
Encryption Policies for Regulating Access to Outsourced Data

SABRINA DE CAPITANI DI VIMERCATI, SARA FORESTI

Università degli Studi di Milano

SUSHIL JAJODIA

George Mason University

STEFANO PARABOSCHI

Università degli Studi di Bergamo

and

PIERANGELA SAMARATI

Università degli Studi di Milano

Current access control models typically assume that resources are under the strict custody of a trusted party, which monitors each access request to verify if it is compliant with the specified access control policy. There are many scenarios where this approach is becoming no longer adequate. Many clear trends in Web technology are creating a need for owners of sensitive information to manage access to it by legitimate users using the services of *honest but curious* third parties, that is, parties trusted with providing the required service but not authorized to read the actual data content. In this scenario, the data owner encrypts the data before outsourcing and stores them at the server. Only the data owner and users with knowledge of the key will be able to decrypt the data. Possible access authorizations are to be enforced by the owner. In this paper, we address the problem of enforcing selective access on outsourced data without need of involving the owner in the access control process. The solution puts forward a novel approach that combines cryptography with authorizations, thus enforcing access control via *selective encryption*. The paper presents a formal model for access control management and illustrates how an authorization policy can be translated into an equivalent encryption policy while minimizing the amount of keys and cryptographic tokens to be managed. The paper also introduces a two-layer encryption approach that allows the data owner to outsource, besides the data, the complete management of the authorization policy itself, thus providing efficiency and scalability in dealing with policy updates. We also discuss experimental results showing that our approach is able to efficiently manage complex scenarios.

This paper extends the previous work by the authors appeared under the title “Over-encryption: Management of Access Control Evolution on Outsourced Data,” in Proc. of VLDB 2007, Sep. 2007, Vienna, Austria [De Capitani di Vimercati et al. 2007].

This work was supported in part by the EU within the FP7 under grant 216483 “PrimeLife”; by NSF grants CT-20013A, CT-0716567, CT-0716323, and CT-0627493; by AFOSR grants FA9550-07-1-0527, FA9550-09-1-0421, and FA9550-08-1-0157; and by ARO grant W911NF-09-01-0352.

Authors’ addresses: S. De Capitani di Vimercati, S. Foresti, P. Samarati, Università degli Studi di Milano, 26013 Crema, Italy, email: {firstname.lastname}@unimi.it; S. Jajodia, George Mason University, Fairfax, VA 22030-4444, USA, email: jajodia@gmu.edu; S. Paraboschi, Università degli Studi di Bergamo, 24044 Dalmine, Italy, email: parabosc@unibg.it.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© 20YY ACM 0362-5915/20YY/0300-0001 \$5.00

Categories and Subject Descriptors: H.2.7 [**Database Management**]: Database Administration—*Security, integrity, and protection*; D.4.6 [**Operating System**]: Security and Protection—*Access Control*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms: Security, Design, Management

Additional Key Words and Phrases: Data outsourcing, encryption policy, privacy

1. INTRODUCTION

Contrary to the vision of a few years ago, where many predicted that Internet users would have in a short time exploited the availability of pervasive high-bandwidth network connections to activate their own servers, users are today, with increasing frequency, resorting to service providers for disseminating and sharing resources they want to make available to others. This trend supports the view that service providers will be more and more requested to be responsible for the storage and the efficient and reliable distribution of content produced by others, realizing a “data outsourcing” architecture on a wide scale. The situation is particularly clear when we look at the success of services like YouTube, Flickr, Blogger, MySpace. These services typically assume that the server has complete access to the stored resources and therefore have limited use for all those scenarios where the server cannot be granted such an access. In many applications, in fact, the server is considered *honest but curious*, that is, is relied upon for the availability of outsourced data but is not authorized to see the actual data content. The most convincing and emerging solutions for these scenarios assume that the *data owner* encrypts data before sending them to the *server* for storage and gives the corresponding key to *users* authorized to access the data (see Figure 1). In this way, the confidentiality of information does not rely on an implicit assumption of trust on the server or on the legal protection offered by specific service contracts, but instead relies on the technical guarantees provided by encryption techniques. Typically, these solutions [Ceselli et al. 2005; Hacigümüs et al. 2002(a); Hacigümüs et al. 2002(b)] focus on the problem of executing queries directly on the encrypted data by exploiting associated metadata and do not explicitly address the problem of supporting different keys or different access privileges (authorizations) for different users.

In this paper, we present an approach to allow selective access to encrypted outsourced data by users. The basic idea behind our approach is to integrate access control and encryption, thus encrypting the data to be outsourced with different keys depending on the authorizations to be enforced on the data. Although it is usually advisable to leave authorization-based access control and cryptographic protections separate, as encryption is traditionally considered a mechanism and should not be adopted in model definition [Samarati and De Capitani di Vimercati 2001], such a combination proves successful and powerful in the data outsourcing scenario. In particular, since neither the data owner nor the remote server can enforce the authorization policy, for either efficiency or security reasons, respectively, implementing selective access via the stored data themselves appears promising. The idea of applying encryption in a selective way depending on the authorizations

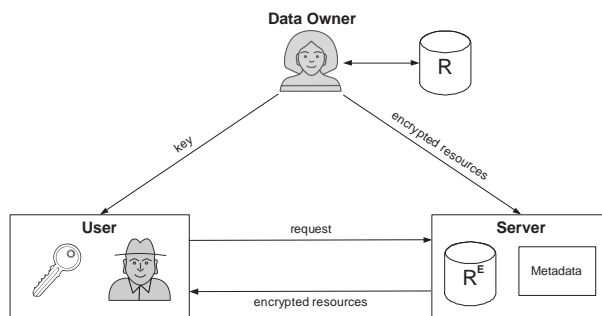


Fig. 1. Outsourcing scenario

holding on data is in itself not new and has been investigated in the context of XML documents, where different keys can be used to encrypt different portions of the XML tree [Miklau and Suciu 2003; XML Encryption Syntax and Processing 2002]. The solutions developed in this context have investigated the management of multiple keys within a single documents but have not addressed the problems related to the definition, management, and evolution of the authorization policy, and therefore of the corresponding encryption, which are the focus and contributions of our work.

The goal of our solution is to translate an authorization policy to be enforced in an equivalent encryption policy regulating which data are encrypted with which key and regulating key release to users. We are guided by the principles of releasing at most one key to each user, and encrypting each resource at most once. To achieve them, we exploit a hierarchical organization of keys allowing the derivation of keys from other keys and public tokens [Akl and Taylor 1983; Atallah et al. 2005; Crampton et al. 2006; Sandhu 1987]. Our goal is then to minimize the number of tokens to be generated and maintained. We also address the problem of enforcing updates to the authorization policy while limiting the cost in terms of bandwidth and computational power (providing a two layer approach that avoids the need for the owner to download the affected resources, decrypt and re-encrypt them, and reload their new versions). Our solution to this problem is particularly appealing as it allows delegating to the server the complete management, not only the enforcement, of the authorization policy. It is important to note that our basic technique is independent from any specific data model and it does not rely on any specific authorization language. In fact, the translation of the authorization policy into a key derivation scheme is completely transparent to the owner. An important strength of our solution is that it does not substitute the current proposals, rather it complements them, enabling them to support encryption in a selective form and easily enforce dynamic policy changes.

The contributions of this paper can be summarized as follows. First (Section 2 and Section 3), we propose a formal model for representing an authorization policy through an *equivalent encryption policy*. We also introduce the definition of minimum encryption policy and we prove that the problem of computing a minimum encryption policy is NP-Hard. Second (Section 4 and Section 5), we present a heuristic algorithm for computing a minimal encryption policy equivalent to a given

authorization policy. We then describe how authorization policy changes can be supported while leaving to the data owner the control on the authorization policy management. Third (Section 6 and Section 7), building on the base model, we propose the use of a two-layer approach to outsource, besides the resource storage and dissemination, the authorization policy management: the first layer of encryption is applied by the data owner at initialization time (when releasing the resources for outsourcing), the second layer of encryption is applied by the service itself to take care of dynamic policy changes. We then characterize the different views of the resources by different users and evaluate potential risks of information exposures (Section 8). We also illustrate experimental results (Section 9) confirming the benefits of our proposal in terms of token reduction and efficiency. Finally, we discuss related work (Section 10) and give our concluding remarks (Section 11). The proofs of the theorems and lemmas are reported in the *electronic Appendix*.

2. AUTHORIZATION AND ENCRYPTION POLICIES

In this section, we describe our model for expressing an authorization policy through encryption and illustrate how users interact with the server to access the outsourced data.

2.1 Authorization policy

We assume that the data owner defines a discretionary authorization policy to regulate access to the outsourced resources, where a resource could be a file, a relational table, or even a tuple within a relation. We assume access by users to the outsourced resources to be read-only, while write operations are to be performed at the owner’s site (typically by the owner itself). Note that write operations require re-encryption and re-uploading of the involved resources on the server. Permissions that need to be enforced through encryption are of the form $\langle user, resource \rangle$.¹ Given a set \mathcal{U} of users and a set \mathcal{R} of resources, we define an authorization policy over \mathcal{U} and \mathcal{R} as follows.

Definition 2.1 Authorization policy. Let \mathcal{U} and \mathcal{R} be the set of users and resources in the system, respectively. An *authorization policy* over \mathcal{U} and \mathcal{R} , denoted \mathcal{A} , is a triple $\langle \mathcal{U}, \mathcal{R}, \mathcal{P} \rangle$, where \mathcal{P} is a set of permissions of the form $\langle u, r \rangle$, with $u \in \mathcal{U}$ and $r \in \mathcal{R}$, stating the accesses to be allowed.

The set of permissions can be represented through an access matrix $\mathcal{M}_{\mathcal{A}}$, with a row for each user $u \in \mathcal{U}$ and a column for each resource $r \in \mathcal{R}$ [Samarati and De Capitani di Vimercati 2001]. Each entry $\mathcal{M}_{\mathcal{A}}[u, r]$ is set to 1 if u can access r ; it is set to 0 otherwise. Given an access matrix $\mathcal{M}_{\mathcal{A}}$ over sets \mathcal{U} and \mathcal{R} , $acl(r)$ denotes the *access control list* of r (i.e., the set of users that can access r).

We model an authorization policy as a directed and bipartite graph $\mathcal{G}_{\mathcal{A}}$ having a vertex for each user $u \in \mathcal{U}$ and for each resource $r \in \mathcal{R}$, and an edge from u to r for each permission $\langle u, r \rangle \in \mathcal{P}$ to be enforced. Since our modeling of the problem and its solution will exploit graphs, we explicitly define $\mathcal{G}_{\mathcal{A}}$ as follows.

¹For the sake of simplicity, we do not deal with the fact that permissions can be specified for groups of users and groups of resources. Our approach supports dynamic grouping, thus subsuming any statically defined group.

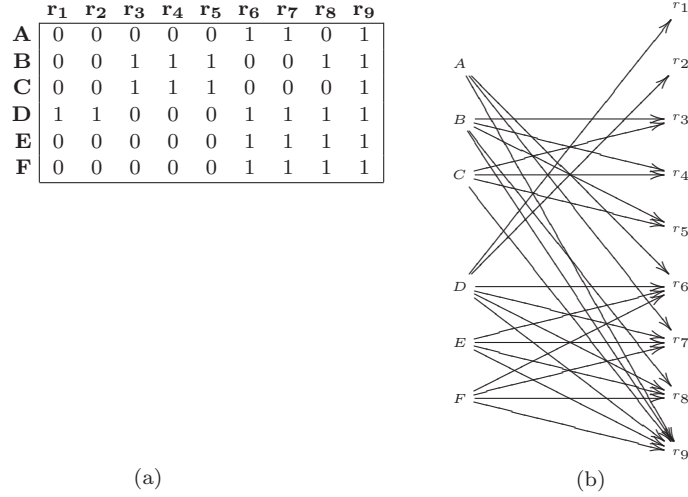


Fig. 2. An example of access matrix (a) and corresponding authorization policy graph (b)

Definition 2.2 Authorization policy graph. Let $\mathcal{A} = \langle \mathcal{U}, \mathcal{R}, \mathcal{P} \rangle$ be an authorization policy. The *authorization policy graph* over \mathcal{A} , denoted $\mathcal{G}_{\mathcal{A}}$, is a graph $\langle V_{\mathcal{A}}, E_{\mathcal{A}} \rangle$, where $V_{\mathcal{A}} = \mathcal{U} \cup \mathcal{R}$ and $E_{\mathcal{A}} = \{(u, r) \mid \langle u, r \rangle \in \mathcal{P}\}$.

In the following, we will use $\xrightarrow{\mathcal{A}}$ to denote reachability of vertices in graph $\mathcal{G}_{\mathcal{A}}$. Consequently, we will use $u \xrightarrow{\mathcal{A}} r$ and $\langle u, r \rangle \in \mathcal{P}$ indistinguishably to denote that user u is authorized to access resource r according to policy \mathcal{A} .

It is easy to see that access matrix $\mathcal{M}_{\mathcal{A}}$ corresponds to the bipartite adjacency matrix of the authorization policy graph $\mathcal{G}_{\mathcal{A}}$. Figure 2 illustrates an example of authorization policy with 6 users, 9 resources, and 26 permissions, reporting the access matrix and the corresponding authorization policy graph.

2.2 Encryption policy

Our goal is to represent the authorization policy by means of proper resource encryption and key distribution. We assume, for efficiency reasons, to adopt symmetric encryption. A naive solution to our goal would consist in encrypting each resource with a different key and assigning to each user the set of keys used to encrypt the resources she can access. Such a solution is clearly unacceptable, since it would require each user to manage 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, we exploit a *key derivation method*. Basically, a key derivation method allows the computation of a key starting from another key and some public information. Among all the key derivation methods (e.g., [Akl and Taylor 1983; Atallah et al. 2005; Ateniese et al. 2006; Crampton et al. 2006; Gudes 1980; Harn and Lin 1990; Hwang and Yang 2003; Liaw et al. 1989; MacKinnon et al. 1985; Sandhu 1987; 1988; De Santis et al. 2004; Shen and Chen 2002]), the proposal in [Atallah et al. 2005] minimizes the amount of re-encrypting and re-keying that must be done to enforce changes to

the authorization policy. The method is based on the definition and computation of *public tokens*. Let \mathcal{K} be the set of symmetric encryption keys in the system. Given two keys k_i and k_j in \mathcal{K} , a token $t_{i,j}$ is defined as $t_{i,j} = k_j \oplus h(k_i, l_j)$, where l_j is a publicly available label associated with k_j , \oplus is the bitwise xor operator, and h is a deterministic cryptographic function. The existence of a public token $t_{i,j}$ allows a user knowing k_i to derive key k_j through token $t_{i,j}$ and public label l_j . Since keys need to remain secret, while tokens are public, the use of tokens greatly simplifies key management. Key derivation via tokens can be applied in chains: a *chain of tokens* is a sequence $t_{i,l} \dots t_{n,j}$ of tokens such that $t_{c,d}$ directly follows $t_{a,b}$ in the chain only if $b = c$.

A major advantage of using tokens is that they are public and allow the user to derive multiple encryption keys, while having to worry about a single one. Exploiting tokens, the release to a user of a set $K = \{k_1, \dots, k_n\}$ of keys can be equivalently obtained by the release to the user of a single key $k_i \in K$ and the publication of a set of tokens allowing the (direct or indirect) derivation of all keys $k_j \in K$, $j \neq i$. In the following, we use \mathcal{T} to denote the set of tokens defined in the system and \mathcal{L} to denote the set of labels associated with the keys in \mathcal{K} and used for computing the tokens in \mathcal{T} .

Since tokens are public information, we assume that they are stored on the remote server (just like the encrypted data), so any user can access them. We model the relationships between keys through tokens allowing derivation of one key from another, via a graph, called the *key and token graph*. The graph has a vertex for each pair $\langle k, l \rangle$ of key k and corresponding label l . There is an edge from a vertex $\langle k_i, l_i \rangle$ to a vertex $\langle k_j, l_j \rangle$ if there exists a token $t_{i,j}$ allowing the derivation of k_j from k_i . The graph is formally defined as follows.

Definition 2.3 Key and token graph. Let \mathcal{K} be a set of keys, \mathcal{L} be a set of publicly available labels, and \mathcal{T} be a set of tokens defined on them. A *key and token graph* over \mathcal{K} , \mathcal{L} , and \mathcal{T} , denoted $\mathcal{G}_{\mathcal{K},\mathcal{T}}$, is a graph $\langle V_{\mathcal{K},\mathcal{T}}, E_{\mathcal{K},\mathcal{T}} \rangle$, where $V_{\mathcal{K},\mathcal{T}} = \{ \langle k_i, l_i \rangle \mid k_i \in \mathcal{K}, l_i \in \mathcal{L} \text{ is the label associated with } k_i \}$ and $E_{\mathcal{K},\mathcal{T}} = \{ (\langle k_i, l_i \rangle, \langle k_j, l_j \rangle) \mid t_{i,j} \in \mathcal{T} \}$.

The graphical representation of keys and tokens nicely captures the derivation relationship between keys that can be either direct, by means of a single token, or indirect, via a chain of tokens (corresponding to a path in the key and token graph).

The definition of tokens allows us to easily support the assumption that each user can be released only a single key and that each resource can be encrypted by using a single key. Note that these are not simplifying or limiting requirements, rather they are desiderata that we want our solution to satisfy. We then require our solution to operate under the following assumption.

Assumption 2.4. Each resource is available in a single instance. Each user can be released only one key.

We also assume that each key k is uniquely identified through the label l associated with it. A *key assignment and encryption schema* ϕ determines the labels of the keys assigned to users and of the keys used for encrypting resources, as stated by the following definition.

Definition 2.5 Key assignment and encryption schema. Let $\mathcal{U}, \mathcal{R}, \mathcal{K}, \mathcal{L}$ be the set of users, resources, keys, and labels in the system, respectively. A *key assignment and encryption schema* over $\mathcal{U}, \mathcal{R}, \mathcal{K}, \mathcal{L}$ is a function $\phi : \mathcal{U} \cup \mathcal{R} \mapsto \mathcal{L}$ that returns for each user $u \in \mathcal{U}$ the label $l \in \mathcal{L}$ associated with the (single) key k in \mathcal{K} released to the user and for each resource $r \in \mathcal{R}$ the label $l \in \mathcal{L}$ associated with the (single) key k in \mathcal{K} with which the resource is encrypted.

We are now ready to introduce the definition of *encryption policy* as follows.

Definition 2.6 Encryption policy. Let \mathcal{U} and \mathcal{R} be the set of users and resources in the system, respectively. An *encryption policy* over \mathcal{U} and \mathcal{R} , denoted \mathcal{E} , is a 6-tuple $\langle \mathcal{U}, \mathcal{R}, \mathcal{K}, \mathcal{L}, \phi, \mathcal{T} \rangle$, where \mathcal{K} is the set of keys defined in the system, \mathcal{L} is the set of corresponding labels, ϕ is a key assignment and encryption schema, and \mathcal{T} is a set of tokens defined on \mathcal{K} and \mathcal{L} .

The encryption policy can be conveniently represented via a graph by extending the key and token graph to include a vertex for each user and each resource, and adding an edge from each user vertex u to the vertex $\langle k, l \rangle$ such that $\phi(u)=l$ and from each vertex $\langle k, l \rangle$ to each resource vertex r such that $\phi(r)=l$. We can think of the encryption policy graph as a graph obtained by merging $\mathcal{G}_{\mathcal{A}}$ with $\mathcal{G}_{\mathcal{K}, \mathcal{T}}$, where instead of directly linking each user u with each resource r she can access, we pass through the vertex $\langle k_i, l_i \rangle$ such that $l_i = \phi(u)$, the vertex $\langle k_j, l_j \rangle$ such that $l_j = \phi(r)$, and possibly a chain of keys/tokens connecting them. The encryption policy graph is formally defined as follows.

Definition 2.7 Encryption policy graph. Let $\mathcal{E} = \langle \mathcal{U}, \mathcal{R}, \mathcal{K}, \mathcal{L}, \phi, \mathcal{T} \rangle$ be an encryption policy. The encryption policy graph over \mathcal{E} , denoted $\mathcal{G}_{\mathcal{E}}$, is the graph $\langle V_{\mathcal{E}}, E_{\mathcal{E}} \rangle$ where:

$$\begin{aligned} -V_{\mathcal{E}} &= V_{\mathcal{K}, \mathcal{T}} \cup \mathcal{U} \cup \mathcal{R}; \\ -E_{\mathcal{E}} &= E_{\mathcal{K}, \mathcal{T}} \cup \{(u, \langle k, l \rangle) \mid u \in \mathcal{U} \wedge l = \phi(u)\} \cup \{(\langle k, l \rangle, r) \mid r \in \mathcal{R} \wedge l = \phi(r)\}, \end{aligned}$$

where $V_{\mathcal{K}, \mathcal{T}}$ and $E_{\mathcal{K}, \mathcal{T}}$ are as in Definition 2.3.

Figure 3 illustrates an example of encryption policy graph, where dotted edges represent the key assignment and encryption schema (function ϕ) and solid edges represent the tokens. In the following, we will use $\xrightarrow{\mathcal{E}}$ to denote the reachability of vertices in graph $\mathcal{G}_{\mathcal{E}}$ (e.g., $A \xrightarrow{\mathcal{E}} r_6$). A user u can then retrieve (via her own key and the set of public tokens) all the keys of the vertices reachable from vertex whose label l is equal to $\phi(u)$. The resources accessible to a user according to an encryption policy are therefore all and only those reachable from u in the encryption policy graph $\mathcal{G}_{\mathcal{E}}$. Our goal is then to translate an authorization policy \mathcal{A} into an equivalent encryption policy \mathcal{E} , meaning that \mathcal{A} and \mathcal{E} allow exactly the same accesses, as formally defined in the following.

Definition 2.8 Policy equivalence. Let $\mathcal{A} = \langle \mathcal{U}, \mathcal{R}, \mathcal{P} \rangle$ be an authorization policy and $\mathcal{E} = \langle \mathcal{U}, \mathcal{R}, \mathcal{K}, \mathcal{L}, \phi, \mathcal{T} \rangle$ be an encryption policy. \mathcal{A} and \mathcal{E} are *equivalent*, denoted $\mathcal{A} \equiv \mathcal{E}$, iff the following conditions hold:

$$\begin{aligned} -\forall u \in \mathcal{U}, r \in \mathcal{R} : u \xrightarrow{\mathcal{E}} r &\implies u \xrightarrow{\mathcal{A}} r \\ -\forall u \in \mathcal{U}, r \in \mathcal{R} : u \xrightarrow{\mathcal{A}} r &\implies u \xrightarrow{\mathcal{E}} r \end{aligned}$$

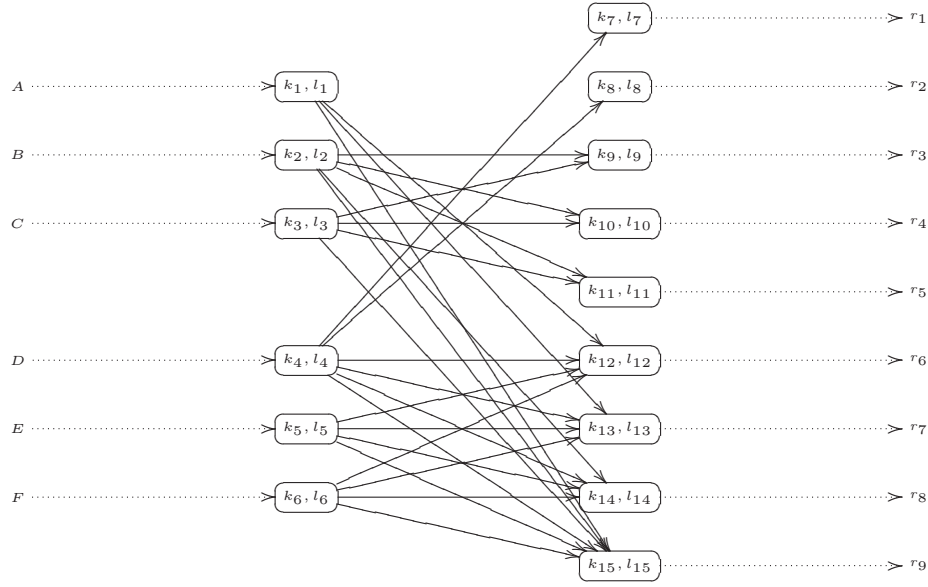


Fig. 3. An example of encryption policy graph

For instance, it is easy to see that the authorization policy in Figure 2 and the encryption policy represented by the encryption policy graph in Figure 3 are equivalent.

2.3 Token management

To allow users to access the outsourced data, a portion of the encryption policy \mathcal{E} must be made publicly available and stored on the server. The only component of the encryption policy \mathcal{E} that cannot be publicly released is the set \mathcal{K} of keys while all the other components can be released without compromising the protection of the outsourced data. The set \mathcal{T} of tokens, the set \mathcal{L} of labels, and the key assignment and encryption schema $\phi(r)$ over \mathcal{R}^2 are therefore stored on the server in the form of a *catalog* composed of two tables: LABELS and TOKENS. Table LABELS corresponds to the key assignment and encryption schema ϕ over \mathcal{R} . For each resource r in \mathcal{R} , table LABELS maintains the correspondence between the identifier of r (attribute *res.id*) and the label $\phi(r)$ (attribute *label*) associated with the key used for encrypting r . Table TOKENS corresponds to the set \mathcal{T} of tokens. For each token $t_{i,j}$ in \mathcal{T} , table TOKENS includes a tuple characterized by three attributes: *source* and *destination* are the labels l_i and l_j associated with k_i and k_j , respectively, and *token.value* is the token value computed as $t_{i,j} = k_j \oplus h(k_i, l_j)$. Figure 4 illustrates tables LABELS and TOKENS corresponding to the encryption policy graph represented in Figure 3.

²Note that the definition of ϕ over \mathcal{U} does not need to be made public, since each user knows her key and therefore the vertex in $\mathcal{G}_{\mathcal{E}}$ from which she can derive keys.

LABELS		TOKENS		
res_id	label	source	destination	token_value
r_1	l_7	l_1	l_{12}	$k_{12} \oplus h(k_1, l_{12})$
r_2	l_8	l_1	l_{13}	$k_{13} \oplus h(k_1, l_{13})$
r_3	l_9	l_1	l_{15}	$k_{15} \oplus h(k_1, l_{15})$
r_4	l_{10}	l_2	l_9	$k_9 \oplus h(k_2, l_9)$
r_5	l_{11}	l_2	l_{10}	$k_{10} \oplus h(k_2, l_{10})$
r_6	l_{12}	l_2	l_{11}	$k_{11} \oplus h(k_2, l_{11})$
r_7	l_{13}	l_2	l_{14}	$k_{14} \oplus h(k_2, l_{14})$
r_8	l_{14}	l_2	l_{15}	$k_{15} \oplus h(k_2, l_{15})$
r_9	l_{15}	l_3	l_9	$k_9 \oplus h(k_3, l_9)$
		l_3	l_{10}	$k_{10} \oplus h(k_3, l_{10})$
		l_3	l_{11}	$k_{11} \oplus h(k_3, l_{11})$
		l_3	l_{15}	$k_{15} \oplus h(k_3, l_{15})$
		l_4	l_7	$k_7 \oplus h(k_4, l_7)$
		l_4	l_8	$k_8 \oplus h(k_4, l_8)$
		l_4	l_{12}	$k_{12} \oplus h(k_4, l_{12})$
		l_4	l_{13}	$k_{13} \oplus h(k_4, l_{13})$
		l_4	l_{14}	$k_{14} \oplus h(k_4, l_{14})$
		l_4	l_{15}	$k_{15} \oplus h(k_4, l_{15})$
		l_5	l_{12}	$k_{12} \oplus h(k_5, l_{12})$
		l_5	l_{13}	$k_{13} \oplus h(k_5, l_{13})$
		l_5	l_{14}	$k_{14} \oplus h(k_5, l_{14})$
		l_5	l_{15}	$k_{15} \oplus h(k_5, l_{15})$
		l_6	l_{12}	$k_{12} \oplus h(k_6, l_{12})$
		l_6	l_{13}	$k_{13} \oplus h(k_6, l_{13})$
		l_6	l_{14}	$k_{14} \oplus h(k_6, l_{14})$
		l_6	l_{15}	$k_{15} \oplus h(k_6, l_{15})$

Fig. 4. Catalog for the encryption policy graph represented in Figure 3

3. MINIMUM ENCRYPTION POLICY

A straightforward approach for translating an authorization policy \mathcal{A} into an equivalent encryption policy \mathcal{E} consists in associating with each user a different key, encrypting each resource with a different key, and producing and publishing a token $t_{u,r}$ for each permission $\langle u, r \rangle \in \mathcal{P}$. The encryption policy graph in Figure 3 corresponds to an encryption policy that has been generated by translating the authorization policy in Figure 2 with this approach. While simple, this translation generates as many keys as the number of users and resources, and as many tokens as the number of permissions in the system. Even if tokens, being public, need not to be remembered or stored by users, producing and managing a token for each single permission can be unfeasible in practice. Indeed, each access to an encrypted resource requires a search across the catalog (see the *electronic Appendix*) and therefore the total number of tokens is a critical factor for the efficiency of access to remotely stored data.

This simple solution can be improved by grouping users with the same access privileges and by encrypting each resource with the key associated with the set of users that can access it. The advantage is that a key can be possibly used to encrypt more than one resource. Since there is a one-to-one mapping between an encryption policy \mathcal{E} and the encryption policy graph $\mathcal{G}_{\mathcal{E}}$ over \mathcal{E} , we exploit the hierarchy among sets of users induced by the partial order relationship based on set containment (\subseteq). We create an encryption policy graph $\mathcal{G}_{\mathcal{E}} = \langle V_{\mathcal{E}}, E_{\mathcal{E}} \rangle$, with $V_{\mathcal{E}} = V_{\mathcal{K}, \mathcal{T}} \cup \mathcal{U} \cup \mathcal{R}$, where $V_{\mathcal{K}, \mathcal{T}}$ includes a vertex for each possible subset U of \mathcal{U} , and $E_{\mathcal{E}}$ includes:

—an edge (v_i, v_j) for each possible pair of vertices $v_i, v_j \in V_{\mathcal{K}, \mathcal{T}}$ such that the set U_i

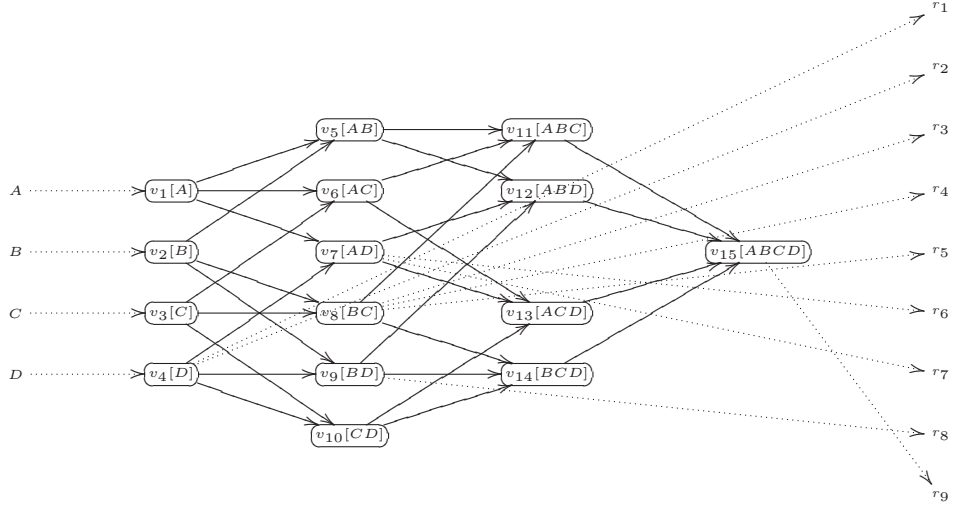


Fig. 5. An example of encryption policy graph over $\{A, B, C, D\}$

of users represented by v_i is a subset of the set U_j of users represented by v_j and the set containment relationship is direct;

- an edge (u_i, v_i) for each user $u_i \in \mathcal{U}$ such that $v_i \in V_{\mathcal{K}, \mathcal{T}}$ and the set of users represented by v_i is $\{u_i\}$;
- an edge (v_j, r_j) for each resource $r_j \in \mathcal{R}$ such that $v_j \in V_{\mathcal{K}, \mathcal{T}}$ and the set of users represented by v_j is $acl(r_j)$.

As an example, consider the portion of the authorization policy in Figure 2 that is defined on the set $\{A, B, C, D\}$ of users.

Figure 5 illustrates the encryption policy graph over $\{A, B, C, D\}$ defined as described above. In the figure, each vertex v_i also reports, between square brackets, the set of users, denoted $v_i.acl$, represented by v_i . It is interesting to note that the subgraph induced by $V_{\mathcal{K}, \mathcal{T}}$ is an n -stratified graph, where n is the number of users in the system (i.e., $n = |\mathcal{U}|$). Each strata, which we call *level*, contains all the vertices that represent sets of users with the same cardinality. For instance, in the encryption policy graph in Figure 5, v_1, v_2, v_3 , and v_4 are vertices at level 1. In the following, the level of a vertex $v \in V_{\mathcal{K}, \mathcal{T}}$ will be denoted as $level(v)$.

By assigning to each vertex $v \in V_{\mathcal{K}, \mathcal{T}}$ of the encryption policy graph a pair $\langle v.key, v.label \rangle$, corresponding to a key and label, the authorization policy can be enforced by: *i*) encrypting each resource with the key of the vertex corresponding to its access control list (e.g., resource r_5 should be encrypted with the key associated with the vertex representing $\{B, C\}$), and *ii*) assigning to each user the key associated with the vertex representing the user in the graph. Note that the encryption policy corresponding to this encryption policy graph is such that:

- the sets \mathcal{K} and \mathcal{L} include all the keys and labels, respectively, associated with vertices in $V_{\mathcal{K}, \mathcal{T}}$;
- the key assignment and encryption schema ϕ is such that $\forall u \in \mathcal{U}, \phi(u) = v.label$,

- with v the vertex representing the user (i.e., $v.acl=\{u\}$), and $\forall r \in \mathcal{R}$, $\phi(r) = v.label$, with v the vertex representing $acl(r)$ (i.e., $v.acl=acl(r)$);
- the set \mathcal{T} includes a token for each edge (v_i, v_j) in $E_{\mathcal{E}}$, with $v_i, v_j \in V_{\mathcal{K}, \mathcal{T}}$, that allows the derivation of key $v_j.key$ from key $v_i.key$.

Although this solution is simple and easy to implement, it defines more keys than actually needed and requires the publication of a great amount of information on the remote server, thus causing an expensive key derivation process at the user-side. For instance, in the encryption policy graph in Figure 5 vertex v_{10} is not needed for enforcing the authorization policy since its key is not used for encrypting any resource. The presence of such a vertex only increases the size of table `TOKENS` stored on the server without giving any benefit. We are then interested in finding a *minimum encryption policy*, equivalent to a given authorization policy and minimizing the number of tokens to be maintained by the server.

Definition 3.1 Minimum encryption policy. Let $\mathcal{A} = \langle \mathcal{U}, \mathcal{R}, \mathcal{P} \rangle$ be an authorization policy and $\mathcal{E} = \langle \mathcal{U}, \mathcal{R}, \mathcal{K}, \mathcal{L}, \phi, \mathcal{T} \rangle$ be an encryption policy such that $\mathcal{A} \equiv \mathcal{E}$. \mathcal{E} is *minimum* with respect to \mathcal{A} iff $\nexists \mathcal{E}' = \langle \mathcal{U}, \mathcal{R}, \mathcal{K}', \mathcal{L}', \phi', \mathcal{T}' \rangle$ such that $\mathcal{A} \equiv \mathcal{E}'$ and $|\mathcal{T}'| < |\mathcal{T}|$.

Given an authorization policy \mathcal{A} , different minimum encryption policies may exist and our goal is to compute one of them, as stated by the following problem definition.

Problem 3.2 Min-EP. Let $\mathcal{A} = \langle \mathcal{U}, \mathcal{R}, \mathcal{P} \rangle$ be an authorization policy. Determine a *minimum encryption policy* $\mathcal{E} = \langle \mathcal{U}, \mathcal{R}, \mathcal{K}, \mathcal{L}, \phi, \mathcal{T} \rangle$.

Unfortunately, Problem 3.2 is *NP-hard*, as proved by the following theorem.

THEOREM 3.3. *The Min-EP problem is NP-hard.*

We then propose a heuristic approach for solving Problem 3.2 that reduces the user’s overhead in deriving keys through a simplification of the encryption policy graph created according to the process previously described. Our heuristic approach is based on two basic observations. First, the encryption policy graph has to include only the vertices that are needed to enforce a given authorization policy, connecting them to ensure a correct key derivability. Second, beside the vertices needed for the enforcement of the authorization policy, other vertices can be included if they are useful for reducing the size of the catalog. We therefore present a factorization procedure that, as the experiments in Section 9 show, improves the performance at the user-side since it allows a great reduction in the number of tokens. Sections 3.1 and 3.2 discuss these two observations, which will then be taken into account by the heuristic approach in Section 4.

3.1 Vertex and edge selection

From the previous discussion, it is immediate to see that the vertices in $V_{\mathcal{K}, \mathcal{T}}$ strictly needed for the enforcement of the authorization policy are the vertices representing: *i*) singleton sets of users, whose keys are needed to derive all the other keys used for decrypting resources in the users’ capabilities; and *ii*) the *acls* of the resources, whose keys are needed for decrypting such resources. In the following, we refer

to these vertices as *material*. According to the definition of policy equivalence (Definition 2.8), the material vertices must be connected in the graph in such a way that each user $u \in \mathcal{U}$ is able to derive keys allowing access to all and only the resources she is authorized to read. This means that the encryption policy graph must include at least one path from the vertex v_i representing user u (i.e., vertex v_i such that $v_i.acl = \{u\}$) to all material vertices v_j such that $u \in v_j.acl$. Since our main goal is to keep at minimum the number of tokens managed by the server and since each edge between vertices in $V_{\mathcal{K},\mathcal{T}}$ corresponds to a token, our problem is to connect the material vertices creating an encryption policy graph such that: *i*) the corresponding encryption policy is equivalent to a given authorization policy, *ii*) the number of edges is minimal. To solve this problem, we observe that the direct ancestors of a material vertex must form a set covering for it. Indeed, since for each user u the encryption policy graph must include a path from the vertex representing u to all vertices v_j such that $u \in v_j.acl$ and, since, by construction, there is an edge (v_i, v_j) , with $v_i, v_j \in V_{\mathcal{K},\mathcal{T}}$, iff $v_i.acl \subset v_j.acl$, vertex v_j must have at least a direct ancestor v_k such that $u \in v_k.acl$. The existence of a set covering for the vertices in the encryption policy graph is formalized via the definition of the following *local cover* property.

Definition 3.4 Local cover property. Let $\mathcal{A} = \langle \mathcal{U}, \mathcal{R}, \mathcal{P} \rangle$ be an authorization policy and $\mathcal{E} = \langle \mathcal{U}, \mathcal{R}, \mathcal{K}, \mathcal{L}, \phi, \mathcal{T} \rangle$ be an encryption policy. The encryption policy graph $\mathcal{G}_{\mathcal{E}} = \langle V_{\mathcal{E}}, E_{\mathcal{E}} \rangle$ over \mathcal{E} , with $V_{\mathcal{E}} = V_{\mathcal{K},\mathcal{T}} \cup \mathcal{U} \cup \mathcal{R}$, satisfies the *local cover* property if $\forall v_j \in V_{\mathcal{K},\mathcal{T}}$, with $|v_j.acl| > 1$, $v_j.acl = \bigcup_i \{v_i.acl \mid (v_i, v_j) \in E_{\mathcal{E}}\}$.

Given an authorization policy \mathcal{A} and an encryption policy \mathcal{E} , it is easy to see that if \mathcal{E} is equivalent to \mathcal{A} , the encryption policy graph over \mathcal{E} satisfies the local cover property, as formally stated by the following theorem.

THEOREM 3.5. *Let \mathcal{A} be an authorization policy and \mathcal{E} be an encryption policy. If \mathcal{E} is equivalent to \mathcal{A} , the encryption policy graph $\mathcal{G}_{\mathcal{E}} = \langle V_{\mathcal{E}}, E_{\mathcal{E}} \rangle$ over \mathcal{E} , with $V_{\mathcal{E}} = V_{\mathcal{K},\mathcal{T}} \cup \mathcal{U} \cup \mathcal{R}$, satisfies the local cover property (Definition 3.4).*

Our approach to generate an encryption policy equivalent to a given authorization policy then starts by creating a key and token graph satisfying Definition 3.4. We apply a bottom up approach, processing vertices in decreasing order of level. For each material vertex v at level l , its possible direct ancestors are first searched among the material vertices at level $l-1$, then at level $l-2$, and so on, until all the material vertices directly connected with v form a set covering for v . The rationale behind this bottom up strategy is that, in principle³, by searching first among the vertices with a greater level value, the number of direct ancestors and therefore of edges for connecting them with v should be less than the number of direct ancestors needed for covering vertex v when such vertices are chosen in increasing order of level. As an example, consider the material vertices corresponding to the authorization policy in Figure 2 that represent the following sets of users: $\{A\}$, $\{B\}$, $\{C\}$, $\{D\}$, $\{E\}$, $\{F\}$, $\{BC\}$, $\{ADEF\}$, $\{BDEF\}$, and $\{ABCDEF\}$. Consider

³Since this bottom up strategy is a heuristic that we apply for solving a NP-hard problem, the solution computed through it may not be always the optimal solution. However, we will see in Section 9 that this heuristic produces good results.

now the material vertex representing $\{ABCDEF\}$ and suppose to compute a set covering for it by choosing the appropriate direct ancestors from the given material vertices. If we apply the bottom up strategy previously described, the possible direct ancestors for $\{ABCDEF\}$ are first chosen among the vertices at level: 5, which is empty; 4, where there are two material vertices (i.e., $\{ADEF\}$, $\{BDEF\}$) that can be chosen as direct ancestors for $\{ABCDEF\}$; 3, which is empty; and then 2, where vertex $\{BC\}$ is chosen. The final set covering for $\{ABCDEF\}$ is $\{\{ADEF\}, \{BDEF\}, \{BC\}\}$, which requires three edges for connecting the vertices in the set covering with the vertex representing $\{ABCDEF\}$. Another possible set covering for $\{ABCDEF\}$ is, for example, $\{\{A\}, \{B\}, \{C\}, \{D\}, \{E\}, \{F\}\}$, which instead requires six edges.

This simple approach for computing a set covering may however introduce redundant edges. For instance, with respect to the previous example, since $\{ADEF\}$ and $\{BDEF\}$ are selected before $\{BC\}$, it is easy to see that the edge from the vertex representing $\{BDEF\}$ to the vertex representing $\{ABCDEF\}$ becomes redundant after choosing $\{BC\}$ since each user in $\{BDEF\}$ is also a member of at least one of the other two direct ancestors of the vertex representing $\{ABCDEF\}$. The redundant edges increase the number of tokens and are not useful for the enforcement of the authorization policy. We are then interested in computing a *non-redundant* encryption policy graph defined as follows.

Definition 3.6 Non-redundant encryption policy graph. Let $\mathcal{A} = \langle \mathcal{U}, \mathcal{R}, \mathcal{P} \rangle$ be an authorization policy and $\mathcal{E} = \langle \mathcal{U}, \mathcal{R}, \mathcal{K}, \mathcal{L}, \phi, \mathcal{T} \rangle$ be an encryption policy equivalent to \mathcal{A} . The encryption policy graph $\mathcal{G}_{\mathcal{E}} = \langle V_{\mathcal{E}}, E_{\mathcal{E}} \rangle$ over \mathcal{E} , with $V_{\mathcal{E}} = V_{\mathcal{K}, \mathcal{T}} \cup \mathcal{U} \cup \mathcal{R}$, is *non-redundant* iff $\forall v_j \in V_{\mathcal{K}, \mathcal{T}}$, with $|v_j.acl| > 1$, $\forall (v_i, v_j) \in E_{\mathcal{E}}$, $\exists u \in v_i.acl$ $\forall (v_l, v_j) \in E_{\mathcal{E}}$, with $v_l \neq v_i$, $u \notin v_l.acl$.

3.2 Vertex factorization

In addition to the material vertices, other vertices can be inserted in the encryption policy graph if they can reduce the number of tokens in the catalog. As an example, consider the material vertices corresponding to the authorization policy in Figure 2. The sets V and V' covering material vertices $\{ADEF\}$ and $\{BDEF\}$, respectively, can only include the vertices representing singleton sets of users, since there are no material vertices representing subsets of $\{ADEF\}$ or of $\{BDEF\}$. The number of edges connecting the vertices in V and V' with $\{ADEF\}$ and $\{BDEF\}$ are then eight. Suppose now to add a non material vertex representing $\{DEF\}$. In this case, the set covering for $\{ADEF\}$ is $\{\{DEF\}, \{A\}\}$ and the set covering for $\{BDEF\}$ is $\{\{DEF\}, \{B\}\}$. The number of edges needed for connecting the vertices in the sets covering with $\{ADEF\}$ and $\{BDEF\}$ is therefore four. Also, three edges are necessary for covering $\{DEF\}$ through $\{\{D\}, \{E\}, \{F\}\}$ for a total of seven edges against the eight edges of the previous case. Generalizing, it is easy to see that whenever there are m vertices v_1, \dots, v_m that share n , with $n > 2$, ancestors v'_1, \dots, v'_n , it is convenient to factorize the common ancestors by inserting an intermediate vertex v' , with $v'.acl = \bigcup_{i=1}^n v'_i.acl$, and to connect each vertex $v'_i, i = 1, \dots, n$, with v' , and v' with $v_j, j = 1, \dots, m$. In this way, the encryption policy graph includes $n + m$, instead of $n \cdot m$, edges (i.e., tokens in the catalog). The advantage may appear small in the example above, but the

ALGORITHM $\mathcal{A}2\mathcal{E}$

```

INPUT authorization policy  $\mathcal{A}=\langle\mathcal{U},\mathcal{R},\mathcal{P}\rangle$ 
OUTPUT encryption policy  $\mathcal{E}$  such that  $\mathcal{A}\equiv\mathcal{E}$  and  $\mathcal{G}_{\mathcal{E}}$  is not redundant

MAIN
 $V_{\mathcal{K},\mathcal{T}} := \emptyset$ 
 $E_{\mathcal{K},\mathcal{T}} := \emptyset$ 
/* Initialization */
 $ACL := \{acl(r)|r\in\mathcal{R}\} \cup \{\{u\}|u\in\mathcal{U}\}$ 
for  $acl\in ACL$  do
  create vertex  $v$ 
   $v.acl := acl$ 
   $v.label := \text{NULL}$ 
   $v.key := \text{NULL}$ 
  for each  $u\in v.acl$  do  $v.counter[u] := 0$ 
   $V_{\mathcal{K},\mathcal{T}} := V_{\mathcal{K},\mathcal{T}} \cup \{v\}$ 
/* Phase 1: cover vertices without redundancies */
for  $l:=|\mathcal{U}|\dots 2$  do
  for each  $v_i\in\{v|v\in V_{\mathcal{K},\mathcal{T}} \wedge level(v)=l\}$  do cover_vertex( $v_i,v_i.acl$ ) /* see Figure 7 */
/* Phase 2: factorize common ancestors */
for  $l:=|\mathcal{U}|\dots 2$  do
  for each  $v_i\in\{v|v\in V_{\mathcal{K},\mathcal{T}} \wedge level(v)=l\}$  do factorize( $v_i$ ) /* see Figure 8 */
/* Phase 3: generate encryption policy */
generate_encryption_policy() /* see Figure 9 */

```

Fig. 6. Algorithm for computing an encryption policy \mathcal{E} equivalent to \mathcal{A}

experiments in Section 9 show that this optimization can produce significant gains in scenarios with complex policies.

Our approach applies this factorization process during the construction of the encryption policy graph, processing vertices in decreasing order of level and comparing a vertex v with each vertex v' at lower level. This bottom up strategy guarantees that the vertex added in the graph (if any) to provide factorization will appear at a level lower than the level of v and v' and therefore it will be compared to the other vertices in the graph when the vertices at that level will be analyzed. To limit the number of pairs of vertices analyzed, we consider only pairs of vertices that have at least one common direct ancestor; a rather straightforward adaptation of the analysis in [Baralis et al. 1997] demonstrates that it is sufficient to consider these pairs, with a significant reduction in the number of comparisons.

4. ALGORITHM $\mathcal{A}2\mathcal{E}$

Our heuristic algorithm for computing a minimal encryption policy is illustrated in Figure 6. The algorithm takes an authorization policy $\mathcal{A}=\langle\mathcal{U},\mathcal{R},\mathcal{P}\rangle$ as input and returns an encryption policy \mathcal{E} equivalent to \mathcal{A} and that satisfies Definition 3.6. To this purpose, the algorithm first creates a key and token graph $\langle V_{\mathcal{K},\mathcal{T}},E_{\mathcal{K},\mathcal{T}}\rangle$ and then generates the corresponding encryption policy, by computing the set \mathcal{T} of tokens and by defining the key assignment and encryption schema ϕ . Each vertex v in $V_{\mathcal{K},\mathcal{T}}$ is associated with four variables: $v.key$ represents the key of the vertex; $v.label$ represents the publicly available label associated with $v.key$; $v.acl$ represents

```

COVER_VERTEX( $v, tocover$ )
   $Eadded := \emptyset$ 
   $l := level(v) - 1$ 
  /* find a correct cover for users in  $tocover$  */
  while  $tocover \neq \emptyset$  do
     $V_l := \{v_i | v_i \in V_{\mathcal{K}, \mathcal{T}} \wedge level(v_i) = l \wedge v_i.acl \subseteq v.acl\}$ 
    while  $tocover \neq \emptyset \wedge V_l \neq \emptyset$  do
      extract  $v_i$  from  $V_l$ 
      if  $v_i.acl \cap tocover \neq \emptyset$  then
         $tocover := tocover \setminus v_i.acl$ 
         $Eadded := Eadded \cup \{(v_i, v)\}$ 
        for each  $u \in v_i.acl$  do  $v.counter[u] := v.counter[u] + 1$ 
       $l := l - 1$ 
  /* remove redundant edges */
  for each  $(v_i, v) \in Eadded$  do
    if  $(\nexists u | u \in v_i.acl \wedge v.counter[u] = 1)$  then
       $Eadded := Eadded \setminus \{(v_i, v)\}$ 
      for each  $u \in v_i.acl$  do  $v.counter[u] := v.counter[u] - 1$ 
   $E_{\mathcal{K}, \mathcal{T}} := E_{\mathcal{K}, \mathcal{T}} \cup Eadded$ 

```

Fig. 7. Procedure for covering material vertices and removing redundant edges

the set of users who can derive $v.key$; $v.counter[]$ is an array with one component for each user u in $v.acl$ such that $v.counter[u]$ is equal to the number of direct ancestors of v whose $acls$ contain user u (as we will see, this information will be used to detect redundant edges).

The algorithm starts by creating the material vertices and by properly initializing the variables associated with them. The algorithm is logically partitioned in three phases: *i) cover vertices* adds edges to the graph in such a way to satisfy both the local cover property and the non-redundancy property (Section 3.1), *ii) factorize common ancestors* adds non material vertices for reducing the number of edges in the graph (Section 3.2), and *iii) generate encryption policy* creates an encryption policy corresponding to the graph calculated in the previous two phases. We now describe these three phases more in details.

Phase 1: Cover vertices. To create a key and token graph that satisfies the local cover and the non-redundancy properties, the algorithm proceeds bottom up, from level $l = |\mathcal{U}|$ to 2, and for each material vertex v at level l , calls procedure **cover_vertex** in Figure 7. Procedure **cover_vertex** takes as input a vertex v and a set $tocover$ of users, corresponding to $v.acl$. The procedure first initializes two local variables: $Eadded$, representing the set of edges that need to be added to the graph, is set to the empty set; and l , representing the level of candidate direct ancestors for v , is set to $level(v) - 1$. At each iteration of the outermost **while** loop, the procedure computes the set V_l of vertices at level l whose acl is a subset of $v.acl$, and the innermost **while** loop checks if there are vertices in V_l that can be part of the set covering for v . To this purpose, the procedure randomly extracts a vertex v_i from V_l and if $v_i.acl$ has at least a user in common with $tocover$, it removes from $tocover$ the set of common users appearing in $v_i.acl$, and adds edge (v_i, v) to $Eadded$. Also, for each user u in $v_i.acl$, the procedure increases $v.counter[u]$ by one. The innermost **while** loop terminates when $tocover$ becomes empty or when

```

FACTORIZE( $v_i$ )
for each  $v_j \in \{v \mid \exists v_a, (v_a, v_i) \in E_{\mathcal{K}, \mathcal{T}} \wedge (v_a, v) \in E_{\mathcal{K}, \mathcal{T}}\}$  do /* children of  $v_i$ 's direct ancestors */
   $E_{added} := \emptyset$ 
   $E_{removed} := \emptyset$ 
   $CommonAnc := \{v_a \mid (v_a, v_i) \in E_{\mathcal{K}, \mathcal{T}} \wedge (v_a, v_j) \in E_{\mathcal{K}, \mathcal{T}}\}$  /* common direct ancestors */
  if  $|CommonAnc| > 2$  then
    /* create a new common ancestor for  $v_i$  and  $v_j$  */
     $U := \bigcup \{v_a.acl \mid v_a \in CommonAnc\}$ 
    find the vertex  $v \in V_{\mathcal{K}, \mathcal{T}}$  with  $v.acl = U$ 
    case  $v$  of
       $\neq v_i \wedge \neq v_j$ :  $E_{added} := E_{added} \cup \{(v, v_i), (v, v_j)\}$ 
        for each  $v_a \in CommonAnc$  do
           $E_{removed} := E_{removed} \cup \{(v_a, v_i), (v_a, v_j)\}$ 
       $= v_i$ :  $E_{added} := E_{added} \cup \{(v_i, v_j)\}$ 
        for each  $v_a \in CommonAnc$  do
           $E_{removed} := E_{removed} \cup \{(v_a, v_j)\}$ 
       $= v_j$ :  $E_{added} := E_{added} \cup \{(v_j, v_i)\}$ 
        for each  $v_a \in CommonAnc$  do
           $E_{removed} := E_{removed} \cup \{(v_a, v_i)\}$ 
      UNDEF: create vertex  $v'$ 
         $v'.acl := U$ 
         $v'.label := NULL$ 
         $v'.key := NULL$ 
        for each  $u \in v'.acl$  do
           $v'.counter[u] := 0$ 
         $V_{\mathcal{K}, \mathcal{T}} := V_{\mathcal{K}, \mathcal{T}} \cup \{v'\}$ 
         $E_{added} := E_{added} \cup \{(v', v_i), (v', v_j)\}$ 
        for each  $v_a \in CommonAnc$  do
           $E_{added} := E_{added} \cup \{(v_a, v')\}$ 
           $E_{removed} := E_{removed} \cup \{(v_a, v_i), (v_a, v_j)\}$ 
    /* update counters */
    for each  $(v_l, v_h) \in E_{added}$  do
      for each  $u \in v_l.acl$  do  $v_h.counter[u] := v_h.counter[u] + 1$ 
    for each  $(v_l, v_h) \in E_{removed}$  do
      for each  $u \in v_l.acl$  do  $v_h.counter[u] := v_h.counter[u] - 1$ 
     $E_{\mathcal{K}, \mathcal{T}} := E_{\mathcal{K}, \mathcal{T}} \cup E_{added} \setminus E_{removed}$ 

```

Fig. 8. Procedure for factorizing the common ancestors between vertices

all vertices in V_l have been processed. Local variable l is then decreased by one and the process is repeated until *tocover* becomes empty. The procedure checks if E_{added} contains redundant edges. For each edge (v_i, v) in E_{added} , if there does not exist a user u in $v_i.acl$ such that $v.counter[u]=1$ (remember that $v.counter[u]$ is the number of direct ancestors of v with user u in their *acls*), then edge (v_i, v) is redundant; it is removed from E_{added} ; and, for each user u in $v_i.acl$, $v.counter[u]$ is decreased by one. The set E_{added} of non-redundant edges is then added to $E_{\mathcal{K}, \mathcal{T}}$.

Phase 2: Factorize acls. The key and token graph resulting from the previous phase guarantees that each user can derive the keys of the resources she is authorized to access. The algorithm now verifies if it is possible to add vertices to reduce the number of edges in the graph. To this purpose, for each level l from $|\mathcal{U}|$ to 2 and for each vertex v_i at level l , the algorithm calls procedure **factorize** in Figure 8

on v_i . For each vertex v_j having at least a common direct ancestor with v_i (first for loop), procedure **factorize** first initializes two local variables: *Eadded* and *Eremoved*, representing the set of edges that need to be added to and removed from the graph, respectively, are both initialized as empty. Procedure **factorize** then determines set *CommonAnc* of direct ancestors common to v_i and v_j . If *CommonAnc* contains more than two vertices, v_i and v_j can be covered by a vertex that factorizes all vertices in *CommonAnc*. In this way, $2 \cdot |CommonAnc|$ edges are removed from the graph and at most $2 + |CommonAnc|$ edges are added. Procedure **factorize** therefore computes the union U among the *acls* associated with vertices in *CommonAnc*. The procedure checks if the graph already includes a vertex v whose *acl* is equal to U and detects the edges that have to be added to, or removed from, the graph. Three cases may occur. First case: vertex v already exists and coincides neither with v_i nor with v_j . The two edges from v to v_i and from v to v_j are inserted in *Eadded*, and all edges from the vertices in *CommonAnc* to v_i and to v_j are inserted in *Eremoved*. Second case: vertex v coincides with v_i (v_j , resp.). The procedure inserts a new edge from v_i to v_j (from v_j to v_i , resp.) in *Eadded* and all edges from the vertices in *CommonAnc* to v_j (v_i , resp.) are inserted in *Eremoved*. Third case: vertex v does not exist in the graph. The procedure creates a new vertex v' and initializes $v'.acl$ to U and both $v'.label$ and $v'.key$ to NULL. The new vertex is then inserted in the graph and the edges from the vertices in *CommonAnc* to v' are inserted in *Eadded* along with the two edges from the new vertex v' to v_i and to v_j . The edges from all the vertices in *CommonAnc* to v_i and to v_j are inserted in *Eremoved*. The procedure then properly updates variables $v.counter[u]$ for all edges (v_l, v_h) in *Eadded* and *Eremoved*. Finally, set $E_{\mathcal{K}, \mathcal{T}}$ of edges is updated by adding edges in *Eadded* and by removing edges in *Eremoved*.

Phase 3: Generate encryption policy \mathcal{E} . The last phase of the algorithm generates the encryption policy corresponding to the key and token graph computed in the previous phases. To this purpose, the algorithm calls procedure **generate_encryption_policy** in Figure 9. First, the procedure initializes the set \mathcal{K} of keys, the set \mathcal{L} of labels, and the set \mathcal{T} of tokens to empty. Then, for each vertex v in $V_{\mathcal{K}, \mathcal{T}}$, the procedure generates a key k and a label l and inserts them in \mathcal{K} and \mathcal{L} , respectively. Also, for each edge (v_i, v_j) in $E_{\mathcal{K}, \mathcal{T}}$, procedure **generate_encryption_policy** computes token $t_{i,j}$, which is inserted in \mathcal{T} and uploaded on the server by inserting a corresponding tuple in table `TOKENS`. Finally, the procedure defines the key assignment and encryption schema ϕ based on the labels previously generated. For each user u , $\phi(u)$ is defined as the label of the vertex representing the singleton set $\{u\}$, and for each resource r , $\phi(r)$ is defined as the label of the vertex representing $acl(r)$ in the graph. Also, each resource r is encrypted with the key of the vertex corresponding to $\phi(r)$ and uploaded on the server; table `LABELS` in the catalog is updated accordingly.

EXAMPLE 4.1. *Figure 10 presents the execution, step by step, of the algorithm in Figure 6, applied to the authorization policy in Figure 2. The algorithm first generates 10 material vertices: v_1, \dots, v_6 represent the singleton sets of users $\{A\}, \dots, \{F\}$, respectively; v_7 represents $\{BC\}$; v_8 represents $\{ADEF\}$; v_9 represents $\{BDEF\}$; and v_{10} represents $\{ABCDE F\}$.*

Figure 10(a) illustrates the key and token graph obtained after the first phase of

```

GENERATE_ENCRYPTION_POLICY()
 $\mathcal{K} := \emptyset; \mathcal{L} := \emptyset; \mathcal{T} := \emptyset$ 
/* generate keys */
for each  $v \in V_{\mathcal{K}, \mathcal{T}}$  do
  generate key  $k$ 
   $v.key := k$ 
  generate label  $l$ 
   $v.label := l$ 
   $\mathcal{K} := \mathcal{K} \cup \{v.key\}$ 
   $\mathcal{L} := \mathcal{L} \cup \{v.label\}$ 
/* compute tokens */
for each  $(v_i, v_j) \in E_{\mathcal{K}, \mathcal{T}}$  do
   $t_{i,j} := v_j.key \oplus h(v_i.key, v_j.label)$ 
   $\mathcal{T} := \mathcal{T} \cup \{t_{i,j}\}$ 
  upload token  $t_{i,j}$  on the server by adding it to table TOKENS
/* define key assignment and encryption schema */
for each  $u \in \mathcal{U}$  do
  find the vertex  $v \in V_{\mathcal{K}, \mathcal{T}}$  with  $v.acl = \{u\}$ 
   $\phi(u) := v.label$ 
for each  $r \in \mathcal{R}$  do
  find the vertex  $v \in V_{\mathcal{K}, \mathcal{T}}$  with  $v.acl = acl(r)$ 
  encrypt  $r$  with key  $v.key$ 
  upload the encrypted version  $r^k$  of  $r$  on the server
   $\phi(r) := v.label$ 
  update table LABELS on the server

```

Fig. 9. Procedure for creating an encryption policy

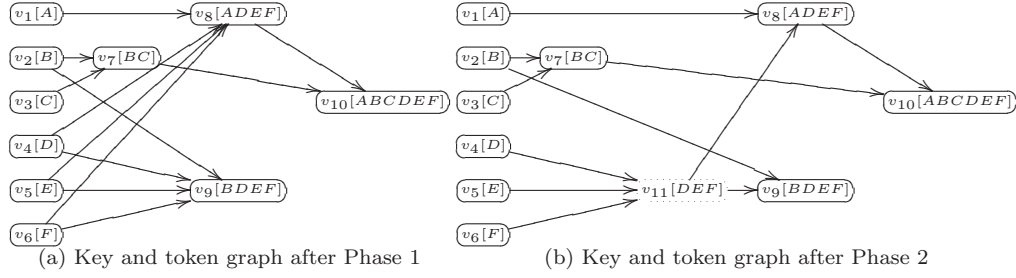
the algorithm. As an example of how this graph has been obtained, consider vertex v_{10} . Procedure **cover_vertex** inserts in E added first edges (v_8, v_{10}) and (v_9, v_{10}) , and then edge (v_7, v_{10}) . Edge (v_9, v_{10}) then becomes redundant since all users in $v_9.acl$ can derive $v_{10}.key$ passing through v_7 or v_8 . The procedure therefore removes such an edge. It is easy to see that the key and token graph satisfies the local cover property and the non-redundancy property.

Figure 10(b) illustrates the graph obtained after the second phase of the algorithm. Here, material vertices are represented with solid lines, while non material vertices are represented with dotted lines. Note that the graph has a new vertex, v_{11} , which has been inserted by procedure **factorize** since vertices v_8 and v_9 in the graph in Figure 10(a) have three common direct ancestors (i.e., v_4, v_5 , and v_6). In this way, the total number of tokens/edges has been decreased by one.

Finally, Figure 10(c) illustrates the key assignment and encryption schema for users in \mathcal{U} , and tables **LABELS** and **TOKENS** uploaded on the server by procedure **generate_encryption_policy**.

4.1 Correctness and complexity

In [Atallah et al. 2005] the authors have proved that the token-based derivation technique is sound, that is, it is secure against key recovery attacks even in presence of collusion: if an adversary compromises a key, she can derive only those keys that are derivable from it; also, no subset of users can collude to gain access to the keys that they cannot already derive. In this section, we prove the correctness



		LABELS		TOKENS		
u	$\phi(u)$	res_id	$label$	$source$	$destination$	$token_value$
A	$v_1.label$	r_1	$v_4.label$	$v_1.label$	$v_8.label$	$t_{1,8}$
B	$v_2.label$	r_2	$v_4.label$	$v_2.label$	$v_7.label$	$t_{2,7}$
C	$v_3.label$	r_3	$v_7.label$	$v_2.label$	$v_9.label$	$t_{2,9}$
D	$v_4.label$	r_4	$v_7.label$	$v_3.label$	$v_7.label$	$t_{3,7}$
E	$v_5.label$	r_5	$v_7.label$	$v_4.label$	$v_{11}.label$	$t_{4,11}$
F	$v_6.label$	r_6	$v_8.label$	$v_5.label$	$v_{11}.label$	$t_{5,11}$
		r_7	$v_8.label$	$v_6.label$	$v_{11}.label$	$t_{6,11}$
		r_8	$v_9.label$	$v_7.label$	$v_{10}.label$	$t_{7,10}$
		r_9	$v_{10}.label$	$v_8.label$	$v_{10}.label$	$t_{8,10}$
				$v_{11}.label$	$v_8.label$	$t_{11,8}$
				$v_{11}.label$	$v_9.label$	$t_{11,9}$

(c) Function ϕ and catalog after Phase 3

Fig. 10. An example of algorithm execution

and complexity of algorithm $\mathcal{A2E}$. The correctness of the algorithm ensures that the hierarchy defined by $\mathcal{A2E}$ will enable derivation of only authorized keys and therefore that each user will be able to decrypt all and only the resources that she is authorized to access according to the authorization policy (i.e., the two policies are equivalent). The correctness of $\mathcal{A2E}$ together with the soundness of the key derivation method guarantee the security of the system.

To prove that the encryption policy generated by algorithm $\mathcal{A2E}$ is equivalent to a given authorization policy, we first introduce some lemmas. First, we prove that users are assigned distinct keys.

LEMMA 4.1 USER KEY UNIQUENESS. *Let $\mathcal{A}=\langle\mathcal{U}, \mathcal{R}, \mathcal{P}\rangle$ be an authorization policy. Algorithm $\mathcal{A2E}$ creates a key and token graph $\mathcal{G}_{\mathcal{K}, \mathcal{T}}=\langle V_{\mathcal{K}, \mathcal{T}}, E_{\mathcal{K}, \mathcal{T}}\rangle$ and the corresponding encryption policy $\mathcal{E}=\langle\mathcal{U}, \mathcal{R}, \mathcal{K}, \mathcal{L}, \phi, \mathcal{T}\rangle$ such that $\forall u_i, u_j \in \mathcal{U}, i \neq j \implies \phi(u_i) \neq \phi(u_j)$.*

We also prove that both Definition 3.4 and Definition 3.6 are satisfied by the encryption policy graph generated by the algorithm in Figure 6.

LEMMA 4.2 LOCAL COVER AND NON-REDUNDANCY. *Let $\mathcal{A}=\langle\mathcal{U}, \mathcal{R}, \mathcal{P}\rangle$ be an authorization policy. Algorithm $\mathcal{A2E}$ creates a key and token graph $\mathcal{G}_{\mathcal{K}, \mathcal{T}}=\langle V_{\mathcal{K}, \mathcal{T}}, E_{\mathcal{K}, \mathcal{T}}\rangle$ and the corresponding encryption policy $\mathcal{E}=\langle\mathcal{U}, \mathcal{R}, \mathcal{K}, \mathcal{L}, \phi, \mathcal{T}\rangle$ such that $\mathcal{G}_{\mathcal{E}}$ satisfies the local cover property (Definition 3.4) and is non-redundant (Definition 3.6).*

By combining Lemma 4.1 and in Lemma 4.2, we can conclude that the encryption policy generated by the algorithm in Figure 6 is equivalent to the authorization policy provided in input.

THEOREM 4.3 POLICY EQUIVALENCE. *Let $\mathcal{A}=\langle\mathcal{U}, \mathcal{R}, \mathcal{P}\rangle$ be an authorization policy. Algorithm $\mathcal{A}2\mathcal{E}$ creates a key and token graph $\mathcal{G}_{\mathcal{K},\mathcal{T}}=\langle V_{\mathcal{K},\mathcal{T}}, E_{\mathcal{K},\mathcal{T}}\rangle$ and the corresponding encryption policy $\mathcal{E}=\langle\mathcal{U}, \mathcal{R}, \mathcal{K}, \mathcal{L}, \phi, \mathcal{T}\rangle$ such that $\mathcal{A} \equiv \mathcal{E}$.*

The following theorem proves that the encryption policy generated by algorithm $\mathcal{A}2\mathcal{E}$ has a total number of keys and tokens that is much less than the number of users, resources, and permissions composing a given authorization policy, thus greatly reducing the overhead on the users in deriving the keys of the resources they are entitled to access, as also the experiments in Section 9 show.

THEOREM 4.4. *Let $\mathcal{A}=\langle\mathcal{U}, \mathcal{R}, \mathcal{P}\rangle$ be an authorization policy. Algorithm $\mathcal{A}2\mathcal{E}$ creates a key and token graph $\mathcal{G}_{\mathcal{K},\mathcal{T}}=\langle V_{\mathcal{K},\mathcal{T}}, E_{\mathcal{K},\mathcal{T}}\rangle$ and the corresponding encryption policy $\mathcal{E}=\langle\mathcal{U}, \mathcal{R}, \mathcal{K}, \mathcal{L}, \phi, \mathcal{T}\rangle$ such that $|\mathcal{K} \cup \mathcal{T}| \ll |\mathcal{U} \cup \mathcal{R} \cup \mathcal{P}|$.*

Finally, we prove that the proposed algorithm has polynomial time complexity.

THEOREM 4.5. *Let $\mathcal{A}=\langle\mathcal{U}, \mathcal{R}, \mathcal{P}\rangle$ be an authorization policy. Algorithm $\mathcal{A}2\mathcal{E}$ creates an encryption policy $\mathcal{E} = \langle\mathcal{U}, \mathcal{R}, \mathcal{K}, \mathcal{L}, \phi, \mathcal{T}\rangle$ such that $\mathcal{A} \equiv \mathcal{E}$ in time $O((|\mathcal{R}| + |V_{\mathcal{K},\mathcal{T}}|^2) \cdot |\mathcal{U}|)$.*

5. POLICY UPDATES

Since the authorization policy is likely to change over time, the corresponding encryption policy needs to be re-arranged accordingly. The possible policy update operations are: 1) insertion/deletion of a user; 2) insertion/deletion of a resource; and 3) grant/revoke of a permission. We note that the insertion/deletion of users has an impact on the encryption policy only when the users gain permissions. In this case, inserting (deleting, resp.) a user implies granting (revoking, resp.) all the permissions in which the user is involved. Analogously, the insertion/deletion of resources has an impact on the encryption policy only when the resources are made accessible to users. Therefore, inserting (deleting, resp.) a resource implies granting (revoking, resp.) all the authorizations in which the resource is involved. For this reason, we focus on the grant and revoke operations. Also, we assume that each operation always refers to a single user u and a single resource r ; extension to sets of users and resources is immediate.

The grant and revoke operations on the authorization policy \mathcal{A} are translated into operations that properly update the encryption policy graph to guarantee that \mathcal{E} is equivalent to \mathcal{A} also after grant/revoke operations. Creating from scratch the encryption policy graph every time there is a grant or revoke operation is obviously too expensive, since it requires to re-generate the whole set of keys and tokens and to re-encrypt all the resources in the system. Therefore, we propose a strategy that updates the existing encryption policy graph, changing only the portions of the graph that are affected by the grant or revoke operation.

5.1 Grant and revoke

Every grant/revoke request for a user u on a resource r has the effect of changing the set of users that can access r and always requires the data owner to decrypt and

```

GRANT_REVOKE( $u, r, operation$ )
/* update the access control list of  $r$  */
find the vertex  $v_{old}$  with  $v_{old}.label = \phi(r)$ 
case  $operation$  of
  'grant':  $acl(r) := v_{old}.acl \cup \{u\}$ 
  'revoke':  $acl(r) := v_{old}.acl \setminus \{u\}$ 
find the vertex  $v_{new}$  with  $v_{new}.acl = acl(r)$ 
if  $v_{new} = \text{UNDEF}$  then
   $v_{new} := \text{create\_new\_vertex}(acl(r))$  /* see Figure 12 */
 $\phi(r) := v_{new}.label$ 
/* re-encrypt resource  $r$  */
download the encrypted version  $r^k$  of  $r$  from the server
decrypt  $r^k$  with key  $v_{old}.key$  to retrieve the original resource  $r$ 
encrypt  $r$  with key  $v_{new}.key$ 
upload the new encrypted version  $r^k$  of  $r$  on the server
update table LABELS on the server
delete\_vertex( $v_{old}$ ) /* see Figure 12 */

```

Fig. 11. Procedure for granting or revoking permission $\langle u, r \rangle$

to re-encrypt the resource with a new key that should be (directly or indirectly) derivable only by the users that belong to the new access control list. Figure 11 illustrates procedure **grant_revoke** that implements the grant and revoke operations. The procedure takes as input a user u , a resource r , and the type of operation that has to be executed, which can be either ‘grant’ or ‘revoke’, and modifies the encryption policy accordingly. First, the procedure retrieves vertex v_{old} whose acl corresponds to the current acl of r and sets $acl(r)$ to the old acl with user u added (grant) or removed (revoke). Since, according to our approach (see Section 3), each resource has to be encrypted with the key associated with the vertex that represents its acl , the procedure checks the existence of a vertex v_{new} in the encryption policy graph representing the new value of $acl(r)$. If such a vertex does not exist, vertex v_{new} is created and inserted in the graph (function **create_new_vertex**). The procedure then downloads the resource from the server, decrypts it with $v_{old}.key$, re-encrypts it with $v_{new}.key$, and uploads the new encrypted version of r on the server. Finally, the procedure calls **delete_vertex** on vertex v_{old} that checks if vertex v_{old} is still needed or if can be removed from the graph.

The insertion and removal of vertices in the encryption policy graph are realized through function **create_new_vertex** in and procedure **delete_vertex** in Figure 12. Note that function **create_new_vertex** and procedure **delete_vertex** are based on the same operations (i.e., **cover_vertex** and **factorize**) used by the algorithm in Figure 6 for initially generating the encryption policy graph. The only difference is that in Figure 12 these operations work locally to the vertex inserted in or removed from the graph.

Function **create_new_vertex** receives as input a set U of users and returns the vertex v , representing U , inserted in the graph. The function first copies the current sets $V_{\mathcal{K}, \mathcal{T}}$ of vertices and $E_{\mathcal{K}, \mathcal{T}}$ of edges in two local variables V and E , respectively. This copy is needed to easily compute the vertices and edges inserted into, or removed from, $V_{\mathcal{K}, \mathcal{T}}$ and $E_{\mathcal{K}, \mathcal{T}}$, respectively, to modify the encryption policy accordingly. In fact, the presence of a new vertex requires the generation of a new

```

CREATE_NEW_VERTEX( $U$ )
/* initial key and token graph
   vertices and edges */
 $V := V_{\mathcal{K},\mathcal{T}}$ 
 $E := E_{\mathcal{K},\mathcal{T}}$ 
/* create the new vertex */
create vertex  $v$ 
 $v.acl := U$ 
 $v.key := \text{NULL}$ 
 $v.label := \text{NULL}$ 
for each  $u \in v.acl$  do
   $v.counter[u] := 0$ 
/* cover  $v$ , remove redundancies,
   and factorize common ancestors */
cover_vertex( $v, v.acl$ )
factorize( $v$ )
/* update encryption policy
   (see Figure 13) */
update_encryption_policy( $V, E$ )
for each  $v_i \in \{v_j | (v_j, v_h) \in (E \setminus E_{\mathcal{K},\mathcal{T}})\}$  do
  delete_vertex( $v_i$ )
return( $v$ )

DELETE_VERTEX( $v$ )
if ( $|v.acl| > 1$ )  $\wedge$  ( $\nexists r \in \mathcal{R}: \phi(r) = v.label$ ) then
/* direct ancestors and descendants of  $v$  */
 $Anc := \{v_i | (v_i, v) \in E_{\mathcal{K},\mathcal{T}}\}$ 
 $Desc := \{v_i | (v, v_i) \in E_{\mathcal{K},\mathcal{T}}\}$ 
if ( $|Desc| \cdot |Anc| \leq (|Desc| + |Anc|)$ ) then
/* initial key and token graph vertices and edges */
 $V := V_{\mathcal{K},\mathcal{T}}$ 
 $E := E_{\mathcal{K},\mathcal{T}}$ 
/* update the key and token graph */
 $E_{\mathcal{K},\mathcal{T}} := E_{\mathcal{K},\mathcal{T}} \setminus (\{(v, v_i) \in E_{\mathcal{K},\mathcal{T}}\} \cup \{(v_i, v) \in E_{\mathcal{K},\mathcal{T}}\})$ 
for each  $(v, v_i): v_i \in Desc$  do
  for each  $u \in v.acl$  do
     $v_i.counter[u] := v_i.counter[u] - 1$ 
   $tocover := \{u | u \in v_i.acl \wedge v_i.counter[u] = 0\}$ 
  cover_vertex( $v_i, tocover$ )
  factorize( $v_i$ )
 $V_{\mathcal{K},\mathcal{T}} := V_{\mathcal{K},\mathcal{T}} - \{v\}$ 
/* update encryption policy (see Figure 13) */
update_encryption_policy( $V, E$ )
for each  $v_i \in \{v_j | (v_j, v_h) \in (E \setminus E_{\mathcal{K},\mathcal{T}})\}$  do
  delete_vertex( $v_i$ )

```

Fig. 12. Function that inserts a new vertex representing U and procedure for deleting vertex v

key and label and the removal of a vertex requires the deletion of the corresponding key and label. Analogously, the presences of a new edge requires the generation of the corresponding token, which is then stored in table `TOKENS`, and the removal of an edge requires the deletion of the corresponding token from table `TOKENS`. Function `create_new_vertex` then creates a vertex v for which $v.acl$ is set to U while $v.key$ and $v.label$ are both set to `NULL`. Vertex v is then covered by other vertices in the graph by calling: 1) procedure `cover_vertex` on v and $v.acl$, to ensure that the vertex is inserted without introducing redundant edges and in such a way that the local cover property is satisfied; and 2) procedure `factorize`, which determines whether the new vertex has more than two direct ancestors in common with other vertices in the graph and possibly factorizes them by adding a non-material vertex. Function `create_new_vertex` then calls procedure `update_encryption_policy` in Figure 13. This procedure takes as input the copies of the old sets of vertices and edges stored in V and E , respectively, and updates the encryption policy by generating and adding the new keys and labels associated with the new vertices (i.e., vertices in $V_{\mathcal{K},\mathcal{T}} \setminus V$), by: *i*) computing and adding the new tokens corresponding to the new edges (i.e., edges in $E_{\mathcal{K},\mathcal{T}} \setminus E$), and *ii*) removing the keys, labels, and tokens that are not needed anymore (i.e., vertices in $V \setminus V_{\mathcal{K},\mathcal{T}}$ and edges in $E \setminus E_{\mathcal{K},\mathcal{T}}$). Finally, for each vertex v_i that appears as starting point of a removed edge, `create_new_vertex` calls procedure `delete_vertex` to check whether vertex v_i can be removed from the graph. Note that we do not call procedure `delete_vertex` on the vertices appearing as ending points of removed edges since, by definition, they correspond to material vertices or have at least two incoming edges and therefore are always useful (or, in the worst case, ineffective) for reducing the number of

```

UPDATE_ENCRYPTION_POLICY( $V, E$ )
for each  $v \in (V_{\mathcal{K}, \mathcal{T}} \setminus V)$  do /* new vertices */
  generate key  $k$ 
   $v.key := k$ 
  generate label  $l$ 
   $v.label := l$ 
   $\mathcal{K} := \mathcal{K} \cup \{v.key\}$ 
   $\mathcal{L} := \mathcal{L} \cup \{v.label\}$ 
for each  $(v_i, v_j) \in (E_{\mathcal{K}, \mathcal{T}} \setminus E)$  do /* new edges */
   $t_{i,j} := v_j.key \oplus h(v_i.key, v_j.label)$ 
   $\mathcal{T} := \mathcal{T} \cup \{t_{i,j}\}$ 
  upload token  $t_{i,j}$  on the server by adding it to table TOKENS
for each  $v \in (V \setminus V_{\mathcal{K}, \mathcal{T}})$  do /* vertices removed */
   $\mathcal{K} := \mathcal{K} \setminus \{v.key\}$ 
   $\mathcal{L} := \mathcal{L} \setminus \{v.label\}$ 
for each  $(v_i, v_j) \in (E \setminus E_{\mathcal{K}, \mathcal{T}})$  do /* edges removed */
   $\mathcal{T} := \mathcal{T} \setminus \{t_{i,j}\}$ 
  remove  $t_{i,j}$  from table TOKENS on the server

```

Fig. 13. Procedure for updating the encryption policy

tokens in the encryption policy graph.

Procedure **delete_vertex** receives as input a vertex v and removes it from the graph if it is neither necessary for policy enforcement nor useful for reducing the number of tokens. Hence, if the key associated with v is no longer used for encrypting any resource and the vertex is no longer needed for factorizing common ancestors, vertex v and all its ingoing and outgoing edges are removed. At this point, the key and token graph violates the local cover property since, by construction (see Lemma 4.2), the graph has no redundant edges and therefore all vertices in $Desc$ (i.e., the direct descendants of v) are no more properly covered. For each direct descendant v_i of v , procedure **delete_vertex** first calls procedure **cover_vertex** on v_i and on the set of users that do not belong to any other direct ancestor of v_i , and then calls procedure **factorize** on v_i . Like for procedure **create_new_vertex**, the encryption policy is then updated by calling procedure **update_encryption_policy**. Finally, for each vertex v_i that appears as starting point of a removed edge, **delete_vertex** recursively calls itself to check if vertex v_i can be removed from the graph and possibly removes it.

EXAMPLE 5.1. Consider the encryption policy illustrated in Figures 10(b) and (c). Figure 14 illustrates the key and token graph and table **LABELS** resulting from granting to D access to r_3 and revoking from F access to r_8 . (Note that for all users $u \in \mathcal{U}$, we do not report $\phi(u)$ since grant/revoke operations do not change it.)

—**grant_revoke**($D, r_3, grant$). First, the procedure identifies the vertex whose key is necessary for decrypting r_3 , which is v_7 . Then, $acl(r_3)$ is updated by inserting D . Since there is not a vertex representing $\{BCD\}$, procedure **create_new_vertex** is called with $U = \{BCD\}$ as a parameter. The procedure creates, and inserts in the graph, a new vertex v_{12} , with $v_{12}.acl = \{BCD\}$. Then, r_3 is downloaded from the server, decrypted with $v_7.key$, encrypted with $v_{12}.key$, and then uploaded on the server. Finally, procedure **delete_vertex** is called with v_7 as a parameter

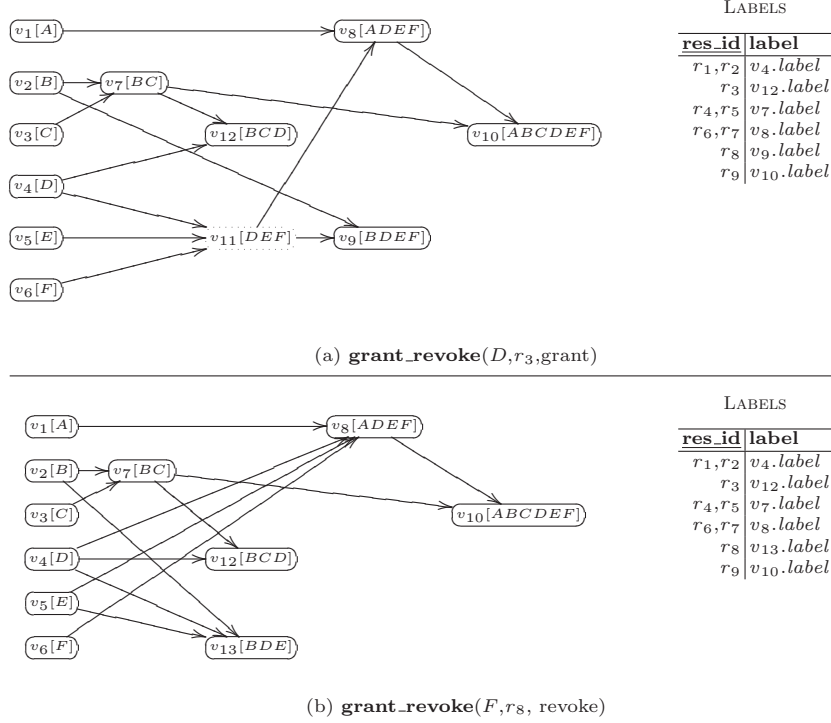


Fig. 14. Examples of grant and revoke operations

and, since $v_7.\text{key}$ is used to encrypt r_4 and r_5 , vertex v_7 is not removed from the graph.

- **grant_revoke**(F, r_8, revoke). First, the procedure identifies the vertex whose key is necessary for decrypting r_8 , which is v_9 . Then, $\text{acl}(r_8)$ is updated by removing F . Since there is not a vertex representing $\{BDE\}$, procedure **create_new_vertex** is called with $U=\{BDE\}$ as a parameter. The procedure creates, and inserts in the graph, a new vertex v_{13} , with $v_{13}.\text{acl}=\{BDE\}$. Then, r_8 is downloaded from the server, decrypted with $v_9.\text{key}$, encrypted with $v_{13}.\text{key}$, and uploaded on the server. Then, procedure **delete_vertex** is called with v_9 as a parameter. Since $v_9.\text{key}$ was only used for encrypting r_8 , v_9 is no longer a useful vertex and it is removed from the graph. The procedure recursively calls itself first with v_2 and then with v_{11} as a parameter. Vertex v_2 is not removed from the graph since it corresponds to user B , while vertex v_{11} is removed.

5.2 Correctness

We now prove that the procedure implementing the grant and revoke operations (Figure 11) preserves the policy equivalence between the resulting authorization and encryption policies. We first prove that procedure **delete_vertex** (Lemma 5.1) and function **create_new_vertex** (Lemma 5.2) modify the encryption policy graph by preserving both the local cover property and the non-redundancy property. This implies that the equivalence of the resulting encryption policy with the given au-

thorization policy is preserved.

LEMMA 5.1. *Let $\mathcal{A} = \langle \mathcal{U}, \mathcal{R}, \mathcal{P} \rangle$ be an authorization policy and $\mathcal{E} = \langle \mathcal{U}, \mathcal{R}, \mathcal{K}, \mathcal{L}, \phi, \mathcal{T} \rangle$ be an encryption policy such that $\mathcal{A} \equiv \mathcal{E}$. Procedure **delete_vertex** generates a new encryption policy $\mathcal{E}' = \langle \mathcal{U}, \mathcal{R}, \mathcal{K}', \mathcal{L}', \phi', \mathcal{T}' \rangle$ such that $\mathcal{A} \equiv \mathcal{E}'$.*

LEMMA 5.2. *Let $\mathcal{A} = \langle \mathcal{U}, \mathcal{R}, \mathcal{P} \rangle$ be an authorization policy and $\mathcal{E} = \langle \mathcal{U}, \mathcal{R}, \mathcal{K}, \mathcal{L}, \phi, \mathcal{T} \rangle$ be an encryption policy such that $\mathcal{A} \equiv \mathcal{E}$. Function **create_new_vertex** generates a new encryption policy $\mathcal{E}' = \langle \mathcal{U}, \mathcal{R}, \mathcal{K}', \mathcal{L}', \phi', \mathcal{T}' \rangle$ such that $\mathcal{A} \equiv \mathcal{E}'$.*

By combining Lemma 5.1 and Lemma 5.2, we conclude that the encryption policy modified by procedure **grant_revoke** in Figure 11 is equivalent to the authorization policy modified by the same procedure through a grant or revoke operation.

THEOREM 5.3. *Let $\mathcal{A} = \langle \mathcal{U}, \mathcal{R}, \mathcal{P} \rangle$ be an authorization policy and $\mathcal{E} = \langle \mathcal{U}, \mathcal{R}, \mathcal{K}, \mathcal{L}, \phi, \mathcal{T} \rangle$ be an encryption policy such that $\mathcal{A} \equiv \mathcal{E}$. Procedure **grant_revoke** generates a new authorization policy $\mathcal{A}' = \langle \mathcal{U}, \mathcal{R}, \mathcal{P}' \rangle$ and a new encryption policy $\mathcal{E}' = \langle \mathcal{U}, \mathcal{R}, \mathcal{K}', \mathcal{L}', \phi', \mathcal{T}' \rangle$ such that $\mathcal{A}' \equiv \mathcal{E}'$.*

6. TWO-LAYER ENCRYPTION FOR POLICY OUTSOURCING

The model described in the previous sections assumes that keys and tokens are computed, on the basis of the existing authorization policy, prior to sending the encrypted resources to the server. As described in Section 5, when permissions are updated by the data owner, the data owner interacts with the service provider for modifying the catalog and for re-encrypting the resources involved in the update. Even if the computation and communication overhead caused by policy updates is limited, the data owner may not have the computational or bandwidth resource availability for managing policy changes.

To further reduce the data owner's overhead, we put forward the idea of outsourcing to the server, besides the resource storage, the authorization policy management as well. Note that this delegation is possible since the server is considered trustworthy to properly carry out the service. Recall, however, that the server is not trusted with confidentiality (i.e., it is honest but curious). For this reason, our solution has been designed taking into account, and therefore minimizing, the risk that the server colludes with users to breach data confidentiality (see Section 8). The solution we propose enforces policy changes on encrypted resources themselves, without need of decrypting them, and can then be performed by the server.

6.1 Two-layer encryption

To delegate policy changes enforcement to the server, avoiding re-encryption for the data owner, we adopt a two-layer encryption approach. The owner encrypts the resources and sends them to the server in encrypted form; the server can impose another layer of encryption, following directions by the data owner. In terms of efficiency, the use of a double layer of encryption does not appear as a significant computational burden. Experience shows that current systems have no significant delay when managing encryption on data coming from either the network or local

disks, as also testified by the widespread use of encryption on network traffic and for protecting the storage of data on local file systems [Schneier et al. 1998].

We then distinguish two layers of encryption.

- **Base Encryption Layer (BEL)**, performed by the data owner before transmitting the resources to the server. It enforces encryption on the resources according to the policy existing at initialization time.
- **Surface Encryption Layer (SEL)**, performed by the server over the resources already encrypted by the data owner. It enforces the dynamic changes over the policy.

Both layers enforce encryption by means of a set of symmetric keys and a set of public tokens between these keys (see Section 2), although some adaptations are necessary, as explained below.

Base Encryption Layer. Compared with the model presented in Section 2, at the BEL we distinguish two kinds of keys: *derivation keys* and *access keys*. Access keys are actually used to encrypt resources, while derivation keys are used to provide the derivation capability via tokens, that is, tokens can be defined only with derivation keys as starting points. Each derivation key k is always associated with an access key k_a obtained by applying a secure hash function to k , that is, $k_a = h(k)$. In other words, keys at the BEL always go in pairs $\langle k, k_a \rangle$. Note that the derivation and the access keys are associated with a unique label, l and l_a , respectively. The rationale for this evolution is to distinguish the two roles associated with keys, namely: enabling key derivation (by applying the corresponding tokens) and enabling resource access. The reason for which such a distinction is needed will be clear in Section 7.

The BEL is characterized by an encryption policy $\mathcal{E}_b = \langle \mathcal{U}, \mathcal{R}, \mathcal{K}_b, \mathcal{L}_b, \phi_b, \mathcal{T}_b \rangle$, where \mathcal{U} , \mathcal{R} , and \mathcal{T}_b are as described in Section 2, \mathcal{K}_b is the set of (derivation and access) keys defined at the BEL, and \mathcal{L}_b is the set of publicly available labels associated with derivation and access keys. The *key assignment and encryption schema* $\phi_b : \mathcal{U} \cup \mathcal{R} \mapsto \mathcal{L}_b$ associates with each user $u \in \mathcal{U}$ the label l corresponding to the *derivation* key released to the user by the data owner, and with each resource $r \in \mathcal{R}$ the label l_a corresponding to the *access* key with which the resource is encrypted by the data owner.

The set \mathcal{K}_b of keys and the set \mathcal{T}_b of tokens can be graphically represented through the corresponding key and token graph, which now has a vertex b characterized by: a derivation key along with the corresponding label, denoted $b.key$ and $b.label$, respectively; and an access key along with the corresponding label, denoted $b.key_a$ and $b.label_a$, respectively. For each token in \mathcal{T}_b that allows the derivation of either k_j or k_{j_a} from k_i , there is an edge (b_i, b_j) in the graph. Graphically, a vertex is simply represented by b and tokens leading to derivation keys are distinguished from tokens leading to access keys by using dotted lines for the latter. The corresponding encryption policy \mathcal{E}_b is graphically represented by an encryption policy graph $\mathcal{G}_{\mathcal{E}_b}$, as described in Section 2, where notation $u \xrightarrow{\mathcal{E}_b} r$ indicates that there exists a path connecting u with r , either following tokens or applying secure hash function h . Note that dotted edges can only appear as the last step of a path in the graph (since they allow the derivation of access keys only). Figure 15(a) illustrates an example of the BEL key and token graph and of the key assignment and encryption

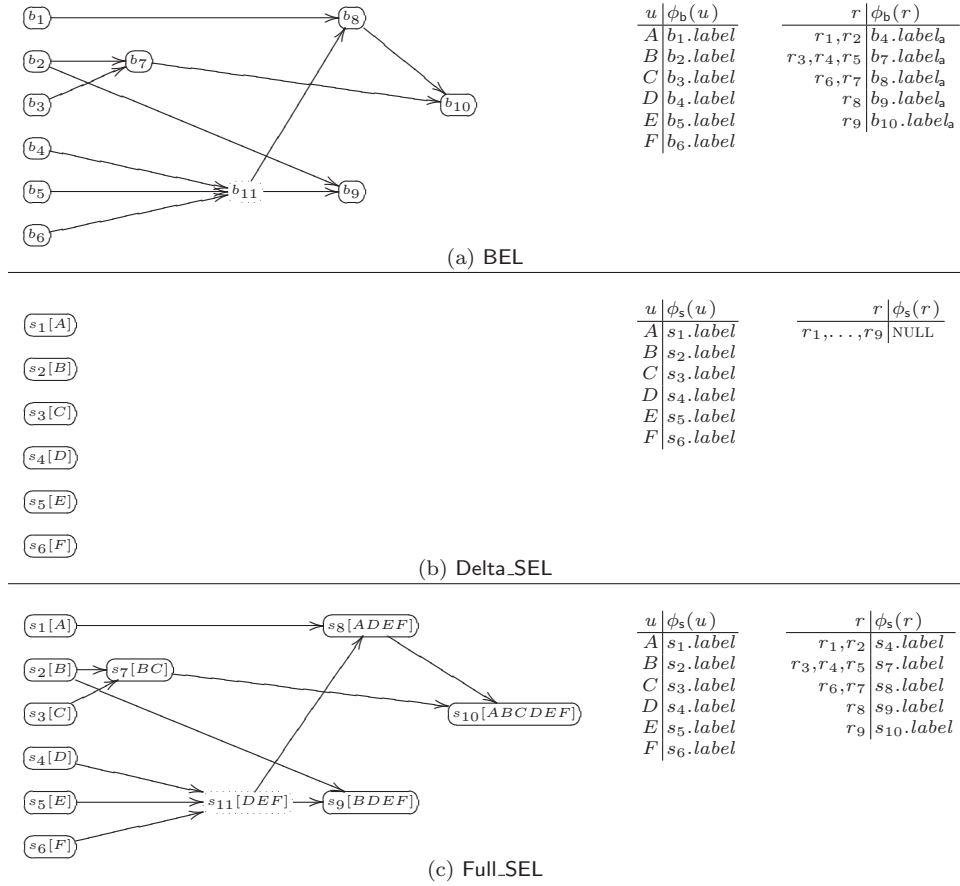


Fig. 15. An example of BEL and SEL combination with the Delta_SEL and the Full_SEL approaches

schema enforcing the authorization policy in Figure 2. In this example, all tokens lead to derivation keys.

Surface Encryption Layer. At the SEL there is no distinction between derivation and access keys (intuitively a single key carries out both functions). The SEL is therefore characterized by an encryption policy $\mathcal{E}_s = \langle \mathcal{U}, \mathcal{R}, \mathcal{K}_s, \mathcal{L}_s, \phi_s, \mathcal{T}_s \rangle$ that is defined and graphically represented as described in Section 2. Hence, as illustrated in Section 2, the set \mathcal{K}_s of keys and the set \mathcal{T}_s of tokens are graphically represented through a key and token graph having a vertex s for each pair $\langle k, l \rangle$ defined at the SEL and an edge (s_i, s_j) if there is a token in \mathcal{T}_s allowing the derivation of k_j from k_i . Each vertex s in the graph is characterized by: a key, denoted $s.key$, and corresponding label, denoted $s.label$; and the set of users, denoted $s.acl$, who can derive $s.key$. The corresponding encryption policy \mathcal{E}_s is graphically represented by an encryption policy graph as described in Section 2, where notation $u \xrightarrow{\mathcal{E}_s} r$ indicates that there is a path connecting u with r .

BEL and SEL combination. In the two-layer approach, each resource can then be encrypted twice: at the BEL first, and then at the SEL. Users can access resources only passing through the SEL. Each user u receives two keys: one to access the BEL and the other to access the SEL.⁴ Users will be able to access resources for which they know both the keys (BEL and SEL) used for encryption.

The consideration of the two layers requires to restate the definition of policy equivalence, which is now defined as follows.

Definition 6.1 Policy equivalence with two-layer encryption. Let $\mathcal{A} = \langle \mathcal{U}, \mathcal{R}, \mathcal{P} \rangle$ be an authorization policy, $\mathcal{E}_b = \langle \mathcal{U}, \mathcal{R}, \mathcal{K}_b, \mathcal{L}_b, \phi_b, \mathcal{T}_b \rangle$ be a BEL encryption policy, and $\mathcal{E}_s = \langle \mathcal{U}, \mathcal{R}, \mathcal{K}_s, \mathcal{L}_s, \phi_s, \mathcal{T}_s \rangle$ be a SEL encryption policy. \mathcal{A} and the pair $\langle \mathcal{E}_b, \mathcal{E}_s \rangle$ are equivalent, denoted $\mathcal{A} \equiv \langle \mathcal{E}_b, \mathcal{E}_s \rangle$, iff the following conditions hold:

$$\begin{aligned} & \neg \forall u \in \mathcal{U}, r \in \mathcal{R} : u \xrightarrow{\mathcal{E}_b} r \wedge u \xrightarrow{\mathcal{E}_s} r \implies u \xrightarrow{\mathcal{A}} r \\ & \neg \forall u \in \mathcal{U}, r \in \mathcal{R} : u \xrightarrow{\mathcal{A}} r \implies u \xrightarrow{\mathcal{E}_b} r \wedge u \xrightarrow{\mathcal{E}_s} r \end{aligned}$$

In principle, the encryption policies at the BEL and at the SEL can be arbitrarily defined, as long as their combination is equivalent to the authorization policy. Let \mathcal{A} be the authorization policy at the initialization time and let \mathcal{E}_b be the encryption policy at the BEL, which is equivalent to \mathcal{A} (i.e., $\mathcal{A} \equiv \mathcal{E}_b$). We envision two approaches that can be followed in the construction of the two layers.

Full_SEL. The SEL encryption policy is initialized to reflect exactly (i.e., to repeat) the BEL encryption policy: for each derivation key in BEL, a corresponding key is defined in SEL; for each token in BEL, a corresponding token is defined in SEL. Note that the set \mathcal{K}_s of keys and the set \mathcal{T}_s of tokens form a key and token graph which is isomorphic to the one existing at the BEL and, therefore, also $\mathcal{G}_{\mathcal{E}_s}$ is isomorphic to $\mathcal{G}_{\mathcal{E}_b}$. The key assignment and encryption schema assigns to each user u a unique label $\phi_s(u) = v_s.label$ (and therefore a unique key $v_s.key$) corresponding to $\phi_b(u) = v_b.label$ (i.e., let f be the isomorphism between $\mathcal{G}_{\mathcal{E}_b}$ and $\mathcal{G}_{\mathcal{E}_s}$, then $v_s = f(v_b)$). Also, it assigns to each resource r a unique label $\phi_s(r) = v_s.label$ (and therefore a unique key $v_s.key$) corresponding to $\phi_b(r) = v_b.label$. The SEL encryption policy models exactly the BEL encryption policy, and hence, by definition, is equivalent to the authorization policy (i.e., $\mathcal{A} \equiv \mathcal{E}_s$).

Delta_SEL. The SEL policy is initialized to not carry out any over-encryption. Each user u is assigned a unique label $\phi_s(u) = v_s.label$, and therefore a unique key $v_s.key$, where $v_s.acl = \{u\}$. No encryption is performed on resources, that is, $\forall r \in \mathcal{R}, \phi_s(r) = \text{NULL}$. Here, the SEL itself does not provide any additional protection at start time, but it does not modify the accesses allowed by BEL.

We note that a third approach could be possible, where the authorization policy enforcement is completely delegated at the SEL and the BEL simply applies a uniform encryption (i.e., with the same key released to all users) to protect the plaintext content from the server's eyes. We do not consider this approach as it presents a significant exposure to collusion (see Section 8).

⁴To simplify key management, the user key for SEL can be obtained by the application of a secure hash function from the user key for BEL. In the initialization phase, the data owner can send to the server the SEL keys.

All the approaches described produce a correct two layer encryption, that is, given a correct encryption policy at the BEL, the approaches produce a SEL such that the authorization policy \mathcal{A} and the pair $\langle \mathcal{E}_b, \mathcal{E}_s \rangle$ are equivalent.

The reason for considering both the Full_SEL and Delta_SEL approaches is the different performance and protection guarantees that they enjoy. In particular, Full_SEL always requires double encryption to be enforced (even when permissions remain unvaried), thus doubling the decryption load of users for each access. By contrast, the Delta_SEL approach requires double encryption only when actually needed to enforce a change in the permissions. However, as we will see in Section 8, the Delta_SEL is characterized by greater information exposure than the Full_SEL approach. The choice between one or the other can then be a trade-off between costs and resilience to attacks.

We close this section with a remark on the implementation. In the illustration of our approach, we always assume over-encryption to be managed with a direct and complete encryption and decryption of the resource, as needed. We note however that the server can, at the SEL, apply a *lazy encryption* approach, similar to the *copy-on-write* (COW) strategy used by most operating systems, and actually over-encrypt the resource when it is first accessed (and then storing the computed encrypted representation). The server may choose also to always store the BEL representation and then dynamically apply the encryption driven by the SEL when users access the resource.

7. POLICY UPDATES IN TWO-LAYER ENCRYPTION

While in the basic model described in Section 2 policy updates are enforced by the owner (Section 5), the two-layer approach enables the enforcement of policy updates without the need for the owner to re-encrypt and to resend resources to the server. By contrast, the owner just adds (if necessary) some tokens at the BEL and delegates policy changes to the SEL by possibly requesting the server to over-encrypt some resources. The SEL (enacted by the server) receives *over-encryption* requests by the BEL (under the control of the data owner) and operates accordingly, adjusting tokens and possibly encrypting (and/or decrypting) resources.

Before analyzing grant and revoke operations in this new scenario, we first describe the working of over-encryption at the SEL.

7.1 Over-encrypt

The SEL regulates the update process by over-encrypting of resources. It receives from the BEL requests of the form **over_encrypt**(U, R) to make the set R of resources accessible only to users U . Note that the semantics is different in the two different encryption modes. In the Full_SEL approach, over-encryption must reflect the actual authorization policy existing at any given time. In other words, it must reflect, besides the - dynamic - policy changes (not reflected at the BEL), also the BEL policy itself. In the Delta_SEL approach, over-encryption is demanded only when additional restrictions (with respect to those enforced by the BEL) need to be enforced. As a particular case, in the Delta_SEL approach the set U of users may be ALL when while processing a grant operation the BEL determines that its protection is sufficient and therefore requests the SEL not to enforce any restriction and to possibly remove an over-encryption previously imposed.

Let us then see how the procedure works. Procedure **over_encrypt** takes a set U of users and a set R of resources as input. First, it determines vertex s such that $s.label = \phi_s(r')$, with r' a resource in R (note that since all resources in R share the same key, it is sufficient to check the condition on $s.label$ on any resource r' in R). If such a vertex s exists and $s.acl = U$, resources in R are over-encrypted with a key ($s.key$) that all and only users in the current acl of resource in R can compute and therefore the procedure terminates. Otherwise, if such a vertex s exists and $s.acl \neq U$, the resources in R are first decrypted with $s.key$ and then procedure **over_encrypt** calls **delete_vertex** on s . At this point, procedure **over_encrypt** verifies whether the set of users that should be allowed access to the resources in R by the SEL is different from ALL. In this case, over-encryption is necessary. (No operation is executed otherwise, since $U = \text{ALL}$ is the particular case of Delta_SEL approach discussed above.) The procedure checks then the existence of a vertex s such that the set of users that can derive key $s.key$ (i.e., belonging to $s.acl$) corresponds to U . If such a vertex does not exist, it is created and inserted in the encryption policy graph at the SEL by function **create_new_vertex**. Then, for each resource r in R , the procedure encrypts r with $s.key$ and updates $\phi_s(r)$ and table LABELS accordingly.

7.2 Grant and revoke

Let us first consider procedure **grant** in Figure 16, which handles a request to grant user u access to resource r . The BEL starts and regulates the update process as follows. First, $acl(r)$ is updated to include u . Then, the procedure retrieves the vertex b_j whose access key $b_j.key_a$ is the key with which r is encrypted. If the resource's access key cannot be derived by u , then a new token from from key $b_i.key$ of the user, with b_i the vertex such that $\phi_b(u) = b_i.label$, to $b_j.key_a$ is generated and added to the token catalog. Note that the separation between derivation and access keys for each vertex allows us to add a token only giving u access to the key used to encrypt resource r , thus limiting the knowledge of u to the information strictly needed to guarantee equivalence with the authorization policy. Indeed, knowledge of $b_j.key_a$ is a necessary condition to make r accessible to u . Even if the knowledge of $b_j.key_a$ does not allow u to further derive keys and therefore to access resources with an acl different from $acl(r)$, there may be other resources that are encrypted with the same key $b_j.key_a$ and which should not be made accessible to u . Since releasing $b_j.key_a$ would make them accessible to u , they need to be over-encrypted so to make them accessible to users in their acs only. Then, the procedure determines if such a set of resources R' exists. If R' is not empty, the procedure partitions R' in sets such that each set $S \subseteq R'$ includes all resources characterized by the same acl , denoted acl_S . For each set S , the procedure calls **over_encrypt**(acl_S, S) to demand SEL to execute an over-encryption of S for users in acl_S . In addition, the procedure requests the SEL to synchronize itself with the policy change. Here, the procedure behaves differently depending on the encryption model assumed. In the case of Delta_SEL, the procedure first controls whether the set of users that can reach the resource's access key (i.e., the set of users $u \in \mathcal{U}$ such that $u \xrightarrow{\mathcal{E}_b} b_i$) corresponds to $acl(r)$. If so, the BEL encryption suffices and no protection is needed at the SEL, and therefore a call **over_encrypt**(ALL, $\{r\}$) is

BEL	SEL
<pre> GRANT(u, r) $acl(r) := acl(r) \cup \{u\}$ find vertex b_j with $b_j.label_a = \phi_b(r)$ if $u \xrightarrow{\mathcal{E}_b} r$ then find vertex b_i with $b_i.label = \phi_b(u)$ $t_{i,j} := b_j.key_a \oplus h(b_i.key, b_j.label_a)$ $\mathcal{T}_b := \mathcal{T}_b \cup \{t_{i,j}\}$ upload token $t_{i,j}$ on the server by storing it in table TOKENS $R' := \{r' \mid r' \neq r \wedge \phi_b(r') = \phi_b(r) \wedge \exists u \in \mathcal{U} : u \xrightarrow{\mathcal{E}_b} r \wedge u \notin acl(r')\}$ if $R' \neq \emptyset$ then Partition R' in sets such that each set S contains resources with the same acl acl_S for each set S do over_encrypt(acl_S, S) case encryption model of Delta_SEL: if $\{u \mid u \in \mathcal{U} : u \xrightarrow{\mathcal{E}_b} b_i\} = acl(r)$ then over_encrypt(ALL, $\{r\}$) else over_encrypt($acl(r)$, $\{r\}$) Full_SEL: over_encrypt($acl(r)$, $\{r\}$) REVOKE(u, r) $acl(r) := acl(r) - \{u\}$ over_encrypt($acl(r)$, $\{r\}$) </pre>	<pre> OVER_ENCRYPT(U, R) let r' be a resource in R find vertex s with $s.label = \phi_s(r')$ if ($s \neq \text{UNDEF} \wedge s.acl = U$) then exit else if $s \neq \text{UNDEF}$ then for each $r \in R$ do decrypt r with $s.key$ delete_vertex(s) if $U \neq \text{ALL}$ then find vertex s with $s.acl = U$ if $s = \text{UNDEF}$ then $s := \text{create_new_vertex}(U)$ for each $r \in R$ do $\phi_s(r) := s.label$ encrypt r with $s.key$ update LABELS on the server </pre>

Fig. 16. Procedures for granting and revoking permission $\langle u, r \rangle$

requested. Otherwise, a call **over_encrypt**($acl(r), \{r\}$) requests the SEL to make r accessible only to users in $acl(r)$. In the case of Full_SEL, procedure **grant** always calls **over_encrypt**($acl(r), r$), requesting the SEL to synchronize its policy so to make r accessible only by the users in $acl(r)$.

Let us now consider procedure **revoke** in Figure 16, which revokes from user u access to resource r . The procedure updates $acl(r)$ to remove user u and calls **over_encrypt**($acl(r), \{r\}$) to demand the SEL to make r accessible only to users in $acl(r)$.

In terms of performance, the grant and revoke procedures only require a direct navigation of the BEL and SEL structures and they determine the requests to be sent to the server in a time which, in typical scenarios, will be less than the time required to send the messages to the server.

EXAMPLE 7.1. Consider the two layer encryption policy in Figure 15. Figures 17 and 18 illustrate the evolution of the corresponding key and token graphs and of $\phi_b(r)$ and $\phi_s(r)$ for resources in \mathcal{R} when the grant and revoke operations listed below are executed. Note that we do not report $\phi_b(u)$ and $\phi_s(u)$ for users in \mathcal{U} since they never change upon grant/revoke operations. Note also that in the Full_SEL scenario the key and token graph at SEL evolves exactly as described in Example 5.1.

—**grant**(D, r_3). First, $acl(r_3)$ is updated by inserting D . Then, since access key

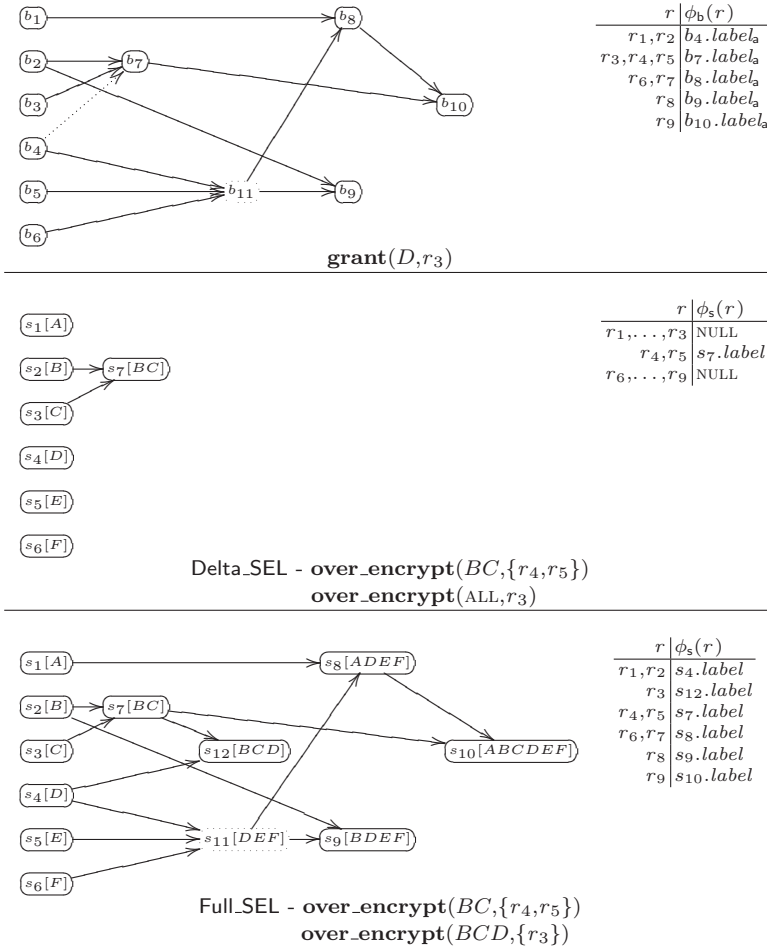


Fig. 17. An example of grant operation

$b_7.key_a$ used to encrypt r_3 cannot be derived from the derivation key of vertex b_4 corresponding to $\phi_b(D)$, a token allowing computation of $b_7.key_a$ from $b_4.key$ is added to BEL. Since $b_7.key_a$ is also used to encrypt resources r_4 and r_5 , which D is not authorized to view, these resources have to be over-encrypted so to make them accessible only to users B and C . In the Delta_SEL scenario, **over_encrypt** creates a new vertex s_7 , with $s_7.acl=\{BC\}$, for resources r_4 and r_5 ; the protection of resource r_3 at BEL level is instead sufficient and no over-encryption is needed (i.e., procedure **over_encrypt** is called with $U=ALL$). In the Full_SEL scenario, resources r_4 and r_5 are already correctly protected, r_3 is instead over-encrypted with the key of vertex s_{12} , which is created and inserted in the graph by function **create_new_vertex**; finally, procedure **delete_vertex** is called with s_7 as a parameter and, since $s_7.key$ is used to encrypt r_4 and r_5 , vertex s_7 is not removed from the graph.

—**revoke**(F, r_8). First, $acl(r_8)$ is updated by removing F . Since $acl(r_8)$ becomes

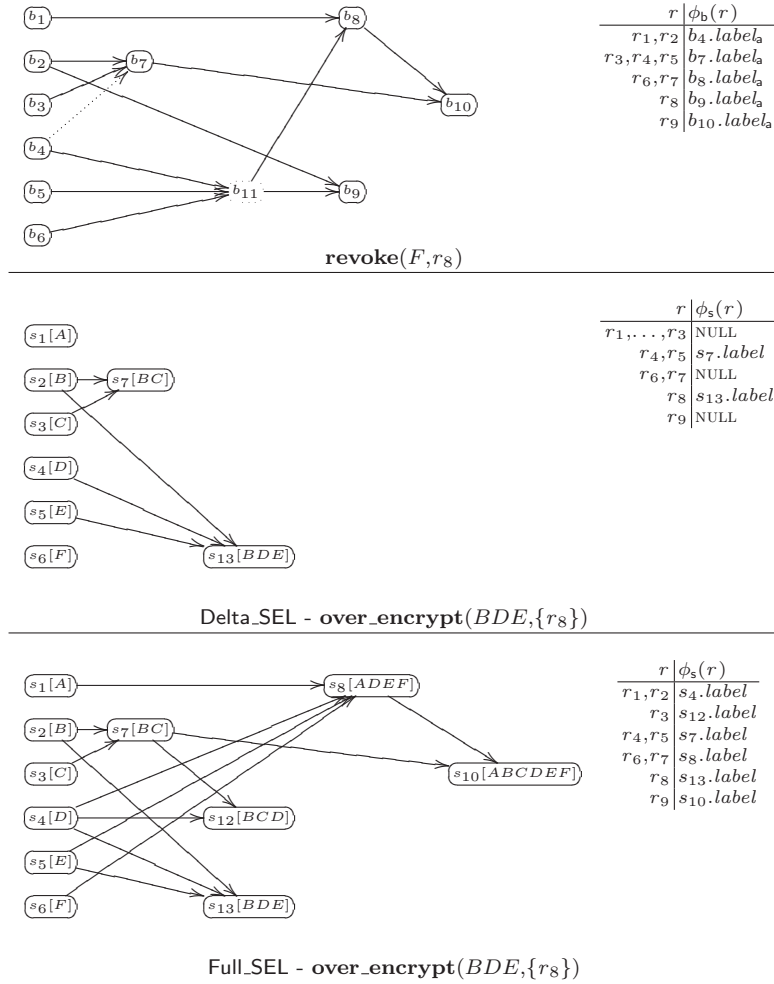


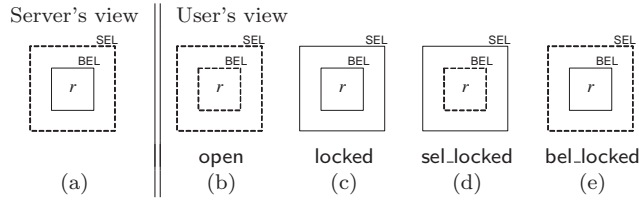
Fig. 18. An example of revoke operation

$\{BDE\}$, resource r_8 has to be over-encrypted with a key that only this set of users can compute. Consequently, in both the Delta_SEL and the Full_SEL scenario, a new vertex s_{13} representing $\{BDE\}$ is created and its key is used to encrypt r_8 . Also, in the Full_SEL scenario, procedure **delete_vertex** is called with s_9 as a parameter. Since s_9 is no longer a useful vertex, it is removed from the graph. The procedure recursively calls itself with s_2 and with s_{11} as a parameter. Vertex s_2 is not removed from the Full_SEL graph since it corresponds to user B while vertex s_{11} is removed.

7.3 Correctness

We now prove that the procedures implementing the grant and revoke operations preserve policy equivalence.

THEOREM 7.1. *Let $\mathcal{A} = \langle \mathcal{U}, \mathcal{R}, \mathcal{P} \rangle$ be an authorization policy,*

Fig. 19. Possible views on resource r

$\mathcal{E}_b = \langle \mathcal{U}, \mathcal{R}, \mathcal{K}_b, \mathcal{L}_b, \phi_b, \mathcal{T}_b \rangle$ be an encryption policy at the BEL, and $\mathcal{E}_s = \langle \mathcal{U}, \mathcal{R}, \mathcal{K}_s, \mathcal{L}_s, \phi_s, \mathcal{T}_s \rangle$ be an encryption policy at the SEL such that $\mathcal{A} \equiv \langle \mathcal{E}_b, \mathcal{E}_s \rangle$. The procedures in Figure 16 generate a new $\mathcal{E}_b' = \langle \mathcal{U}, \mathcal{R}, \mathcal{K}_b', \mathcal{L}_b', \phi_b', \mathcal{T}_b' \rangle$, $\mathcal{E}_s' = \langle \mathcal{U}, \mathcal{R}, \mathcal{K}_s', \mathcal{L}_s', \phi_s', \mathcal{T}_s' \rangle$, and \mathcal{A}' such that $\mathcal{A}' \equiv \langle \mathcal{E}_b', \mathcal{E}_s' \rangle$.

8. PROTECTION EVALUATION

Since the BEL and SEL encryption policies are jointly equivalent to the authorization policy at initialization time, the correctness of the procedures in Figure 16 ensures that the authorization policy \mathcal{A} and the pair $\langle \mathcal{E}_b, \mathcal{E}_s \rangle$ are equivalent. In other words, at any point in time, users will be able to access only resources for which they have - directly or indirectly - the necessary keys both at the BEL and at the SEL.

The key derivation process is proved to be secure [Atallah et al. 2005]. We also assume that all the encryption functions and the tokens are robust and cannot be broken, even combining the information available to many users. Moreover, we assume that each user correctly manages her keys, without the possibility for a user to steal keys from another user.

It still remains to evaluate whether the approach is vulnerable to attacks from users who access and store all information offered by the server, or from *collusion* attacks, where different users (or a user and the server) combine their knowledge to access resources they would not otherwise be able to access. Note that for collusion to exist, both parties should gain in the exchange (as otherwise they will not have any incentive in colluding).

To model exposure, we first examine the different views that one can have on a resource r by exploiting a graphical notation with resource r in the center and with fences around r denoting the barriers to the access imposed by the knowledge of the keys used for r 's encryption at the BEL (inner fence) and at the SEL (outer fence). The fence is continuous if there is no knowledge of the corresponding key (the barrier cannot be passed); it is discontinuous otherwise (the barrier can be passed). Figure 19 illustrates the different views that can exist on the resource. On the left, Figure 19(a), there is the view of the server itself, which knows the key at the SEL but does not have access to the key at the BEL. On the right, there are the different possible views of users, for whom the resource can be:

- open: the user knows the key at the BEL as well as the key at the SEL (Figure 19(b));
- locked: the user knows neither the key at the BEL nor the key at the SEL (Figure 19(c));

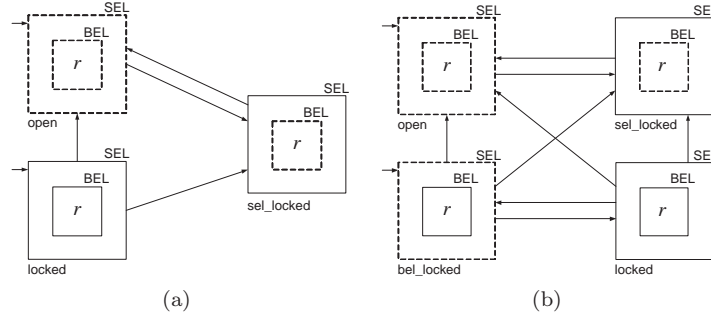


Fig. 20. View transitions in the Full_SEL (a) and in the Delta_SEL (b)

- `sel_locked`: the user knows only the key at the BEL but does not know the key at the SEL (Figure 19(d));
- `bel_locked`: the user knows only the key at the SEL but does not know the key at the BEL (Figure 19(e)). Note that this latter view corresponds to the view of the server itself.

By the authorization policy and encryption policy equivalence (Theorem 7.1), the `open` view corresponds to the view of authorized users, while the remaining views correspond to the views of non authorized users.

8.1 Exposure risk

We now discuss possible information exposure, with the conservative assumption that users are not oblivious (i.e., they have the ability to store and keep indefinitely all information they were entitled to access).

Full_SEL. In the Full_SEL approach, at initialization time, BEL and SEL are completely synchronized. For each user, a resource is then protected by both keys or by neither: authorized users will have the `open` view, while non authorized users will have the `locked` view. Figure 20(a) summarizes the possible view transitions starting from these two views.

Let us first examine the evolution of the `open` view. Since resources at the BEL are not re-encrypted, the view of an authorized user can change only if the user is revoked the permission. In this case, the resource is over-encrypted at the SEL, then becoming `sel_locked` for the user. The view can be brought back to be `open` if the user is granted the permission again (i.e., over-encryption is removed).

Let us now examine the evolution of the `locked` view. For how the SEL is constructed and maintained in the Full_SEL approach, it cannot happen that the SEL grants a user an access that is blocked at the BEL, and therefore the `bel_locked` view can never be reached. The view can instead change to `open`, if the user is granted the permission to access the resource; or to `sel_locked`, if the user is given the access key at the BEL but she is not given that at the SEL. This latter situation can happen if the release of the key at the BEL is necessary to make accessible to the user another resource r' that is, at the BEL, encrypted with the same key as r . To illustrate, suppose that at initialization time resources r and r' are both encrypted

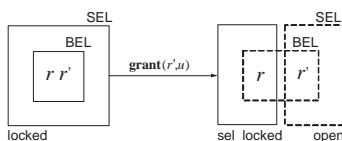


Fig. 21. From locked to sel_locked views

with the same key and they are not accessible by user u (see the leftmost view in Figure 21). Suppose then that u is granted the permission for r' . To make r' accessible at the BEL, a token is added to make the key corresponding to label $\phi_b(r)$ derivable by u , where however $\phi_b(r) = \phi_b(r')$. Hence, r' will be over-encrypted at the SEL and the key corresponding to label $\phi_s(r')$ made derivable by u . The resulting situation is illustrated in Figure 21, where r' is **open** and r results **sel_locked**.

We now analyze what are the possible views of users that may collude. Users having the **open** and the **locked** view need not be considered as they have nothing to gain in colluding. Also, recall that, as noted above, in the Full_SEL approach nobody (but the server) can have a **bel_locked** view. This leaves us only with users having the **sel_locked** view. Since users having the same views will not gain anything in colluding, the only possible collusion can happen between the server (who has a **bel_locked** view) and a user who has a **sel_locked** view. In this situation, the knowledge of the server allows lowering the outer fence, while the knowledge of the user allows lowering the inner fence: merging their knowledge, they would then be able to bring down both fences and enjoy the **open** view on the resource. The risk of collusion then arises on resources for which a user holds a **sel_locked** view and the user never had the permission to access the resource (i.e., the user never belonged to the *acl* of the resource). Indeed, if a user would get access to a resource she previously had permission for, the user has no gain in colluding with the server.

Besides collusion between different parties, we also need to consider the risk of exposure due to a single user (or server) merging her own views on a resource at different points in time. It is easy to see that, in the Full_SEL approach, where all non authorized users start with a **locked** view on the resource (and transitions are as illustrated in Figure 20(a)), there is no risk of exposure. Trivially, if the user is released the key at the SEL (i.e., it is possible for her to bring down the outer fence) it is because the user has the permission for r at some point in time and therefore she is (or has been) authorized for the resource. There is therefore no exposure risk.

Delta_SEL. In the Delta_SEL approach, users not authorized to see a resource have, at initialization time, the **bel_locked** view on it. From there, the view can evolve to be **sel_locked**, **open**, or **locked**. The view evolves from **bel_locked** to **open** for a user u if she is given the permission for the resource. The view evolves from **bel_locked** to **locked** for a user u if a user u' , with the **bel_locked** view on r , is given the permission for a resource r' encrypted, at the BEL, with the same key as r . In this case, r is over-encrypted with a SEL key that neither u nor u' know. View transitions are illustrated in Figure 20(b). It is easy to see that, in this case, a single user by herself can then hold, at different points in time, the two different views: **sel_locked** and **bel_locked**. In other words a (planning-ahead) user could retrieve the

resource at initialization time, when she is not authorized, getting and storing at her side r 's `bel_locked` view. If, at a later point in time the user is released the key corresponding to label $\phi(r)$ to make accessible to her another resource r' , she will acquire the `sel_locked` view on r . Merging this with the past `bel_locked` view, she can enjoy the `open` view on r . Note that the set of resources potentially exposed to a user coincides with the resources exposed to collusion between that user and the server in the `Full_SEL` approach.

It is important to note that in both cases (`Full_SEL` and `Delta_SEL`), this exposure only impacts resources that have been involved in a policy split to make other resources, encrypted with the same `BEL` key, available to the user. Exposure is therefore limited and well identifiable. This allows the owner to counteract it, when the owner feels specific risks have to be minimized, via explicit selective re-encryption or by proper design (as discussed in the next section).

The collusion analysis clarifies why we did not consider the third possible encryption scenario illustrated in Section 6, that is, the scenario where the authorization policy enforcement is completely delegated at the `SEL` and the `BEL` simply applies a uniform encryption (i.e., with the same key released to all users) to protect the plaintext content from the server's eyes. In this scenario, all users non authorized to access a resource would always have the `sel_locked` view on it and could potentially collude with the server. The fact that the `BEL` key is the same for all resources would make all the resources exposed (as the server would just need to collude with one user to be able to access all resources).

8.2 Design considerations

From the analysis above, we can make the following observations on the `Delta_SEL` and the `Full_SEL` approaches.

- *Exposure protection.* The `Full_SEL` approach provides superior protection, as it reduces the risk of exposure, which is limited to collusion with the server. By contrast, the `Delta_SEL` approach exposes also to single (planning-ahead) users.
- *Performance.* The `Delta_SEL` approach provides superior performance, as it imposes over-encryption only when required by a change in permissions. By contrast, the `Full_SEL` approach always imposes a double encryption on the resources, and therefore an additional load.

From these observations we can draw some criteria that could be followed by a data owner when choosing between the use of `Delta_SEL` or `Full_SEL`. If the data owner knows that:

- the access policy will be relatively static, or
- sets of resources sharing the same *acl* at initialization time represent a strong semantic relationship rarely split by policy evolution, or
- resources are grouped at the `BEL` in fine granularity components where most of the `BEL` vertices are associated with a single or few resources,

then the risk of exposing the data to collusion is limited also in the `Delta_SEL` approach, which can then be preferred for performance reasons.

By contrast, if permissions have a more dynamic and chaotic behavior, the Full_SEL approach can be preferred to limit exposure due to collusion. Also, the collusion risk can be minimized by a proper organization of the resources to reduce the possibility of policy splits. This could be done either by producing a finer granularity of encryption and/or better identifying resource groups characterized by a persistent semantic affinity (in both cases, using at the BEL different keys for resources with identical *acl*).

9. EXPERIMENTAL RESULTS

An important issue for the success of the presented techniques is their scalability. The potential for their adoption would be greatly compromised if they were not applicable in large-scale scenarios. We performed two series of experiments. The first series of experiments evaluate the number of tokens needed for representing an authorization policy. Such a metric allows us to estimate the load in terms of storage required server-side to support the authorization policy. The second series of experiments evaluate the performance of over-encryption in terms of the time required for deriving keys and for downloading and decrypting resources.

9.1 Evaluation of the number of tokens

A natural verification of the adaptability of the presented techniques to large configurations could start from the extraction of a complex authorization policy from a large system, with the goal of computing an equivalent encryption policy using the approach presented above. Unfortunately, there is no large scale access control system available today that would allow us to produce a significant test. The most structurally rich authorization policies are today those that characterize large enterprise scenarios, but these policies typically exhibit a relatively poor structure, which can be represented in our system with a limited number of tokens and almost no effort on the part of the construction algorithm. We then need to follow a different strategy to obtain a robust guarantee on the ability of the proposed system to scale well, building a simulated scenario exhibiting large scale and articulated policies. As we describe later, a single experiment was not sufficient and we designed two series of experiments, covering different configurations that solicited the system in two distinct ways.

The first scenario starts from the premise that data outsourcing platforms are used to support the exchange and dissemination of resources among the members of a user community. The idea then is to use a description of the structure of a large social network to derive a number of resource dissemination requests. We identified as a source for the construction of a large social network the coauthor relationship represented within the DBLP bibliography index. DBLP [DBLP Bibliography] is a well-known bibliographic database that currently indexes more than one million articles. The assumption at the basis of the first series of experiments is that each paper represents a resource that must be accessible by all its authors.

The social network of DBLP coauthors has been the subject of several investigations (e.g., [Cormode et al. 2008; Nascimento et al. 2003]); this network has a structure representative of that of other social networks, synthetically classified as a *power-law* or *self-similar* structure, with a sparse graph, and non-random structure of links. We implemented a C++ program that starts from a random author and

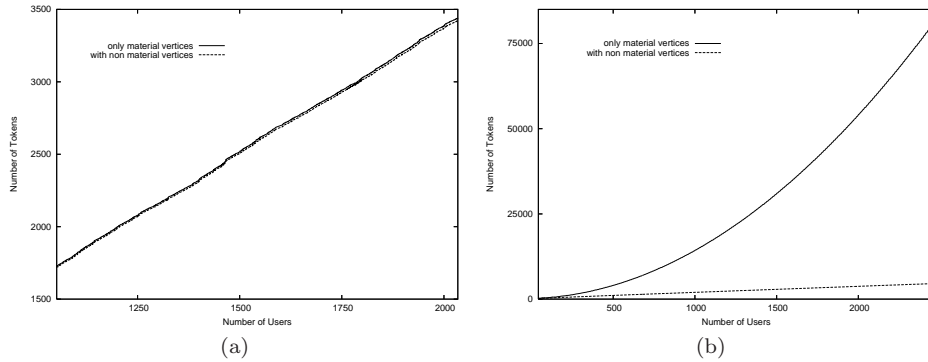


Fig. 22. Number of tokens for the DBLP scenario (a) and for the championship scenario (b)

considers all his/her publications and coauthors; then, one of the coauthors is randomly chosen and his/her publications and corresponding coauthors are iteratively retrieved, extending the user population and the set of resources. We then built a token-based encryption policy corresponding to the authorization policy, where every author has access to all the papers that he/she has authored or co-authored.

The first metric we considered in the experiments is the number of tokens required for the representation of the authorization policy. The graph in Figure 22(a) illustrates how the number of tokens increases with the number of users. We observe that the growth is linear and that the number of tokens remains low (with 2000 authors, we have 3369 tokens).

Another important metric was the one evaluating the impact of the vertices factorization process on the number of tokens. The optimization presented a very limited benefit in the DBLP scenario, as visible from Figure 22(a) (18 tokens gained out of 3369, thanks to the introduction of 12 non material vertices). The rationale is that the structure of the social network is relatively sparse. As it has been demonstrated by other investigations, self-similar networks are characterized by a few nodes which present a high level of connectedness, whereas most of the network nodes are loosely connected with a few other nodes and form small strictly connected communities. Then, the construction of a token-based encryption policy for a situation like this, produces a relatively simple graph, with relatively few tokens. This is a positive and important property, which demonstrates that our approach is immediately applicable to large social networks, with an efficient construction.

Taking into account the behavior emerging from the above experimental scenario, it became interesting to test the behavior of the system in a more difficult configuration, with a complex authorization policy. We were specifically interested in evaluating the benefit produced by the application of the optimizations introduced in the paper. As representative of a potential selective dissemination scenario, we consider the case study, also analyzed in [Damiani et al. 2007], of a sport news database. The chosen service manages a system with t teams, where each team is composed by pt players and is coordinated by one manager. The service is supposed to be used by s team supporters, referred in the following as *subscribers*. Moreover,

a set of *reporters* follows the league and uses the service to work with *tr* teams. The *reporters* are grouped into sets of *rm* elements, each of which coordinated by one *manager*. In the considered scenario, each user (team manager, reporter, reporter manager, and subscriber) can subscribe (i.e., gain read permission) to any number of resources, partitioned between *player news* and *team news*. Consistently with [Damiani et al. 2007], the set of permissions granted to subscribers is modeled to be quite large to evaluate our approach in a significant scenario. The number of team news accessed by each subscriber, along with the player news of the same team, follows a *Zipf* distribution that increases with the number *s* of subscribers. The results of the experiments, presented in Figure 22(b), (continuous line) show the number of tokens required for the representation of the policy. It is easy to observe that the number of tokens required per user is significantly higher than in the DBLP scenario, because of the more intricate structure of the policy in this experimental setup. Still, the number of tokens after the application of the optimization techniques increases linearly with the increase in the number of users, with no sign of divergence for extremely large configurations. The graph in Figure 22(b) shows the advantage produced by the identification of non material vertices, that is, by the factorization process. The advantage is significant, with an average reduction of 82% on the number of tokens.

Overall, the experiments allow us to make two important claims. First, the approach presented in the paper is able to manage large scenarios, particularly when the authorization policy presents a structure analogous to that exhibited by social networks. Second, for complex authorization policies that present a complex structure and would otherwise require a significant number of tokens per user, the use of the optimization techniques introduced in this paper is able to provide a significant reduction in the complexity, keeping at a manageable level the total number of tokens required for the representation of the policy.

9.2 Evaluation of the performance of over-encryption

We also run another series of experiments to investigate the run time costs. We implemented a prototype of a Web-based file sharing application, with a Java server answering requests originated in the client by a Firefox plugin. The extension was integrated with the XUL model underlying the Firefox interface, uses JavaScript to control the interaction with the user, and invokes the services offered by a binary library (originally written in C++) to realize the encryption functions. Open source implementations of the SHA-1 hash function and of the AES algorithm have been used. The extension is multi-platform (Windows, MacOS X, Linux).

The experiments have been executed using two distinct machines as server and client. The two computers were common PCs running Linux on the server and Windows XP SP2 on the client. The two PCs were connected by a local 100Mb/s Ethernet connection. The experiments have considered requests on resources varying in size from 1KB to 100MB, with a 10X increase at each step. The values reported in the graphs in Figure 23(a) and Figure 23(b) illustrate the average computed from 128 file retrieval requests, with length of the token chain on average equal to 2.5 (values chosen to be consistent with the analysis of large scale token configurations reported in [De Capitani di Vimercati et al. 2008]).

Figure 23(a) shows the time required to complete the retrieval of a resource. The

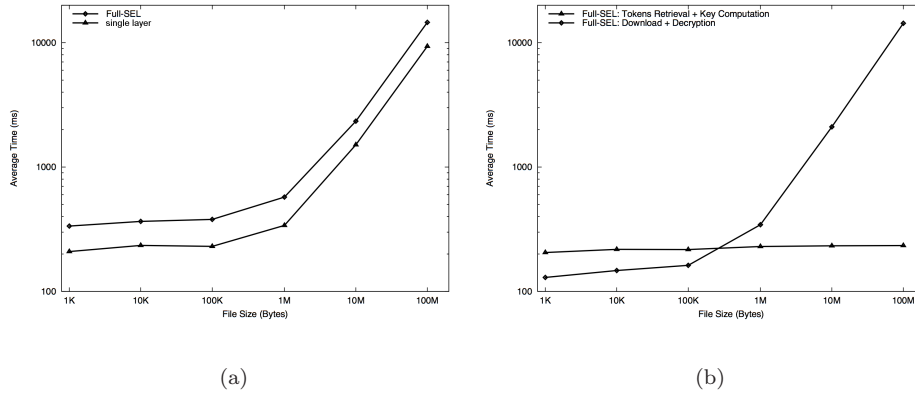


Fig. 23. Total time required for retrieving keys and resources with single layer encryption and Full_SEL over-encryption (a), and times required for retrieving keys and for retrieving resources with Full_SEL over-encryption

graph compares the time required to complete the request with a system using only BEL protection and a system using over-encryption with the Full_SEL mode. As the graph shows, for small resources the time required is doubled, whereas for large resources there is a 36% increase. The motivation is that for small resources the dominant factor is the retrieval of tokens and key derivation, which is executed twice when using over-encryption. For large resources, the difference is due to the prototype writing on disk the result of the SEL decryption before applying the BEL decryption (if the resource had been kept in memory, the difference would have been negligible). Confirming this analysis, Figure 23(b) splits the execution time for resource retrieval when using over-encryption into the time required for obtaining the key (sum of the time required to retrieve tokens from the server and the time required to compute the hash functions and derive the keys in the chain) and the time required to transfer and twice decrypt the resource. The graph clearly shows that the time required for key derivation is, as expected, independent of resource size, whereas the time required to transfer and decrypt the resource grows linearly with the increase in the resource size and is bound by the performance of the network connection. In this local network configuration, the time required to transfer and decrypt the resource is the dominant factor for resources of size larger than 1MB. Considering a geographic network, with lower bandwidth, the overhead introduced by token computation becomes irrelevant even for smaller resources.

10. RELATED WORK

Previous work close to our is in the area of “database-as-a-service” paradigm [Hacigümüs et al. 2002(a); Hacigümüs et al. 2002(b)], which considers the problem of database outsourcing. Its intended purpose is to enable data owners to outsource distribution of data on the Internet to service providers. Different security aspects of the outsourced scenario have been addressed in the last few years (e.g., execution of queries on encrypted outsourced data, inference exposure, integrity,

physical security measures). The majority of existing efforts on this topic focuses on techniques allowing the execution of queries on encrypted outsourced data, trying to support all SQL clauses and different kinds of conditions over attributes [Agrawal et al. 2004; Ceselli et al. 2005; Hacigümüs et al. 2002(a); Hacigümüs et al. 2002(b); Shmueli et al. 2005; Wang and Lakshmanan 2006]. One of the first proposals in this direction [Hacigümüs et al. 2002(a); Hacigümüs et al. 2002(b)] is based on the definition of indexing information that is stored together with the encrypted database and is used by the DBMS to select the data to be returned in response to a query. In [Ceselli et al. 2005] the authors propose a hash-based index technique for equality queries, together with a B+ tree technique applicable to range queries. They also provide an evaluation of the inference exposure in encrypted databases enriched with indexing information. The work demonstrates that even a limited number of indexes can greatly facilitate the task for an adversary that aims at violating the confidentiality provided by encryption. In [Wang and Lakshmanan 2006] the authors propose an indexing method that exploits B-trees for supporting both equality and range queries, while reducing inference exposure due to an almost flat distribution of the frequencies of index values.

In addition to the application-based approaches above-mentioned, hardware-based approaches to the problem of secure outsourced storage have been investigated [Bouganim and Pucheral 2002]. The basic idea is to use a security hardware component, which can support secure computations at both client and server sides.

A related effort [Mykletun et al. 2006] focuses on the design of mechanisms for protecting the integrity and authenticity of data from both malicious outsider attacks and the service provider itself. In [Sion 2005] the authors propose an approach for proving the correct execution of queries on outsourced data. Another interesting problem considered is privacy of queries, supported by Private Information Retrieval (PIR) techniques [Chor et al. 1998; Kushilevitz and Ostrovsky 1997]. PIR techniques could complement our solution for providing resource integrity and query correctness. However, PIR techniques unfortunately are today still too inefficient to be applicable in real systems, particularly for large data collections.

A different, but related, line of work has addressed the protection of sensitive associations existing among attributes of a relational table to be stored at external servers. Essentially, these proposals protect sensitive associations breaking them by storing data in separate tables that cannot be joined (i.e., that are stored at non-communicating servers or that have no attributes in common). The first proposal suggesting the protection of sensitive associations via fragmentation has been presented in [Aggarwal et al. 2005], and a more recent proposal is described in [Ciriani et al. 2007]. In [Aggarwal et al. 2005] sensitive associations are protected by splitting the involved attributes over two independent and non-communicating database servers, thus enforcing a fragmentation with at most two fragments, and resorting to encryption whenever necessary. In [Ciriani et al. 2007] the authors propose an alternative solution that removes the assumption of the servers be non-communicating and allows for multiple fragments, thus limiting the need to resort to encryption.

A few research efforts have directly tackled the issues of access control in an outsourced scenario. In [Miklau and Suciu 2003] 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. Our work is complementary to this proposal, as we look at the problems of exploiting key derivation techniques [Akl and Taylor 1983; Atallah et al. 2005; Crampton et al. 2006; Sandhu 1987] for access control enforcement and of outsourcing, besides the data, the management of policy changes.

The two-layer approach for policy management in outsourced encrypted databases presented in this paper has been first introduced in [De Capitani di Vimercati et al. 2007]. The paper considerably extends this prior work presenting a complete framework for policy management via encryption in outsourcing scenarios, introducing the novel problem of translating an authorization policy into an equivalent encryption policy and addressing it from its formalization and resolution to the experimental results.

11. CONCLUSIONS

We addressed the problem of enforcing access control in a scenario where data are outsourced to external servers that, while trusted for data management, are not authorized to read the data content (honest but curious servers). Our solution puts forward a novel approach combining authorizations and encryption. We provided a formal characterization of the problem of translating authorization policies into equivalent encryption policies, while minimizing the overhead in terms of storage and computation needed for the enforcement. We also described a novel solution that allows the data owner to outsource the complete management of the authorization policy by providing two layers of encryption. These two layers allow the server to directly enforce policy changes demanded by the data owner, avoiding resource transfer and re-encryption that would otherwise be required.

Our experimental results show that the solution presented in this paper is efficient and can manage complex situations. Our solution can therefore be immediately used in all database-centered scenarios and, more in general, in all scenarios where sensitive data have to be distributed and made available through a variety of external servers. We are confident that approaches that implement our proposal can have an important role in the design of the security infrastructure of future network-based information systems.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library by visiting the following URL: <http://www.acm.org/pubs/citations/journals/tods/20YY-V-N/p1-URLend>.

REFERENCES

- AGGARWAL, G., BAWA, M., GANESAN, P., GARCIA-MOLINA, H., KENTHAPADI, K., MOTWANI, R., SRIVASTAVA, U., THOMAS, D., AND XU, Y. 2005. Two can keep a secret: A distributed architecture for secure database services. In *Proc. of CIDR'05*. VLDB Endowment, 186–199.
- AGRAWAL, R., KIERMAN, J., SRIKANT, R., AND XU, Y. 2004. Order preserving encryption for numeric data. In *Proc. of ACM SIGMOD'04*. ACM, New York, 563–574.
- AKL, S. AND TAYLOR, P. 1983. Cryptographic solution to a problem of access control in a hierarchy. *ACM Trans. Comput. Syst.* 1, 3 (Aug.), 239–248.

- ATALLAH, M., FRIKKEN, K., AND BLANTON, M. 2005. Dynamic and efficient key management for access hierarchies. In *Proc. of ACM CCS'05*. ACM, New York, 190–202.
- ATENIESE, G., DE SANTIS, A., FERRARA, A.L., AND MASUCCI, B. 2006. Provably-secure time-bound hierarchical key assignment schemes. In *Proc. of ACM CCS'06*. ACM, New York, 288–297.
- BARALIS, E., PARABOSCHI, S., AND TENIENTE, E. 1997. Materialized views selection in a multidimensional database. In *Proc. of VLDB'97*. Morgan Kaufmann Publishers Inc., San Francisco, 156–165.
- BOUGANIM, L. AND PUCHERAL, P. 2002. Chip-secured data access: Confidential data on untrusted servers. In *Proc. of VLDB'02*. VLDB Endowment, 131–142.
- CESELLI, A., DAMIANI, E., DE CAPITANI DI VIMERCATI, S., JAJODIA, S., PARABOSCHI, S., AND SAMARATI, P. 2005. Modeling and assessing inference exposure in encrypted databases. *ACM Trans. on Information and System Security* 8, 1 (Feb.), 119–152.
- CHOR, B., KUSHILEVITZ, E., GOLDBREICH, O., AND SUDAN, M. 1998. Private information retrieval. *Journal of the ACM* 45, 6 (Nov.), 965–981.
- CIRIANI, V., DE CAPITANI DI VIMERCATI, S., FORESTI, S., JAJODIA, S., PARABOSCHI, S., AND SAMARATI, P. 2007. Fragmentation and encryption to enforce privacy in data storage. In *Proc. of ESORICS'07*. Springer, Berlin/Heidelberg, 225–239.
- CORMODE, G., SRIVASTAVA, D., YU, T., AND ZHANG, Q. 2008. Anonymizing bipartite graph data using safe groupings. In *Proc. of VLDB'08*. VLDB Endowment, 833–844.
- CRAMPTON, J., MARTIN, K., AND WILD, P. 2006. On key assignment for hierarchical access control. In *Proc. of IEEE CSFW'06*. IEEE Computer Society, Washington, 98–111.
- DAMIANI, E., DE CAPITANI DI VIMERCATI, S., FORESTI, S., JAJODIA, S., PARABOSCHI, S., AND SAMARATI, P. 2007. An experimental evaluation of multi-key strategies for data outsourcing. In *Proc. of IFIP SEC'07*. Springer, Boston, 385–396.
- DE CAPITANI DI VIMERCATI, S., FORESTI, S., JAJODIA, S., PARABOSCHI, S., PELOSI, G., AND SAMARATI, P. 2008. Preserving confidentiality of security policies in data outsourcing. In *Proc. of WPES'08*. ACM, New York, 75–84.
- DE CAPITANI DI VIMERCATI, S., FORESTI, S., JAJODIA, S., PARABOSCHI, S., AND SAMARATI, P. 2007. Over-encryption: Management of access control evolution on outsourced data. In *Proc. of VLDB'07*. VLDB Endowment, 123–134.
- DE SANTIS, A., FERRARA, A.L., AND MASUCCI, B. 2004. Cryptographic key assignment schemes for any access control policy. *Inf. Process. Lett.* 92, 4 (Nov.), 199–205.
- GUDES, E. 1980. The design of a cryptography based secure file system. *IEEE Trans. Softw. Eng.* 6, 5 (Sept.), 411–420.
- HACIGÜMÜS, H., IYER, B., AND MEHROTRA, S. 2002(a). Providing database as a service. In *Proc. of ICDE'02*. IEEE Computer Society, Washington, 29–39.
- HACIGÜMÜS, H., IYER, B., MEHROTRA, S., AND LI, C. 2002(b). Executing SQL over encrypted data in the database-service-provider model. In *Proc. of ACM SIGMOD'02*. ACM, New York, 216–227.
- HARN, L. AND LIN, H. 1990. A cryptographic key generation scheme for multilevel data security. *Computers and Security* 9, 6 (Oct.), 539–546.
- HWANG, M. AND YANG, W. 2003. Controlling access in large partially ordered hierarchies using cryptographic keys. *J. Syst. Softw.* 67, 2 (Aug.), 99–107.
- KUSHILEVITZ, E. AND OSTROVSKY, R. 1997. Replication is not needed: Single database, computationally-private information retrieval. In *Proc. of IEEE FOCS'97*. IEEE Computer Society, Washington, 364.
- LIAW, H., WANG, S., AND LEI, C. 1989. On the design of a single-key-lock mechanism based on Newton's interpolating polynomial. *IEEE Trans. Softw. Eng.* 15, 9 (Sept.), 1135–1137.
- MACKINNON, S., P. TAYLOR, MEIJER, H., AND AKL, S. 1985. An optimal algorithm for assigning cryptographic keys to control access in a hierarchy. *IEEE Trans. Comput.* 34, 9 (Sept.), 797–802.
- MIKLAU, G. AND SUCIU, D. 2003. Controlling access to published data using cryptography. In *Proc. of VLDB'03*. VLDB Endowment, 898–909.

- MYKLETUN, E., NARASIMHA, M., AND TSUDIK, G. 2006. Authentication and integrity in outsourced databases. *ACM Trans. Storage* 2, 2 (May), 107–138.
- NASCIMENTO, M., SANDER, J., AND POUND, J. 2003. Analysis of SIGMOD’s co-authorship graph. *ACM SIGMOD Records* 32, 3 (Sept.), 8–10.
- OLSON, L., ROSULEK, M., AND WINSLETT, M. 2007. Harvesting credentials in trust negotiation as an honest-but-curious adversary. In *Proc. of ACM WPES’07*. ACM, New York, 64–67.
- SAMARATI, P. AND DE CAPITANI DI VIMERCATI, S. 2001. Access control: Policies, models, and mechanisms. In *Foundations of Security Analysis and Design*, R. Focardi and R. Gorrieri, Eds. Springer-Verlag, London, 137–196.
- SANDHU, R. 1987. On some cryptographic solutions for access control in a tree hierarchy. In *Proc. of the 1987 Fall Joint Computer Conf. on Exploring Technology: Today and Tomorrow*. IEEE Computer Society Press, Los Alamitos, 405–410.
- SANDHU, R. 1988. Cryptographic implementation of a tree hierarchy for access control. *Inf. Process. Lett.* 27, 2 (Feb.), 95–98.
- SCHNEIER, B., KELSEY, J., WHITING, D., WAGNER, D., HALL, C., AND FERGUSON, N. 1998. On the twofish key schedule. In *Proc. of SAC’98*. Springer, Berlin/Heidelberg, 27–42.
- SHEN, V. AND CHEN, T. 2002. A novel key management scheme based on discrete logarithms and polynomial interpolations. *Computer and Security* 21, 2 (Mar.), 164–171.
- SHMUELI, E., WAISENBERG, R., ELOVICI, Y., AND GUDES, E. 2005. Designing secure indexes for encrypted databases. In *Proc. of IFIP DBSec’05*. Springer, Berlin/Heidelberg, 54–68.
- SION, R. 2005. Query execution assurance for outsourced databases. In *Proc. of VLDB’05*. VLDB Endowment, 601–612.
- SION, R. 2007. Secure data outsourcing. In *Proc. of VLDB’07*. VLDB Endowment, 1431–1432.
- SION, R. AND WINSLETT, M. 2007. Regulatory-compliant data management. In *Proc. of VLDB’07*. VLDB Endowment, 1433–1434.
- The DBLP Computer Science Bibliography. The DBLP computer science bibliography. <http://dblp.uni-trier.de>.
- WANG, H. AND LAKSHMANAN, L. V. S. 2006. Efficient secure query evaluation over encrypted XML databases. In *Proc. of VLDB’06*. VLDB Endowment, 127–138.
- XML ENCRYPTION SYNTAX AND PROCESSING, W3C REC. 2002. <http://www.w3.org/TR/xmlenc-core/>.

THIS DOCUMENT IS THE ONLINE-ONLY APPENDIX TO:

Encryption Policies for Regulating Access to Outsourced Data

SABRINA DE CAPITANI DI VIMERCATI, SARA FORESTI

Università degli Studi di Milano

SUSHIL JAJODIA

George Mason University

STEFANO PARABOSCHI

Università degli Studi di Bergamo

and

PIERANGELA SAMARATI

Università degli Studi di Milano

ACM Transactions on Database Systems, Vol. V, No. N, M 20YY, Pages 1–45.

A. ACCESS TO RESOURCES

The catalog stored on the server and created by algorithm $\mathcal{A2E}$ contains the necessary information (i.e., the set \mathcal{T} of tokens and the encryption schema $\phi(r)$ over \mathcal{R}) that users query whenever they wish to access a resource r . As a matter of fact, to access a resource r , a user u needs first to retrieve a chain of tokens that, starting from her own key k , ends in the one used to encrypt r . Figure 1 illustrates the algorithm that receives as input the resource identifier r , the key k of u , and the label $\phi(u)$ associated with k , and computes the key k_{dest} with which resource r is encrypted. The algorithm is basically composed of two steps.

The first step is performed server-side and consists in executing function **Find_Path** that, given a label $\phi(u)$ and a resource r , retrieves the shortest token chain from $\phi(u)$ to $\phi(r)$ by querying table `TOKENS`. Function **Find_Path** first determines $\phi(r)$ by querying table `LABELS` and then computes the shortest path in the key and token graph through a shortest path algorithm (an improved version of Dijkstra working on DAGs), which exploits the topological order of vertices. The function then builds backward the path from $current=\phi(r)$ to $\phi(u)$. At each iteration of the **while** loop, the function follows $pred[current]$, which is an array that contains the label of the predecessor of vertex $current$ in the path previously computed, and adds to stack $chain$ the token in `TOKENS` from $pred[current]$ to $current$.

The second step is evaluated user-side and consists in deriving keys following the chain of tokens (if not empty) returned by **Find_Path** and stored in stack $chain$, and terminating with the derivation of the key used for encrypting resource

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© 20YY ACM 0362-5915/20YY/0300-0001 \$5.00

```

INPUT
resource  $r$  to be accessed
user's key  $k$ 
label  $\phi(u)$  of the user's key

OUTPUT
key  $k_{dest}$  with which  $r$  is encrypted

MAIN
/* server-side query */
chain := Find_Path( $\phi(u), r$ )
/* user-side computation */
k_source := k
if chain  $\neq \emptyset$  then /* user  $u$  is authorized to access  $r$  */
  t := Pop(chain)
  repeat
    k_dest := t[token_value]  $\oplus$  h(k_source, t[destination])
    k_source := k_dest
    t := Pop(chain)
  until t=NULL
  return(k_dest)

FIND_PATH(from, r)
let t  $\in$  LABELS | t[res_id]=r
to := t[label]
topologically sort  $V_{\mathcal{K}, \mathcal{T}}$  in  $\mathcal{G}_{\mathcal{K}, \mathcal{T}}$ 
for each v  $\in V_{\mathcal{K}, \mathcal{T}}$  do
  dist[v] :=  $\infty$ 
  pred[v] := NULL
dist[from] := 0
for each vi  $\in V_{\mathcal{K}, \mathcal{T}}$  do /* visit vertices in topological order */
  for each (vi, vj)  $\in E_{\mathcal{K}, \mathcal{T}}$  do /* the weight of each arc is 1 */
    if dist[vj] > dist[vi] + 1 then
      dist[vj] := dist[vi] + 1
      pred[vj] := vi.label
chain :=  $\emptyset$ 
current := to
while current  $\neq$  from  $\wedge$  current  $\neq$  NULL do
  let t  $\in$  TOKENS | t[source]=pred[current]  $\wedge$  t[destination]=current
  Push(chain, t)
  current := pred[current]
if current=NULL then return( $\emptyset$ )
else return(chain)

```

Fig. 1. Key derivation process

r . For instance, consider the catalog in Figure 10(c) and suppose that C , with $\phi(C) = v_3.label$, wants to access r_9 . Function **Find_Path**($v_3.label, r_9$) first queries table LABELS for retrieving the label associated with resource r_9 , which is $\phi(r_9) = v_{10}.label$, and then finds the shortest path from v_3 to v_{10} , thus setting $pred[v_{10}]$ to v_7 and $pred[v_7]$ to v_3 . The returned *chain* is composed of two tokens, corresponding to tuples $(v_7.label, v_{10}.label, t_{7,10})$ and $(v_3.label, v_7.label, t_{3,7})$, respectively, of table TOKENS. The algorithm then derives key k_7 through user's secret key k_3 and token $(v_3.label, v_7.label, t_{3,7})$; it then derives k_{10} (i.e., the key used for encrypting

r_9) through the just computed k_7 and token $(v_7.label, v_{10}.label, t_{7,10})$, extracted from *chain*.

B. NP-HARDNESS ANALYSIS

THEOREM 3.3. *The Min-EP problem is NP-hard.*

PROOF. The considered problem is NP-hard since it can be reduced to the *Minimum Set Cover* (MSC) problem, which can be formulated as follows: *given a universal set $Uset = \{a_1, \dots, a_n\}$ and a set of subsets of $Uset$, $\mathcal{S} = \{S_1, \dots, S_m\}$, find the smallest subset \mathcal{C} of \mathcal{S} such that $\bigcup_{S_i \in \mathcal{C}} S_i = Uset$.*

Given a universal set $Uset$ and a set \mathcal{S} of its subsets, we define a corresponding authorization policy $\mathcal{A} = \langle \mathcal{U}, \mathcal{R}, \mathcal{P} \rangle$ in polynomial time as follows. For each a_i in $Uset$, there is a user u_i in \mathcal{U} . For each subset $S_j = \{a_{j,1}, \dots, a_{j,m_j}\}$ in \mathcal{S} , there is a resource r_j in \mathcal{R} with $acl(r_j) = \{u_{j,1}, \dots, u_{j,m_j}\}$ and a set R_j of $m_j - 1$ resources $r_{j,k}$, $k = 1, \dots, m_j - 1$, with $acl(r_{j,k}) = \{u_{j,1}, \dots, u_{j,k}\}$. Finally, a further resource r_\perp with $acl(r_\perp) = \{u_1, \dots, u_n\}$ is added to \mathcal{R} .

A possible encryption policy $\mathcal{E} = \langle \mathcal{U}, \mathcal{R}, \mathcal{K}, \mathcal{L}, \phi, \mathcal{T} \rangle$ equivalent to \mathcal{A} can be characterized by a key and token graph $\mathcal{G}_{\mathcal{K}, \mathcal{T}}$ with: a vertex for each user, whose key is known to the user herself; a vertex for each acl , whose key is used to encrypt the resources characterized by the acl ; and a path from each vertex representing a user u to each vertex representing an acl containing u . This implies that, each vertex v in $\mathcal{G}_{\mathcal{K}, \mathcal{T}}$ but vertices representing singleton users must have at least two incoming edges in the graph (i.e., tokens) and that the sets of users represented by the direct ancestors must cover all users represented by v . By construction, for each vertex v representing a set $\{u_1, \dots, u_k\}$ of users but the vertex representing \mathcal{U} , there is a vertex v' representing $\{u_1, \dots, u_{k-1}\}$. Therefore, the direct ancestors of v are v' and the vertex representing $\{u_k\}$. A minimum encryption policy is then the encryption policy minimizing the number of incoming edges/tokens in vertex v_\perp representing \mathcal{U} (for all other vertices, the number of incoming edges is already minimum). Note also that the addition of further vertices in $\mathcal{G}_{\mathcal{K}, \mathcal{T}}$ does not produce any benefit.

The solution to the minimum set covering problem is obtained from the solution of the corresponding Min-EP problem as follows. For each edge (v, v_\perp) , v can either represent a subset of \mathcal{U} belonging to \mathcal{S} or not. In the latter case, v is substituted with its nearest descendant representing a subset belonging to \mathcal{S} . Such a descendant must exist since, by construction, for each S_j we create vertices representing only subsets of the sets in \mathcal{S} . Since the set of direct ancestors of v_\perp represents a cover for \mathcal{U} , then the subsets they represent are a minimum set cover for $Uset$. \square

THEOREM 3.5. *Let \mathcal{A} be an authorization policy and \mathcal{E} be an encryption policy. If \mathcal{E} is equivalent to \mathcal{A} , the encryption policy graph $\mathcal{G}_{\mathcal{E}} = \langle V_{\mathcal{E}}, E_{\mathcal{E}} \rangle$ over \mathcal{E} , with $V_{\mathcal{E}} = V_{\mathcal{K}, \mathcal{T}} \cup \mathcal{U} \cup \mathcal{R}$, satisfies the local cover property (Definition 3.4).*

PROOF. By induction, we prove that $\forall v_i \in V_{\mathcal{K}, \mathcal{T}}$ the local cover property is satisfied.

- For all v_i such that $|v_i.acl| = 1$, v_i is correctly covered by definition.
- Let us suppose that for all v_i such that $|v_i.acl| \leq n$, v_i is correctly covered. We now prove that also all vertices v_j with $|v_j.acl| = n + 1$ are correctly covered.

By definition, $\forall u \in v_j.acl$, there exists a path in $\mathcal{G}_{\mathcal{E}}$ from u to v_j , that is, there exists a path from the vertex v , such that $v.acl=\{u\}$, to v_j . Therefore, there exists an edge $(v', v_j) \in E_{\mathcal{K}, \mathcal{T}}$ such that $u \in v'.acl$. Since, by construction, $v'.acl \subset v_j.acl$ (i.e., $|v'.acl| < n$) and, by hypothesis, v' is correctly covered, we can conclude that v_j is correctly covered. \square

C. CORRECTNESS AND COMPLEXITY OF ALGORITHM $\mathcal{A}2\mathcal{E}$

LEMMA 4.1 USER KEY UNIQUENESS. *Let $\mathcal{A}=\langle \mathcal{U}, \mathcal{R}, \mathcal{P} \rangle$ be an authorization policy. Algorithm $\mathcal{A}2\mathcal{E}$ creates a key and token graph $\mathcal{G}_{\mathcal{K}, \mathcal{T}}=\langle V_{\mathcal{K}, \mathcal{T}}, E_{\mathcal{K}, \mathcal{T}} \rangle$ and the corresponding encryption policy $\mathcal{E}=\langle \mathcal{U}, \mathcal{R}, \mathcal{K}, \mathcal{L}, \phi, \mathcal{T} \rangle$ such that $\forall u_i, u_j \in \mathcal{U}, i \neq j \implies \phi(u_i) \neq \phi(u_j)$.*

PROOF. During the initialization phase, algorithm $\mathcal{A}2\mathcal{E}$ creates a distinct vertex for each user u in \mathcal{U} . Since the algorithm never removes vertices from the graph, when the algorithm calls procedure **generate_encryption_policy** such vertices are again in the graph. Also, since we assume that procedure **generate_encryption_policy** correctly generates keys (i.e., avoiding duplicates), at each iteration of the first **for** loop the procedure assigns a unique key and a unique label to each vertex v in the graph, and therefore also to vertices representing singleton sets of users. The key assignment and encryption schema function ϕ is then defined based on the keys associated with the vertices representing singleton sets of users. For each user u , the procedure sets $\phi(u)$ to $v.key$, where v is the unique vertex in the graph such that $v.acl=\{u\}$. Consequently, we have the guarantee that different users are associated with different labels and, also, with different keys. \square

LEMMA 4.2 LOCAL COVER AND NON-REDUNDANCY. *Let $\mathcal{A}=\langle \mathcal{U}, \mathcal{R}, \mathcal{P} \rangle$ be an authorization policy. Algorithm $\mathcal{A}2\mathcal{E}$ creates a key and token graph $\mathcal{G}_{\mathcal{K}, \mathcal{T}}=\langle V_{\mathcal{K}, \mathcal{T}}, E_{\mathcal{K}, \mathcal{T}} \rangle$ and the corresponding encryption policy $\mathcal{E}=\langle \mathcal{U}, \mathcal{R}, \mathcal{K}, \mathcal{L}, \phi, \mathcal{T} \rangle$ such that $\mathcal{G}_{\mathcal{E}}$ satisfies the local cover property (Definition 3.4) and is non-redundant (Definition 3.6).*

PROOF. We first prove that procedure **cover_vertex**($v, tocover$) terminates and that vertex v is properly covered without redundant edges. Then, we prove that procedure **factorize**(v_i) terminates and modifies the key and token graph without violating the local cover and non-redundancy properties of the key and token graph.

During the initialization phase, for each material vertex v and for each user u in $v.acl$, algorithm $\mathcal{A}2\mathcal{E}$ sets variable $v.counter[u]$ to 0. For each material vertex v in $V_{\mathcal{K}, \mathcal{T}}$ the algorithm first calls procedure **cover_vertex** with v and $v.acl$ as parameters, respectively.

*Procedure **cover_vertex**.* The procedure first initializes local variables E_{added} and l to \emptyset and $level(v)-1$, respectively, finds a correct cover for v , and then removes redundant edges. The computation of a cover for v is implemented through two nested **while** loops. The innermost **while** loop analyzes the set V_l of vertices at level l that represent subsets of $v.acl$. For each vertex v_i randomly extracted from V_l , $v_i.acl$ is removed from $tocover$ only if $v_i.acl \cap tocover \neq \emptyset$. In this case, edge (v_i, v) is inserted in E_{added} and for each user u in $v_i.acl$, $v.counter[u]$ is increased by one. Therefore, at each iteration of the innermost **while** loop, the size of V_l always decreases by one and the size of $tocover$ possibly decreases. Since both V_l

and $toCover$ are two finite sets, the innermost **while** loop terminates. Since at each iteration of the outermost **while** loop l is decreased by one, in the worst case l will also assume the value 1. When l becomes 1, the innermost **while** loop iterates on the set V_l of vertices representing singleton sets of users. It is then easy to see that $toCover$ will become empty during the iterations on this set of vertices. We can then conclude that also the outermost **while** loop terminates and that v is correctly covered since a user u is removed from $toCover$ iff an edge (v_i, v) has been inserted in E_{added} where $u \in v_i.acl$.

Redundant edges are detected through a **for** loop that processes each edge (v_i, v) in E_{added} . Since the first two nested **while** loops terminate, E_{added} contains a finite number of edges and therefore also the **for** loop on E_{added} terminates. Edge (v_i, v) is removed from E_{added} (and therefore not inserted in $E_{\mathcal{K}, \mathcal{T}}$) only if for each user u in $v_i.acl$, $v.counter[u]$ is greater than 1, meaning that there is at least another direct ancestor v_j of v (besides v_i) such that u belongs to $v_j.acl$. When (v_i, v) is removed from E_{added} , for each user u in $v_i.acl$, $v.counter[u]$ is decreased by one to keep the counter $v.counter[u]$ consistent with the number of direct ancestors of v that include u . Also, since all incoming edges of v belong to E_{added} and each edge in E_{added} is evaluated by the procedure, the non-redundancy property is satisfied for v .

Procedure factorize. For each material vertex v_i in $V_{\mathcal{K}, \mathcal{T}}$ the algorithm calls procedure **factorize** with v_i as parameter.

The first **for** loop of the procedure evaluates each vertex v_j in $V_{\mathcal{K}, \mathcal{T}}$ having at least a common direct ancestor with v_i . Since the number of vertices in $V_{\mathcal{K}, \mathcal{T}}$ is finite, also the set of vertices with at least a common ancestor with v_i is finite and therefore the **for** loop terminates. Analogously, the **for** loops inside the **case** instruction terminates since they iterate on a finite set $CommonAnc$ of vertices as well as the last two **for** loops of the procedure, since they operate on E_{added} and $E_{removed}$ that are two finite sets. For each pair of vertices v_i and v_j , procedure **factorize** changes the set of direct ancestors of v_i and v_j iff they have more than 2 common ancestors. In this case, the edges from the common ancestors, say v_1, \dots, v_m , to v_i and v_j are removed and replaced by two edges from v' to v_i and v_j , where v' is a vertex such that $v'.acl = v_1.acl \cup \dots \cup v_m.acl$, and by the edges from the common ancestors v_1, \dots, v_m to v' . It immediately follows that the local cover property for v_i and v_j is satisfied as well as for vertex v' , which is covered by v_1, \dots, v_m that, by definition, form a cover for v' . The same discussion applies when vertex v' coincides with v_i or v_j .

For each new edge (v_l, v_h) in E_{added} and for each removed edge (v_l, v_h) in $E_{removed}$, variables $v_h.counter[u]$, with u in $v_l.acl$, are updated accordingly.

We can conclude that, since both **cover_vertex** and **factorize** are called on each vertex v in $V_{\mathcal{K}, \mathcal{T}}$ and since these procedures guarantee that vertex v is properly covered without redundant edges, $\mathcal{G}_{\mathcal{E}}$ satisfies both the local cover and the non-redundancy properties. \square

THEOREM 4.3 POLICY EQUIVALENCE. *Let $\mathcal{A} = \langle \mathcal{U}, \mathcal{R}, \mathcal{P} \rangle$ be an authorization policy. Algorithm $\mathcal{A}2\mathcal{E}$ creates a key and token graph $\mathcal{G}_{\mathcal{K}, \mathcal{T}} = \langle V_{\mathcal{K}, \mathcal{T}}, E_{\mathcal{K}, \mathcal{T}} \rangle$ and the corresponding encryption policy $\mathcal{E} = \langle \mathcal{U}, \mathcal{R}, \mathcal{K}, \mathcal{L}, \phi, T \rangle$ such that $\mathcal{A} \equiv \mathcal{E}$.*

PROOF.

$\mathcal{E} \implies \mathcal{A}$.

Procedure **generate_encryption_policy** defines an encryption policy \mathcal{E} that is based on the key and token graph created by the first two phases of algorithm $\mathcal{A}2\mathcal{E}$. In particular, the procedure defines an encryption policy such that: for each user u , $\phi(u)$ corresponds to the label of vertex v_i representing the singleton set $\{u\}$ (i.e., $v_i.acl = \{u\}$); and for each resource r , $\phi(r)$ corresponds to the label of vertex v_j representing $acl(r)$ (i.e., $v_j.acl = acl(r)$). Consider now the encryption policy graph corresponding to the encryption policy \mathcal{E} created by procedure **generate_encryption_policy** and suppose that $u \xrightarrow{\mathcal{E}} r$. This is equivalent to say that the key and token graph includes a path from the vertex v_i with label $\phi(u)$ to the vertex v_j with label $\phi(r)$. Also, since the key and token graph satisfies the local cover property (Lemma 4.2), we know that u belongs to $v_j.acl = acl(r)$ and therefore the authorization policy \mathcal{A} includes permission $\langle u, r \rangle$.

$\mathcal{E} \Leftarrow \mathcal{A}$.

Suppose that $u \xrightarrow{\mathcal{A}} r$. During the initialization phase, algorithm $\mathcal{A}2\mathcal{E}$ inserts in the key and token graph a vertex for each user in the system and for each acl value for the resources in the system. Therefore, there is a material vertex v_i such that $v_i.acl = \{u\}$, and there is a material vertex v_j such that $v_j.acl = acl(r)$ in the key and token graph. Since the algorithm never removes vertices and creates a key and token graph that satisfies the local cover property (Lemma 4.2), it is immediate to conclude that the key and token graph includes a path from v_i to v_j and that the encryption policy graph obtained by defining an encryption policy complementing the key and token graph by means of procedure **generate_encryption_policy** includes a path from u to r . \square

THEOREM 4.4. *Let $\mathcal{A} = \langle \mathcal{U}, \mathcal{R}, \mathcal{P} \rangle$ be an authorization policy. Algorithm $\mathcal{A}2\mathcal{E}$ creates a key and token graph $\mathcal{G}_{\mathcal{K}, \mathcal{T}} = \langle V_{\mathcal{K}, \mathcal{T}}, E_{\mathcal{K}, \mathcal{T}} \rangle$ and the corresponding encryption policy $\mathcal{E} = \langle \mathcal{U}, \mathcal{R}, \mathcal{K}, \mathcal{L}, \phi, \mathcal{T} \rangle$ such that $|\mathcal{K} \cup \mathcal{T}| \ll |\mathcal{U} \cup \mathcal{R} \cup \mathcal{P}|$.*

PROOF. Since all the sets involved in the union operations are disjoint, we need to prove that $|\mathcal{K}| + |\mathcal{T}| \ll |\mathcal{U}| + |\mathcal{R}| + |\mathcal{P}|$.

The number of keys created by the algorithm is equal to the number of vertices in the key and token graph while the number of tokens is equal to the number of edges. With respect to the vertices, the algorithm creates a vertex for each user in \mathcal{U} , for each acl associated with resources in \mathcal{R} , plus some additional vertices inserted by procedure **factorize** (Phase 2). Since two or more resources may share the same acl , it is easy to see that we need to prove that the number of vertices inserted by procedure **factorize** plus the number of tokens is less than the number of permissions. First, consider the graph created after Phase 1, where there are only the material vertices. Each material vertex v representing the acl of one or more resources has a number of incoming edges that, in the worst case, is equal to $|acl|$. Therefore, in the worst case, the number of tokens is equal to the number of permissions. However, if there are m resources with the same acl that is composed by n users, the number of tokens is n against the $n \cdot m$ permissions. Consider now Phase 2. Here, procedure **factorize** adds a vertex iff the pair of vertices currently

analyzed have $n > 2$ common parents. In this case, $2 \cdot n$ edges are removed from the graph and at most $n + 2$ edges are inserted. This means that the number of tokens decreases at least by one and that the number of additional vertices plus the number of tokens remains lower than the number of permissions. \square

THEOREM 4.5. *Let $\mathcal{A} = \langle \mathcal{U}, \mathcal{R}, \mathcal{P} \rangle$ be an authorization policy. Algorithm $\mathcal{A}2\mathcal{E}$ creates an encryption policy $\mathcal{E} = \langle \mathcal{U}, \mathcal{R}, \mathcal{K}, \mathcal{L}, \phi, \mathcal{T} \rangle$ such that $\mathcal{A} \equiv \mathcal{E}$ in time $O((|\mathcal{R}| + |V_{\mathcal{K},\mathcal{T}}|^2) \cdot |\mathcal{U}|)$.*

PROOF. The complexity of the algorithm is obtained by evaluating the complexity of the operations performed during the initialization and the three phases composing it.

Initialization. The **for** loop composing the initialization phase requires time proportional to $|\mathcal{U}| + |\mathcal{R}| \cdot |\mathcal{U}|$, since the inner most **for** loop has constant cost for vertices representing singleton sets of users.

Phase 1. The algorithm calls procedure **cover_vertex** for each material vertex v in $V_{\mathcal{K},\mathcal{T}}$. In the worst case, the two nested **while** loops check all vertices v_i in $V_{\mathcal{K},\mathcal{T}}$ such that $level(v_i) < level(v)$. The computational cost is then proportional to $|V_{\mathcal{K},\mathcal{T}}|^2 \cdot |\mathcal{U}|$.

The following **for** loop checks each edge $(v_i, v) \in E_{\mathcal{K},\mathcal{T}}$ and evaluates and possibly updates the value of variable $v.counter[u]$ for each u in $acl(v_i)$. In the worst case, the cost of this loop is proportional to $|E_{\mathcal{K},\mathcal{T}}| \cdot |\mathcal{U}|$.

Since $|E_{\mathcal{K},\mathcal{T}}|$ is upperbounded by $|V_{\mathcal{K},\mathcal{T}}|^2$, the overall complexity of the first phase of the algorithm is proportional to $|V_{\mathcal{K},\mathcal{T}}|^2 \cdot |\mathcal{U}|$.

Phase 2. The algorithm calls procedure **factorize** for each vertex v_i in $V_{\mathcal{K},\mathcal{T}}$. The first **for** loop checks all vertices with at least a common ancestor with v_i , which in the worst case are all vertices in $V_{\mathcal{K},\mathcal{T}}$. The procedure then finds the common direct ancestors by considering the edges incident in v_i and v_j . Since the maximum number of direct ancestors of a vertex v_i is equal to $|v_i.acl|$, the costs of this operation is proportional to $|\mathcal{U}|$. The **for** loops nested in the **case** instruction evaluate all the vertices in $CommonAnc$, which are at most $|\mathcal{U}|$. Since both $E_{\mathcal{K},\mathcal{T}}$ and $E_{removed}$ are filled in by these loops, they contain a number of elements linear in $|\mathcal{U}|$.

The overall complexity of the second phase of the algorithm is therefore proportional to $|V_{\mathcal{K},\mathcal{T}}|^2 \cdot |\mathcal{U}|$.

Phase 3. The algorithm finally calls procedure **generate_encryption_policy**, which is composed of four **for** loops, checking vertices, edges, users, and resources in the order.

The overall complexity of the third phase of the algorithm is therefore proportional to $|V_{\mathcal{K},\mathcal{T}}|^2 + |\mathcal{U}| + |\mathcal{R}|$.

If we assume that all operations performed by procedures **cover_vertex**, **factorize**, and **generate_encryption_policy** have a constant cost and c_{max} is the

maximum cost, the overall time complexity is in $O(c_{max}((|\mathcal{R}| + |V_{\mathcal{K},\mathcal{T}}|^2) \cdot |\mathcal{U}|)) = O((|\mathcal{R}| + |V_{\mathcal{K},\mathcal{T}}|^2) \cdot |\mathcal{U}|)$. \square

D. CORRECTNESS OF POLICY UPDATES

LEMMA 5.1. *Let $\mathcal{A} = \langle \mathcal{U}, \mathcal{R}, \mathcal{P} \rangle$ be an authorization policy and $\mathcal{E} = \langle \mathcal{U}, \mathcal{R}, \mathcal{K}, \mathcal{L}, \phi, \mathcal{T} \rangle$ be an encryption policy such that $\mathcal{A} \equiv \mathcal{E}$. Procedure **delete_vertex** generates a new encryption policy $\mathcal{E}' = \langle \mathcal{U}, \mathcal{R}, \mathcal{K}', \mathcal{L}', \phi', \mathcal{T}' \rangle$ such that $\mathcal{A} \equiv \mathcal{E}'$.*

PROOF. Since we assume that $\mathcal{A} \equiv \mathcal{E}$, when procedure **delete_vertex** is called on vertex v , we need to consider only the keys and tokens updated by the procedure. First, we note that procedure **delete_vertex** can remove vertex v iff v is non material. Therefore, if v is removed, it is non material and it is no more useful for reducing the number of tokens. In this case, the effect of its removal is that all direct descendants of v are no more properly covered. To this reason, for each direct descendant v_i of v , procedure **delete_vertex** calls **cover_vertex** on v_i and *tocover*, where *tocover* contains the subset of users in $v_i.acl$ such that $v_i.counter[u]=0$ (since $v_i.counter[u]$ always correctly represents the number of direct ancestors of v_i such that u belongs to their *acls*, it is not necessary to cover the users for which the corresponding counter is greater than or equal to one). Procedure **delete_vertex** then calls **factorize** on v_i . The updates on $\mathcal{G}_{\mathcal{K},\mathcal{T}}$ are then translated from procedure **update_encryption_policy** in equivalent updates on \mathcal{E} . Since procedure **delete_vertex** preserves the local cover and non-redundancy property of $\mathcal{G}_{\mathcal{K},\mathcal{T}}$, we can conclude that after the removal of vertex v , the key and token graph obtained represents an encryption policy equivalent to \mathcal{A} . \square

LEMMA 5.2. *Let $\mathcal{A} = \langle \mathcal{U}, \mathcal{R}, \mathcal{P} \rangle$ be an authorization policy and $\mathcal{E} = \langle \mathcal{U}, \mathcal{R}, \mathcal{K}, \mathcal{L}, \phi, \mathcal{T} \rangle$ be an encryption policy such that $\mathcal{A} \equiv \mathcal{E}$. Function **create_new_vertex** generates a new encryption policy $\mathcal{E}' = \langle \mathcal{U}, \mathcal{R}, \mathcal{K}', \mathcal{L}', \phi', \mathcal{T}' \rangle$ such that $\mathcal{A} \equiv \mathcal{E}'$.*

PROOF. Since we assume that $\mathcal{A} \equiv \mathcal{E}$, when function **create_new_vertex** is called, we need to consider only the keys and tokens updated by the function. In particular, we need to prove that the insertion of a new vertex representing a set U of users is performed in such a way that the local cover and non-redundancy properties of $\mathcal{G}_{\mathcal{K},\mathcal{T}}$ are satisfied. To this purpose, we first note that **create_new_vertex** removes vertices only through procedure **delete_vertex**, which preserves the equivalence between the authorization policy and the encryption policy (Lemma 5.1). **create_new_vertex** calls procedures **cover_vertex** and **factorize** on the new vertex v , thus enforcing the local cover and non-redundancy on v (Lemma 4.2). The updates on $\mathcal{G}_{\mathcal{K},\mathcal{T}}$ are then translated from procedure **update_encryption_policy** in equivalent updates on \mathcal{E} . Since procedure **create_new_vertex** preserves the local cover and non-redundancy properties, we can conclude that after the insertion of vertex v , the corresponding graph represents an encryption policy equivalent to \mathcal{A} . \square

THEOREM 5.3. *Let $\mathcal{A} = \langle \mathcal{U}, \mathcal{R}, \mathcal{P} \rangle$ be an authorization policy and $\mathcal{E} = \langle \mathcal{U}, \mathcal{R}, \mathcal{K}, \mathcal{L}, \phi, \mathcal{T} \rangle$ be an encryption policy such that $\mathcal{A} \equiv \mathcal{E}$. Procedure*

grant_revoke generates a new authorization policy $\mathcal{A}' = \langle \mathcal{U}, \mathcal{R}, \mathcal{P}' \rangle$ and a new encryption policy $\mathcal{E}' = \langle \mathcal{U}, \mathcal{R}, \mathcal{K}', \mathcal{L}', \phi', \mathcal{T}' \rangle$ such that $\mathcal{A}' \equiv \mathcal{E}'$.

PROOF. Since we assume that $\mathcal{A} \equiv \mathcal{E}$ when procedure **grant_revoke** is called, we need to consider only users and resources for which the encryption and authorization policies change.

Grant.

$\mathcal{E}' \implies \mathcal{A}'$

Consider user u and resource r . From the procedure, it is easy to see that r is encrypted with a key such that from the key of the vertex with label $\phi'(u)$ it is possible to derive the key of the vertex with label $\phi'(r)$ through \mathcal{T}' , since $\phi'(r)$ is set to $v_{new}.label$, where $v_{new}.key$ is the key that can be reached from $v = \{u\}$ (Lemma 5.2). Therefore, we have that $u \xrightarrow{\mathcal{A}'} r$.

$\mathcal{E}' \longleftarrow \mathcal{A}'$

Consider user u and resource r . From the insertion of u in $acl(r)$, we have that $u \xrightarrow{\mathcal{A}'} r$. Also, r is encrypted with a key such that the key of the vertex with label $\phi'(r)$ can be derived from the key of the vertex with label $\phi'(u)$, for the correctness of function **create_new_vertex** (Lemma 5.2). Therefore, we have that $u \xrightarrow{\mathcal{E}'} r$.

Revoke.

$\mathcal{E}' \implies \mathcal{A}'$

Consider user u and resource r . From the procedure, it is easy to see that r is encrypted with a key such that from the key of the vertex with label $\phi'(u)$ it is not possible to derive the key of the vertex with label $\phi'(r)$ through \mathcal{T}' , since $\phi'(r)$ is set to $v_{new}.label$, which cannot be reached from $v = \{u\}$ (Lemma 5.1). Therefore, we have that $u \not\xrightarrow{\mathcal{A}'} r$.

$\mathcal{E}' \longleftarrow \mathcal{A}'$

Consider user u and resource r . From the removal of u from $acl(r)$, we have that $u \not\xrightarrow{\mathcal{A}'} r$. Also, r is encrypted with a key such that the key of the vertex with label $\phi'(r)$ cannot be derived from the key of the vertex with label $\phi'(u)$ (Lemma 5.1).

Therefore, we have that $u \not\xrightarrow{\mathcal{E}'} r$. \square

E. CORRECTNESS OF THE TWO-LAYER ENCRYPTION

THEOREM 7.1. *Let $\mathcal{A} = \langle \mathcal{U}, \mathcal{R}, \mathcal{P} \rangle$ be an authorization policy, $\mathcal{E}_b = \langle \mathcal{U}, \mathcal{R}, \mathcal{K}_b, \mathcal{L}_b, \phi_b, \mathcal{T}_b \rangle$ be an encryption policy at the BEL, and $\mathcal{E}_s = \langle \mathcal{U}, \mathcal{R}, \mathcal{K}_s, \mathcal{L}_s, \phi_s, \mathcal{T}_s \rangle$ be an encryption policy at the SEL such that $\mathcal{A} \equiv \langle \mathcal{E}_b, \mathcal{E}_s \rangle$. The procedures in Figure 16 generate a new $\mathcal{E}_b' = \langle \mathcal{U}, \mathcal{R}, \mathcal{K}_b', \mathcal{L}_b', \phi_b', \mathcal{T}_b' \rangle$, $\mathcal{E}_s' = \langle \mathcal{U}, \mathcal{R}, \mathcal{K}_s', \mathcal{L}_s', \phi_s', \mathcal{T}_s' \rangle$, and \mathcal{A}' such that $\mathcal{A}' \equiv \langle \mathcal{E}_b', \mathcal{E}_s' \rangle$.*

PROOF. Since we assume that $\mathcal{A} \equiv \langle \mathcal{E}_b, \mathcal{E}_s \rangle$ when procedures **grant** and **revoke** are called, we need to consider only users and resources for which the encryption and authorization policies change. Grant and revoke are based on the correctness of the over-encryption operations. We then examine it first.

Over-encrypt. We need to prove that **over_encrypt**(U, R) possibly encrypts all resources in R with a key in such a way that a user u' can derive such a key if and only if $u' \in U$. If the condition in the first **if** statement is evaluated to true, resources in R are already correctly protected and since the procedure terminates, the result is correct. Otherwise, resources in R are first possibly decrypted and then encrypted. The only case we need to consider for encryption is when the set of users U is different from ALL (when $U = \text{ALL}$, resources in R are not needed to be over-encrypted). If $U \neq \text{ALL}$, resources in R are encrypted with the correct key $s.\text{key}$ or with a key assigned to vertex s created through function **create_new_vertex**(U). The correctness is guaranteed by the correctness of both function **create_new_vertex** and procedure **delete_vertex** (Lemmas 5.1 and 5.2).

Grant.

$$\langle \mathcal{E}_b', \mathcal{E}_s' \rangle \Longrightarrow \mathcal{A}'$$

Consider user u and resource r . From the pseudocode in Figure 16, it is easy to see that $\phi_b'(r) = \phi_b(r)$ and also that there is a (set of) token allowing to derive the key of the vertex with label $\phi_b'(r)$ from the key of the vertex with label $\phi_b'(u)$. From the **case** instruction and by the correctness of **over_encrypt**, either $\phi_s'(r) = \text{NULL}$ or r is over-encrypted with a key such that from the key of the vertex with label $\phi_s'(u)$ it is possible to derive the key of the vertex with label $\phi_s'(r)$ through \mathcal{T}_s' (user u is included in the current $\text{acl}(r)$). Since the key of the vertex with label $\phi_b'(r)$ can be derived from the key of the vertex with label $\phi_b'(u)$ and the key of the vertex with label $\phi_s'(r)$ can be derived from the key of the vertex with label $\phi_s'(u)$, we have that $u \xrightarrow{\mathcal{A}'} r$.

Consider now the set of resources R' and suppose that R' is not empty. For each subset S of resources in R' characterized by the same acl , denoted acl_S , user u can now derive the key used to encrypt such a set of resources. This implies that $\forall r' \in S, \phi_b'(r') = \phi_b(r')$ whose corresponding key can be computed starting from the key of the vertex with label $\phi_b'(u)$. By the correctness of **over_encrypt**, a call **over_encrypt**(acl_S, S) guarantees that all resources r' in S are over-encrypted with a key such that $\forall r' \in S$, the key of the vertex with label $\phi_s'(r')$ is not derivable from the key of the vertex with label $\phi_s'(u)$ because acl_S does not include user u .

$$\langle \mathcal{E}_b', \mathcal{E}_s' \rangle \Leftarrow \mathcal{A}'$$

Consider user u and resource r . From the first instruction in the procedure, we have that $u \xrightarrow{\mathcal{A}'} r$. From the pseudocode in Figure 16, it is easy to see that $\phi_b'(r) = \phi_b(r)$ and that the corresponding key can be computed from the key of the vertex with label $\phi_b'(u)$. Also, from the **case** instruction and by the correctness of **over_encrypt**, either $\phi_s'(r) = \text{NULL}$ or r is over-encrypted with the key of the vertex with label $\phi_s'(r)$ that can be derived from the key of the vertex with label $\phi_s'(u)$.

Revoke.

$$\langle \mathcal{E}_b', \mathcal{E}_s' \rangle \Longrightarrow \mathcal{A}'$$

Consider user u and resource r . A call **over_encrypt**($\text{acl}(r), \{r\}$) is requested to demand the SEL to make r accessible only to users in the current $\text{acl}(r)$. We know

that $u \xrightarrow{\mathcal{E}'_b} r$. Also, from the **over_encrypt** correctness, it is easy to see that the key of the vertex with label $\phi'_s(r)$ cannot be computed from the key of the vertex with label $\phi'_s(u)$.

$\langle \mathcal{E}'_b, \mathcal{E}'_s \rangle \Leftarrow \mathcal{A}'$

Consider user u and resource r . From the first instruction in the procedure we have that $u \xrightarrow{\mathcal{A}'} r$. The subsequent call **over_encrypt**($acl(r), \{r\}$) makes resource r no more accessible to user u because r is over-encrypted with a key that is no more derivable by u (this property is a consequence of the **over_encrypt** correctness), that is, the key of the vertex with label $\phi'_b(r)$ is still derivable from the key of the vertex with label $\phi'_b(u)$ but the key of the vertex with label $\phi'_s(r)$ is not derivable from the key of the vertex with label $\phi'_s(u)$. \square