Managing Key Hierarchies for Access Control Enforcement: Heuristic Approaches[☆]

Carlo Blundo[a], Stelvio Cimato[b], Sabrina De Capitani di Vimercati[b], Alfredo De Santis[a], Sara Foresti[b], Stefano Paraboschi[c], Pierangela Samarati[*,b]

[a] *DIA - Università di Salerno, 84084 Fisciano - Italy*
[b] *DTI - Università degli Studi di Milano, 26013 Crema - Italy*
[c] *DIIMM - Università di Bergamo, 24044 Dalmine - Italy*

Abstract

Data outsourcing is emerging today as a successful paradigm allowing individuals and organizations to resort to external servers for storing their data, and sharing them with others. The main problem of this trend is that sensitive data are stored on a site that is not under the data owner's direct control. This scenario poses a major security problem since often the external server is relied upon for ensuring high availability of the data, but it is not authorized to read them. Data need therefore to be encrypted. In such a context, the application of an access control policy requires different data to be encrypted with different keys so to allow the external server to directly enforce access control and support selective dissemination and access. The problem therefore emerges of designing solutions for the efficient management of an encryption policy enforcing access control, with the goal of minimizing the number of keys to be maintained by the system and distributed to users.

In this paper, we prove that the problem of minimizing the number of keys is NP-hard and present alternative approaches for its solution. We first formu-

[☆] A preliminary version of this paper appeared under the title "Efficient Key Management for Enforcing Access Control in Outsourced Scenarios," in Proc. of the 24th IFIP TC-11 International Information Security Conference (SEC 2009), Cyprus, May 2009 [1].

[*] Corresponding author
Email addresses: carblu@dia.unisa.it (Carlo Blundo), stelvio.cimato@unimi.it (Stelvio Cimato), sabrina.decapitani@unimi.it (Sabrina De Capitani di Vimercati), ads@dia.unisa.it (Alfredo De Santis), sara.foresti@unimi.it (Sara Foresti), parabosc@unibg.it (Stefano Paraboschi), pierangela.samarati@unimi.it (Pierangela Samarati)

late the minimization problem as an instance of an integer linear programming problem and then propose three different families of heuristics, which are based on a key derivation tree exploiting the relationships among user groups. Finally, we experimentally evaluate the performance of our heuristics, comparing them with previous approaches.

Key words: Data outsourcing, encryption policy, confidentiality

---

1. Introduction

Data outsourcing has become increasingly popular in recent years. The main advantage of data outsourcing is that it promises higher availability and more effective disaster protection than in-house operations. However, since data owners physically release their information to external servers that are not under their control, data confidentiality (and even integrity) may be put at risk. Besides protecting such data from attackers and unauthorized users, there is the need to protect the privacy of the data from the so called *honest-but-curious* servers, that is, servers that, while trustworthy to properly manage the data, may not be trusted by the data owner to read their content. The problem of protecting data when outsourcing them to an external honest-but-curious server has emerged to the attention of researchers very recently. Existing proposals (e.g., [2, 3, 4]) in the data outsourcing area typically store the data in encrypted form and associate with the encrypted data additional indexing information. Such indexes are used by the external DBMS to select the data to be returned in response to a query. Existing approaches however do not address the problem of supporting different access privileges (authorizations) for different users and result therefore limited in today's scenarios, where remotely stored data may need to be made accessible in a selective way (i.e., different users may be authorized to access different views of the data).

There is therefore an increasing interest in the definition of security solutions that allow the enforcement of access control policies on outsourced data. A promising solution in this direction consists in integrating access control and

encryption. Although traditional approaches assume a strict separation between policies and mechanisms, and such a separation has often been beneficial, in the data outsourcing scenario their combination is proving successful. Combining cryptography with access control essentially requires that resources should be encrypted differently depending on the access authorizations holding on them, so to make their decryption possible only to authorized users [5, 6]. The application of this approach in data outsourcing scenarios allows owners to encrypt data, according to an encryption policy regulated by authorizations, outsource the data to the external servers, and distribute to users the needed encryption keys. Proper encryption and key distribution automatically ensure therefore obedience of the access control policy, while not requiring the data owner to maintain control on the data storage and accesses. In this paper, we address such a problem and propose a heuristic approach to minimize the number of keys to be maintained by the system and distributed to users. Like other proposals in the literature [5, 6, 7], we base our solution on *key derivation* exploiting a *key derivation tree* that allows users to derive new keys from other keys they know. In [1] we presented an early version of our proposal that here is extended to consider alternative approaches for determining the minimal number of keys that correctly enforce an authorization policy defined by the data owner. In particular, we provide a new formulation of the problem in terms of an integer linear programming problem and present two new families of heuristics (i.e., leaves-based and mixed-based heuristics). We also formally prove that the problem of minimizing the number of keys is NP-Hard.

The remainder of the paper is organized as follows. Section 2 illustrates the basic concepts of access control systems based on selective encryption. Section 3 introduces our minimization problem. Section 4 formulates the minimization problem as a minimum weight problem. Section 5 illustrates the integer linear programming problem corresponding to the weight minimization problem. Section 6 presents three families of heuristic, which are based on the computation of a minimum spanning tree (MST) and on vertices factorization to improve the quality of the solution. Section 7 presents some experimental results showing

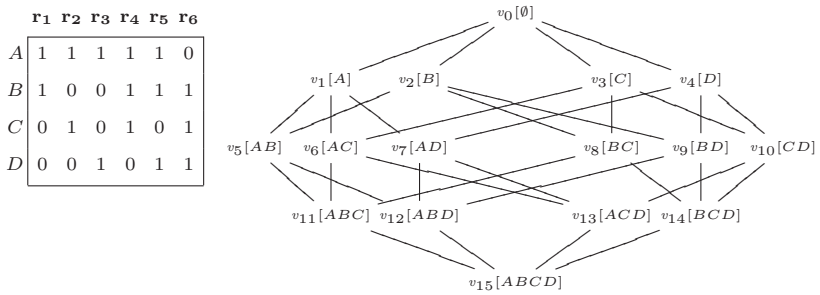


Figure 1: An example of access matrix (a) and of user graph over $\mathcal{U}$={A,B,C,D} (b)

that, compared with previous proposals, our heuristics prove efficient and effective in the computation of a key derivation graph. Section 8 discusses related work. Finally, Section 9 draws our conclusions.

2. Basic concepts

We assume that the data owner defines an authorization policy to regulate read access to the outsourced resources.[1] Given a set $\mathcal{U}$ of users and a set $\mathcal{R}$ of resources, an *authorization policy* over $\mathcal{U}$ and $\mathcal{R}$ is defined as a set of pairs $\langle u,r\rangle$, where $u\in\mathcal{U}$ and $r\in\mathcal{R}$, meaning that user u can access resource r. An authorization policy can be modeled via an *access matrix* $\mathcal{A}$, with a row for each user $u\in\mathcal{U}$, a column for each resource $r\in\mathcal{R}$, and $\mathcal{A}[u, r]$ is equal to 1 (0, resp.) if u has (does not have, resp.) authorization to access r. Given an access matrix $\mathcal{A}$, $acl(r)$ denotes the *access control list* of r (i.e., the set of users that can access r), and $cap(u)$ denotes the *capability list* of u (i.e., the set of resources that u can access). Figure 1(a) illustrates an example of access matrix with four users (A, B, C, D) and six resources ($r_1,\ldots,r_6$), where, for example, $acl(r_2)$={A,C} and $cap(C)$={r_2,r_4,r_6}.

In the data outsourcing scenario, the enforcement of the authorization policy

[1] Write operations require re-encryption and re-uploading of the resources on the server and are performed at the owner's site, typically by the owner itself.

cannot be delegated to the remote server, which is trusted neither for accessing the data content nor for enforcing the authorization policy. Consequently, the data owner has to be involved in the access control enforcement. To avoid the owner's involvement in managing access and enforcing authorizations, recently *selective encryption* techniques have been proposed [5, 6, 8]. Selective encryption means that the *encryption policy* (i.e., which data are encrypted with which key) is dictated by the authorizations to be enforced on the data. The basic idea is to use different keys for encrypting data and to release to each user the set of keys necessary to decrypt all and only the resources the user is authorized to access. For efficiency reasons, selective encryption is realized through symmetric keys.

A straightforward solution for implementing selective encryption associates a key with each resource r and communicates to each user u the keys used to encrypt the resources in $cap(u)$. It is easy to see that this solution, while correctly enforcing the authorization policy, is too expensive to manage, due to the high number of keys each user has to keep. Indeed, any user $u \in \mathcal{U}$ would need to hold as many keys as the number of resources she is authorized to access.

To avoid users having to store and manage a huge number of (secret) keys, consistently with other proposals in the literature [5, 6], we exploit a *key derivation method* that allows the derivation of a key starting from another key and some public information [9, 10, 11, 12, 13, 14]. In our scenario, the derivation relationship between keys can be represented through a *user graph*, where there is a vertex v for each possible set of users and a key associated with it, and there is an edge (v_i, v_j) for all pairs of vertices such that the set of users represented by v_i is a subset of the set of users represented by v_j. In the following, we use $v.acl$ to denote the set of users represented by vertex v and $v.key$ to denote the key associated with v. Formally, a user graph is defined as follows.

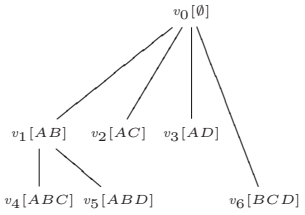
Definition 2.1 (User Graph). *Given a set $\mathcal{U}$ of users, a* user graph *over $\mathcal{U}$, denoted $G_{\mathcal{U}}$, is a graph $\langle V_{\mathcal{U}}, E_{\mathcal{U}} \rangle$, where $V_{\mathcal{U}} = P(\mathcal{U})$ is the power set of $\mathcal{U}$, and $E_{\mathcal{U}} = \{(v_i, v_j) \mid v_i.acl \subset v_j.acl\}$.*

As an example, consider the set of users $\mathcal{U}$=$\{A,B,C,D\}$. Figure 1(b) reports the user graph, where, for each vertex v_i, the users in the square brackets represent $v_i.acl$. For clarity of the picture, edges that are implied by other edges (relationships between sets differing for more than one user) are not reported.

By exploiting the user graph defined above, the authorization policy can be enforced: *i)* by encrypting each resource with the key of the vertex corresponding to its access control list (e.g., resource r_4 should be encrypted with $v_{11}.key$ since $acl(r_4)$=$v_{11}.acl$=$\{A,B,C\}$), and *ii)* by assigning to each user the key associated with the vertex representing the user in the graph. Since edges represent the possible key derivations, each user u, starting from her own key, can directly compute the keys of all vertices v such that u∈$v.acl$. It is easy to see that this approach to design the encryption policy *correctly enforces* the authorization policy represented by matrix $\mathcal{A}$, meaning that each user u can only derive the keys of the resources she is authorized to access. For instance, with reference to the user graph in Figure 1(b), user A knows the key associated with vertex v_1 from which she can derive, following the edges outgoing from v_1, the keys associated with vertices v_5, v_6, v_7, v_{11}, v_{12}, v_{13}, and v_{15}.

3. Problem formulation

Although the solution based on a user hierarchy is conceptually simple and potentially easy to implement, it defines significantly more keys than actually needed. Furthermore, a crucial aspect for the success of a solution supporting selective encryption is the efficiency of the key management and distribution activities required. For these reasons, since key derivation methods working on trees are in general more convenient and simpler than those working on DAGs and require a lower amount of publicly available information, we transform, according with the proposal in [5], the user graph $G_{\mathcal{U}}$ in a *user tree*, denoted T, enforcing the authorization policy in $\mathcal{A}$. The user tree must include the set, denoted $\mathcal{M}$, of all vertices, called *material vertices*, representing acl values (i.e., vertices whose keys are used for encrypting resources) and the empty set of



(a)

user	**key_ring**
A	$v_1.key$, $v_2.key$, $v_3.key$
B	$v_1.key$, $v_6.key$
C	$v_2.key$, $v_4.key$, $v_6.key$
D	$v_3.key$, $v_5.key$, $v_6.key$

(b)

Figure 2: A user tree (a) and the corresponding key rings (b)

users (i.e., $\mathcal{M} = \{v \in V_{\mathcal{U}} \mid v.acl{=}\emptyset \vee \exists\ r \in \mathcal{R}$ with $v.acl = acl(r)\}$), as formally defined in the following.

Definition 3.1 (User tree). *Let $\mathcal{A}$ be an access matrix over a set $\mathcal{U}$ of users and a set $\mathcal{R}$ of resources, and $G_{\mathcal{U}} = \langle V_{\mathcal{U}}, E_{\mathcal{U}} \rangle$ be the user graph over $\mathcal{U}$. A* user tree*, denoted T, is a tree $T = \langle V, E \rangle$, subgraph of $G_{\mathcal{U}}$, rooted at vertex v_0, with $v_0.acl{=}\emptyset$, where $\mathcal{M} \subseteq V \subseteq V_{\mathcal{U}}$, and $E \subseteq E_{\mathcal{U}}$.*

In other words, a user tree is a tree, rooted at the vertex representing the empty user group $\emptyset$, subgraph of $G_{\mathcal{U}}$, and spanning all vertices in $\mathcal{M}$.

To grant the correct enforcement of the authorization policy, each user u has a key ring, denoted $key\_ring_T(u)$, containing all the keys necessary to derive the keys of all vertices v such that $u \in v.acl$. More precisely, the key ring of each user u must include the keys associated with all vertices v such that $u \in v.acl$ and $u \notin v_p.acl$, where v_p is the parent of v. If $u \in v_p.acl$, u must already have access to the key in v_p and must be able to derive $v.key$ through the key of v_p, which she knows either by derivation or by direct communication.

Clearly, given a set of users and an authorization policy $\mathcal{A}$, more user trees may exist. Among all possible user trees, we are interested in determining a *minimum user tree*, correctly enforcing a given authorization policy and minimizing the number of keys in users' key rings.

Definition 3.2 (Minimum user tree). *Let $\mathcal{A}$ be an access matrix and T be*

a user tree correctly enforcing $\mathcal{A}$. T *is* minimum *with respect to* $\mathcal{A}$ *iff* $\nexists T'$ *such that* T' *correctly enforces* $\mathcal{A}$ *and* $\sum_{u \in \mathcal{U}} |key\_ring_{T'}(u)| < \sum_{u \in \mathcal{U}} |key\_ring_{T}(u)|$.

Figure 2(a) illustrates an example of user tree and Figure 2(b) reports the corresponding user key rings.[2]

Given an access matrix $\mathcal{A}$, different minimum user trees may exist and our goal is to compute one of them, as stated by the following problem definition.

Problem 3.1. *Let* $\mathcal{A}$ *be an access matrix. Determine a minimum user tree* T.

Since Problem 3.1 is NP-hard, in [5] we proposed a heuristic algorithm that builds a user tree over the set of vertices obtained by closing $\mathcal{M}$ with respect to the intersection operator. For each vertex, the algorithm selects a parent choosing first vertices representing larger sets of users, and then material vertices. Finally, the algorithm prunes non necessary vertices.

In the following, we further investigate the problem and elaborate alternative solutions, that, as our experimental results prove, demonstrate more efficient and generally better than our previous proposal [5].

4. Minimum weight user tree

Our solution is based on a reformulation of Problem 3.1 in terms of a weight minimization problem. We start by introducing the concept of weight in association with a user tree.

Definition 4.1 (Weight function). *Let* $T=\langle V,E\rangle$ *be a user tree.*

- $w{:}E \rightarrow \mathbb{N}$ *is a weight function such that* $\forall (v_i, v_j) \in E$,
  $w(v_i, v_j) = | v_j.acl \setminus v_i.acl |$

[2] Note that the keys in each key ring could be managed with the use of *tokens*, public pieces of information that allow the reconstruction of a secret from another one [10, 11]. The minimality of the user tree implies the minimization of the number of tokens, making the approach presented in this paper applicable also to scenarios using tokens.

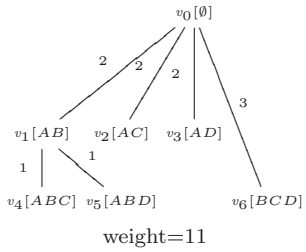


Figure 3: Weighted version of the user tree in Figure 2

- $\texttt{weight}(T) = \sum_{(v_i,v_j)\in E} w(v_i, v_j)$.

According to this definition, given a user tree T=$\langle V,E\rangle$, the weight $w(v_i,v_j)$ of edge (v_i,v_j) in E is the number of users in $v_j.acl \backslash v_i.acl$ and $\texttt{weight}(T)$ is the sum of the weights of its edges. Figure 3 illustrates an example of weighted user tree. Problem 3.1 can now be reformulated as the problem of finding a *minimum weight* user tree. In fact, the presence of an edge $(v_i,v_j) \in E$ implies that users in $v_i.acl$ should know both keys $v_i.key$ and $v_j.key$ while users in $v_j.acl \backslash v_i.acl$ need only to know $v_j.key$. It is then sufficient to include key $v_i.key$ in the key rings of all users in $v_i.acl$, since $v_j.key$ can be derived from $v_i.key$, and to include key $v_j.key$ in the key rings of users in $v_j.acl \backslash v_i.acl$. This is equivalent to say that $w(v_i, v_j)$ corresponds to the number of users whose key ring must include key $v_j.key$. Generalizing, it is immediate to conclude that $\texttt{weight}(T)$ is equal to the sum of the total number of keys stored in users' key rings (i.e., $\texttt{weight}(T) = \sum_{u\in\mathcal{U}} |key\_ring_T(u)|$).

The problem of computing a user tree with minimum weight is NP-hard since the Vertex Cover problem can be reduced to it, as formally stated by the following theorem.

Theorem 4.1 (NP-hardness). *Let $\mathcal{A}$ be an access matrix over a set $\mathcal{U}$ of users and a set $\mathcal{R}$ of resources. The problem of computing a minimum weight user tree T correctly enforcing $\mathcal{A}$ is NP-hard.*

Proof. See Appendix A. ■

To solve the problem of computing a minimum weight user tree, we propose two alternative approaches. First, we formulate the minimization problem as an Integer Linear Programming (ILP) problem, which can be solved adopting known algorithms and tools. Second, we introduce three families of heuristics, which are based on the computation of a minimum spanning tree induced by material vertices $\mathcal{M}$ over the user graph $G_{\mathcal{U}}$. The three families of heuristics differ in the strategy adopted to reduce the cost of the minimum spanning tree. In the following, we discuss in more details these two approaches.

5. Linear programming approach

The translation of the minimum weight user tree problem into a linear programming problem exploits the interpretation of the edges in the user graph $G_{\mathcal{U}}=\langle V_{\mathcal{U}},E_{\mathcal{U}}\rangle$ as boolean variables. For each edge (v_i,v_j) in $E_{\mathcal{U}}$, we define a boolean variable, denoted $x_{i,j}$, representing whether (v_i,v_j) is an edge of a minimum weight user tree $T = \langle V,E\rangle$. The vertices composing V are all and only the vertices in $V_{\mathcal{U}}$ having at least an incident edge in E.

The formulation of our minimization problem as an integer linear programming problem requires the specification of the objective function and of a set of linear constraints. The objective function corresponds to the minimization of `weight`(T), modeled as the sum of the weights of the edges (v_i,v_j) such that $x_{i,j}$ is equal to one. The linear constraints impose that: *1)* the edges (and vertices) selected form a tree structure; *2)* all material vertices belong to the user tree.

The first property is satisfied if there is a root vertex (vertex v_0 in our case) and each vertex v in the tree has exactly one direct ancestor, that is, the number of incoming edges of each vertex v in $V_{\mathcal{U}}$ is either 1 or 0, depending on whether v belongs to the computed user tree. The fact that a vertex of a user tree may have only one incoming edge is equivalent to say that given a vertex v_i, if at least one among the variables representing its outgoing edges $x_{i,k}$ is equal to 1 (i.e., v_i belongs to V), exactly one among the variables representing its incoming edges $x_{l,i}$ must be equal to 1, meaning that v_i has to be connected to the tree

with one incoming edge. Since the maximum number of children of a vertex v in T is upper-bounded by the number of material vertices and, therefore, by the number of resources $|\mathcal{R}|$, by imposing that the number of incoming edges in v_i multiplied by $|\mathcal{R}|$ is greater than or equal to the number of edges outgoing from v_i, vertex v_i can have outgoing edges only if it has at least an incoming edge. Vertex v_0 is instead implicitly set as the root of the resulting tree structure.

The second property, that is, the fact that all material vertices must belong to the computed user tree is modeled by requiring that each material vertex, but v_0, has exactly one incoming edge. Note that we can only impose constraints on incoming edges, since a material vertex could be a leaf of the computed user tree. The weight minimization problem can then be formally modeled as an integer linear programming problem as follows.

Definition 5.1 (ILP minimum user tree). *Given an access matrix $\mathcal{A}$ over a set $\mathcal{U}$ of users and a set $\mathcal{R}$ of resources and the weight function w in Definition 4.1, the* ILP minimum user tree problem *is a problem formulated as follows.*

Minimize $$\sum_{(v_i,v_j)\in E_{\mathcal{U}}} w(v_i, v_j)\, x_{i,j}$$

subject to

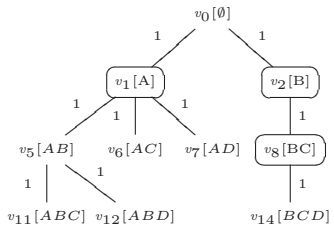
1) $$\sum_{v_i\in V_{\mathcal{U}},(v_i,v_j)\in E_{\mathcal{U}}} x_{i,j} \leq 1, \qquad \forall\ v_j\in V_{\mathcal{U}}\backslash\mathcal{M};$$

2) $$|\mathcal{R}|\cdot \sum_{v_i\in V_{\mathcal{U}},(v_i,v_j)\in E_{\mathcal{U}}} x_{i,j} \geq \sum_{v_k\in V_{\mathcal{U}},(v_j,v_k)\in E_{\mathcal{U}}} x_{j,k}, \quad \forall\ v_j\in V_{\mathcal{U}}\backslash\{v_0\};$$

3) $$\sum_{v_i\in V_{\mathcal{U}},(v_i,v_j)\in E_{\mathcal{U}}} x_{i,j} = 1, \qquad \forall\ v_j\in\mathcal{M}\backslash\{v_0\};$$

4) $x_{i,j}\in\{0,1\}$, $\forall\ (v_i, v_j)\in E_{\mathcal{U}}$.

It is immediate to see that the integer linear programming formulation of the minimum user tree problem exactly represents the problem of computing a minimum weight user tree rooted in v_0. The objective function models the minimization requirement. The first set of constraints requires that each non



user	key_ring
A	$v_1.key$
B	$v_2.key$, $v_5.key$
C	$v_6.key$, $v_8.key$, $v_{11}.key$
D	$v_7.key$, $v_{12}.key$, $v_{14}.key$

weight = 9

(a) (b)

Figure 4: User tree and key rings resulting from the resolution of the ILP minimum user tree problem

material vertex in the user graph has at most one incoming edge in the user tree. The second set of constraints requires that only vertices having at least an outgoing edge must have an incoming edge in the user tree. The third set of constraints requires that each material vertex has exactly an incoming edge in the user tree and therefore that it belongs to the user tree. Finally, the fourth set of constraints states that the variables associated with the edges of the user graph can only assume value 1 or 0, modeling the presence or not of the corresponding edge in the computed user tree.

As an example, consider the access matrix in Figure 1(a) and the user graph in Figure 1(b). The solution computed by the LPsolve integer linear programming tool [15] of the corresponding ILP minimum user tree problem assigns value 1 to variables $x_{0,1}$, $x_{0,2}$, $x_{1,5}$, $x_{1,6}$, $x_{1,7}$, $x_{2,8}$, $x_{5,11}$, $x_{5,12}$, $x_{8,14}$, and value 0 to all other variables. Figure 4 illustrates the resulting optimal user tree and key rings.

6. Minimum spanning tree heuristics

We now propose three families of heuristics for solving Problem 3.1. All the proposed heuristics are based on the computation of a minimum spanning tree (MST) over a graph $G = \langle V, E', w \rangle$, with $V = \mathcal{M}$, $E' = \{(v_i, v_j) \mid v_i, v_j \in$

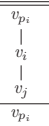
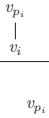
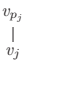
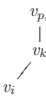
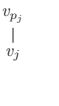
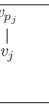
	Case (U=$v_k.acl$=$v_i.acl$∩$v_j.acl$)	Initial configuration	Final configuration	weight_red(v_{p_i},v_{p_j},v_i,v_j)
1	U=$v_i.acl$			$\lvert v_i.acl \rvert - \lvert v_{p_j}.acl \rvert$
	U=$v_j.acl$			$\lvert v_j.acl \rvert - \lvert v_{p_i}.acl \rvert$
2	$v_k \in V$ and $v_k \neq v_i$ and $v_k \neq v_j$			$2\,\lvert U \rvert - (\lvert v_{p_i}.acl \rvert + \lvert v_{p_j}.acl \rvert)$
3	$v_k \notin V$ and either ∘ $v_{p_i}.acl \subset U$ and $v_{p_j}.acl \not\subset U$ or ∘ $v_{p_i}.acl \subset U$, $v_{p_j}.acl \subset U$, and $\lvert U \rvert - \lvert v_{p_j}.acl \rvert \geq \lvert U \rvert - \lvert v_{p_i}.acl \rvert$			$\lvert U \rvert - \lvert v_{p_j}.acl \rvert$
	$v_k \notin V$ and either ∘ $v_{p_j}.acl \subset U$ and $v_{p_i}.acl \not\subset U$ or ∘ $v_{p_i}.acl \subset U$, $v_{p_j}.acl \subset U$, and $\lvert U \rvert - \lvert v_{p_i}.acl \rvert > \lvert U \rvert - \lvert v_{p_j}.acl \rvert$			$\lvert U \rvert - \lvert v_{p_i}.acl \rvert$
	$v_k \notin V$ and $v_{p_i}.acl \not\subset U$ and $v_{p_j}.acl \not\subset U$			$\lvert v_t.acl \rvert + \lvert U \rvert - (\lvert v_{p_i}.acl \rvert + \lvert v_{p_j}.acl \rvert)$

Figure 5: Possible updates to the user tree

$V \wedge v_i.acl \subset v_j.acl\}$, w the weight function defined in Definition 4.1, and where the root vertex is v_0. The MST over G is a user tree whose weight can be further reduced with the addition of non-material vertices that represent sets of users resulting from the intersection of the acls of at least two vertices already in the MST. To better understand the reason for which the insertion of such a kind of vertices may cause a reduction of the weight, consider two vertices, say v_i and v_j, in the MST. The possibly insertion of a new vertex $v_k.acl$=$v_i.acl \cap v_j.acl$ as a parent of v_i and v_j can reduce the weight of the tree since the key ring of users in $v_k.acl$ should only include $v_k.key$ instead of both $v_i.key$ and $v_j.key$. Let v_{p_i} and v_{p_j} be the unique direct ancestor of vertex v_i and of vertex v_j, respectively. The weight reduction is formally defined by function $weight\_red$:$V \times V \times V \times V \rightarrow \mathbb{N}$, that given vertices v_i, v_j, v_{p_i}, and v_{p_j} returns a value computed according to whether vertex $v_k.acl$=$v_i.acl \cap v_j.acl$ exists in the tree. The following three

cases, represented in Figure 5, may occur.

Case 1 v_k=v_i (or v_k=v_j), that is, one of the two vertices represents a subset of the users represented by the other vertex. The user tree can be updated by removing the edge connecting vertex v_{p_j} with v_j (v_{p_i} with v_i, resp.) and by inserting the edge connecting v_i with v_j (v_j with v_i, resp.). As a consequence, the weight of the tree is reduced by $w(v_{p_j}, v_j) - w(v_i, v_j)$ ($w(v_{p_i}, v_i) - w(v_j, v_i)$, resp.), which is equal to $|v_i.acl| - |v_{p_j}.acl|$ ($|v_j.acl| - |v_{p_i}.acl|$, resp.).

Case 2 $v_k \in V$ and $v_k \neq v_i$ and $v_k \neq v_j$, that is, there is a vertex in the tree representing the set $U = v_i.acl \cap v_j.acl$ of users. The user tree can be updated by removing the edge connecting vertex v_{p_i} with v_i and the edge connecting v_{p_j} with v_j, and by inserting two new edges connecting v_k with v_i and v_j, respectively. As a consequence, the weight of the tree is reduced by $w(v_{p_i}, v_i) + w(v_{p_j}, v_j) - (w(v_k, v_i) + w(v_k, v_j))$, which is equal to $2|U| - (|v_{p_i}.acl| + |v_{p_j}.acl|)$.

Case 3 $v_k \notin V$, that is, there is no vertex representing the set $U = v_i.acl \cap v_j.acl$ of users in the tree. The user tree can be updated by removing the edges connecting v_{p_i} and v_{p_j} with v_i and v_j, respectively, and inserting three new edges connecting: *1)* v_k with v_i, *2)* v_k with v_j, and *3)* a vertex in V with v_k that can be chosen among the following three vertices.

- v_{p_i}: if either $v_{p_i}.acl \subset U$ and $v_{p_j}.acl \not\subset U$, or if $v_{p_i}.acl \subset U$, $v_{p_j}.acl \subset U$, and the cardinality of the set of users represented by v_{p_i} is greater than or equal to the cardinality of the set of users represented by v_{p_j} (i.e., $|U| - |v_{p_j}.acl| \geq |U| - |v_{p_i}.acl|$). The weight of the tree is reduced by $w(v_{p_i}, v_i) + w(v_{p_j}, v_j) - (w(v_k, v_i) + w(v_k, v_j) + w(v_{p_i}, v_k))$, which is equal to $|U| - |v_{p_j}.acl|$.
- v_{p_j}: if either $v_{p_j}.acl \subset U$ and $v_{p_i}.acl \not\subset U$, or if $v_{p_i}.acl \subset U$, $v_{p_j}.acl \subset U$, and the cardinality of the set of users represented by v_{p_j} is greater than the cardinality of the set of users represented by v_{p_i} (i.e., $|U| -$

$|v_{p_i}.acl| > |U| - |v_{p_j}.acl|$). The weight of the tree is reduced by $w(v_{p_i}, v_i) + w(v_{p_j}, v_j) - (w(v_k, v_i) + w(v_k, v_j) + w(v_{p_j}, v_k))$, which is equal to $|U| - |v_{p_i}.acl|$.

- v_t: if $v_{p_i}.acl \not\subset U$, $v_{p_j}.acl \not\subset U$, and v_t is a vertex in the set $P=\{v \in V | v.acl \subseteq v_k\}$ of candidate parents of v_k such that $|v_t.acl|$ contains the maximum number of users. The weight of the tree is then reduced by $w(v_{p_i}, v_i) + w(v_{p_j}, v_j) - (w(v_k, v_i) + w(v_k, v_j) + w(v_t, v_k))$, which is equal to $|v_t.acl| + |U| - (|v_{p_i}.acl| + |v_{p_j}.acl|)$.
  Note that in principle the direct ancestor of vertex v_k can always be chosen as the vertex in the set P of candidate parent vertices whose acl contains the maximum number of users. However, since this selection process is expensive, we decide to use v_{p_i} or v_{p_j} as a direct ancestor of v_k whenever it is possible.

The three families of heuristics differ in the pairs of vertices considered for weight reduction: the *sibling-based* (S) family considers only pairs of sibling vertices; the *leaf-based* (L) family considers only pairs of vertices that are not sibling and where at least one vertex is a leaf; and the *mixed* (M) family considers pairs of vertices without imposing any constraint on the vertices involved. Among all possible candidate pairs of vertices, the pairs $\langle v_i, v_j \rangle$ that maximize the reduction of the weight of the user tree is then chosen. Note that different pairs of vertices may provide the same maximum weight reduction. In this case, different preference criteria may be applied for choosing a pair. In particular, we propose the following three criteria:

- *rnd*: at random;
- *max*: in such a way that $|v_i.acl| + |v_j.acl|$ is maximum, ties are broken randomly;
- *min*: in such a way that $|v_i.acl| + |v_j.acl|$ is minimum, ties are broken randomly.

INPUT
set $\mathcal{U}$ of users
set $\mathcal{R}$ of resources
access matrix $\mathcal{A}$
heuristic (S, L, or M)
criterion (*rnd*, *max*, or *min*)

OUTPUT
user tree T= $\langle V,E \rangle$

MAIN
$V := \emptyset$
$E := \emptyset$
/* **Phase 1**: select material vertices */
$Acl_{\mathcal{M}} := \{acl(r) | r \in \mathcal{R}\} \cup \{\emptyset\}$
for each $acl \in Acl_{\mathcal{M}}$ **do**
    create vertex v
    $v.acl := acl$
    $V := V \cup \{v\}$
/* **Phase 2**: compute a minimum spanning tree */
$E' := \{(v_i,v_j) \mid v_i,v_j \in V \wedge v_i.acl \subset v_j.acl\}$
let w be a weight function such that $\forall (v_i, v_j) \in E'$, $w(v_i,v_j) = |v_j.acl \setminus v_i.acl|$
$G := (V,E',w)$
let v_0 be the vertex in V with $v_0.acl = \emptyset$
$T :=$ **Minimum_Spanning_Tree**(G,v_0)
/* **Phase 3**: insert non-material vertices */
case *heuristic* **of**
 S: $T :=$ **Factorize_Siblings**$(T,$ *criterion*$)$
 L: $T :=$ **Factorize_Leaves**$(T,$ *criterion*$)$
 M: $T :=$ **Factorize_Vertices**$(T,$ *criterion*$)$
return(T)

Figure 6: Heuristic algorithm for computing a minimal user tree

Any of these three preference criteria in combination with the three families of heuristics discussed above can be used to compute an approximation of the minimum user tree. Figure 6 illustrates a heuristic algorithm that, given an authorization policy represented through an access matrix $\mathcal{A}$, the identification of a heuristic family (i.e., S, L, or M), and a preference criterion (i.e., *rnd*, *max*, or *min*) as input, creates a user tree correctly enforcing the policy. The algorithm creates the set V of material vertices and builds a graph G, where the set of vertices coincides with the set V of material vertices and the set E' of edges

includes an edge (v_i,v_j) for each pair of vertices $v_i,v_j \in V$ such that $v_i.acl \subset v_j.acl$. The algorithm then calls function **Minimum_Spanning_Tree**[3] on G and vertex v_0, with $v_0.acl = \emptyset$, and returns a minimum spanning tree of G rooted at v_0. On such a minimum spanning tree, the algorithm calls a function aimed at reducing the weight of the user tree. The function called depends on the identification of the family of heuristics given in input to the algorithm: functions **Factorize_Siblings**, **Factorize_Leaves**, and **Factorize_Vertices** correspond to the sibling-based, leaves-based, and mixed family, respectively. All these functions take a minimum spanning tree and a preference criterion as input and return a minimal user tree. We now describe the three families of heuristics in more details.

6.1. Sibling-based heuristic

The basic idea behind this family of heuristics is that for each internal vertex v of the minimum spanning tree with at least two children the set of candidate pairs $CC_v = \{\langle v_i,v_j\rangle \mid v_i \neq v_j \wedge (v,v_i), (v,v_j) \in E \wedge v_i.acl \cap v_j.acl \neq v.acl\}$ is computed. For each pair $\langle v_i,v_j\rangle$ in CC_v we then evaluate if the insertion in T of vertex v_k representing $U = v_i.acl \cap v_j.acl$ can reduce $\texttt{weight}(T)$. The three preference criteria previously discussed can be used to choose, among all pairs in CC_v, the pair $\langle v_i,v_j\rangle$ that, when v_k is possibly inserted in the tree (and it becomes the parent of vertices v_i and v_j), produces the highest reduction of the weight of the tree.

As an example, consider the weighted user tree in Figure 3 and suppose to compute the intersection between the pairs of children of the root vertex v_0. In this case, all possible intersections correspond to user A that is not already represented in the tree and therefore each intersection requires the addition of a new vertex in the tree as child of v_0 and as parent of the considered pair of children.

[3]This function may correspond to any algorithm commonly used for computing a minimum spanning tree. Our implementation is based on Prim's algorithm.

```
FACTORIZE_SIBLINGS(ST,criterion)
let ST be ⟨V,E⟩
for each v∈{v_i | v_i∈V ∧ ∃(v_i,v_j),(v_i,v_l) ∈E, v_j≠v_l} do
   repeat
      CC_v := {⟨v_i,v_j⟩ | v_i≠v_j ∧ (v,v_i), (v,v_j) ∈ E ∧ v_i.acl ∩ v_j.acl ≠ v.acl}
      if CC_v≠∅ then
         max_red := max{weight_red(v,v,v_i,v_j) | ⟨v_i,v_j⟩ ∈ CC_v}
         ST := Update_Tree(ST,CC_v,max_red,criterion)
   until CC_v=∅
return(ST)
```

Figure 7: Function **Factorize_Siblings** implementing the sibling-based heuristic

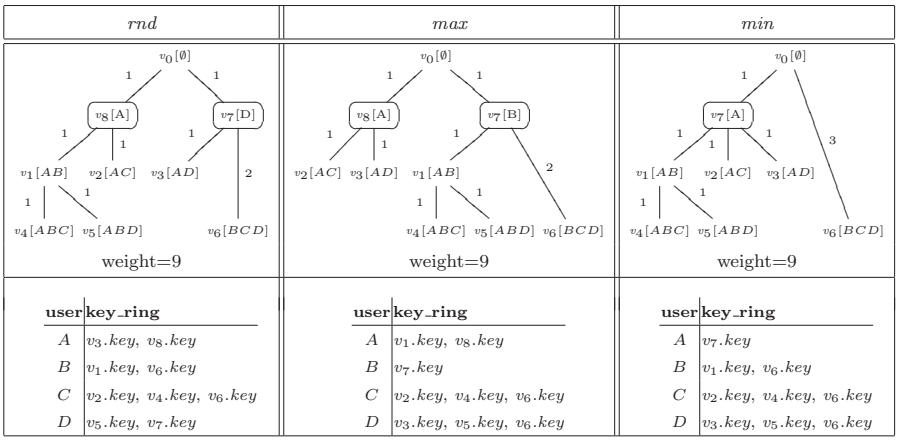
Figure 7 illustrates the pseudocode of function **Factorize_Siblings**, which implements the weight reduction strategy above described. Function **Factorize_Siblings** takes a minimum spanning tree ST and a preference criterion as input and returns a minimal user tree. For each internal vertex v in ST with at least two children (first **for** loop in the function), the function evaluates the possible insertion of a vertex v_k as a parent for each pair of children of v. At each iteration of the internal **repeat-until** loop, the set CC_v of pairs of candidate children of v is computed.[4] If CC_v is not empty, the function determines the maximum reduction *max_red* of the weight of the tree that any of these pairs can cause and calls function **Update_Tree**. This function, illustrated in Figure 8, receives as input a tree T, a set CC of pairs of vertices, the maximum weight reduction *max_red* that these pairs can cause, and a preference criterion *criterion*. Depending on the criterion given in input, function **Update_Tree** chooses a pair in CC that maximizes the weight reduction and then modifies the tree as illustrated in Figure 5. The **repeat-until** loop stops when CC_v becomes empty. Function **Factorize_Siblings** terminates when all internal vertices have been evaluated (i.e., when the **for** loop has iterated on all internal vertices).

[4]Note that CC_v does not need to be recomputed at each iteration of the **repeat-until** loop, since it can be simply updated by removing the pairs involving the vertices appearing in the pair selected by function **Update_Tree**, and possibly inserting the pairs that include the vertex (if any) added to the user tree.

UPDATE_TREE(T,CC,max_red,$criterion$)
let T be $\langle V,E\rangle$
$MC := \{\langle v_i,v_j\rangle \mid \langle v_i,v_j\rangle \in CC \wedge weight\_red(v_{p_i},v_{p_j},v_i,v_j)=max\_red\}$
case $criterion$ **of**
 rnd: choose $\langle v_i,v_j\rangle \in MC$ randomly
 max: choose $\langle v_i,v_j\rangle \in MC : |v_i.acl|+|v_j.acl|$ is maximum
 min: choose $\langle v_i,v_j\rangle \in MC : |v_i.acl|+|v_j.acl|$ is minimum
$U := v_i.acl \cap v_j.acl$
find $v_k \in V : v_k.acl = U$
case v_k **of**
 /* case 1 */
 $=v_i$: $E := E \setminus \{(v_{p_j},v_j)\} \cup \{(v_i,v_j)\}$
 $=v_j$: $E := E \setminus \{(v_{p_i},v_i)\} \cup \{(v_j,v_i)\}$
 /* case 2 */
 $\neq v_i \wedge \neq v_j$: $E := E \setminus \{(v_{p_i},v_i),(v_{p_j},v_j)\} \cup \{(v_k,v_i),(v_k,v_j)\}$
 /* case 3 */
 UNDEF: create a vertex v_k
 $v_k.acl := U$
 $V := V \cup \{v_k\}$
 if $v_{p_i}.acl \subset U \wedge v_{p_j}.acl \not\subset U$ **then** $v_{p_k} := v_{p_i}$
 if $v_{p_j}.acl \subset U \wedge v_{p_i}.acl \not\subset U$ **then** $v_{p_k} := v_{p_j}$
 if $v_{p_i}.acl \not\subset U \wedge v_{p_j}.acl \not\subset U$ **then**
 let $v_t \in \{v \in V \mid v.acl \subseteq U\} : |v_t.acl|$ is maximum
 $v_{p_k} := v_t$
 if $v_{p_i}.acl \subset U \wedge v_{p_j}.acl \subset U$ **then**
 if $|U| - |v_{p_j}.acl| \geq |U| - |v_{p_i}.acl|$ **then** $v_p := v_{p_i}$
 else $v_{p_k} := v_{p_j}$
 $E := E \setminus \{(v_{p_i},v_i),(v_{p_j},v_j)\} \cup \{(v_{p_k},v_k),(v_k,v_i),(v_k,v_j)\}$
return(T)

Figure 8: Function **Update_Tree** implementing the user tree updates in Figure 5

As an example, consider the authorization policy $\mathcal{A}$ in Figure 1(a). The table in Figure 9 is composed of three columns, one for each preference criteria (i.e., *rnd*, *max*, and *min*). Each column represents the user tree and the user key rings computed by the sibling-based heuristic following the corresponding preference criterion. Note that in Figure 9 the non-material vertices inserted by function **Factorize_Siblings** are circled.



user	key_ring
A	$v_3.key,\ v_8.key$
B	$v_1.key,\ v_6.key$
C	$v_2.key,\ v_4.key,\ v_6.key$
D	$v_5.key,\ v_7.key$

user	key_ring
A	$v_1.key,\ v_8.key$
B	$v_7.key$
C	$v_2.key,\ v_4.key,\ v_6.key$
D	$v_3.key,\ v_5.key,\ v_6.key$

user	key_ring
A	$v_7.key$
B	$v_1.key,\ v_6.key$
C	$v_2.key,\ v_4.key,\ v_6.key$
D	$v_3.key,\ v_5.key,\ v_6.key$

Figure 9: User trees and key rings computed by the sibling-based heuristic over the MST in Figure 3

6.2. Leaves-based heuristics

This family of heuristics applies the same factorization process described for the sibling-based family but on the leaves of the tree only. For each leaf vertex v of the minimum spanning tree $ST = \langle V,E \rangle$, the set of candidate pairs is computed as $CC_v = \{\langle v,v_i \rangle \mid v_i \in V \setminus (S_v \cup A_v) \wedge v.acl \cap v_i.acl \neq \emptyset\}$, where S_v and A_v are the sets of siblings and ancestors of v in ST, respectively. Like for the sibling-based family, for each pair $\langle v_i,v_j \rangle$ in CC_v we evaluate if the insertion in T of vertex v_k representing $U = v_i.acl \cap v_j.acl$ can reduce $\texttt{weight}(T)$. Among all pairs producing the same highest weight reduction, a pair is chosen according to the given preference criterion.

Figure 10 illustrates the pseudocode of function **Factorize_Leaves**, which implements the weight reduction strategy above described over a given spanning tree ST. Function **Factorize_Leaves** takes a minimum spanning tree ST and a preference criterion as input and returns a minimal user tree. For each leaf vertex v in ST, the function computes the sets S_v and A_v of the sibling and ancestor vertices of v in ST, respectively. It then determines the set CC_v of candidate pairs of vertices $\langle v,v_i \rangle$, where v_i is a vertex in the tree that is neither a sibling nor an ancestor of v and such that v and v_i represent two sets of users with at least a common element. The function computes the maximum reduction

FACTORIZE_LEAVES(ST,$criterion$)
let ST be $\langle V,E\rangle$
for each $v\in\{v_i \mid v_i\in V \wedge \nexists(v_i,v_j)\in E\}$ **do**
  $S_v := \{v_i \mid v_i\in V \wedge \exists v_p\in V:(v_p,v_i),(v_p,v)\in E\}$
  $A_v := \{v_i \mid v_i\in V \wedge \exists$ a path in ST from v_i to $v\}$
  $CC_v := \{\langle v,v_i\rangle \mid v_i\in V\setminus(S_v\cup A_v) \wedge v.acl\cap v_i.acl\neq\emptyset\}$
  $max\_red :=$ **max**$\{weight\_red(v_{p_i},v_{p_j},v_i,v_j) \mid \langle v_i,v_j\rangle \in CC_v\}$
  if $max\_red>0$ **then** $ST :=$ **Update_Tree**(ST,CC_v,max_red,$criterion$)
return(ST)

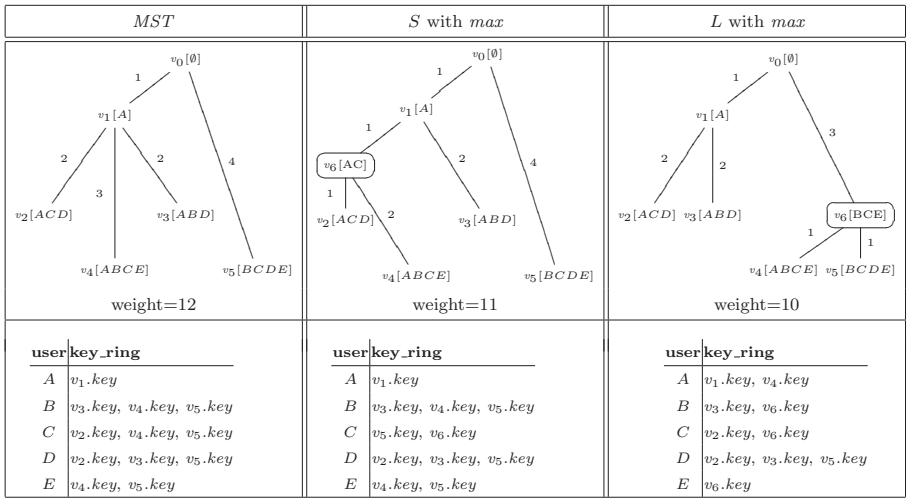
Figure 10: Function **Factorize_Leaves** implementing the leaves-based heuristic

	$\mathbf{r_1}$	$\mathbf{r_2}$	$\mathbf{r_3}$	$\mathbf{r_4}$	$\mathbf{r_5}$
A	1	1	1	1	0
B	0	0	1	1	1
C	0	1	1	0	1
D	0	1	0	1	1
E	0	0	1	0	1

Figure 11: An example of access matrix

max_red of the weight of the tree that any of the pairs in CC_v can cause and, if such a reduction is a positive value, calls function **Update_Tree** that works as discussed for the previous family of heuristics. Function **Factorize_Leaves** terminates when all the leaves in the tree have been evaluated (i.e., when the **for** loop has iterated on all leaves).

We note here that, since the sibling-based and leaves-based families work on disjoint sets of candidate pairs of vertices, they may compute different user trees with different weights. As an example, consider the access control policy represented by the access matrix in Figure 11. The table in Figure 12 represents the user trees and the corresponding key rings computed by the two heuristics. The first column in the table reports the minimum spanning tree over material vertices. The second and the third columns report the user tree and the user key rings computed by the sibling-based heuristic and the leaves-based heuristic, respectively, with the *max* preference criterion. Note that, in this case, the leaves-based heuristic computes a user tree with a lower weight than the one



user	key_ring
A	$v_1.key$
B	$v_3.key,\ v_4.key,\ v_5.key$
C	$v_2.key,\ v_4.key,\ v_5.key$
D	$v_2.key,\ v_3.key,\ v_5.key$
E	$v_4.key,\ v_5.key$

user	key_ring
A	$v_1.key$
B	$v_3.key,\ v_4.key,\ v_5.key$
C	$v_5.key,\ v_6.key$
D	$v_2.key,\ v_3.key,\ v_5.key$
E	$v_4.key,\ v_5.key$

user	key_ring
A	$v_1.key,\ v_4.key$
B	$v_3.key,\ v_6.key$
C	$v_2.key,\ v_6.key$
D	$v_2.key,\ v_3.key,\ v_5.key$
E	$v_6.key$

Figure 12: User trees and key rings computed by the sibling-based and leaves-based heuristics for the access matrix in Figure 11

computed by the sibling-based heuristic.

6.3. Mixed heuristics

This family of heuristics is obtained by combining the strategies proposed for the sibling-based and leaves-based families. In this case, the computation of a minimal user tree is performed by taking into consideration a set of candidate pairs which is obtained by merging the candidate pairs considered by the two previous families. Again, one of the three preference criteria previously discussed can be applied for selecting one of the pairs that produces the highest reduction of the weight of the tree.

Figure 13 illustrates the pseudocode of function **Factorize_Vertices**, implementing a weight reduction process operating on both the internal vertices and the leaves of a given spanning tree ST. Function **Factorize_Vertices** takes a minimum spanning tree ST and a preference criterion as input and returns a minimal user tree. At each iteration of the **repeat-until** loop, the function builds the set of all possible candidate pairs of vertices CC for the current topol-

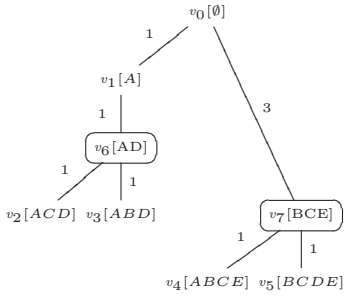
FACTORIZE_VERTICES(ST,$criterion$)
let ST be $\langle V,E\rangle$
repeat
  $CC := \emptyset$
  for each $v \in \{v_i \mid v_i \in V \wedge \exists (v_i,v_j) \in E\}$ **do**
    $CC := CC \cup \{\langle v_i,v_j\rangle \mid v_i \neq v_j \wedge (v,v_i),\ (v,v_j) \in E \wedge v_i.acl \cap v_j.acl \neq v.acl\}$
  for each $v \in \{v_i \mid v_i \in V \wedge \nexists (v_i,v_j) \in E\}$ **do**
    $S_v := \{v_i \mid v_i \in V \wedge \exists v_p \in V : (v_p,v_i),(v_p,v) \in E\}$
    $A_v := \{v_i \mid v_i \in V \wedge \exists$ a path in ST from v_i to $v\}$
    $CC := CC \cup \{\langle v,v_i\rangle \mid v_i \in V \setminus (S_v \cup A_v) \wedge v.acl \cap v_i.acl \neq \emptyset \wedge weight\_red(v_p,v_{p_i},v,v_i) \geq 0\}$
  if $CC \neq \emptyset$ **then**
    $max\_red := \mathbf{max}\{weight\_red(v_{p_i},v_{p_j},v_i,v_j) \mid \langle v_i,v_j\rangle \in CC\}$
    $ST :=$ **Update_Tree**(ST,CC,max_red,$criterion$)
until $CC = \emptyset$
return(ST)

Figure 13: Function **Factorize_Vertices** implementing the factorization process for both sibling and leaf vertices

ogy of the tree.[5] For each internal vertex v in ST (first **for** loop in the function), the function computes the set of candidate pairs $\langle v_i,v_j\rangle$ of children of v such that $v_i.acl \cap v_j.acl \neq v.acl$ and inserts these pairs in CC. Then, for each leaf v in ST (second **for** loop in the function), it computes and inserts in CC the set of pairs $\langle v,v_i\rangle$, where v_i is a vertex in the tree that is neither a sibling nor an ancestor of v and such that $v.acl$ and $v_i.acl$ have at least a user in common and cause a positive weight reduction. If CC is not empty, the function determines the maximum reduction max_red of the weight of the tree that any of the pairs in CC can cause and calls function **Update_Tree**. The function terminates when CC is empty and, therefore, when the weight of the user tree cannot be further reduced.

To conclude the discussion about the mixed heuristic we note that, since this heuristic takes advantage of both the strategies adopted by the sibling-

[5] As already pointed out for the sibling-based heuristic, the set CC of candidate pairs of vertices does not need to be recomputed at each iteration of the **repeat-until** loop, but it can be updated considering the updates to the topology of the tree performed by function **Update_Tree**.



weight=9

(a)

user	key_ring
A	$v_1.key,\ v_4.key$
B	$v_3.key,\ v_7.key$
C	$v_2.key,\ v_7.key$
D	$v_5.key,\ v_6.key$
E	$v_7.key$

(b)

Figure 14: User tree (a) and key rings (b) computed by the mixed heuristics for the access matrix in Figure 11

based and the leaves-based heuristics, it computes a user tree that can have a weight less that the weight of the trees computed by the other two heuristics. As an example, consider the access control policy represented by the access matrix in Figure 11. Figure 14 illustrates the user tree and the corresponding user key rings computed by the mixed heuristic, when the *max* preference criterion is applied. It is immediate to see that the computed solution merges the benefits of both the sibling-based and the leaves-based heuristics, thus producing a tree with a lower weight. We also note that if on one hand the solution computed by this heuristic is in general better than the solution computed by the other heuristics, on the other hand the computation of all the possible candidate pairs of vertices is more expensive than the corresponding process of the other heuristics. However, as the experimental results described in Section 7 show, the computational time of this third heuristic is acceptable, while the quality of the solution outperforms both the sibling-based and the leaves-based heuristics.

Number of resources	5 users				6 users				10 users			
	tot	*min*	*max*	*rnd*	*tot*	*min*	*max*	*rnd*	*tot*	*min*	*max*	*rnd*
5	937	932	924	927	865	863	830	834	828	802	692	709
10	879	872	849	849	778	693	648	657	709	633	219	269
15	947	946	936	936	735	720	637	634	729	685	168	205
20	987	983	979	982	780	751	671	685	717	626	118	120
25	1000	998	998	998	781	763	705	714	694	598	90	131
30	1000	1000	1000	1000	846	835	808	815	626	543	77	131
35					891	886	853	858	554	484	64	104
40					943	940	924	928	570	538	59	85
45					981	978	966	973	501	488	57	68
50					993	992	989	991	501	478	55	67

Figure 15: Number of times that the sibling-based heuristic with different preference criteria is better than the heuristic in [5]

7. Experimental results

In large scale access control systems with a huge number of users and resources, the time needed to set the right key assignment scheme can be considerably large. A correct evaluation of the performance of the proposed heuristics is requested to provide the system designer with a valid set of tools she can use for the selection of the strategy that provides the best trade-off between the quality of the solution returned by the selected heuristic and the amount of time invested in obtaining such a result. The heuristics have been implemented by using Scilab [15] Version 4-1 on Windows XP operating system. We ran the experiments on a computer equipped with Centrino 1,7 Mhz CPU, with randomly generated access matrices, considering different numbers of users and resources in the system.

A first set of experiments has been devoted to compare the solutions returned by our three families of heuristics on the basis of the *rnd*, *max*, and *min* preference criteria with the heuristic in [5]. The goal is to verify whether there is a preference criterion that works better than the others. In these experiments, the number of users varies from 5 to 10 and, for a given number n of users, the number of resources varies from 5 to 2^n, since 2^n is the maximum number of vertices in the user tree. For each configuration (i.e., for a fixed number of

#	[5]			S			L			M		
Res.	$d=0$	$d=1,2$	$d>2$	$d=0$	$d=1,2$	$d>2$	$d=0$	$d=1,2$	$d>2$	$d=0$	$d=1,2$	$d>2$
5	93	7	0	82.3	16.2	1.5	1.8	11.9	86.3	92.8	7.2	0
10	50.8	44.1	5,1	61.3	34.5	4,2	0	3,5	96,5	79,4	20.1	0.5
15	35.7	46.4	17.9	46.4	42.2	11.4	0.1	1.7	98.2	75.5	23.4	5.1
20	29.2	43.1	27.7	36.9	47	16.1	0.1	0.5	99.4	76.2	22.1	1.7
25	30.3	38.3	31.4	31.5	45.5	23	0.1	0.3	99.6	72.8	24.1	3.1
30	29	37.4	33.6	28.4	42.2	29.4	0	0	100	74	23.6	2.4
35	31.3	34.7	34	24.1	39.5	36.4	0	0.3	99.7	74.5	21.4	4.1
40	34	34.9	31.1	20.6	40.2	39.2	0	0.4	99.6	70.5	24.4	5.1
45	38.1	33.9	28	16.8	35.4	47.8	0	0.1	99.9	70.4	23.2	6.4
50	36	37.2	26.8	15	33.5	51.5	0	0	100	69.5	23.4	7.1

Figure 16: Percentage of times each heuristic returns a solution at distance d from the lowest weight solution computed.

users and resources), we generated 1000 access matrices and, for each access matrix, we applied the heuristic proposed in [5] and the three families of heuristics previously discussed, considering all possible preference criteria. Since the experimental results for the three families of heuristics are quite similar, Figure 15 illustrates only the results for the sibling-based heuristic. This figure reports the number of times the sibling-based heuristic, adopting the preference criterion indicated as the label of the column, computes a user tree with a weight lower than or equal to the weight of the tree obtained by running the heuristic in [5]. Column *tot* lists the number of times that the sibling-based heuristic (adopting one of the three preference criteria) returns a solution better than the one returned by the heuristic in [5]. By comparing each column *tot* with the corresponding *min*, *max*, and *rnd* columns we see that, in the majority of the cases, the better solution is obtained with the *min* criterion while there are only few cases where the *max* or *rnd* criteria perform better. This behavior has been confirmed by the results obtained by running the same experiments with the other two families of heuristics (leaves-based and mixed).

A second set of experiments has been run to compare the solutions returned by our three families of heuristics adopting the *min* preference criterion (which results to be the better criterion) with the heuristic in [5]. The goal is to evaluate

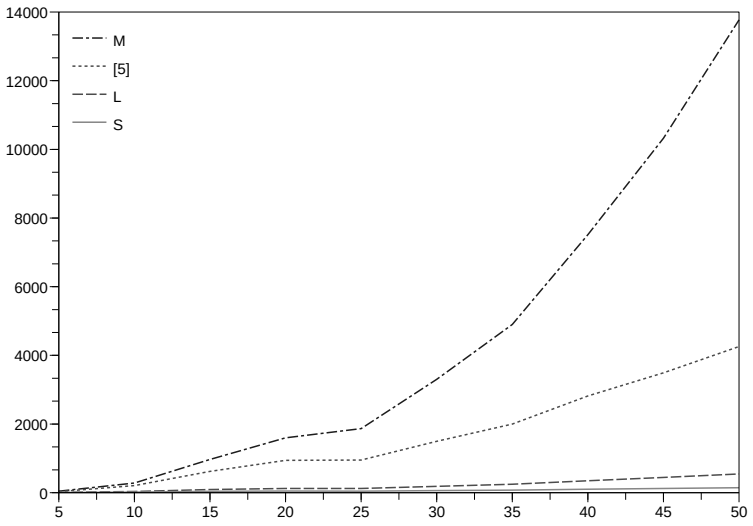


Figure 17: Execution time (in seconds) for the heuristics for 10 users (1000 runs)

the famility of heuristics that works better and to analyze the execution time of all the heuristics. To this purpose, we considered configurations with 10 users and with a number of resources that varies in the range 5 to 50. Like in the previous set of experiments, for each configuration we generated 1000 access matrices. Figure 16 reports the results obtained. Each column in the table is dedicated to a different heuristic and gives the percentage of times the weight of the tree computed with the heuristic labeling the column is at distance 0, included between 1 and 2, or greater than 2 from the weight of the better user tree (i.e., the tree with the lowest weight) computed by one of the four implemented algorithms. On the basis of the data reported in Figure 16, we observe that the sibling-based and the mixed heuristics compute a solution that, in many cases, is better than the one returned by the heuristic in [5]. The leaves-based heuristic provides worse quality solutions, even if in few cases it returns the user tree with the lowest weight among the four heuristics considered in the comparison.

Figure 17 reports the execution times for all the considered configurations. The execution time is composed of the time for the construction of the graph G (see Section 4), the time for the construction of the minimum spanning tree on G, and the time for the execution of the selected heuristic. As shown in the figure, the sibling-based and the leaves-based heuristics are very efficient compared with the heuristic in [5]. Considering that, in most of the cases, the sibling-based heuristic returns a solution better than or equal to the one computed by the heuristic in [5], it represents a good trade-off between quality of the solution and execution time. Compared with the sibling-based and leaf-based heuristics, the mixed heuristic returns better quality solutions, computing the user tree with the lowest weight for most of the considered access matrices. However, the execution time of the mixed heuristic is higher than the time requested by the other heuristics.

We have also performed experiments on the integer linear programming formulation of the problem. The experiments showed high variability of the time necessary to solve the ILP problem, even assuming a fixed number of users and resources in the system. This variability is due to the fact that the execution time depends on the number of constraints and variables of the ILP problem, which is dictated by the number of authorizations. For instance, with a configuration of 10 users and 10 resources we randomly generated 1000 access matrices that result in 1000 different instances of the ILP problem, where the number of constraints and variables vary in the range of 35–80 and 40–140, respectively. The corresponding execution time varies in the range of $0,4$–8000 seconds. We however observed that, on average, the number of constraints and variables in the ILP problem grows with the number of users and resources in the system. Also, as expected, the time necessary to solve the ILP problem is always higher than the execution time of our heuristics, while the weight of the user tree is often very close to the weight of the trees computed by the heuristics.

8. Related work

Previous related work is in the area of the "database-as-a-service" paradigm [3, 16], which considers the problem of database outsourcing with the main goal of enabling data owners to outsource their data to, possibly non fully trusted, third parties. This new scenario requires the evaluation of different security issues, which have been recently addressed in the literature (e.g., evaluation of queries on encrypted data, inference exposure control, integrity). Most of the existing efforts on this topic however focus on the proposal of techniques for the evaluation of queries on encrypted outsourced data, with the goal of supporting different SQL clauses and different kinds of conditions over attributes [2, 3, 4, 16, 17, 18]. These proposals are based on the definition of indexing information, which is stored together with the encrypted data, that reflects certain characteristics of the original data and that can therefore be exploited for partial query evaluation at the server side. The first proposals in this direction [3, 16] support simple queries with equality conditions only. In [2] the authors propose a solution, based on the B+ tree data structure used by relational DBMSs for physically index data, that supports range queries besides equality queries. They also first provide an evaluation of the inference exposure that the public availability of indexing information can cause, proving that even a limited number of indexes can greatly facilitate the task for an adversary that wants to violate the confidentiality provided by encryption. A recent proposal, illustrated in [4], tries to minimize inference exposure by defining indexes with an almost flat distribution of the frequencies of the values in the index domain. This proposal exploits B-trees for supporting both equality and range queries.

A few research efforts have directly tackled the issues of access control in an outsourced scenario. In [8] the authors first present a framework for enforcing access control on published XML documents by using different cryptographic keys over different portions of the XML tree and by introducing special metadata nodes in the structure. In [5] the authors instead address the issue of access control enforcement in the database outsourcing context, by exploiting selective

encryption and hierarchical key assignment schemes on trees. To grant users efficient access to resources, the authors propose an algorithm that minimizes the number of secret keys in users' key rings. A related line of research has studied solutions for efficient policy updates [6, 19]. Here, two layers of encryption are imposed on data: the inner layer is imposed by the owner for providing initial protection, the outer layer is imposed by the server to reflect policy modifications (i.e., grant/revoke of authorizations). Even if these works exploit DAG key derivation hierarchies, they can be easily adapted to the solution proposed in this paper.

9. Conclusions and future work

There is an emerging trend towards scenarios where resource management is outsourced to an external service providing storage capabilities and high-bandwidth distribution channels. In this context, selective dissemination of data requires enforcing measures to protect the resource confidentiality from both unauthorized users and "honest-but-curious" servers. In this paper, we addressed this issue by integrating access control and encryption and by exploiting key derivation methods as a way for minimizing the number of keys distributed to users. In particular, we presented three families of heuristics for building a key derivation tree that correctly enforces the authorization policy defined by the data owner and an integer linear programming formulation of the minimization problem. The experimental results obtained by the implementation of the heuristics prove their efficiency with respect to previous solutions.

Future work includes the investigation of the problem of updating the derivation tree upon changes in the authorization policy. Possible directions include: *1)* re-executing the heuristics every time there is a change in the authorization, *2)* performing a simple adaptation of the tree to reflect the authorization changes (and possibly re-execute the heuristics periodically if the quality of the tree degenerates), or *3)* applying over-encryption solutions [6].

Acknowledgments

This work was supported in part by the EU within the FP7 under grant 216483 "PrimeLife"; by the EU within the FP7 under contract ICT-2007-216646 "ECRYPT II"; and by the EU within the FP6 under contract FP6-1596 "AEOLUS".

References

[1] C. Blundo, S. Cimato, S. De Capitani di Vimercati, A. De Santis, S. Foresti, S. Paraboschi, P. Samarati, Efficient key management for enforcing access control in outsourced scenarios, in: Proc. of IFIP SEC 2009, Cyprus, 2009.

[2] A. Ceselli, E. Damiani, S. De Capitani di Vimercati, S. Jajodia, S. Paraboschi, P. Samarati, Modeling and assessing inference exposure in encrypted databases, ACM Transactions on Information and System Security 8 (1) (2005) 119–152.

[3] H. Hacigümüs, B. Iyer, S. Mehrotra, C. Li, Executing SQL over encrypted data in the database-service-provider model, in: Proc. of SIGMOD 2002, Madison, USA, 2002.

[4] H. Wang, L. V. Lakshmanan, Efficient secure query evaluation over encrypted XML databases, in: Proc. of the 32nd VLDB 2006 Conference, Seoul, Korea, 2006.

[5] E. Damiani, S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, P. Samarati, Selective data encryption in outsourced dynamic environments, in: Proc. of VODCA 2006, Bertinoro, Italy, 2006.

[6] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, P. Samarati, Over-encryption: Management of access control evolution on outsourced data, in: Proc. of the 33rd VLDB Conference, Vienna, Austria, 2007.

[7] E. Damiani, S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, P. Samarati, An experimental evaluation of multi-key strategies for data outsourcing, in: Proc. of IFIP SEC 2007, Sandton, South Africa, 2007.

[8] G. Miklau, D. Suciu, Controlling access to published data using cryptography, in: Proc. of the 29th VLDB Conference, Berlin, Germany, 2003.

[9] S. Akl, P. Taylor, Cryptographic solution to a problem of access control in a hierarchy, ACM Transactions on Computer System 1 (3) (1983) 239–248.

[10] M. Atallah, K. Frikken, M. Blanton, Dynamic and efficient key management for access hierarchies, in: Proc. of the ACM CCS05, Alexandria, USA, 2005.

[11] G. Ateniese, A. De Santis, A. Ferrara, B. Masucci, Provably-secure time-bound hierarchical key assignment schemes, in: Proc. of ACM CCS06, Alexandria, USA, 2006.

[12] S. MacKinnon, P. Taylor, H. Meijer, S.Akl, An optimal algorithm for assigning cryptographic keys to control access in a hierarchy, IEEE Transactions on Computers 34 (9) (1985) 797–802.

[13] R. Sandhu, Cryptographic implementation of a tree hierarchy for access control, Information Processing Letters 27 (2) (1988) 95–98.

[14] A. De Santis, A. Ferrara, B. Masucci, Cryptographic key assignment schemes for any access control policy, Information Processing Letters 92 (4) (2004) 199–205.

[15] Scilab Consortium, Scilab, the open source platform for numerical computation, http://www.scilab.org, V. 4-1.

[16] H. Hacigümüs, B. Iyer, S. Mehrotra, Providing database as a service, in: Proc. of ICDE'02, San Jose, USA, 2002.

[17] R. Agrawal, J. Kierman, R. Srikant, Y. Xu, Order preserving encryption for numeric data, in: Proc. of SIGMOD 2004, Paris, France, 2004.

[18] E. Shmueli, R. Waisenberg, Y. Elovici, E. Gudes, Designing secure indexes for encrypted databases, in: Proc. of IFIP DBSec'05, Storrs, USA, 2005.

[19] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, P. Samarati, A data outsourcing architecture combining cryptography and access control, in: Proc. of CSAW 2007, Fairfax, USA, 2007.

[20] M. Garey, D. Johnson, Computers and Intractability; a Guide to the Theory of *NP*-Completeness, W.H. Freeman, 1979.

A. NP-hardness of the minimization problem

To prove the hardness of Problem 3.1, we reduce it, in polynomial time, to an instance of the *Vertex Cover* problem. The problem of determining the minimum vertex cover of an undirected graph is a classical NP-hard optimization problem [20] that can be formulated as follows.

Definition A.1. Vertex Cover. *Given an undirected graph $G = (V, E)$, find a smallest subset $S \subseteq V$ such that every edge in E is incident on at least one of the vertices in S. The set S is called a* vertex cover *of graph G.*

Before proving the NP-hardness of Problem 3.1, we prove the following lemma

Lemma A.1. *Let $\mathcal{A}$ be an access matrix over a set $\mathcal{U}$ of users and a set $\mathcal{R}$ of resources, and $G_{\mathcal{U}} = \langle V_{\mathcal{U}}, E_{\mathcal{U}} \rangle$ be the user graph over $\mathcal{U}$. The computation of a minimum user tree T subgraph of $G_{\mathcal{U}}$ that correctly enforces $\mathcal{A}$ is equivalent to the computation of a minimum user tree T subgraph of $G'_{\mathcal{U}} = (\mathcal{I}(\mathcal{M}), E'_{\mathcal{U}})$, where $\mathcal{I}(\mathcal{M}) \subseteq P(\mathcal{U})$ is the smallest set including all material vertices $\mathcal{M}$ for $\mathcal{A}$ and closed under the intersection operator, and $E'_{\mathcal{U}} = \{(v_i, v_j) \mid v_i.acl \subset v_j.acl$ and $v_i, v_j \in \mathcal{I}(\mathcal{M})\}$.*

Proof. For the sake of contradiction, assume that a vertex v is such that $v \notin \mathcal{I}(\mathcal{M})$ and v belongs to T. Notice that, only one vertex $w \in \mathcal{M}$ can belong to the subtree T_v rooted at v (otherwise, vertex v has to belong to $\mathcal{I}(\mathcal{M})$). Hence, we can construct a new user tree T' by removing subtree T_v and connecting vertex w to v's parent. The new user tree T' is such that $\texttt{weight}(T') \leq \texttt{weight}(T)$. We can then conclude that any vertex $v \notin \mathcal{I}(\mathcal{M})$ does not belong to a minimum user tree. ■

We also note that to compute a minimum user tree, instead of considering the graph $G'_{\mathcal{U}} = (\mathcal{I}(\mathcal{M}), E'_{\mathcal{U}})$, we can consider its transitive reduction[6] $\widehat{G}_{\mathcal{U}} =$

[6]The transitive reduction of a graph $G = (V, E)$ is the smallest subgraph $G' = (V, E')$ such that that the transitive closure of G' is the same as the transitive closure of G.

$(\mathcal{I}(\mathcal{M}), \widehat{E}_{\mathcal{U}})$. Since the graph $G'_{\mathcal{U}}$ is finite and acyclic, then it is known that its transitive reduction $\widehat{G}_{\mathcal{U}}$ is unique. We are now ready to prove the NP-hardness of Problem 3.1

Theorem 4.1 (NP-hardness). *Let $\mathcal{A}$ be an access matrix over a set $\mathcal{U}$ of users and a set $\mathcal{R}$ of resources. The problem of computing a minimum weight user tree T correctly enforcing $\mathcal{A}$ is NP-hard.*

Proof. Let $G = (V_G, E_G)$ be an undirected graph. Construct the access matrix $\mathcal{A}$ over the set $\mathcal{U}$ of users and the set $\mathcal{R}$ of resources such that, for any vertex $v \in V_G$ there is a user $u_v \in \mathcal{U}$ and for any edge $(v, w) \in E_G$ there is a resource r with $acl(r) = \{u_v, u_w\}$. This reduction can be done in polynomial time. Notice that, by a little abuse of notation: the set of material vertices $\mathcal{M}$ is equal to $\{v_\emptyset\} \cup E_G$; its closure, with respect to the intersection, $\mathcal{I}(\mathcal{M})$ is equal to $\{v_\emptyset\} \cup V_G \cup E_G$; and $|E_G| = |\mathcal{R}|$.
Let T be the minimum user tree correctly enforcing $\mathcal{A}$ computed over the graph $\widehat{G}_{\mathcal{U}} = (\mathcal{I}(\mathcal{M}), \widehat{E}_{\mathcal{U}})$. Since all edges in $\widehat{E}_{\mathcal{U}}$ have weight one and all vertices in $\mathcal{M}$ must belong to T, then T contains $t + |\mathcal{M}| = t + |\mathcal{R}| + 1$ vertices, for some non-negative integer t, and $\texttt{weight}(T) = t + |\mathcal{R}|$. Such an observation implies that the remaining t vertices in T are from V_G and they are connected to all vertices representing $\mathcal{R}$ (e.g., E_G) – such t vertices correspond to a vertex cover in G. Thus, the graph G has a vertex cover of size t if and only if the minimum user tree T correctly enforcing $\mathcal{A}$ computed over the graph $\widehat{G}_{\mathcal{U}} = (\mathcal{I}(\mathcal{M}), \widehat{E}_{\mathcal{U}})$ has cost equal to $t+|\mathcal{R}|$. This proves that the problem of computing a minimum user tree T correctly enforcing $\mathcal{A}$ is NP-hard. ■