Selective data encryption in outsourced dynamic environments

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

[1] *DTI - Università degli Studi di Milano*
26013 Crema - Italy

[2] *George Mason University*
Fairfax, VA 22030-4444

[3] *DIGI - Università 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, investigating 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.

1 Introduction

Nowadays, databases hold a critical concentration of sensitive information and their volume is increasing very quickly. In such a scenario, *database outsourcing* is becoming increasingly popular. A client's database is stored at an external service provider that should provide mechanisms for clients to access the outsourced databases. The main advantage of outsourcing is related to the costs of in-house versus outsourced hosting (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

[1] Email: {damiani,decapita,foresti,samarati}@dti.unimi.it
[2] Email: jajodia@gmu.edu
[3] Email: parabosc@unibg.it

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 techniques are usually adopted [7]. 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 [4,6,8,9] that associate indexes with the outsourced 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, operate 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 assumption 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]. 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 introduces the main concepts of access control and presents our selective encryption solution. Section 3 describes the algorithm we propose to efficiently enforce access control through selective encryption in the specific scenario. Section 4 illustrates how changes in the access control policies can be efficiently managed. Finally, Section 5 contains our conclusions.

2 Scenario and basic definitions

Given a system with a set $\mathcal{U}$ of users and a set $\mathcal{T}$ of resources, we assume that the access control policies are represented via an access matrix $\mathcal{A}$ with $|\mathcal{U}|$ rows and $|\mathcal{T}|$ columns, where $\mathcal{A}[\mathtt{u},\mathtt{t}]$ contains the operations that user $\mathtt{u}$ can perform on $\mathtt{t}$. Since we consider the read operation only, each entry in the access matrix can simply assume two values: $\mathcal{A}[\mathtt{u},\mathtt{t}]$=1 if $\mathtt{u}$ can read $\mathtt{t}$; 0 otherwise. Figure 1 represents an example of access matrix for a system with 6 tuples ($\mathtt{t}_1 \ldots \mathtt{t}_6$) and 4 users ($A$, B, C, and D). Given an access matrix $\mathcal{A}$, $acl_{\mathtt{t}}$ denotes the access control list for tuple $\mathtt{t}$, that is, the set of users that can access $\mathtt{t}$; $cap_{\mathtt{u}}$ denotes the capability list of user $\mathtt{u}$, that is, the set of tuples she can access. For instance, with reference to Figure 1, $acl_{\mathtt{t}_2}$=$\{A,D\}$ and cap_B=$\{\mathtt{t}_1,\mathtt{t}_3,\mathtt{t}_4,\mathtt{t}_5,\mathtt{t}_6\}$.

In the considered scenario, the enforcement of access control policies cannot be

	$\mathbf{t_1}$	$\mathbf{t_2}$	$\mathbf{t_3}$	$\mathbf{t_4}$	$\mathbf{t_5}$	$\mathbf{t_6}$
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

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* [2,10] 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 $\mathcal{K}$ be the set of symmetric encryption keys used to protect data. We introduce two functions:

- a *user key assignment* $\phi : \mathcal{U} \mapsto 2^{\mathcal{K}}$, which associates with each user $\mathtt{u} \in \mathcal{U}$ the set of keys $\mathtt{k} \in \mathcal{K}$ in the user's key ring;
- a *resource key assignment* $\lambda : \mathcal{K} \mapsto 2^{\mathcal{T}}$, which associates with each key $\mathtt{k} \in \mathcal{K}$ the set of tuples $\mathtt{t} \in \mathcal{T}$ encrypted with $\mathtt{k}$.

Given a user key assignment function ϕ, a resource key assignment λ, and an access matrix $\mathcal{A}$, the pair (ϕ, λ) is said to be *complete* with respect to $\mathcal{A}$, denoted as $(\phi, \lambda) \Rightarrow \mathcal{A}$, if each user can decrypt all tuples she can access according to $\mathcal{A}$. The pair (ϕ, λ) is said to be *sound* with respect to $\mathcal{A}$, denoted as $(\phi, \lambda) \Leftarrow \mathcal{A}$, if no user can decrypt tuples that she cannot access according to $\mathcal{A}$. The pair (ϕ, λ) is said to *correctly enforce* $\mathcal{A}$, denoted as $(\phi, \lambda) \Leftrightarrow \mathcal{A}$, iff it is both sound and complete with respect to $\mathcal{A}$.

A straightforward solution for adopting selective encryption in our scenario associates a key with each tuple $\mathtt{t}$ and communicates to each user $\mathtt{u}$ the keys used to encrypt tuples in $cap_{\mathtt{u}}$. It is easy to see that this solution correctly enforces $\mathcal{A}$, 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] or DAGs [1]. In our scenario, it is natural to introduce the definition of a user hierarchy $\mathsf{UH}=(2^{|\mathcal{U}|}, \preceq)$, where the domain of the hierarchy is the powerset of $\mathcal{U}$ and $\preceq$ is the subset containment partial order relation. Graphically, the UH hierarchy can be represented as a graph, called *user graph*, with a vertex for each element in $2^{|\mathcal{U}|}$ and a path connecting a with b iff $b \preceq a$. A vertex $\mathtt{v}$ of the graph is therefore characterized by the set of users to which it corresponds. In the following, given a vertex $\mathtt{v}$ of the user graph, it will be used to denote the set of users that it represents. Figure 2 illustrates an example of UH hierarchy for $\mathcal{U}=\{A,B,C,D\}$, 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

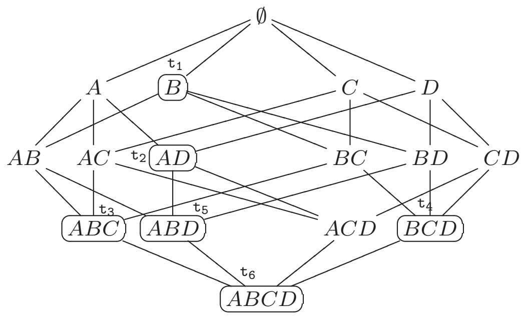


Fig. 2. An example of UH

v in the user graph is associated with a key $\mathtt{k_v}$ and a path $\mathtt{v_1} \rightarrow \mathtt{v_2} \rightarrow \ldots \rightarrow \mathtt{v_n}$ in the graph represents a *key derivation path*, meaning that starting from the key associated with vertex $\mathtt{v_1}$ and following the path, it is possible to derive the keys of all vertices $\mathtt{v_i}, \mathtt{i} = 2, \ldots, n$, in the path. For instance, path $AB \rightarrow ABC \rightarrow ABCD$ in the UH hierarchy in Figure 2 means that key $\mathtt{k_{ABCD}}$ can be derived from key $\mathtt{k_{ABC}}$ that in turn can be derived from key $\mathtt{k_{AB}}$. According to the UH definition, $\phi(\mathtt{u})=\mathtt{k_{v_i}}$ with $\mathtt{v_i}=\{\mathtt{u}\}$ (i.e., the key ring of each user contains one key only); and $\mathtt{t}\in \lambda(\mathtt{k_v})$ with $acl_\mathtt{t}=\mathtt{v}$. Consequently, each user can derive all keys $\mathtt{k_v}$ such that $\mathtt{u}\in\mathtt{v}$.

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 theorems (e.g., modular exponentiation), which make the support of a dynamic scenario complex and not efficient. By contrast, methods proposed for tree hierarchies exploit 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, $\mathtt{t}\in \lambda(\mathtt{k_v})$ with $\mathtt{v}=acl_\mathtt{t}$. 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.

3 A transformation algorithm

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

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

Step 1: select vertices

Given the access matrix representing the policies to enforce, it is first of all necessary to select which vertices should be part of TUH. In particular, since each tuple t is encrypted with the key of the vertex representing $acl_\mathtt{t}$, all vertices corresponding to resource *acl*s have to be part of the tree. The set of these vertices, called

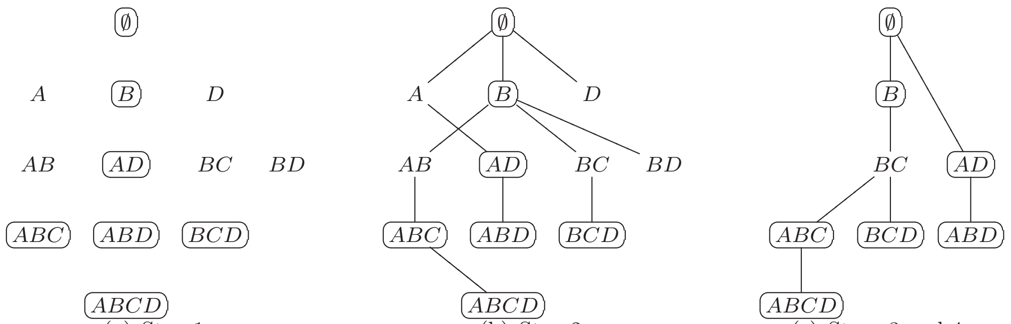
Algorithm 1 (TUH **building**)

```
MAIN
/* Initialization */
M := {∅} ∪ {$acl_t$: $t \in \mathcal{T}$}
NM := ∅
$E$ := ∅

/* Step 1: select Vertices */
/* Choose non material vertices for TUH */
For $l := |\mathcal{U}| \ldots 1$ do
  For $v_i \in \{(\mathsf{M} \cup \mathsf{NM}) : |v_i| = l\}$ do
    For $l' := l \ldots 1$ do
      For $v_j \in \{(\mathsf{M} \cup \mathsf{NM}) : |v_j| = l'\}$ do
        $v_k := v_i \cap v_j$
        If($v_k \notin$ M) then NM := NM ∪ {$v_k$}

/* Step 2: TUH Construction */
/* Edges selection for TUH */
For $l := |\mathcal{U}| \ldots 1$ do
  For $v_i \in \{(\mathsf{M} \cup \mathsf{NM}) : |v_i| = l\}$ do parent($v_i$,$l$-1)

/* Step 3: prune Tree */
/* Delete non useful vertices from TUH */
For $l := |\mathcal{U}| \ldots 1$ do
  For $v_i \in \{\mathsf{NM} : |v_i| = l\}$ do
    $p := parent(v_i)$
    If $|children(v_i)|$=1 then /* $v_i$ is a non material vertex with one child */
      $c := children(v_i)$
      $E := E - \{\langle v_i,c\rangle, \langle p,v_i\rangle\}$
      NM := NM − {$v_i$}
      parent($c$, $l$)
    else If $|children(v_i)|$=0 then /* $v_i$ is a non material leaf vertex */
          $E := E - \{\langle p,v_i\rangle\}$; NM := NM − {$v_i$}

/* Step 4: key Assignment */
/* Define each user's key set*/
For $i$= 1 ... $|\mathcal{U}|$ do $\phi(\mathtt{u_i}) := \emptyset$
For $l$= 1 ... $|\mathcal{U}|$ do
  For $v \in \{(\mathsf{M} \cup \mathsf{NM}) : |v| = l\}$ do
    For $\mathtt{u} \in \{v - parent(v)\}$ do $\phi(\mathtt{u}) := \phi(\mathtt{u}) \cup \{k_v\}$

PARENT($v$, $l$)
$found\_parent := false$; $p := \emptyset$
While $found\_parent$=$false$ do
   $N := \{v_i \in (\mathsf{M} \cup \mathsf{NM}): |v_i|=l\}$
   $found := false$
   While $(N \neq \emptyset) \wedge (found$=$false)$ do
      Choose $v_i \in N$; $N := N - \{v_i\}$
      If $v_i \subset v$ then
        $found\_parent := true$
        If $v_i \in$M then
          $p := \{v_i\}$; $found := true$
        else If $|children(v_i)|$=1 then /* $v_i$ is a non material candidate parent with 1 child */
              $p := \{v_i\}$
            else If $p \neq \emptyset$ then /* Choose the candidate parent with more children */
                   If$(|children(p)| \neq 1) \wedge (|children(p)| < |children(v_i)|)$ then $p$:= {$v_i$}
                 else $p := \{v_i\}$ /* $v_i$ is the first candidate parent */
   $l := l - 1$
$E := E \cup \langle p,v\rangle$
```

Fig. 3. Algorithm that builds TUH

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 $\mathtt{v_1}$ and $\mathtt{v_2}$ in the tree. In this case, the key of a vertex $\mathtt{v}$, parent of the two vertices $\mathtt{v_1}$ and $\mathtt{v_2}$ can be communicated to all users in $\mathtt{v}$, instead



(a) Step 1 (b) Step 2 (c) Steps 3 and 4

Fig. 4. TUH building example

of separately communicating both $\mathtt{k}_{\mathtt{v}_1}$ and $\mathtt{k}_{\mathtt{v}_2}$. 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. The set of vertices selected by the algorithm is represented in Figure 4(a), 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 hierarchy, enforcing the policy in $\mathcal{A}$ 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.

(i) Lower level vertices are preferred to higher level vertices.

(ii) In case of more candidate vertices at the same level, material vertices are preferred to non material vertices.

(iii) Among non material vertices, the vertices with exactly one child are preferred.

(iv) Among other non material vertices, the vertices with more children are preferred.

For instance, with respect to the set of vertices in Figure 4(a), the structure obtained through the second step of the algorithm is represented in Figure 4(b). 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 algorithm 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) 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 $\texttt{v}$ belongs to the key ring of $\texttt{u}$ iff $\texttt{u}\in\texttt{v}$ and $\texttt{u}\notin \mathit{parent}(\texttt{v})$. If $\texttt{u}\in\mathit{parent}(\texttt{v})$, $\texttt{u}$ would be able to derive $\texttt{k}_\texttt{v}$ through the key of $\mathit{parent}(\texttt{v})$, which she knows either by derivation or by direct communication.

For instance, with respect to the tree in Figure 4(c), $\phi(A) = \{\texttt{k}_{\texttt{ABC}}, \texttt{k}_{\texttt{AD}}\}$ (from which user A can derive $\texttt{k}_{\texttt{ABCD}}$ and $\texttt{k}_{\texttt{ABD}}$); $\phi(B) = \{\texttt{k}_{\texttt{B}}, \texttt{k}_{\texttt{ABD}}\}$; $\phi(C) = \{\texttt{k}_{\texttt{BC}}\}$; and $\phi(D) = \{\texttt{k}_{\texttt{AD}}, \texttt{k}_{\texttt{BCD}}, \texttt{k}_{\texttt{ABCD}}\}$. The system has then eight different keys to manage.

The algorithm in Figure 3 builds a tree that correctly enforces the access control policy represented by the access matrix $\mathcal{A}$ (a proof sketch can be found in the Appendix). Its time complexity is $O(|\mathsf{M}\cup\mathsf{NM}|^2)$, that is, polynomial in the number of vertices selected to build the tree hierarchy. Since in the worst case $\mathsf{M}\cup\mathsf{NM}$ coincides with the powerset of $\mathcal{U}$ (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]. In the worst case, it is $O(2^{|\mathcal{U}|})$, if the tree has all the vertices in UH. The quality of the solutions computed through our algorithm has been experimentally proved in [5], 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.

4 Key management and dynamic access control policies

The algorithm in Figure 3 builds TUH on the basis of the policy $\mathcal{A}$ defined at construction 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 solution 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 recompute, 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. Obviously, 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 $\mathtt{v}$ that corresponds to a group of users that is not currently represented in the tree. Vertex $\mathtt{v}$ is inserted in TUH as a leaf and then function **parent** is used to choose an adequate parent for it. When $\mathtt{v}$ has been properly connected to the tree, its key $\mathtt{k_v}$ is computed starting from the parent's key. Key $\mathtt{k_v}$ is then communicated to all users in $\mathtt{v}$ 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), we insert the new vertex as child of B and $\mathtt{k_{AB}}$ 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 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) 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) and suppose to insert vertices CD, AC, and C. Figure 5(a) represents the TUH obtained inserting these vertices according to the procedure above-mentioned, and Figure 5(b) 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:

(i) a new vertex is inserted as a leaf, even if it could be the root of a subtree in TUH;

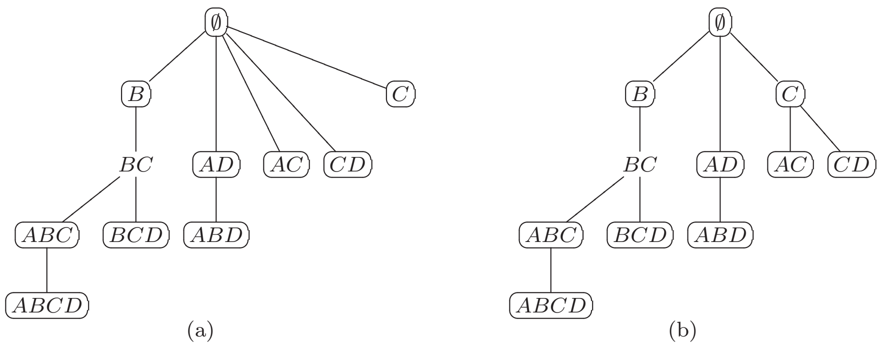


Fig. 5. An example of vertex insertion

(ii) 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 decrease under a predefined threshold.

Insert/delete tuple

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

For instance, with respect to the tree in Figure 4(c), suppose that tuple $\mathtt{t}_7$ is inserted and that $acl_{\mathtt{t}_7}=\{A, C, D\}$. Since vertex ACD does not belong to the tree, it is inserted as a child of AD. Suppose now to insert $\mathtt{t}_8$ with $acl_{\mathtt{t}_8}=\{B, C\}$. In this case BC belongs to TUH and therefore it is sufficient to use its key to encrypt $\mathtt{t}_8$ and to move this vertex from NM to M.

When the data owner removes a tuple $\mathtt{t}$ from the remote database, she needs also to eventually update the TUH structure. If the key of the vertex corresponding to $acl_{\mathtt{t}}$ 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), suppose that tuple $\mathtt{t}_2$ with $acl_{\mathtt{t}_2}=\{A, D\}$ is deleted. Since $\mathtt{k}_{\mathtt{AD}}$ 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 $\mathtt{t}_6$ with $acl_{\mathtt{t}_6}=\{A, B, C, D\}$ 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) $\mathtt{u}$ is granted (or revoked) access to a tuple $\mathtt{t}$, $acl_{\mathtt{t}}$ changes and, consequently, also the key used for encrypting $\mathtt{t}$ is changed. Let $\mathtt{v}_{\mathtt{old}}$ be the vertex in TUH representing $acl_{\mathtt{t}}$ before the change, and let $\mathtt{v}_{\mathtt{new}}$ be

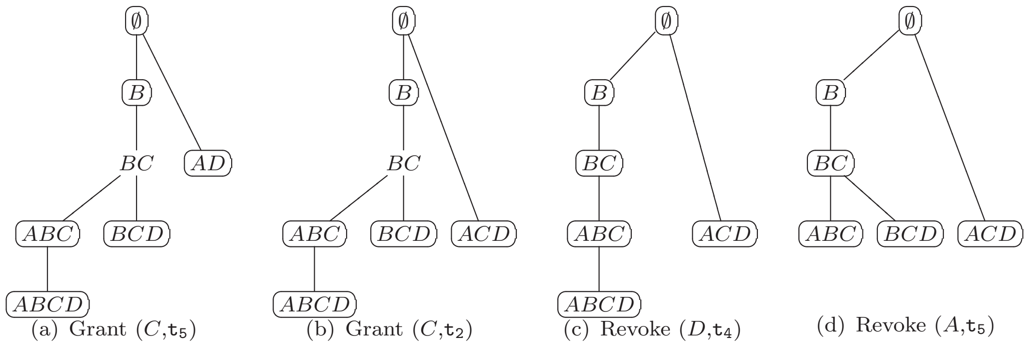


Fig. 6. Grant and revoke operations

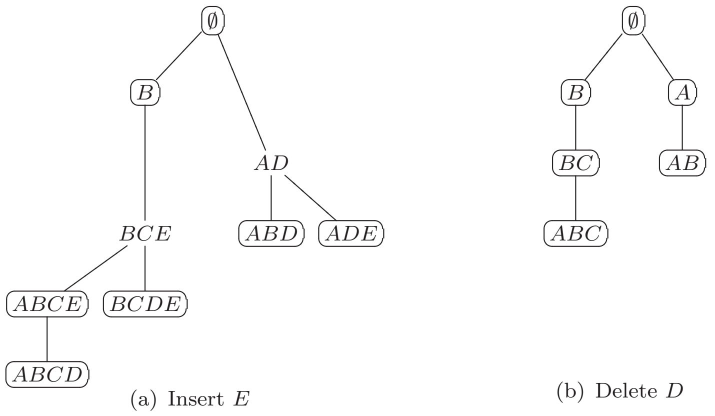
the vertex representing $acl_{\mathtt{t}}$ after the change. If $\mathtt{k}_{\mathtt{v}_{\mathtt{old}}}$ is no more used for tuple encryption, vertex $\mathtt{v}_{\mathtt{old}}$ becomes non material and is eventually removed from the tree. If vertex $\mathtt{v}_{\mathtt{new}}$ does not belong to TUH, it is inserted as a new material vertex. Otherwise, if $\mathtt{v}_{\mathtt{new}}$ belongs to TUH as a non material vertex, it is moved to the set M of material vertices. Tuple $\mathtt{t}$ 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 $\mathtt{v}_{\mathtt{old}}\subset\mathtt{v}_{\mathtt{new}}$ (i.e., for grant operations only) and $\mathtt{v}_{\mathtt{old}}$ has to be removed and $\mathtt{v}_{\mathtt{new}}$ has to be inserted. In this case, key $\mathtt{k}_{\mathtt{v}_{\mathtt{old}}}$ can be associated with $\mathtt{v}_{\mathtt{new}}$, thus avoiding the re-encryption of tuple $\mathtt{t}$ and $\mathtt{v}_{\mathtt{new}}$ takes the place of $\mathtt{v}_{\mathtt{old}}$ in TUH. Moreover, $\mathtt{k}_{\mathtt{v}_{\mathtt{old}}}$ has to be communicated to the set of users $\mathtt{v}_{\mathtt{new}}-\mathtt{v}_{\mathtt{old}}$ only.

For instance, with respect to the tree in Figure 4(c), suppose that the data owner grants access to tuple $\mathtt{t}_5$ to user C. In this case, $\mathtt{v}_{\mathtt{old}}=ABD$ and $\mathtt{v}_{\mathtt{new}}=ABCD$ and $\mathtt{v}_{\mathtt{old}}$ is removed from the tree because it becomes a non material leaf. Also, $\mathtt{t}_5$ is re-encrypted through $\mathtt{k}_{\mathtt{ABCD}}$ because $ABCD$ already belongs to the hierarchy. Now, if the data owner grants access to $\mathtt{t}_2$ to C, $\mathtt{v}_{\mathtt{old}}=AD$ becomes a non material leaf and can be removed. However, since $\mathtt{v}_{\mathtt{new}}=ACD$ does not belong to the hierarchy, it is sufficient to associate $\mathtt{k}_{\mathtt{v}_{\mathtt{old}}}$ with $\mathtt{v}_{\mathtt{new}}$, thus avoiding re-encryption. The results of these operations are represented in Figure 6(a) and in Figure 6(b).

Suppose now to revoke the access to $\mathtt{t}_4$ from D. The new value of $acl_{\mathtt{t}_4}$ 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). If we then revoke access to $\mathtt{t}_5$ 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 $\mathtt{k}_{\mathtt{ABCD}}$ with the new vertex BCD, as A knows $\mathtt{k}_{\mathtt{ABCD}}$ and she can read $\mathtt{t}_5$ even if she is not allowed to. Moreover, it would be possible to derive $\mathtt{k}_{\mathtt{BCD}}$ from $\mathtt{k}_{\mathtt{ABC}}$ but this derivation is not allowed by the partial order relation in TUH. Figure 6(d) represents the TUH after the execution of this revoke operation.

Insert/delete user

When the data owner adds a new user $\mathtt{u}$ to the system, all tuples $\mathtt{t}\in cap_{\mathtt{u}}$ should be re-encrypted because their acls change (i.e., user $\mathtt{u}$ is added in these acls) 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



(a) Insert E (b) Delete D

Fig. 7. User insertion and deletion

fact that, if the data owner communicates to the new user a key $\mathtt{k}$ associated with a vertex in the tree, the new user will be able to access all the tuples encrypted with any key derivable from $\mathtt{k}$. A more efficient solution consists then of visiting TUH and verifying, for each vertex $\mathtt{v_i}$ visited, whether the following four cases arise.

(i) $\mathtt{u}$ can access all the tuples associated with vertices in the subtree rooted in $\mathtt{v_i}$. In this case, $\mathtt{k_{v_i}}$ is communicated to $\mathtt{u}$ and no re-encryption is needed.

(ii) $\mathtt{u}$ can access all the tuples associated with $\mathtt{v_i}$, but not the tuples associated with the vertices in its subtree. In this case, we create a new vertex $\mathtt{v_j}=\mathtt{v_i}\cup\{\mathtt{u}\}$, child of $\mathtt{v_i}$, and the corresponding key $\mathtt{k_{v_j}}$ is communicated to $\mathtt{u}$ and is used to re-encrypt the tuples protected through $\mathtt{k_{v_i}}$. At that point, $\mathtt{v_i}$ becomes non material but, as it has at least two children, it is maintained in TUH.

(iii) $\mathtt{u}$ can access a subset of the tuples associated with $\mathtt{v_i}$. In this case, we create a new vertex $\mathtt{v_j}=\mathtt{v_i}\cup\{\mathtt{u}\}$, child of $\mathtt{v_i}$, and the corresponding key $\mathtt{k_{v_j}}$ is used to re-encrypt the tuples protected through $\mathtt{k_{v_i}}$ and accessible to $\mathtt{u}$.

(iv) $\mathtt{u}$ cannot access the tuples associated with $\mathtt{v_i}$. No action is needed.

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

For instance, with respect to the tree in Figure 4(c), suppose that user E is inserted and that $cap_E=\{\mathtt{t_2},\mathtt{t_3},\mathtt{t_4},\mathtt{t_6}\}$. 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 $\mathtt{k_{BC}}$ to E, and change the groups of users represented by these vertices, adding E. By contrast, with respect to AD, only $\mathtt{t_2}$ but not $\mathtt{t_5}$ is in cap_E, so we add vertex ADE as a child of AD, which becomes non material. Figure 7(a) represents the resulting TUH.

When the data owner removes a user $\mathtt{u}$ from the system, she has to re-encrypt all the tuples $\mathtt{t}\in cap_{\mathtt{u}}$ because $acl_{\mathtt{t}}$ changes, and tuples in $cap_{\mathtt{u}}$ 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.

(i) Delete each vertex v in TUH such that $\mathtt{u}\in\mathtt{v}$.

(ii) For each $\mathtt{t}\in cap_{\mathtt{u}}$, insert in TUH the vertex representing the new $acl_{\mathtt{t}}$, starting from high-level vertices and going down in the hierarchy. In this way, previously inserted vertices can be parent of subsequently inserted vertices.

(iii) Re-encrypt tuples in $cap_{\mathtt{u}}$ with the correct keys.

For instance, with respect to the tree in Figure 4(c), suppose that user D with $cap_D=\{\mathtt{t}_2,\mathtt{t}_4,\mathtt{t}_5,\mathtt{t}_6\}$ 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) illustrates the resulting tree.

Update optimization

As previously noted, TUH updates result in a lower quality hierarchy than rebuilding 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 disadvantages, we propose to adopt a *preallocation* strategy. According to this strategy, a tree can only include edges $\langle \mathtt{v_i},\mathtt{v_j}\rangle$ such that $\mathtt{v_i}\subset\mathtt{v_j}$ and $|\mathtt{v_j}-\mathtt{v_i}| = 1$. For instance, with reference to the tree in Figure 5(a), edge $\langle\emptyset,AC\rangle$ 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 $\langle\emptyset,AC\rangle$, we add a preallocated vertex C between $\emptyset$ 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 $\emptyset$ and AC, we do not have any advantage.

5 Conclusions

In this paper we introduced the problem of access control enforcement in the database outsourced scenario and proposed an interesting solution based on selective 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 enforcement, 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].

Issues to be investigated will include: the testing, on a real or simulated system, 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] Hacigümüs, 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] Hacigümüs, 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.

A Proof sketches

A.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 $\langle v_i, v_j \rangle$ in TUH is allowed iff $v_i \subseteq v_j$. Each edge $\langle v_i, v_j \rangle$ costs $|v_j - v_i|$.

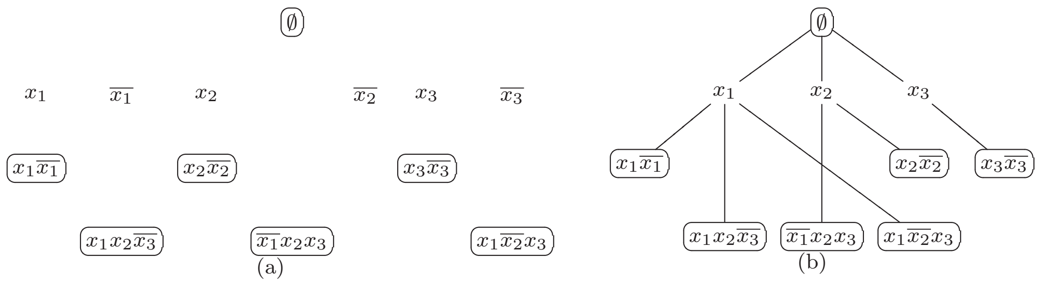


Fig. A.1. Demonstration example

3-SAT. Let f be a logic formula obtained ANDing l clauses $c_1, c_2, \ldots c_l$ defined over boolean variables $x_1, x_2 \ldots x_n$, where each clause contains three literals (i.e., a clause contains a variable x_i or its negation $\overline{x_i}$) 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 $x_1, x_2 \ldots x_n$ and clauses $c_1, c_2, \ldots c_l$, we build M and NM sets as follows.

- $\emptyset$ ∈M, is the root.
- For each x_i, x_i ∈NM, $\overline{x_i}$ ∈NM, and $x_i\overline{x_i}$ ∈M.
- For each c_i, c_i ∈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 x_i and $\overline{x_i}$ for each $i = 1 \ldots n$, f is satisfiable. This is because, if vertex x_i belongs to TUH, a true value is assigned to x_i; if vertex $\overline{x_i}$ belongs to TUH, a false value is assigned to x_i. According to this assignment, each clause c_i, which is connected with one of the literals in it, will be evaluated to true 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 x_i and $\overline{x_i}$ are both non material and, consequently, one of them is removed if it is not needed (one is always maintained as parent for $x_i\overline{x_i}$).

As an example, let f be $(x_1 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3)$. Figure 1(a) represents the material vertices M (circled) and the non material vertices NM (non circled). Figure 1(b) represents the minimum TUH, which corresponds to a solution for f where $x_1 = x_2 = x_3 =$ true.

A.2 Correctness

To demonstrate the correctness of Algorithm 1, 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 $\langle p, \texttt{v}\rangle$ to the tree and is called exactly once for each $\texttt{v}$ in $\mathsf{M}\cup\mathsf{NM}$. 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 $\texttt{v}$, the hierarchy cannot have cycles.

λ **correctness.** Each tuple is correctly encrypted with the key of the vertex representing its *acl* because we initially add a vertex $\texttt{v}$ in M for each $acl_{\texttt{t}}$ such that $\texttt{t}\in\mathcal{T}$. Moreover, material nodes are never removed from the tree.

ϕ **correctness.** Each user $\texttt{u}$ is directly communicated key $\texttt{k}_{\texttt{v}}$, such that $\texttt{u}\in\texttt{v}$ and $\texttt{u}$ does not belong to the set of users represented by the parent of $\texttt{v}$. Since edge $\langle \texttt{v}_{\texttt{i}}, \texttt{v}_{\texttt{j}}\rangle$ belongs to TUH only if $\texttt{v}_{\texttt{i}}\subset\texttt{v}_{\texttt{j}}$ (see the condition in function **parent**), $\texttt{u}$ can derive only keys of vertices she belongs to. Moreover, as the whole tree is visited for key assignment, $\texttt{u}$ knows all these keys.

A.3 Complexity analysis

To evaluate the time complexity of Algorithm 1, we compute the cost of the steps composing it.

Step 1: select vertices. The selection of material vertices has linear cost in $|\mathcal{T}|$, 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 $\mathsf{M}\cup\mathsf{NM}$. Given a set of n elements, the number of non ordered couples of distinct elements is: $\binom{n}{2} = 1/2 \cdot n \cdot (n-1)$. The cost of NM computation is then $O(|\mathsf{M}\cup\mathsf{NM}|^2)$.

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 $|\mathsf{M}\cup\mathsf{NM}|$ times. The function is composed of two nested cycles, which look for a parent for $\texttt{v}$ in a subset of $\mathsf{M}\cup\mathsf{NM}$. The cost of this step is then $O(|\mathsf{M}\cup\mathsf{NM}|^2)$.

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 $O(|\mathsf{M}\cup\mathsf{NM}|^2)$.

Step 4: key assignment. This step visits the tree and computes, for each vertex, its key. The cost of this step is then $O(|\mathsf{M}\cup\mathsf{NM}| \cdot \alpha)$, 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: $|\mathcal{T}|+3 \cdot O(|\mathsf{M}\cup\mathsf{NM}|^2)$, that is, $O(|\mathsf{M}\cup\mathsf{NM}|^2)$.