removed from TUH because it is a leaf vertex.

Grant/revoke authorization

When a user (or a set thereof) u is granted (or revoked) access to a tuple t, *acl*t changes and, consequently, also the key used for encrypting t is changed. Let vold be the vertex in TUH representing *acl*t before the change, and let vnew be the vertex representing *acl*t after the change. If kvold is no more used for tuple encryption, vertex vold becomes non material and is eventually removed from the tree. If vertex vnew does not belong to TUH, it is inserted as a new material vertex. Otherwise, if vnew belongs to TUH as a non material vertex, it is moved to the set M of material vertices. Tuple t is then re-encrypted by the data owner using the new key. However, it is important to note that re-encryption may be avoided whenever vold⊂vnew (i.e., for grant operations only) and vold has to be removed and vnew has to be inserted. In this case, key kvold can be associated with vnew, thus avoiding the re-encryption of tuple t and vnew takes the place of vold in TUH. Moreover, kvold has to be communicated to the set of users vnew−vold only.

For instance, with respect to the tree in Figure [4(c)](#_bookmark11), suppose that the data owner grants access to tuple t5 to user *C*. In this case, vold=*ABD* and vnew=*ABCD* and vold is removed from the tree because it becomes a non material leaf. Also, t5 is re-encrypted through kABCD because *ABCD* already belongs to the hierarchy. Now, if the data owner grants access to t2 to *C*, vold=*AD* becomes a non material leaf and can be removed. However, since vnew=*ACD* does not belong to the hierarchy, it is sufficient to associate kvold with vnew, thus avoiding re-encryption. The results of these operations are represented in Figure [6(a)](#_bookmark15) and in Figure [6(b)](#_bookmark16).

Suppose now to revoke the access to t4 from *D*. The new value of *acl*t4 is {*B, C*}. Vertex *BCD* becomes a non material leaf and it is removed from the tree, while *BC* becomes a material vertex. The tree resulting from this operation is represented in Figure [6(c)](#_bookmark16). If we then revoke access to t5 from *A*, vertex *ABCD* becomes a non material leaf and is removed, while material vertex *BCD* is inserted, as a child for *BC*. In this case it is not possible to associate kABCD with the new vertex *BCD*, as *A* knows kABCD and she can read t5 even if she is not allowed to. Moreover, it would

*,,,*

,, ¸∅ ,*,*,,

,, ¸*B* ,,,

*,,,*

*,,,*

*BC* ,, ¸*AD* ,,,

*,,,*

*,,,*

,, ¸∅ ,*,*,,*,*

,, ¸*B* ,,,

*,,*

*BC ,,*

*,,,*

*,,,*

*ccc*

, ,

*c*,¸∅,*,*,

*,,,*

, ,

,

¸ ,,

*B*

*c*

,,¸ *BC* ,,,

,, ¸*ABC*, ,,

,,¸ *ABCD*, ,,

*,,*

,¸*B*,,

*,,,*

, ,

,¸∅,*,*,

, ,

,,¸ *BC¸* ,,,

*,,,*

*,,,*

*,,*

*,,,*

,,¸ *ABC* ,,,

,, ¸*ABCD* ,,,

,, ¸*ABC* ,,,

,,¸ *ABCD* ,,,

,, ¸*ACD* ,,,

,, ¸*ACD* ,,,

*,,,*

*¸¸¸¸¸*

*,,,*

,,¸ *BCD* ,,,

,,¸ *BCD* ,,,

,, ¸*ACD* ,,,

,, ¸*ABC* ,,, ,, ¸*BCD*, ,,

* 1. Grant (*C*,t5 )
  2. Grant (*C*,t2 )
  3. Revoke (*D*,t4 )
  4. Revoke (*A*,t5 )

Fig. 6. Grant and revoke operations

be possible to derive kBCD from kABC but this derivation is not allowed by the partial order relation in TUH. Figure [6(d)](#_bookmark16) represents the TUH after the execution of this revoke operation.

Insert/delete user

When the data owner adds a new user u to the system, all tuples t∈*cap*u should be re-encrypted because their *acl*s change (i.e., user u is added in these *acl*s) and therefore they are associated with a different vertex in TUH. Although user insertion can be treated as a set of grant operations, this solution is not efficient. In this situation, it is possible to avoid re-encryption operations simply considering the fact that, if the data owner communicates to the new user a key k associated with a vertex in the tree, the new user will be able to access all the tuples encrypted with any key derivable from k. A more efficient solution consists then of visiting TUH and verifying, for each vertex vi visited, whether the following four cases arise.

1. u can access all the tuples associated with vertices in the subtree rooted in vi. In this case, kvi is communicated to u and no re-encryption is needed.
2. u can access all the tuples associated with vi, but not the tuples associated with the vertices in its subtree. In this case, we create a new vertex vj=vi∪{u}, child of vi, and the corresponding key kvj is communicated to u and is used to re-encrypt the tuples protected through kvi . At that point, vi becomes non material but, as it has at least two children, it is maintained in TUH.
3. u can access a subset of the tuples associated with vi. In this case, we create a new vertex vj=vi∪{u}, child of vi, and the corresponding key kvj is used to re-encrypt the tuples protected through kvi and accessible to u.
4. u cannot access the tuples associated with vi. No action is needed.

It is important to note that, in cases (ii) and (iii), vertex vj is inserted in the tree without calling function parent, because it is simply a new child of vi. This procedure reduces the time needed for updating TUH and works well because vi is in the lowest possible level where we can find a candidate parent for vj. Moreover, in case (iii) vi is a material vertex, and in case (ii) vi is a non material vertex with at least a child, consequently it is maintained in TUH, independently from vj.

*,,,*

*cccc ,,,*

,, ¸∅ ,*,*,,

, ,

,¸ ,,

*B*

*c*

*BCE*

*E* ,,, ,, ¸*BCDE* ,,,

*eeee ,,,,*

,, ¸∅ ,,,*,*

, ,

,

¸ ,,

*B*

*e*

,,¸ *BC* ,,,

,, ¸*ABC*, ,,

,,¸ *A* ,,,

,, ¸*AB* ,,,

*,,,*

*AD¸*

*¸¸*

*¸¸¸*

,, ¸*ABD* ,,, ,, ¸*ADE* ,,,

* 1. Insert *E*

,, ¸*ABC*

,, ¸*ABCD* ,,,

* 1. Delete *D*

Fig. 7. User insertion and deletion

For instance, with respect to the tree in Figure [4(c)](#_bookmark11), suppose that user *E* is inserted and that *capE* ={t2,t3,t4,t6}. Visiting the tree, we note that user *E* can access all the tuples associated with vertices in the subtree rooted in *BC*. The data owner can then communicate kBC to *E* , and change the groups of users represented by these vertices, adding *E*. By contrast, with respect to *AD*, only t2 but not t5 is in *capE* , so we add vertex *ADE* as a child of *AD*, which becomes non material. Figure [7(a)](#_bookmark17) represents the resulting TUH.

When the data owner removes a user u from the system, she has to re-encrypt all the tuples t∈*cap*u because *acl*t changes, and tuples in *cap*u are associated with a different vertex in TUH. Like for insertion, although user deletion can be treated as a series of revoke operations, this solution is not efficient. Therefore, the removal of user u is partitioned into three main steps, described in the following.

1. Delete each vertex v in TUH such that u∈v.
2. For each t∈*cap*u , insert in TUH the vertex representing the new *acl*t , starting from high-level vertices and going down in the hierarchy. In this way, previously inserted vertices can be parent of subsequently inserted vertices.
3. Re-encrypt tuples in *cap*u with the correct keys.

For instance, with respect to the tree in Figure [4(c)](#_bookmark11), suppose that user *D* with *capD*={t2,t4,t5,t6} is deleted. First, we delete all the vertices containing *D*, that is, *ABCD*, *BCD*, *ABD* and *AD*. The vertices needed for re-encryption reasons are *ABC*, *BC*, *AB*, and *A*. Following the increasing level order, *A* is inserted as a child of the root; *AB* is inserted as a child of *A*; *BC* is already part of the tree and becomes material; *ABC* is already part of the tree and is a material vertex. Figure [7(b)](#_bookmark18) illustrates the resulting tree.

Update optimization

As previously noted, TUH updates result in a lower quality hierarchy than re- building the tree. This is also due to the fact that non material vertices are inserted in the tree during the initial construction phase only. To mitigate such disadvan- tages, we propose to adopt a *preallocation* strategy. According to this strategy, a

tree can only include edges ⟨vi,vj⟩ such that vi⊂vj and |vj−vi| = 1. For instance, with reference to the tree in Figure [5(a)](#_bookmark13), edge ⟨∅,*AC*⟩ is not allowed.

To fill in gaps due to not allowed edges, we add *preallocated* vertices in the tree, between vertices that cannot be adjacent. A preallocated vertex is not adopted for encryption reasons and is not useful for key rings reduction, it is only needed to better accommodate updates. For instance, with respect to edge ⟨∅,*AC*⟩, we add a preallocated vertex *C* between ∅ and *AC*. In this case, when vertex *CD* is inserted, it is added as a child of preallocated vertex *C*, which now becomes non material. Moreover, when the *acl* corresponding to *C* is inserted, vertex *C* becomes material. However, also the preallocation strategy has some drawbacks. First, the tree is composed of a high number of vertices and each of them has a key. Second, since the insertion and deletion operations cannot be known a priori, it is not possible to choose, among the sets of users that each preallocated vertex can represent, the set that is more convenient. For instance, with respect to the previous example, if we associate *A* with the preallocated vertex between ∅ and *AC*, we do not have any

advantage.

# Conclusions

In this paper we introduced the problem of access control enforcement in the database outsourced scenario and proposed an interesting solution based on selec- tive encryption, which exploits hierarchical key derivation methods. In particular, we proposed to build a user-based tree hierarchy and discussed how this structure can be modified in case of a dynamic scenario, where the access control policy may change. The proposed solution has the advantage of outsourcing access control en- forcement, thus it does not request the constant online presence of the data owner. Moreover, this mechanism reduces the number of private keys that each client has to keep, as empirically demonstrated in [[5](#_bookmark24)].

Issues to be investigated will include: the testing, on a real or simulated sys- tem, of the methods proposed for managing access control policy updates; and the management of write privileges.

# References

1. Akl, S. and P. Taylor, *Cryptographic solution to a problem of access control in a hierarchy*, ACM Transactions on Computer System 1 (1983), pp. 239–248.
2. Birget, J., X. Zou, G. Noubir and B. Ramamurthy, *Hierarchy-based access control in distributed environments*, in: *Proc. of IEEE International Conference on Communications*, Helsinki, Finland, 2002.
3. Damiani, E., S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi and P. Samarati, *Metadata management in outsourced encrypted databases*, in: *Proc. of the 2nd VLDB Workshop on Secure Data Management (SDM’05)*, Trondheim, Norway, 2005.
4. Damiani, E., 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, 2003.
5. Damiani, E., S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi and P. Samarati, *Key management for multiuser encrypted databases*, in: *Proc. of the International Workshop on Storage Security and Survivability*, Fairfax Virginia, USA, 2005.
6. Damiani, E., 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, 2003.
7. Davida, G., D. Wells and J. Kam, *A database encryption system with subkeys*, ACM Transactions on Database Systems 6 (1981), pp. 312–328.
8. Hacigu¨mu¨s, H., B. Iyer and S. Mehrotra, *Providing database as a service*, in: *Proc. of 18th International Conference on Data Engineering*, San Jose, California, USA, 2002.
9. Hacigu¨mu¨s, H., 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, 2002.
10. Miklau, G. and D. Suciu, *Controlling access to published data using cryptography*, in: *Proc. of the 29th International Conference on Very Large Data Bases*, Berlin, Germany, 2003.
11. Sandhu, R., *Cryptographic implementation of a tree hierarchy for access control*, Information Processing Letters 27 (1988), pp. 95–98.

# Proof Sketches

* 1. *Minimal Key Rings*

We now present a sketch of a demonstration for the NP-completeness of the problem of minimizing the size of users’ key rings while building TUH. Such a demonstration is based on the polynomial reduction of the minimization problem to the 3-SAT problem, which is NP-complete. We start by describing these two problems.

Minimum TUH. Given two sets of vertices, M and NM, build a minimum TUH connecting all vertices in M and a subset of NM, such that edge ⟨vi,vj⟩ in TUH is allowed iff vi⊆vj. Each edge ⟨vi,vj⟩ costs |vj-vi|.

3-SAT. Let *f* be a logic formula obtained anding *l* clauses *c*1*, c*2*,... cl* defined over boolean variables *x*1*, x*2 *... xn*, where each clause contains three literals (i.e., a clause contains a variable *xi* or its negation *xi*) composed through the or logic operator. Evaluate if there exists an assignment to the *n* variables in *f* that makes the formula true.

To demonstrate that our problem is at least as complex as the 3-SAT, we need to show that, if there exists an algorithm solving the minimum TUH problem, this algorithm solves also 3-SAT. To this purpose, we first show that each instance of 3-SAT can be mapped in an instance of minimum TUH. Given a formula *f* with variables *x*1*, x*2 *... xn* and clauses *c*1*, c*2*,... cl*, we build M and NM sets as follows.

* + - ∅ ∈M, is the root.
    - For each *xi*, *xi* ∈NM, *xi* ∈NM, and *xixi* ∈M.
    - For each *ci*, *ci* ∈M.

We suppose now that there exists an algorithm that solves the minimum TUH problem. Consequently, we just need to map the solution found by this algorithm in a solution for the corresponding instance of 3-SAT. The idea is that, if the obtained TUH contains just one between *xi* and *xi* for each *i* = 1 *... n*, *f* is satisfiable. This is because, if vertex *xi* belongs to TUH, a true value is assigned to *xi*; if vertex *xi* belongs to TUH, a false value is assigned to *xi*. According to this assignment, each clause *ci*, which is connected with one of the literals in it, will be evaluated to true

,, ¸∅ ,,,*¸¸¸*

,, ¸∅ ,,,

*¸¸¸*

*x*1 *x*

*x*2 *x*

*x*3 *x*

*x*1

*x*2 *¸*

*¸¸*

*¸*

*x*3 *¸*

1 2 3

*¸¸¸*

*¸¸¸*

*¸¸¸¸¸*

*¸¸¸¸¸*

,, ¸*x*1 *x*1 ,,,

,, ¸*x*2*x*2 ,,,

,, ¸*x*3*x*3 ,,,

,, ¸*x*1*x*1 ,,,

*¸¸¸¸*

,,¸ *x*2*x*2 ,,,

,, ¸*x*3 *x*3, ,,

(a)

,, ¸*x*1*x*2*x*3 ,,,

,, ¸*x*1*x*2 *x*3 ,,,

,, ¸*x*1 *x*2*x*3 ,,,

*¸¸¸¸*

,, ¸*x*1*x*2*x*3, ,,

,,¸ *x*1*x*2 *x*3, ,,

(b)

Fig. A.1. Demonstration example

,, ¸*x*1 *x*2*x*3 ,,,

and therefore *f* will be satisfied. Since the algorithm computes the minimal TUH, non material vertices that are not useful for minimization are removed because they need at least an edge to be connected to the rest of the tree. Vertices *xi* and *xi* are both non material and, consequently, one of them is removed if it is not needed (one is always maintained as parent for *xixi*).

As an example, let *f* be (*x*1 ∨ *x*2 ∨ *x*3) ∧ (*x*1 ∨ *x*2 ∨ *x*3) ∧ (*x*1 ∨ *x*2 ∨ *x*3). Figure [1(a)](#_bookmark31) represents the material vertices M (circled) and the non material vertices NM (non circled). Figure [1(b)](#_bookmark32) represents the minimum TUH, which corresponds to a solution for *f* where *x*1 = *x*2 = *x*3 = true.

* 1. *Correctness*

To demonstrate the correctness of Algorithm [1](#_bookmark8), it is necessary to prove the following three different assertions.

TUH is a tree. Each vertex in a tree has exactly one parent. In our algorithm, it is assigned through function parent, which adds edge ⟨*p,* v⟩ to the tree and is called exactly once for each v in M∪NM. Moreover, as the cardinality of the set of users corresponding to *p* is always lower than the cardinality of the set of users corresponding to v, the hierarchy cannot have cycles.

*λ* correctness. Each tuple is correctly encrypted with the key of the vertex rep- resenting its *acl* because we initially add a vertex v in M for each *acl*t such that t∈T . Moreover, material nodes are never removed from the tree.

*φ* correctness. Each user u is directly communicated key kv, such that u∈v and

u does not belong to the set of users represented by the parent of v. Since edge

⟨vi*,* vj⟩ belongs to TUH only if vi⊂vj (see the condition in function parent), u can derive only keys of vertices she belongs to. Moreover, as the whole tree is visited for key assignment, u knows all these keys.

* 1. *Complexity Analysis*

To evaluate the time complexity of Algorithm [1](#_bookmark8), we compute the cost of the steps composing it.

Step 1: select vertices. The selection of material vertices has linear cost in |T |, as we scan all tuples to find out their *acl*s. The selection of non material instead

requires the computation of all possible couples of vertices in M∪NM. Given a set of *n* elements, the number of non ordered couples of distinct elements is: (*n*)= 1*/*2 · *n* · (*n* − 1). The cost of NM computation is then *O*(|M ∪ NM|2).

2

Step 2: TUH construction. The edges selection scans all the vertices in the tree, to find out a good direct ancestor through function parent, which is called |M ∪ NM| times. The function is composed of two nested cycles, which look for a parent for v in a subset of M∪NM. The cost of this step is then *O*(|M ∪ NM|2).

Step 3: prune tree. Also this third step scans all the vertices in the tree, calling sometimes function parent. The cost of the pruning phase is then *O*(|M ∪ NM|2).

Step 4: key assignment. This step visits the tree and computes, for each vertex, its key. The cost of this step is then *O*(|M ∪ NM|· *α*), where *α* is the cost of key derivation; it can be considered a constant as its value does not depend on the size of the tree.

The computational cost of the proposed algorithm is therefore: |T |+3

·*O*(|M∪NM|2), that is, *O*(|M ∪ NM|2).