Enforcing Dynamic Write Privileges in Data Outsourcing[☆]

Sabrina De Capitani di Vimercati[a], Sara Foresti[a], Sushil Jajodia[b], Giovanni Livraga[a], Stefano Paraboschi[c], Pierangela Samarati[a]

[a] *DI - Università degli Studi di Milano, 26013 Crema, Italy*
firstname.lastname*@unimi.it*
[b] *CSIS - George Mason University, Fairfax, VA 22030-4444, USA*
jajodia@gmu.edu
[c] *Università degli Studi di Bergamo, 24044 Dalmine, Italy*
parabosc@unibg.it

Abstract

Users and companies are more and more resorting to external providers for storing their data and making them available to others. Since data sharing is typically selective (i.e., accesses to certain data should be allowed only to authorized users), there is the problem of enforcing authorizations on the outsourced data. Recently proposed approaches based on selective encryption provide convenient enforcement of read privileges, but are not directly applicable for supporting write privileges.

In this paper, we extend selective encryption approaches to the support of write privileges. Our proposal enriches the approach based on key derivation of existing solutions and complements it with a hash-based approach for supporting write privileges. Enforcement of write privileges and of possible policy updates relies on the - controlled - cooperation of the external provider. Our solution also allows the data owner and the users to verify the integrity of the outsourced data.

Keywords: Data outsourcing, data protection, write authorization enforcement, policy updates, data integrity.

[☆] A preliminary version of this paper appeared under the title "Support for Write Privileges on Outsourced Data," in Proc. of SEC 2012, Heraklion, Crete, June 2012 [1].

1. Introduction

Data outsourcing gives to end users and companies the opportunity to benefit from the lower costs, higher availability, and larger elasticity that are offered by the rapidly growing market of cloud providers. A major obstacle to the adoption of cloud storage services is commonly recognized to be the uncertainty and concerns about the correct management of security requirements. Users ask for robust guarantees about the confidentiality and integrity of the outsourced data, and the research community has recently proposed several techniques addressing this need (e.g., [2, 3, 4]). A common requirement is that data should remain confidential to both unauthorized users and the external server storing them, which is considered *honest-but-curious* (i.e., trustworthy for managing resources but not for accessing their content). To provide such confidentiality guarantee, existing proposals typically assume data to be encrypted before being outsourced to the external server, and they associate with the encrypted data additional indexing information that can be used by the server to perform queries on encrypted data. For efficiency reasons, encryption is based on symmetric keys. Earlier proposals typically consider data to be encrypted with a single key, assuming either all users to have complete visibility of the resources in the data collection, or the data owner to mediate access requests to the data to enforce read authorizations. More recent proposals, addressing the problem of allowing the users to have selective visibility over the data (so that different sets of users be able to access different resources), have proposed the application of a 'selective encryption' approach. Intuitively, different keys are used to encrypt different resources and users have visibility on subsets of resources depending on the keys they know. Proper modeling and key derivation techniques have been devised to ensure limited key management overhead in approaches based on selective encryption.

While interesting and promising, all the solutions above assumed outsourced data to be read-only. In other words, the owner can modify resources while all other users can only read them. Such an assumption can result restrictive in several scenarios where a data owner outsourcing the data to the external server may also want to authorize other users (again selectively) to write and update the outsourced resources. Like for read authorizations [5], the enforcement of write privileges in outsourcing scenarios is complicated by the fact that data are not under the direct control of their owner. In this paper, we provide a solution for enforcing write authorizations on encrypted

outsourced data. Our solution is based on the same principles as previous proposals, since it relies on encryption for enforcing read and write access restrictions having efficiency and manageability as primary goal. In [1], we presented an early version of our proposal for the enforcement of write privileges over outsourced data. Here, we extend this approach with the support of grant and revoke of write authorizations. We therefore provide a more general solution, applicable to scenarios where static write authorizations may result limiting as the sets of users authorized to modify the content of a resource can dynamically change over time. Our solution for enforcing updates to write authorizations results appealing for its efficiency and flexibility, as it avoids expensive re-keying and re-encryption operations. A key feature is that it delegates the enforcement of updates on the write access control policy to the external server, thus reducing the burden left at the data owner side. Our proposal is complemented by a mechanism that allows both the data owner and the authorized writers to verify the integrity of the resources externally stored (i.e., to verify that resources have not been modified by unauthorized users or by the server). We also extend the write integrity check technique proposed in [1] to reflect updates in the write access policy.

The remainder of this paper is organized as follows. Section 2 introduces the basic concepts on which our proposal is based, and presents the problem of enforcing write privileges in outsourcing scenarios. Section 3 illustrates our solution for enforcing write authorizations exploiting selective encryption. Section 4 discusses our approach for enforcing grant and revoke of write privileges. Section 5 presents a mechanism for allowing the data owner and writers to check the write operations executed and detect possible misbehaviors by the server or by the users. Section 6 extends the integrity check mechanism to support updates to the write access policy. Section 7 discusses related work. Finally, Section 8 gives our final remarks and concludes the paper. The proofs of the theorems are reported in Appendix A.

2. Basic concepts and problem statement

Our work builds upon and extends a previous proposal [5] for confidential data outsourcing. According to this proposal, a data owner outsourcing data to a honest-but-curious server and wishing to provide selective visibility over them to other users encrypts resources before sending them to the external server for storage, and reflects the authorization policy in the encryption itself. Therefore, each resource o is encrypted with a key to be made known

only to the users authorized to read o, that is, to users who belong to the access control list of o. Symmetric encryption is used and different keys are assumed: one for each user and one for each group of users that corresponds to an access control list. The adoption of a *key derivation technique* based on public tokens allows users to access the resources of the system while having to manage only one key. In further detail, each key k_i is identified by a public label l_i and, given keys k_i and k_j, token $t_{i,j}$ is computed as $k_j \oplus h(k_i, l_j)$, with $\oplus$ the bitwise `xor` operator, and h a deterministic cryptographic function. Token $t_{i,j}$ permits to derive key k_j from the knowledge of key k_i and public label l_j [6]. All keys with which resources are encrypted are then connected in a *key derivation graph*. A key derivation graph is a DAG whose nodes correspond to keys (of users and acls) and whose edges correspond to tokens that ensure that each user can - via a sequence of public tokens - derive the keys corresponding to the sets to which she belongs. Each user is then communicated the key of the node representing herself in the graph. Each resource is encrypted with the key corresponding to its acl. Encrypted resources as well as the tokens are outsourced to the server. In particular, for each resource o, the external server stores the encrypted version of the resource together with the resource identifier and the label of the key with which the resource is encrypted. A user authorized to read a resource (i.e., who belongs to its acl) can, via the tokens available on the server, derive the key corresponding to the acl of the resource and decrypt it.

Example 2.1. *Consider a system with four users* $\mathcal{U}=\{A,B,C,D\}$ *and four resources* $\mathcal{O}=\{o_1,o_2,o_3,o_4\}$*, whose access control lists are reported in Figure 1(a). Figure 1(b) illustrates the encrypted resources stored at the server, where:* r_label *is the label of the key used to encrypt the resource (i.e., the key associated with its access control list);* o_id *is the resource identifier; and* encr_resource *is the encrypted resource. Figure 1(c) illustrates the key derivation graph enforcing the authorizations. For the sake of readability, in the key derivation graph we denote a key corresponding to a given acl* U *(i.e., a key with label* l_U *and value* k_U*) with* U*. Figure 1(d) illustrates the tokens corresponding to the key derivation graph in Figure 1(c).*

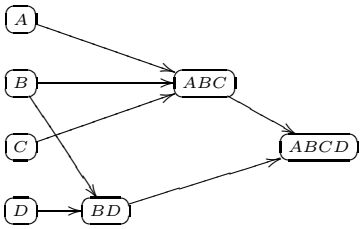
The encryption-based model described in this section nicely fits a scenario in which the authorization policy regulates only read access privileges, selectively restricting resource visibility to subsets of users. The support of read accesses without consideration of write privileges may result however

o	**acl**
o_1	$ABCD$
o_2	$ABCD$
o_3	ABC
o_4	BD

(a)

r_label	**o_id**	**encr_resource**
l_{ABCD}	1	zKZlJxVcCC0g
l_{ABCD}	2	t9qdJqC7ImXU
l_{ABC}	3	AxalPH8v37Ts
l_{BD}	4	xwfPJSn.MVqY

(b)



(c)

from	**to**	**val**
l_A	l_{ABC}	$k_{ABC} \oplus h(k_A, l_{ABC})$
l_B	l_{ABC}	$k_{ABC} \oplus h(k_B, l_{ABC})$
l_B	l_{BD}	$k_{BD} \oplus h(k_B, l_{BD})$
l_C	l_{ABC}	$k_{ABC} \oplus h(k_C, l_{ABC})$
l_D	l_{BD}	$k_{BD} \oplus h(k_D, l_{BD})$
l_{BD}	l_{ABCD}	$k_{ABCD} \oplus h(k_{BD}, l_{ABCD})$
l_{ABC}	l_{ABCD}	$k_{ABCD} \oplus h(k_{ABC}, l_{ABCD})$

(d)

Figure 1: An example of four resources with their acls (a), encrypted resources (b), key derivation graph (c), and tokens (d)

limiting in emerging data sharing scenarios (e.g., document sharing), where the data owner may wish to grant other users the privilege to modify some of her resources. Unfortunately, the keys associated with resources for regulating the read accesses to them cannot be used for restricting write accesses as well. As a matter of fact, we can imagine that in many situations the set of users authorized to write a resource is different from (typically being a subset of) the set of users authorized to read it. A straightforward solution for enforcing write authorizations might consist in simply outsourcing to the external server the authorization policy (for write privileges) as is. The server would then perform traditional (authorization-based) access control, adopting user authentication and policy enforcement. This solution would however present the main drawback of requesting a considerable management overhead. Also, it would not be in line with the goal pursued by outsourcing approaches, aimed at minimizing the server's involvement and responsibility in access control enforcement. Our goal is to enforce write privileges following the same spirit of the proposal in [5]: for this reason, we propose to exploit selective encryption for the enforcement also of write authorizations. As a matter of fact, having resources tied to access restrictions by means of cryptographic solutions can provide a more robust and flexible control, whose enforcement is less exposed to server misbehaviors. However, while the encryption of a resource with a key known to all and only the users au-

thorized to read it suffices for enforcing read authorizations, enforcement of write privileges requires cooperation from the external server. In the following sections, we will describe an approach, based on selective encryption, for the effective outsourcing to the external server of the enforcement of both read and write privileges, as well as of grant and revoke operations.

3. Authorization policy

The basic idea of our approach for the enforcement of both read and write privileges consists in associating each resource with a *write tag* defined by the data owner, and in adopting selective encryption techniques to regulate both access to resource contents and to their write tags. Our intuition is to encrypt the tag of a given resource with a key known only by the users authorized to write the resource and by the external server. In this way, only the server and authorized writers will have access to the plaintext write tag of each resource. The server will then accept a write operation on a resource when the requesting user shows knowledge of the corresponding write tag. Since the key used for encrypting the write tag has to be shared by the server and the writers, we leverage on the underlying structure already in place for regulating the necessary read operations. In this section, we illustrate our key derivation structure for managing the encryption keys of the system (Section 3.1), and we discuss how to use it for enforcing read and write access restrictions (Section 3.2).

3.1. Key derivation structure

Elaborating on the approach in [5], and adapting it to our context, we introduce a *set-based key derivation graph* as follows.

Definition 3.1 (Set-based key derivation graph). *Let $\mathcal{U}$ be a set of users and $\mathtt{U}\subseteq 2^{\mathcal{U}}$ be a family of subsets of users in $\mathcal{U}$ such that $\forall u\in\mathcal{U}$, $\{u\}\in\mathtt{U}$. A* set-based key derivation graph *over $\mathcal{U}$ and $\mathtt{U}$ is a triple $\langle\mathcal{K},\mathcal{L},\mathcal{T}\rangle$, with $\mathcal{K}$ a set of keys, $\mathcal{L}$ the set of corresponding labels, and $\mathcal{T}$ a set of tokens, such that:*

1. *$\forall U\in\mathtt{U}$, there exist a* derivation key *$k_U\in\mathcal{K}$;*
2. *$\forall u\in\mathcal{U}$, $\forall U\in\mathtt{U}\backslash\{u\}$ s.t. $u\in U$, there exists a token $t_{\{u\},U}$ or a sequence $\langle t_{\{u\},U_i},\ldots,t_{U_j,U}\rangle$ of tokens in $\mathcal{T}$, with $t_{c,d}$ following $t_{a,b}$ in the sequence if $b=c$.*

Definition 3.1 ensures that, for each set $U \in \mathtt{U}$ of users, there exists a derivation key, and that each user u in the system can derive (through either a single token or a chain of tokens) all the derivation keys of all the groups $U \in \mathtt{U}$ to which she belongs.

Since our approach requires each resource to be associated with a write tag that must be encrypted with a key shared by the server and the authorized writers of the resource, we extend the set-based key derivation graph in Definition 3.1 with the external server. However, since the server cannot access the plaintext of the outsourced resources, it cannot be treated the same way as authorized users (i.e., considering it as an additional user). We then define a *key derivation structure* by extending the set-based key derivation graph to include also the keys that will be shared with the server, and will be used to encrypt the write tags for enforcing write privileges (see Section 3.2). These additional keys are defined in such a way that authorized users can compute them applying a secure hash function h^s to a key they already know (or can derive via a sequence of tokens), while the server can derive them through a token specifically added to the key derivation structure. Compared with the set-based key derivation graph in Definition 3.1, in the key derivation structure we also distinguish between two kinds of keys (possibly associated with each set of users): *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 set of users in $\mathtt{U}$ is therefore associated with a derivation key k and, when needed, also with an access key k^a obtained by applying a secure hash function h^a to k (i.e., $k^a{=}h^a(k)$). 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 access to resources.

Formally, a key derivation structure is defined as follows.

Definition 3.2 (Key derivation structure). *Let $\mathcal{U}$ be a set of users, $\mathcal{S}$ be an external server, $\mathtt{U} \subseteq 2^{\mathcal{U}}$ be a family of subsets of users in $\mathcal{U}$ such that $\forall u \in \mathcal{U}$, $\{u\} \in \mathtt{U}$, $\mathtt{U}^s$ and $\mathtt{U}^a$ be two subsets of $\mathtt{U}$, and $\langle \mathcal{K}', \mathcal{L}', \mathcal{T}' \rangle$ be a set-based key derivation graph over $\mathcal{U}$ and $\mathtt{U}$. A* key derivation structure *implied by $\mathtt{U}^s$ and $\mathtt{U}^a$ over $\langle \mathcal{K}', \mathcal{L}', \mathcal{T}' \rangle$ is a triple $\langle \mathcal{K}, \mathcal{L}, \mathcal{T} \rangle$, with $\mathcal{K}$ a set of keys, $\mathcal{L}$ the set of corresponding labels, and $\mathcal{T}$ a set of tokens, such that:*

1. $\mathcal{K} = \mathcal{K}' \cup \{k_{\mathcal{S}}\} \cup \{k_{U \cup \{\mathcal{S}\}}{=}h^s(k_U) \mid U \in \mathtt{U}^s\} \cup \{k^a_U{=}h^a(k_U) \mid U \in \mathtt{U}^a\}$, *with h^s and h^a two secure hash functions;*

2. $\mathcal{T} = \mathcal{T}' \cup \{t_{\mathcal{S},U\cup\{\mathcal{S}\}} \mid U \in \mathtt{U}^s\}$.

A key derivation structure therefore extends a set-based key derivation graph by including: *i)* a derivation key $k_{\mathcal{S}}$ assigned to the server; *ii)* a key $k_{U\cup\{\mathcal{S}\}}$ shared by the users in U and the server, for each set U of users in $\mathtt{U}^s$; *iii)* an access key k^a_U shared by the users in U, for each set U of users in $\mathtt{U}^a$; and *iv)* a token $t_{\mathcal{S},U\cup\{\mathcal{S}\}}$ that allows the server to derive key $k_{U\cup\{\mathcal{S}\}}$ starting from its key $k_{\mathcal{S}}$, for each set U of users in $\mathtt{U}^s$. For each set U of users in $\mathtt{U}^s$, both a derivation key k_U and a key $k_{U\cup\{\mathcal{S}\}}$ shared with the server belong to $\mathcal{K}$. Analogously, for each set U of users in $\mathtt{U}^a$, both a derivation key k_U and an access key k^a_U belong to the set $\mathcal{K}$ of keys in the key derivation structure.

Figure 2 illustrates function **Define_Key_Derivation_Structure** that builds a key derivation structure. The function receives as input a set $\mathcal{U}$ of users, an external server $\mathcal{S}$, three families $\mathtt{U}$, $\mathtt{U}^s$, and $\mathtt{U}^a$ of subsets of users in $\mathcal{U}$, with $\mathtt{U}^s \subseteq \mathtt{U}$ and $\mathtt{U}^a \subseteq \mathtt{U}$, and two secure hash functions h^s and h^a. It returns the key derivation structure $\langle \mathcal{K},\mathcal{L},\mathcal{T}\rangle$ implied by $\mathtt{U}^s$ and $\mathtt{U}^a$ over $\langle \mathcal{K}',\mathcal{L}',\mathcal{T}'\rangle$ (Definition 3.2). The function operates in two steps: the first step defines the set-based key derivation graph over $\mathcal{U}$ and $\mathtt{U}$; the second step extends the key derivation graph with the server, for defining the key derivation structure of interest. In the first step, the function leverages on the algorithms in [5] to define the set-based key derivation graph $\langle \mathcal{K}',\mathcal{L}',\mathcal{T}'\rangle$. To this aim, for each set $U \in \mathtt{U}$ of users the function generates a derivation key and the corresponding label, and inserts them into the sets $\mathcal{K}'$ of keys and $\mathcal{L}'$ of labels, respectively (lines 5–8). The function then defines a set $\mathcal{T}'$ of tokens such that, for each user u in the set $\mathcal{U}$, there is a token (or a sequence of tokens) in $\mathcal{T}'$ that permits to derive, starting from k_u, all those keys k_U associated with a set $U \in \mathtt{U}$ of users with $u \in U$ (lines 10–12). In the second step, function **Define_Key_Derivation_Structure** extends the set-based key derivation graph computed in the previous step to obtain the key derivation structure of interest. To this aim, the function first generates a derivation key $k_{\mathcal{S}}$ for the server and the corresponding label $l_{\mathcal{S}}$, and inserts them into sets $\mathcal{K}$ and $\mathcal{L}$, respectively (lines 14–16). The set $\mathcal{T}$ of tokens is initialized to the set $\mathcal{T}'$ of tokens in the set-based key derivation graph (line 17). For each set U of users in $\mathtt{U}^s$, the function computes key $k_{U\cup\{\mathcal{S}\}}$ (shared by the server and U) applying secure hash function h^s to k_U, generates the corresponding label, and inserts them into the set $\mathcal{K}$ of keys and into the set $\mathcal{L}$ of labels in the key derivation structure, respectively (lines 18–22). The set $\mathcal{T}$ of tokens is then updated by inserting a token that permits to

DEFINE_KEY_DERIVATION_STRUCTURE($\mathcal{U}$, $\mathcal{S}$, $\mathtt{U}$, $\mathtt{U}^s$, $\mathtt{U}^a$, h^s, h^a)

/* **Input** $\mathcal{U}$: users of the system */
/* $\mathcal{S}$: external server */
/* $\mathtt{U} \subseteq 2^{\mathcal{U}}$: family of subsets of users in $\mathcal{U}$ */
/* $\mathtt{U}^s \subseteq \mathtt{U}$, $\mathtt{U}^a \subseteq \mathtt{U}$: subsets of $\mathtt{U}$ */
/* h^s, h^a : secure hash functions */
/* **Output** $\langle \mathcal{K}, \mathcal{L}, \mathcal{T} \rangle$: key derivation structure implied by $\mathtt{U}^s$ and $\mathtt{U}^a$ over $\langle \mathcal{K}', \mathcal{L}', \mathcal{T}' \rangle$ */

1: /* **Step 1**: define the set-based key derivation graph */
2: $\mathcal{K}' := \emptyset$
3: $\mathcal{L}' := \emptyset$
4: $\mathcal{T}' := \emptyset$
5: **for each** $U \in \mathtt{U}$ **do** /* generate a derivation key for each $U \in \mathtt{U}$ (C1 in Def. 3.1) */
6: generate a derivation key k_U and a label l_U
7: $\mathcal{K}' := \mathcal{K}' \cup \{k_U\}$
8: $\mathcal{L}' := \mathcal{L}' \cup \{l_U\}$
9: /* define a set of tokens s.t. $\forall U \in \mathtt{U}$ and $\forall u \in \mathcal{U}$, k_U is derivable from k_u iff $u \in U$ (C2 in Def. 3.1) */
10: **for each** $U_j \in \mathtt{U}$, $|U_j| > 1$ **do**
11: $cover_j := \{U_1, \ldots, U_n \in \mathtt{U} \mid \bigcup_{i=1}^{n} U_i = U_j\}$
12: $\mathcal{T}' := \mathcal{T}' \cup \{t_{U_i,U_j} = k_{U_j} \oplus h(k_{U_i}, l_{U_j}) \mid U_i \in cover_j\}$
13: /* **Step 2**: define a key derivation structure */
14: generate a key $k_{\mathcal{S}}$ and a label $l_{\mathcal{S}}$ /* generate a key for the external server (C1 in Def. 3.2) */
15: $\mathcal{K} := \mathcal{K}' \cup \{k_{\mathcal{S}}\}$
16: $\mathcal{L} := \mathcal{L}' \cup \{l_{\mathcal{S}}\}$
17: $\mathcal{T} := \mathcal{T}'$
18: **for each** $U \in \mathtt{U}^s$ **do** /* for each $U \in \mathtt{U}^s$, compute $k_{U \cup \{\mathcal{S}\}}$ as the result of h^s over k_U (C1 in Def. 3.2) */
19: $k_{U \cup \{\mathcal{S}\}} := h^s(k_U)$
20: generate a label $l_{U \cup \{\mathcal{S}\}}$
21: $\mathcal{K} := \mathcal{K} \cup \{k_{U \cup \{\mathcal{S}\}}\}$
22: $\mathcal{L} := \mathcal{L} \cup \{l_{U \cup \{\mathcal{S}\}}\}$
23: $\mathcal{T} := \mathcal{T} \cup \{t_{\mathcal{S},U \cup \{\mathcal{S}\}} = k_{U \cup \{\mathcal{S}\}} \oplus h(k_{\mathcal{S}}, l_{U \cup \{\mathcal{S}\}})\}$ /* token from $k_{\mathcal{S}}$ to $k_{U \cup \{\mathcal{S}\}}$ (C2 in Def. 3.2) */
24: **for each** $U \in \mathtt{U}^a$ **do** /* for each $U \in \mathtt{U}^a$, compute k^a_U as the result of h^a over k_U (C1 in Def. 3.2) */
25: $k^a_U := h^a(k_U)$
26: generate a label l^a_U
27: $\mathcal{K} := \mathcal{K} \cup \{k^a_U\}$
28: $\mathcal{L} := \mathcal{L} \cup \{l^a_U\}$
29: **return**($\langle \mathcal{K}, \mathcal{L}, \mathcal{T} \rangle$)

Figure 2: Function that defines a key derivation structure

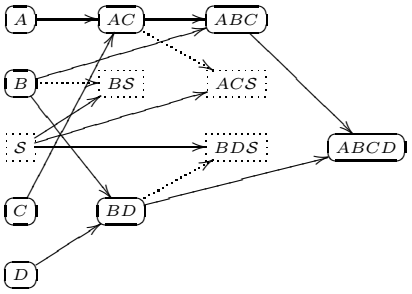
derive $k_{U \cup \{\mathcal{S}\}}$ from $k_{\mathcal{S}}$ for each set U of users in $\mathtt{U}^s$ (line 23). The function generates an access key k^a_U (and the corresponding label) for each set U of users in $\mathtt{U}^a$ by applying secure hash function h^a to the derivation key k_U associated with the same set of user, and inserts the key and the label into $\mathcal{K}$ and $\mathcal{L}$, respectively (lines 24–28). The function terminates returning the resulting key derivation structure $\langle \mathcal{K}, \mathcal{L}, \mathcal{T} \rangle$ (line 29). The following theorem formally shows that function **Define_Key_Derivation_Structure** correctly computes a key derivation structure.

o	r[o]	w[o]
o_1	$ABCD$	BD
o_2	$ABCD$	BD
o_3	ABC	AC
o_4	BD	B

(a)

r_label	w_label	o_id	encw_tag	encr_resource
l_{ABCD}	l_{BDS}	1	α	zKZlJxVcCrC0g
l_{ABCD}	l_{BDS}	2	β	t9qdJqC7AImXU
l_{ABC}	l_{ACS}	3	γ	AxalPH8Kv37Ts
l_{BD}	l_{BS}	4	δ	xwfPJSLn.MVqY

(b)



(c)

from	to	val
l_A	l_{AC}	$k_{AC} \oplus h(k_A, l_{AC})$
l_B	l_{ABC}	$k_{ABC} \oplus h(k_B, l_{ABC})$
l_B	l_{BD}	$k_{BD} \oplus h(k_B, l_{BD})$
l_C	l_{AC}	$k_{AC} \oplus h(k_C, l_{AC})$
l_D	l_{BD}	$k_{BD} \oplus h(k_D, l_{BD})$
l_{AC}	l_{ABC}	$k_{ABC} \oplus h(k_{AC}, l_{ABC})$
l_{BD}	l_{ABCD}	$k_{ABCD} \oplus h(k_{BD}, l_{ABCD})$
l_{ABC}	l_{ABCD}	$k_{ABCD} \oplus h(k_{ABC}, l_{ABCD})$
l_S	l_{BS}	$k_{BS} \oplus h(k_S, l_{BS})$
l_S	l_{ACS}	$k_{ACS} \oplus h(k_S, l_{ACS})$
l_S	l_{BDS}	$k_{BDS} \oplus h(k_S, l_{BDS})$

(d)

Figure 3: An example of read and write acls (a), encrypted resources (b), key derivation structure (c), and tokens (d)

Theorem 3.1 (Correctness of procedure Define_Key_Derivation_Structure). *Let $\mathcal{U}$ be a set of users, $\mathcal{S}$ be an external server, $\mathtt{U} \subseteq 2^{\mathcal{U}}$ be a family of subsets of users in $\mathcal{U}$ such that $\forall u \in \mathcal{U}$, $\{u\} \in \mathtt{U}$, and $\mathtt{U}^s$ and $\mathtt{U}^a$ be two subsets of $\mathtt{U}$. Triple $\langle \mathcal{K}, \mathcal{L}, \mathcal{T} \rangle$ computed by function* **Define_Key_Derivation_Structure** *in Figure 2 is a key derivation structure (Definition 3.2).*

Proof. See Appendix A.

Example 3.1. *Consider a system with four users $\mathcal{U}=\{A,B,C,D\}$, a family $\mathtt{U}=\{A,B,C,D,AC,BD,ABC,ABCD\}$ of subsets of users, and two subsets $\mathtt{U}^s=\{B,AC,BD\}$ and $\mathtt{U}^a=\{BD,ABC,ABCD\}$ of $\mathtt{U}$. Figure 3(c) illustrates the key derivation structure computed by function* **Define_Key_Derivation_Structure** *in Figure 2. In the figure, nodes drawn with a continuous line represent derivation keys, and nodes drawn with a dotted line represent keys shared with the external server (for the sake of readability, access keys are not reported in the figure). Continuous edges represent tokens, and dotted edges correspond to hash-based derivations computed via secure hash function h^s.*

3.2. Access control enforcement

We now illustrate our proposal for enforcing both read and write access restrictions. Each resource o is associated with two (possibly different) access control lists: *i)* a read access list $r[o]$ reporting the set of users authorized to read o, and *ii)* a write access list $w[o]$ reporting the set of users authorized to write o. Consistently with most real-world scenarios, we assume the users authorized to write a resource to also read it, that is, $\forall o \in \mathcal{O}$: $w[o] \subseteq r[o]$.

Read authorizations are enforced through selective encryption. Each resource o in the set $\mathcal{O}$ of resources is then encrypted with the *access key* corresponding to the set of users in its read access list $r[o]$, which is known or can be derived by all and only the users authorized to view the resource content.

Enforcement of write authorizations, as mentioned at the beginning of this section, relies on the definition of a write tag for each resource and on the cooperation with the external server. Each resource $o \in \mathcal{O}$ is associated with a write tag $tag[o]$, defined by the data owner using a secure random function to ensure independence of the tag from both the resource identifier and its content. To guarantee that only the server $\mathcal{S}$ and the set $w[o]$ of authorized writers know the plaintext value of the write tag of resource o, $tag[o]$ is encrypted with a key that is known or can be derived only by the users in $w[o]$ and by the server.

Each resource $o \in \mathcal{O}$ is stored at the external server in encrypted form, together with the following metadata.

- `r_label`: label of the key with which the resource is encrypted, which is the access key of the set $r[o]$ of users authorized to read o (i.e., $l^a_{r[o]}$).
- `w_label`: label of the key shared by the set $w[o]$ of users authorized to write o and the server $\mathcal{S}$ (i.e., $l_{w[o] \cup \{\mathcal{S}\}}$).
- `encw_tag`: write tag $tag[o]$ of resource o, which is used by the server to enforce restrictions on write privileges. The tag is encrypted with the key identified by the label in `w_label` (i.e., $E(tag[o], k_{w[o] \cup \{\mathcal{S}\}})$, where E is a symmetric encryption function computed over $tag[o]$ with key $k_{w[o] \cup \{\mathcal{S}\}}$).
- `encr_resource`: encrypted version of resource o, encrypted with the access key identified by the label in `r_label` (i.e., $E(o, k^a_{r[o]})$).

INITIALIZE_SYSTEM($\mathcal{U}$, $\mathcal{O}$, $\mathcal{S}$, h^s, h^a)

/* **Input** $\mathcal{U}$: users of the system */
/* $\mathcal{O}$: resources of the system */
/* $\mathcal{S}$: external server */
/* h^s, h^a : secure hash functions */

1: /* **Step 1**: define the key derivation structure */
2: $\mathtt{U}^s := \bigcup_{o \in \mathcal{O}} w[o]$
3: $\mathtt{U}^a := \bigcup_{o \in \mathcal{O}} r[o]$
4: $\mathtt{U} := \mathtt{U}^s \cup \mathtt{U}^a \cup \{\{u\} \mid u \in \mathcal{U}\}$
5: $\langle \mathcal{K}, \mathcal{L}, \mathcal{T} \rangle$:= **Define_Key_Derivation_Structure**($\mathcal{U}$, $\mathcal{S}$, $\mathtt{U}$, $\mathtt{U}^s$, $\mathtt{U}^a$, h^s, h^a)
6: /* **Step 2**: distribute keys */
7: **for each** $u \in \mathcal{U}$ **do** /* communicate derivation keys to users */
8: send k_u to u
9: send $k_{\mathcal{S}}$ to $\mathcal{S}$ /* communicate the derivation key to the server */
10: /* **Step 3**: outsource resources and tokens */
11: $\mathcal{O}^k := \emptyset$ /* outsourced relation */
12: **for each** $o \in \mathcal{O}$ **do** /* define the outsourced relation */
13: create a new tuple t
14: $t[\mathtt{r\_label}] := l_{r[o]}$
15: $t[\mathtt{w\_label}] := l_{w[o] \cup \{\mathcal{S}\}}$
16: $t[\mathtt{o\_id}] := \mathbf{Id}(o)$
17: randomly generate a value for $tag[o]$
18: $t[\mathtt{encw\_tag}] := E(tag[o], k_{w[o] \cup \{\mathcal{S}\}})$
19: $t[\mathtt{encr\_resource}] := E(o, k_{r[o]})$
20: insert t into $\mathcal{O}^k$
21: send relation $\mathcal{O}^k$ to the server
22: TOKEN := $\emptyset$ /* relation storing public tokens */
23: **for each** $t_{i,j} \in \mathcal{T}$ **do**
24: create a new tuple t
25: $t[\mathtt{from}] := l_i$
26: $t[\mathtt{to}] := l_j$
27: $t[\mathtt{val}] := t_{i,j}$
28: insert t into TOKEN
29: send relation TOKEN to the server

Figure 4: Procedure that enforces the access control policy defined by the data owner before outsourcing resources

Given the set $\mathcal{U}$ of users and the set $\mathcal{O}$ of resources in the system, where each resource is associated with read and write access control lists as mentioned above, the data owner must compute keys and tokens composing the key derivation structure before outsourcing resources in $\mathcal{O}$. To this aim, it calls procedure **Initialize_System** in Figure 4, which in turn calls function **Define_Key_Derivation_Structure** in Figure 2 to properly define the key derivation structure. The procedure receives as input the set $\mathcal{U}$ of users and the set $\mathcal{O}$ of resources in the system, an external server $\mathcal{S}$, and two secure hash functions h^s and h^a. The procedure first needs to define three families $\mathtt{U}$, $\mathtt{U}^s$, and $\mathtt{U}^a$ of subsets of users in $\mathcal{U}$. $\mathtt{U}$ corresponds to the set of groups of

users whose keys must be represented in the system for the correct enforcement of the authorizations. It then includes the singleton sets $\{u\}$ of users u in $\mathcal{U}$, and the sets U of users representing read and write access lists ($r[o]$ and $w[o]$, respectively) of resources o in $\mathcal{O}$. $\mathtt{U}^s$ is the subset of $\mathtt{U}$ representing those sets of users that have to share a key with the external server. It then includes all the sets of users corresponding to the write access lists $w[o]$ of resources o in $\mathcal{O}$. $\mathtt{U}^a$ is the subset of $\mathtt{U}$ representing those sets of users for which an access key needs to be defined. It then includes all the sets corresponding to the read access lists $r[o]$ of resources o in $\mathcal{O}$ (lines 2–4). The procedure then calls function **Define_Key_Derivation_Structure**, which returns a key derivation structure (line 5). Finally, the procedure:

1. communicates to each user u derivation key k_u, and to the external server derivation key $k_{\mathcal{S}}$ (lines 7–9);
2. computes and stores at the external server the encrypted resources and the associated metadata (lines 11–21);
3. stores at the external server all the tokens in the key derivation structure (i.e., tokens in $\mathcal{T}$) as a set of triples of the form $\langle l_i, l_j, t_{i,j}\rangle$ indicating that the key with label l_j can be directly derived from the key with label l_i through token $t_{i,j}$ (lines 22–29).

Example 3.2. *Consider a system with four users $\mathcal{U}$=$\{A,B,C,D\}$ and four resources $\mathcal{O}$=$\{o_1,o_2,o_3,o_4\}$, and assume read and write acls of resources to be as in Figure 3(a) (read acls are the same as in Example 2.1). Figure 3(c) illustrates the key derivation structure computed as described in Example 3.1. Figure 3(b) and Figure 3(d) illustrate the encrypted resources and associated metadata, and the tokens outsourced to the external server, respectively.*

It is easy to see that our approach guarantees: *i)* correct *read authorization enforcement*; *ii)* correct *write authorization enforcement*; and *iii) write control* by the server. Read authorization enforcement is guaranteed as each resource $o\in\mathcal{O}$ is encrypted with an access key (i.e., $k^a_{r[o]}$) that only authorized readers in $r[o]$ know or can derive. In fact, each user u can compute any access key k^a_U such that $u\in U$ by applying hash function h^a to derivation key k_U, which u knows or can derive as she belongs to U. Write authorization enforcement is guaranteed since the write tag $tag[o]$ of each resource $o\in\mathcal{O}$ is encrypted with a key (i.e., $k_{w[o]\cup\{\mathcal{S}\}}$) that only authorized writers in $w[o]$ (and the server) can derive. Also, the server is assumed to be

honest-but-curious and therefore not interested in tampering with resources (see Sections 5 and 6). Write control by the server is guaranteed since the server has visibility over the write tag of all resources, which is encrypted with a key that the server can directly derive.

The correct enforcement of the authorization policy is formally proved by the following theorem.

Theorem 3.2 (Correct enforcement of authorizations). *Let $\mathcal{U}$ be a set of users, $\mathcal{S}$ be an external server, $\mathcal{O}$ be a set of resources such that $\forall o \in \mathcal{O}$ $r[o]$ and $w[o]$ are the read and write access lists of o, respectively. Our access control system satisfies the following conditions:*

1. *$\forall u \in \mathcal{U}$ and $\forall o \in \mathcal{O}$, u can decrypt* `encr_resource`$[o]$ *iff $u \in r[o]$* (read authorization enforcement)*;*
2. *$\forall u \in \mathcal{U}$ and $\forall o \in \mathcal{O}$, u can decrypt* `encw_tag`$[o]$ *iff $u \in w[o]$* (write authorization enforcement)*;*
3. *$\forall o \in \mathcal{O}$, $\mathcal{S}$ can decrypt* `encw_tag`$[o]$ (write control)*.*

Proof. See Appendix A.

4. Policy updates

Policy updates must be managed with special care in our scenario, since they might require expensive re-encryption and/or key re-distribution operations by the data owner, thus limiting the advantages of data outsourcing. The problem of granting and revoking read authorizations with limited overhead for the data owner has been already investigated, and we can therefore assume to solve it by using the proposal in [5], which is based on over-encryption. In this section, we will focus on the management of write privileges, with the goal of outsourcing the enforcement of grant and revoke operations to the external server. Since both grant and revoke operations translate into the insertion of keys (and tokens) in the key derivation structure, we first illustrate how to manage this operation (Section 4.1). We then describe how grants and revokes of privileges can be enforced to correctly reflect updates in the write authorizations (Section 4.2).

GLOBAL VARIABLES

$\mathcal{U}$: users of the system
$\mathcal{S}$: external server
$\langle \mathcal{K}, \mathcal{L}, \mathcal{T} \rangle$: key derivation structure
h^s, h^a : secure hash functions

GET_KEY(U)

1: let k_U be the key in $\mathcal{K}$ associated with U
2: **if** k_U = NULL **then** /* if $\mathcal{K}$ does not include a derivation key for U */
3: generate k_U and label l_U
4: $k^a_U := h^a(k_U)$ /* compute k^a_U as the result of h^a over k_U */
5: generate a label l^a_U
6: $\mathcal{K} := \mathcal{K} \cup \{k_U, k^a_U\}$
7: $\mathcal{L} := \mathcal{L} \cup \{l_U, l^a_U\}$
8: let $\mathcal{U}'$ be the family of subsets of $2^{\mathcal{U}}$ such that $\forall U \in \mathcal{U}'$, $k_U \in \mathcal{K}$
9: $cover := \{U_1, \ldots, U_n \subseteq \mathcal{U}' \mid \bigcup_{i=1}^{n} U_i = U_j\}$
10: **for each** $U_i \in cover$ **do**
11: $t_{U_i,U} := k_U \oplus h(k_{U_i}, l_U)$
12: $\mathcal{T} := \mathcal{T} \cup \{t_{U_i,U}\}$
13: create a new tuple t
14: t[`from`] := l_{U_i}
15: t[`to`] := l_U
16: t[`val`] := $t_{U_i,U}$
17: insert t into TOKEN /* at the server side */
18: **return**(k_U)

GET_SHARED_KEY(U)

19: let $k_{U \cup \{\mathcal{S}\}}$ be the key in $\mathcal{K}$ shared by U and $\mathcal{S}$
20: **if** $k_{U \cup \{\mathcal{S}\}}$ = NULL **then** /* if $\mathcal{K}$ does not include a key shared by U and $\mathcal{S}$ */
21: k_U := **Get_Key**(U) /* retrieve or create the derivation key associated with U */
22: $k_{U \cup \{\mathcal{S}\}} := h^s(k_U)$ /* compute $k_{U \cup \{\mathcal{S}\}}$ as the result of h^s over k_U */
23: generate a label $l_{U \cup \{\mathcal{S}\}}$
24: $\mathcal{K} := \mathcal{K} \cup \{k_{U \cup \{\mathcal{S}\}}\}$
25: $\mathcal{L} := \mathcal{L} \cup \{l_{U \cup \{\mathcal{S}\}}\}$
26: $\mathcal{T} := \mathcal{T} \cup \{t_{\mathcal{S},U \cup \{\mathcal{S}\}} = k_{U \cup \{\mathcal{S}\}} \oplus h(k_{\mathcal{S}}, l_{U \cup \{\mathcal{S}\}})\}$ /* insert into $\mathcal{T}$ the token from $k_{\mathcal{S}}$ to $k_{U \cup \{\mathcal{S}\}}$ */
27: create a new tuple t
28: t[`from`] := $l_{\mathcal{S}}$
29: t[`to`] := $l_{U \cup \{\mathcal{S}\}}$
30: t[`val`] := $t_{\mathcal{S},U \cup \{\mathcal{S}\}}$
31: insert t into TOKEN /* at the server side */
32: **return**($k_{U \cup \{\mathcal{S}\}}$)

Figure 5: Pseudocode of functions **Get_Key** and **Get_Shared_Key**

4.1. Updates to the key derivation structure

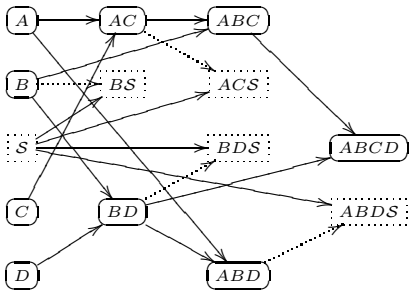
The basic operations on the key derivation structure necessary to manage grant and revoke operations consist in the retrieval/insertion of derivation and access keys.

Function **Get_Key** in Figure 5 receives as input a set $U \subseteq \mathcal{U}$ of users and returns the derivation key associated with it. The function first checks

whether the set $\mathcal{K}$ of keys in the key derivation structure already includes a derivation key for U (line 1). If this is not the case, the function generates a new derivation key for the set of users together with its label, and computes the corresponding access key together with its label. It then inserts keys and labels in the sets $\mathcal{K}$ and $\mathcal{L}$ of keys and labels of the key derivation structure (lines 2–7). The function then updates the set $\mathcal{T}$ of tokens in the key derivation structure by inserting the tokens necessary to guarantee that each user u in U can derive k_U from her key k_u (lines 8–12). The function then updates relation TOKEN at the server side accordingly (lines 13–17). Finally, the function returns derivation key k_U (line 18).

Function **Get_Shared_Key** in Figure 5 receives as input a set $U \subseteq \mathcal{U}$ of users and returns the key shared by the server and U. The function first checks whether the set $\mathcal{K}$ of keys already includes the key of interest (line 19). If this is not the case, the function first retrieves the derivation key associated with the set U of users by calling function **Get_Key** over U (lines 20–21). It then computes the hash of k_U through secure hash function h^s, obtaining $k_{U \cup \{\mathcal{S}\}}$ (line 22). The function then generates the corresponding label and inserts the key into $\mathcal{K}$ and the label into $\mathcal{L}$ (lines 23–25). The function inserts into $\mathcal{T}$ a token that permits the server to derive $k_{U \cup \{\mathcal{S}\}}$ from $k_{\mathcal{S}}$ (line 26). The function then updates relation TOKEN at the server side accordingly (lines 27–31). Finally, the function returns $k_{U \cup \{\mathcal{S}\}}$ (line 32).

Example 4.1. *Consider the key derivation structure of Figure 3(c), and assume that a key has to be shared by the server $\mathcal{S}$ and the set ABD of users. Figure 6 illustrates the key derivation structure, and the corresponding set of tokens, resulting from the call to function* **Get_Shared_Key** *in Figure 5 over ABD. Since $\mathcal{K}$ does not include a key shared by $\mathcal{S}$ and ABD, function* **Get_Shared_Key** *calls function* **Get_Key** *over ABD, which inserts derivation key k_{ABD} and access key k^a_{ABD} into $\mathcal{K}$, labels l_{ABD} and l^a_{ABD} into $\mathcal{L}$, and tokens $t_{A,ABD}$ and $t_{BD,ABD}$ into $\mathcal{T}$. It returns the derivation key of ABD. Function* **Get_Shared_Key** *then computes key $k_{ABD\mathcal{S}}$ by applying secure hash function h^s to k_{ABD}, inserts $k_{ABD\mathcal{S}}$ into $\mathcal{K}$, the corresponding label $l_{ABD\mathcal{S}}$ into $\mathcal{L}$, and token $t_{\mathcal{S},ABD\mathcal{S}}$ into $\mathcal{T}$. In Figure 6(b) and in the following figures, we denote tokens inserted by functions* **Get_Key** *and* **Get_Shared_Key** *with a bullet $\bullet$.*



(a)

from	to	val
l_A	l_{AC}	$k_{AC} \oplus h(k_A, l_{AC})$
l_A	l_{ABD}	$k_{ABD} \oplus h(k_A, l_{ABD})$
l_B	l_{ABC}	$k_{ABC} \oplus h(k_B, l_{ABC})$
l_B	l_{BD}	$k_{BD} \oplus h(k_B, l_{BD})$
l_C	l_{AC}	$k_{AC} \oplus h(k_C, l_{AC})$
l_D	l_{BD}	$k_{BD} \oplus h(k_D, l_{BD})$
l_{AC}	l_{ABC}	$k_{ABC} \oplus h(k_{AC}, l_{ABC})$
l_{BD}	l_{ABD}	$k_{ABD} \oplus h(k_{BD}, l_{ABD})$
l_{BD}	l_{ABCD}	$k_{ABCD} \oplus h(k_{BD}, l_{ABCD})$
l_{ABC}	l_{ABCD}	$k_{ABCD} \oplus h(k_{ABC}, l_{ABCD})$
l_S	l_{BS}	$k_{BS} \oplus h(k_S, l_{BS})$
l_S	l_{ACS}	$k_{ACS} \oplus h(k_S, l_{ACS})$
l_S	l_{BDS}	$k_{BDS} \oplus h(k_S, l_{BDS})$
l_S	l_{ABDS}	$k_{ABDS} \oplus h(k_S, l_{ABDS})$

(b)

Figure 6: Key derivation structure (a) and tokens (b) after the insertion of key k_{ABDS} in the key derivation structure in Figure 3(c)

4.2. Grant and revoke

Despite effective for enforcing changes to read authorizations, over-encryption falls short when it is necessary to grant or revoke write privileges. In fact, in a worst case scenario, users are not oblivious (i.e., they have the ability to store and keep indefinitely all information they have been entitled to access), and the users in the write access list of a resource have knowledge of the value of the corresponding write tag. These users can therefore exploit such knowledge to modify the resource even when they lost the write privilege. To illustrate, consider a resource o with write access list $w[o]$ and assume that, at a given point in time, the data owner revokes from user $u \in w[o]$ the write privilege for o. To enforce the revoke operation, write tag $tag[o]$ should be encrypted with a key known only to the users in $w[o] \backslash \{u\}$. However, since u was previously included in $w[o]$ she might know the plaintext value of the write tag $tag[o]$. Even without being able to decrypt the encrypted write tag sent by the server, user u would then still be able to correctly reply to the challenge of the server, thus violating the write access policy defined by the data owner. For instance, consider the key derivation structure in Figure 3(c), and suppose that the data owner revokes the write privilege over resource o_2 from user B. If B already knows the plaintext value of $tag[o_2]$, she can still answer the challenge of the server, and then improperly modify o_2. Since this problem depends on previous knowledge of the revoked user and not on her ability to decrypt the write tag received

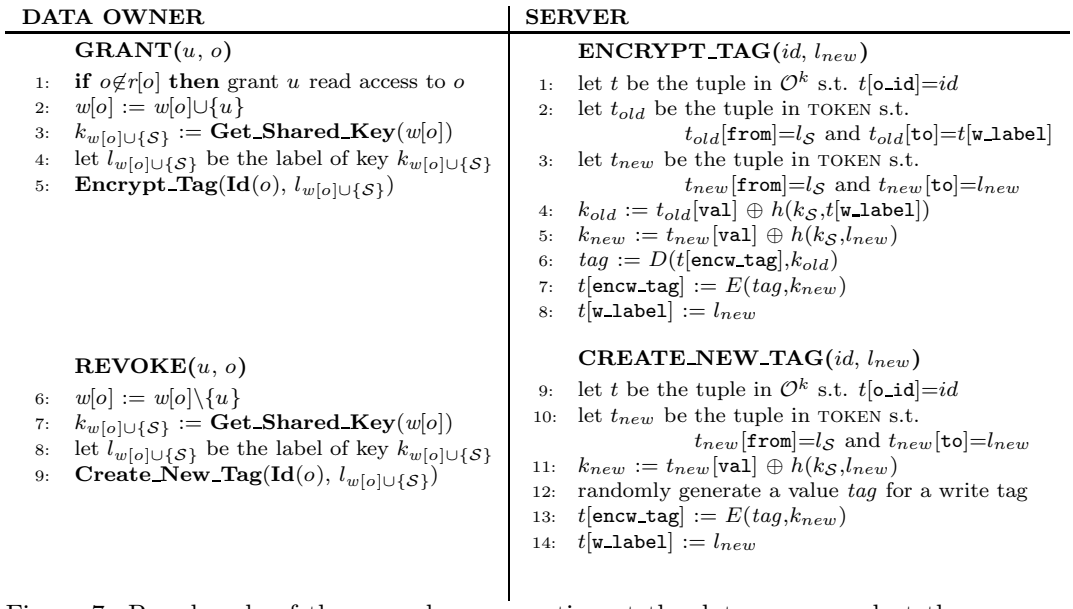


Figure 7: Pseudocode of the procedures operating at the data owner and at the server side to grant and revoke write privileges

from the server, it is necessary to associate a fresh write tag with the revoked resource to effectively enforce the policy change.

We now illustrate in details how write authorizations can be granted and revoked upon decision of the data owner.

Grant. We consider the case of the data owner granting user u write privilege over resource o. Note that, if u is not a reader of o, the access control policy is first modified granting u read access to o. To ensure that write requests by u are accepted by the server, the data owner must encrypt the write tag associated with o with a key known to: the server, the authorized writers in $w[o]$, and the user u who is being granted the write privilege. In other words, $tag[o]$ must be encrypted with a key shared by the server and the new set $w[o]\cup\{u\}$ of writers. Clearly, if the key derivation structure does not include a key known by the server and by all and only the users in $w[o]\cup\{u\}$, then the data owner must first update the key derivation structure to include it.

Procedure **Grant** in Figure 7 receives as input a user u and a resource o and grants u the privilege of modifying o. The procedure first updates the read access list (if necessary) and the write access list of the resource (lines 1–2). It then retrieves the derivation key (and the corresponding label) that will be used to encrypt the write tag of the resource (i.e., the key shared by

the authorized writers of o, including u, and the server) by calling function **Get_Shared_Key** on the updated write access list of the resource (lines 3–4). The procedure then calls procedure **Encrypt_Tag**, which is executed by the server, to update the representation of the resource at the server side (line 5). Procedure **Encrypt_Tag** in Figure 7 receives as input a resource identifier id and a label l_{new} and encrypts the write tag of the resource identified by id with the key identified by l_{new}. To this purpose, it first determines the tuple t in the outsourced table representing resource o with identifier id (line 1). It then finds the token that permits to derive key k_{old} with which t[`encw_tag`] is currently encrypted (i.e., the token from $l_{\mathcal{S}}$ to t[`w_label`]), and the token that permits to derive key k_{new} with which the write tag must be encrypted to reflect the policy change (i.e., the token from $l_{\mathcal{S}}$ to l_{new}) (lines 2–3). The procedure then uses these tokens to derive both k_{old} and k_{new} (lines 4–5). It decrypts t[`encw_tag`] with k_{old}, re-encrypts the write tag with k_{new}, and updates t[`w_label`], setting it to l_{new} to reflect the policy update (lines 6–8).

Example 4.2. *Consider the key derivation structure, outsourced resources, and tokens in Figure 3 and assume that the data owner grants A write privilege over o_2 (i.e., $w[o_2]=w[o_2]\cup\{A\}=ABD$). Since the key derivation structure includes neither a key shared by ABD and $\mathcal{S}$, nor a derivation key for ABD, the structure is first updated to accommodate the new keys (see Example 4.1). Then, the write tag of o_2 is re-encrypted by the server with $k_{ABD\mathcal{S}}$.*

Assume now that the data owner grants D write privilege over o_4 (i.e., $w[o_4]=w[o_4]\cup\{D\}=BD$). Since the key derivation structure already contains a key for the updated write access list of o_4, no update is necessary to the key derivation structure. Hence, the only operation performed to enforce this authorization update consists in encrypting the write tag of o_4 with key $k_{BD\mathcal{S}}$. Figure 8 illustrates the read and write access lists, the encrypted resources, the key derivation structure, and the tokens after these two grant operations.

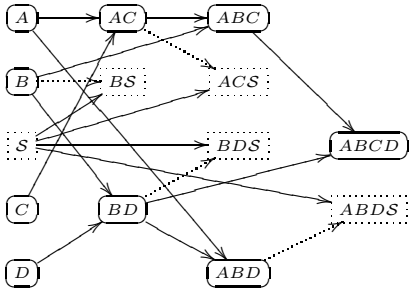
Revoke. We consider the case of the data owner revoking from user u the write privilege over resource o. To ensure that u cannot exploit her knowledge of the plaintext write tag $tag[o]$ of the revoked resource to perform unauthorized write operations on o, a new write tag must be defined for o, whose value must be independent from the former value of $tag[o]$ (i.e., it has to be chosen adopting a secure random function). Since the server is authorized to know the write tag of each and every resource in the system to

o	r[o]	w[o]
o_1	$ABCD$	BD
o_2	$ABCD$	ABD
o_3	ABC	AC
o_4	BD	BD

(a)

r_label	w_label	o_id	encw_tag	encr_resource
l_{ABCD}	$l_{BD\mathcal{S}}$	1	α	zKZlJxVcCrC0g
l_{ABCD}	$l_{ABD\mathcal{S}}$	2	ϵ	t9qdJqC7AImXU
l_{ABC}	$l_{AC\mathcal{S}}$	3	γ	AxalPH8Kv37Ts
l_{BD}	$l_{BD\mathcal{S}}$	4	ζ	xwfPJSLn.MVqY

(b)



(c)

from	to	val
l_A	l_{AC}	$k_{AC} \oplus h(k_A, l_{AC})$
l_A	l_{ABD}	$k_{ABD} \oplus h(k_A, l_{ABD})$
l_B	l_{ABC}	$k_{ABC} \oplus h(k_B, l_{ABC})$
l_B	l_{BD}	$k_{BD} \oplus h(k_B, l_{BD})$
l_C	l_{AC}	$k_{AC} \oplus h(k_C, l_{AC})$
l_D	l_{BD}	$k_{BD} \oplus h(k_D, l_{BD})$
l_{AC}	l_{ABC}	$k_{ABC} \oplus h(k_{AC}, l_{ABC})$
l_{BD}	l_{ABD}	$k_{ABD} \oplus h(k_{BD}, l_{ABD})$
l_{BD}	l_{ABCD}	$k_{ABCD} \oplus h(k_{BD}, l_{ABCD})$
l_{ABC}	l_{ABCD}	$k_{ABCD} \oplus h(k_{ABC}, l_{ABCD})$
$l_{\mathcal{S}}$	$l_{B\mathcal{S}}$	$k_{B\mathcal{S}} \oplus h(k_{\mathcal{S}}, l_{B\mathcal{S}})$
$l_{\mathcal{S}}$	$l_{AC\mathcal{S}}$	$k_{AC\mathcal{S}} \oplus h(k_{\mathcal{S}}, l_{AC\mathcal{S}})$
$l_{\mathcal{S}}$	$l_{BD\mathcal{S}}$	$k_{BD\mathcal{S}} \oplus h(k_{\mathcal{S}}, l_{BD\mathcal{S}})$
$l_{\mathcal{S}}$	$l_{ABD\mathcal{S}}$	$k_{ABD\mathcal{S}} \oplus h(k_{\mathcal{S}}, l_{ABD\mathcal{S}})$

(d)

Figure 8: Read and write acls (a), encrypted resources (b), key derivation structure (c), and tokens (d) of Figure 3 after B is granted write permission over o_2 and D is granted write permission over o_4

correctly enforce write privileges, the data owner can delegate to the external server both the generation and encryption with the correct key of the write tag of resource o. In fact, the data owner does not need to known or keep track of the write tag of her resources.

Procedure **Revoke** in Figure 7 receives as input a user u and a resource o and revokes u the privilege of modifying o. The procedure first updates the write access list $w[o]$ of the resource by removing user u (line 6). It then retrieves the derivation key (and the corresponding label) that will be used to encrypt the write tag of the resource (i.e., the key shared by the authorized writers of o, except u, and the server) by calling function **Get_Shared_Key** on the updated write access list of the resource (lines 7–8). The procedure then calls procedure **Create_New_Tag**, which is executed by the server, to generate a new write tag for the resource and update its representation at the server (line 9). Procedure **Create_New_Tag** in Figure 7 receives as input the identifier id of resource o and a label l_{new}, and generates a new write tag

for o, which is then encrypted with the key identified by l_{new}. The procedure first determines the tuple t in the outsourced table representing the resource with identifier id (line 9). It then finds the token that permits to derive the key k_{new} with which the new write tag must be encrypted to reflect the policy change (i.e., the token from $l_{\mathcal{S}}$ to l_{new}) (line 10). The procedure uses this token to derive k_{new} (line 11), randomly generates a value for the write tag (line 12), and encrypts this value with key k_{new} (line 13). Finally, the procedure updates t[`w_label`], setting it to l_{new} to reflect the policy update (line 14).

Example 4.3. *Consider the key derivation structure, outsourced resources, and tokens in Figure 8, and assume that the data owner revokes from A the write privilege over resource o_3 (i.e., $w[o_3]$=$w[o_3]\backslash\{A\}$=C). Since the key derivation structure does not include a key shared by the server and C, such a key is first computed as the hash of derivation key k_C with secure hash function h^s. Then, a new write tag is generated for o_3 and encrypted with $k_{C\mathcal{S}}$. Figure 9 illustrates the read an write access lists, the encrypted resources, the key derivation structure, and the tokens after this revocation.*

The following theorem formally proves that procedures **Grant** and **Revoke** correctly enforce updates to the write authorizations in the system.

Theorem 4.1 (Correct enforcement of policy updates). *Let $\mathcal{U}$ be a set of users, $\mathcal{S}$ be an external server, $\mathcal{O}$ be a set of resources with $r[o]$ and $w[o]$ the read and write access lists of o, respectively, and $\langle\mathcal{K},\mathcal{L},\mathcal{T}\rangle$ a key derivation structure. Procedures* **Grant** *and* **Revoke** *in Figure 7 guarantee that the following conditions are satisfied:*

1. *$\forall u \in \mathcal{U}$ and $\forall o \in \mathcal{O}$, u can decrypt* `encr_resource`$[o]$ *iff $u \in r[o]$* (read authorization enforcement)*;*
2. *$\forall u \in \mathcal{U}$ and $\forall o \in \mathcal{O}$, u can decrypt* `encw_tag`$[o]$ *iff $u \in w[o]$* (write authorization enforcement)*;*
3. *$\forall o \in \mathcal{O}$, $\mathcal{S}$ can decrypt* `encw_tag`$[o]$ (write control).

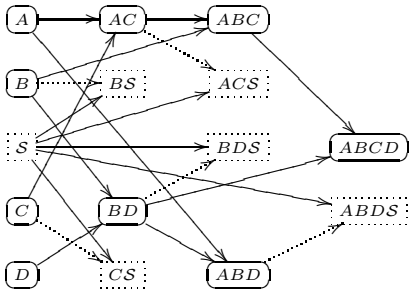
Proof. See Appendix A. □

o	r[o]	w[o]
o_1	$ABCD$	BD
o_2	$ABCD$	ABD
o_3	ABC	C
o_4	BD	BD

(a)

r_label	w_label	o_id	encw_tag	encr_resource
l_{ABCD}	l_{BDS}	1	α	zKZlJxVcCrC0g
l_{ABCD}	l_{ABDS}	2	ϵ	t9qdJqC7AImXU
l_{ABC}	l_{CS}	3	η	AxalPH8Kv37Ts
l_{BD}	l_{BDS}	4	ζ	xwfPJSLn.MVqY

(b)



(c)

from	to	val
l_A	l_{AC}	$k_{AC} \oplus h(k_A, l_{AC})$
l_A	l_{ABD}	$k_{ABD} \oplus h(k_A, l_{ABD})$
l_B	l_{ABC}	$k_{ABC} \oplus h(k_B, l_{ABC})$
l_B	l_{BD}	$k_{BD} \oplus h(k_B, l_{BD})$
l_C	l_{AC}	$k_{AC} \oplus h(k_C, l_{AC})$
l_D	l_{BD}	$k_{BD} \oplus h(k_D, l_{BD})$
l_{AC}	l_{ABC}	$k_{ABC} \oplus h(k_{AC}, l_{ABC})$
l_{BD}	l_{ABD}	$k_{ABD} \oplus h(k_{BD}, l_{ABD})$
l_{BD}	l_{ABCD}	$k_{ABCD} \oplus h(k_{BD}, l_{ABCD})$
l_{ABC}	l_{ABCD}	$k_{ABCD} \oplus h(k_{ABC}, l_{ABCD})$
l_S	l_{BS}	$k_{BS} \oplus h(k_S, l_{BS})$
l_S	l_{CS}	$k_{CS} \oplus h(k_S, l_{CS})$
l_S	l_{ACS}	$k_{ACS} \oplus h(k_S, l_{ACS})$
l_S	l_{BDS}	$k_{BDS} \oplus h(k_S, l_{BDS})$
l_S	l_{ABDS}	$k_{ABDS} \oplus h(k_S, l_{ABDS})$

(d)

Figure 9: Read and write acls (a), encrypted resources (b), key derivation structure (c), and tokens (d) of Figure 8 after A is revoked write permission over o_3

5. Write integrity control

Although the server can be assumed trustworthy to manage resources and delegated actions, it is important to provide a means to the data owner to verify that the server and users are behaving properly. Providing such a control has a double advantage: *i)* it allows detecting resource tampering, due to the server not performing the required check on the write tags or directly tampering with resources, and *ii)* it discourages improper behavior by the server and by the users since they know that their improper behavior can be easily detected, and their updates recognized as invalid and discarded. In this section, we illustrate our approach for providing the data owner with a means to verify that modifications to a resource have been produced only by users authorized to write the resource. In the following section, we will extend our solution to the management of updates to write privileges. As discussed in previous sections, if the server performs the correct control on the write tags, data integrity is automatically guaranteed. We therefore

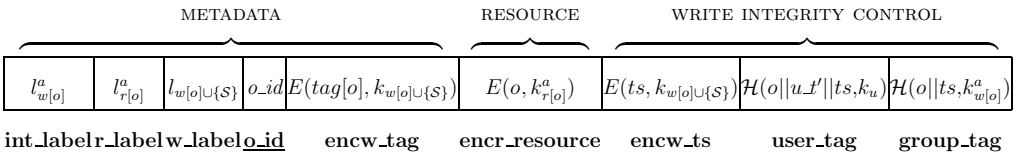


Figure 10: Structure of outsourced resources

illustrate how to perform a write integrity control to detect misbehavior (or laziness) by the server as well as misbehavior by users that can happen with the help of the server (not enforcing the control on the write tags since it is either colluding with the user or just behaving lazily) or without the help of the server (if the user improperly acquires the write tag for a resource by others).

A straightforward approach to provide such a write integrity control would be to apply a signature-based approach. This requires each user to have a pair ⟨`private,public`⟩ of keys and, when updating a resource, to sign the new resource content with her private key. The data owner can then check the write integrity by verifying that the signature associated with a resource correctly reflects the resource content and that it has been produced by a user authorized for the operation. Such an approach, while intuitive and simple, has however the main drawback of being computationally expensive (asymmetric encryption is considerably less efficient than symmetric encryption) and not well aligned with our approach, which - as a matter of fact - exploits symmetric encryption, tokens, and hash functions to provide efficiency in storage and processing. In the spirit of our approach, we then build our solution for controlling write integrity on HMAC functions [7]. In fact, for common platforms, the ratio between the execution times of digital signatures and of HMAC is more than three orders of magnitude. We then associate with each resource the following three integrity control fields (namely, `encw_ts`, `user_tag`, and `group_tag`) and metadata field (namely, `int_label`) to the fields introduced in Section 3 (see Figure 10).

- `encw_ts`: timestamp of the write operation, encrypted with the key $k_{w[o]\cup\{\mathcal{S}\}}$ corresponding to the group including the server and all the users in the write access list of o (i.e., $E(ts, k_{w[o]\cup\{\mathcal{S}\}})$);
- `user_tag`: HMAC $\mathcal{H}$ computed with the key k_u of the user who performed the write operation over the resource, concatenated with the

`user_tag` u_t' of the resource prior to the write operation,[1] and the timestamp ts of the write operation (i.e., $\mathcal{H}(o||u\_t'||ts,k_u)$);

- `group_tag`: HMAC $\mathcal{H}$ computed with the access key $k^a_{w[o]}$ corresponding to the write access list of o over the resource, concatenated with the timestamp of the write operation (i.e., $\mathcal{H}(o||ts,k^a_{w[o]})$).

- `int_label`: label of the key used to compute the `group_tag` (i.e., $l^a_{w[o]}$).

At time zero, when the data owner outsources her resources to the server, the values of the `user_tag` and of the `group_tag` are those computed by the owner with her own key for the `user_tag`, and with the key of the write access list of the resource (to which the owner clearly belongs) for the `group_tag`. Every time a user updates a resource, it also updates its `user_tag`, `group_tag`, and `int_label`.

A `user_tag` is considered valid if it matches the resource content and it is produced by a user in the write access list of the resource. The `user_tag` provides write integrity (meaning the resource has been written by an authorized user) and accountability of user actions (i.e., the user cannot repudiate her write actions). In fact, since the data owner knows the key k_u of every user u (which she generated and distributed), she can check the validity of the `user_tag` and detect possible mismatches, corresponding to unauthorized writes. In addition, every write operation considered valid (according to the control on the `user_tag`) cannot be repudiated by the user u whose key k_u generated the HMAC. The consideration of `group_tag` extends the ability of checking the validity of the write operations (i.e., write integrity) also to all the users in the write access list of the resource. Note that allowing writers to check resource integrity is not less important than allowing the data owner to perform the check, as it guarantees that, even in cases of data owner absence, all write operations are performed on resources that have not been improperly modified. Indeed, before modifying a resource content, the writer will check its integrity to be sure that she is operating on genuine data.

While we assume the server to be trustworthy and therefore not interested in tampering with the resources, we note that the `user_tag` would allow also

[1]The reason for including the `user_tag` of the resource prior to the write operation is to provide the data owner with a hash chain connecting all the resource versions (we assume the server to never overwrite resources but to maintain all their versions).

to detect possible tampering of the server with the resource (since not being an authorized writer, the server will not be able to produce a valid `user_tag`). The server could also tamper with the write authorizations, by decrypting the write tag and encrypting it with the key corresponding to a different write access list. However, the improper inclusion of a user in the write access list does not have any different effect than when the server does not perform the control, since the user improperly included in the write access list will not be able to produce a valid `user_tag`. Analogously, the improper removal of a user from the write access list has the same effects as when the server refuses its services.

Unauthorized write operations, in the case of a well behaving server, can only happen if a user has improperly acquired or received from other authorized users the write tag of a resource. Whichever the case, the user will be able to provide neither a valid `user_tag` nor a valid `group_tag` for the resource. Also, the data owner and any user authorized to write the resource will be able to detect the invalidity of the `group_tag`, since the key used to compute the HMAC will not correspond to the access key of $w[o]$.

6. Write integrity control with policy updates

A change in the write authorizations of a resource also requires a change in the write integrity fields associated with the resource. In particular, when user u gains the privilege of writing resource o as a consequence of a grant operation, the set $w[o] \cup \{u\}$ of users should be able to generate and check the `group_tag` of o. If this were not the case, u would not be able to verify the integrity of the resource before modifying its content. Analogously, when u is revoked the write privilege over o, the set $w[o] \setminus \{u\}$ of users should be able to generate and check the `group_tag` of o. If this were not the case, u could possibly collude with the server to modify the content of resource o without being detected by the other writers of the resource. A naive strategy to compute a `group_tag` that guarantees the correct enforcement of integrity checks would require the data owner, when granting/revoking a write privilege, to: *i)* download the encrypted resource from the external server, *ii)* decrypt its content, *iii)* compute the HMAC of the resource with the access key of the new set of writers, and *iv)* send the new value of the `group_tag` back to the server. However, this approach causes a high computation and communication overhead for the data owner, who should interact with the external server at every update of the write authorizations. To reduce this overhead,

we put forward the idea of modifying the key derivation structure to prevent the re-computation of the `group_tag`, and therefore the need for the data owner to download the resource at every policy update. In the remainder of this section, we first describe our approach for efficiently supporting integrity verification in case of policy updates (Section 6.1), and we then discuss its exposure to integrity violations (Section 6.2).

6.1. Integrity keys

Let us assume that the data owner grants user u the write privilege for resource o. Since the `group_tag` of o is computed using key $k^a_{w[o]}$ that u does neither know nor can derive, a straightforward approach that would permit u to verify the integrity of o consists in inserting into the key derivation structure a token from k_u to $k^a_{w[o]}$. This solution has however two drawbacks: *i)* it does not handle revoke operations; and *ii)* it permits u to derive access key $k^a_{w[o]}$ used to encrypt resources o' with $r[o']{=}w[o]$ (and to generate the `group_tag` of resources o' with $w[o']{=}w[o]$). With respect to the first drawback, we note that the data owner can always detect the misbehavior of users who modify revoked resources since they are not able to generate correct user tags for these resources. With respect to the second drawback, this solution has the side effect of permitting used u to access the content of resources she is not authorized to read. For instance, with reference to Example 4.2, granting A write access to o_2 causes the insertion of a token from k_A to k^a_{BD}, used to compute the `group_tag` of o_2. However, k^a_{BD} is also used to encrypt o_4 ($r[o_4]{=}BD$), which A is not authorized to read. This confidentiality breach is due to the fact that the same key is used for two different purposes: protect data confidentiality (when encrypting the content of resources), and provide integrity guarantees to outsourced data (when computing the `group_tag` of resources). A simple and effective solution to this problem consists in using two different keys for protecting data confidentiality and for providing integrity. We then associate an *integrity key* (and corresponding label) with a derivation key whenever needed, and we use integrity keys to compute group tags. We note that, like access keys, integrity keys do not provide derivation capability via tokens (i.e., tokens cannot have integrity keys as starting point). Given derivation key k_U associated with a group U of users, the corresponding integrity key k^i_U is obtained by applying a secure hash function h^i to k_U (i.e., $k^i_U{=}h^i(k_U)$). The `group_tag` of a resource o is then the HMAC, computed with the integrity key $k^i_{w[o]}$ of the write access list of o, over the resource concatenated with the timestamp of the write operation

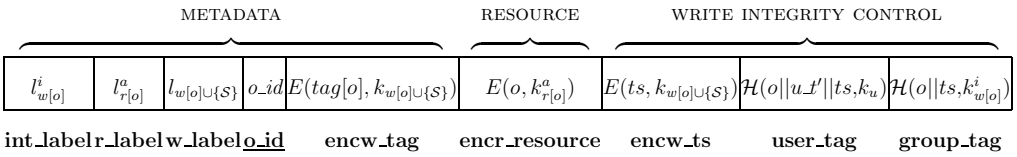


Figure 11: Structure of outsourced resources adopting integrity keys

(see Figure 11). When user u is granted the privilege of modifying o, the data owner inserts into the key derivation structure a token that permits u to derive integrity key $k^i_{w[o]}$. With reference to the example above, when A is granted write access to o_2, the data owner inserts a token from k_A to k^i_{BD}, which permits A to verify the integrity of o_2 without compromising the confidentiality of o_4.

It is interesting to note that, when inserting a token from k_u to k^i_U, the set of users who know or can derive k^i_U becomes $U\cup\{u\}$, and is therefore different from the set of users who know or can derive the corresponding derivation key k_U. As a consequence, when granting u write access to o, the integrity field `int_label` of o remains unchanged and is equal to l^i_U, where U corresponds to the write access list of resource o before the grant operation (i.e., $u\notin U$). To limit this mismatch between $w[o]$ and the label of the key used for the `group_tag`, at each write operation the user who modifies the resource content generates a new `group_tag` using the integrity key associated with the current write access list of the resource, which reflects the grant/revoke operation. For instance, with reference to Example 4.2, after granting A write privilege over o_2 (but before any further update to o_2), integrity field `int_label` for resource o_2 has value l^i_{BD} since the `group_tag` had been computed using the integrity key of $w[o_2]$ before inserting A into the write access list of the resource. Assume now that user B modifies resource o_2. She will compute the `group_tag` for the resource as $\mathcal{H}(o_2||ts,k^i_{ABD})$ and, when uploading the new resource content and the corresponding `group_tag`, she will also update the value of `int_label`, setting it to l^i_{ABD}.

6.2. Exposure risk

We now discuss two cases of possible exposure of data integrity that might occur as a consequence of a policy update.

Revoke. According to the mechanism illustrated above, when the data owner revokes u write access to o neither the `group_tag` of the resource nor the key

derivation structure are modified. As a consequence, u is able to verify and to generate a valid `group_tag` for o till the first update of the resource content by an authorized writer. In this time window, u is not able to decrypt `encw_tag` for o but, colluding with the server, she could possibly modify the resource content and compute a valid `group_tag` for o (i.e., a tag that authorized writers would accept). In fact, u can derive the integrity key identified by `int_label`, and then compute a `group_tag` that is compliant with the new resource content, using the key identified by `int_label`. Note that this collusion has the effect that we have when the server does not check write requests.

Policy split. A similar situation can happen when a user u is granted the write privilege for a resource o that has the same write access list of other resources. In fact, the integrity key k^i used to compute the `group_tag` of o is also used to compute the `group_tag` of all the resources o' with $w[o']{=}w[o]$ before the grant operation. Since u, as a consequence of the grant operation, can derive k^i to verify the integrity of o, she can (as a side-effect) also verify and compute a valid `group_tag` for all those resources with the same `int_label`. Also in this situation, u can collude with the server (or exploit the laziness of the server not checking write requests) to modify the content of o without being detected by authorized users.

The misbehaviors described above for the revoke and policy split cases do not go undetected by the data owner. In fact, users cannot compute a valid user tag for a resource that she is not authorized to write. Also, exposure to integrity violations is limited and well identifiable. The data owner can then counteract them by explicitly recomputing the `group_tag` of the resource subject to the revoke/grant operation when she considers the communication and computation overhead worth to protect the exposed resources. The risk of integrity violations caused by policy splits can be mitigated by a proper organization of the resources, that is, adopting the same integrity key only if the write access list of the resources is likely to evolve in the same way.

7. Related work

In the last few years, several research efforts have been devoted to enable data owners to outsource the storage and management of their data to possibly non-fully trusted third parties (e.g., [4, 8, 9]). Most proposals

have addressed the problem of efficiently performing queries on outsourced encrypted data, without decrypting sensitive information at the server side. These approaches typically define indexes that are stored together with the encrypted data and are used by the external server to select the data to be returned in response to a query (e.g., [2, 3, 10, 11, 12]). Alternative solutions instead adopt specific encryption functions (usually exploiting homomorphic properties) that permit keyword searches at the server side over encrypted data, without compromising the confidentiality of the outsourced data (e.g., [13]). A related line of work is represented by the design of mechanisms for easily verifying that neither an external malicious user nor the external server improperly modify data in storage (e.g., [14, 15, 16]) and that the external server provides a correct response to users queries (e.g., [17, 18, 19, 20, 21, 22]). However, these approaches do not take into consideration possible access restrictions that users may have on the outsourced data.

The problem of specifying and enforcing an authorization policy on outsourced data, without the need for the data owner to filter query results, has recently received the attention of the research community (e.g., [5, 23, 24, 25]). The first proposal in this direction focuses on protecting access to published XML documents [23]. To this purpose, different portions of the XML tree are encrypted using different keys and specific metadata nodes are inserted into the XML structure. The solution in [5], which can be adopted independently from the data organization, first proposes the combined use of selective encryption and key derivation strategies, to guarantee a limited key management overhead at the client side while correctly enforcing the data owner access control policy. This approach also permits to delegate to the external server the management of updates to read authorizations. The solution in [24] uses attribute-based encryption for enforcing access restrictions to outsourced data to provide system scalability. The solution in [26] adopts selective encryption for enforcing a subscription-based access control policy. The work in [25] proposes an approach that does not require complete trust in the external provider w.r.t. both resource content and authorization management.

All the access control approaches mentioned above however only focused on the enforcement of read access privileges and do not support restrictions on write operations, which are assumed to be an exclusive privilege of the data owner. In the literature, few works have addressed this issue. The solution in [27] adopts selective encryption to enforce the data owner's au-

thorization policy on outsourced data, adopting asymmetric encryption to enforce both read and write privileges and defining two key derivation hierarchies: one for private keys (to enforce read privileges) and one for public keys (to enforce write privileges). The solution also proposes to replicate resources and perform updates on a different copy of the data, to prevent unauthorized write operations from destroying valuable data content. The proposal in [28] adopts Attribute-Based Encryption (ABE) and Attribute-Based Signature (ABS) techniques to enforce read and write access privileges, respectively. This approach, although effective, has the disadvantage of requiring the presence of a trusted party for correct policy enforcement. The work in [29] investigates a similar approach based on the combined use of ABE and ABS for supporting both read and write privileges. This solution has the advantage over the approach in [28] of being suited also to distributed scenarios. All these approaches, however, do not address the problem of efficiently supporting changes to the authorization policy by the owner of the data, which may require expensive data re-encryption operations.

8. Conclusions

In this paper, we presented an approach for supporting both read and write privileges on outsourced encrypted data that does not require the data owner intervention to filter query results and/or access requests. Our solution also efficiently supports updates in the access control policy, while minimizing the data owner's overhead and resulting transparent to the users. Data integrity can be easily verified by the data owner and by the users authorized to write resources, thus providing guarantees on the fact that the data externally stored have not been tampered with by unauthorized parties without being detected. The proposed solution relies on the use of symmetric encryption, hashing, and HMAC functions for enforcing access control and integrity checks in an efficient and effective way. Our proposal then performs a step towards the development of solutions actually applicable in real-world scenarios where efficiency and scalability are mandatory.

Acknowledgments

This work was supported in part by the EC within the 7FP under grant agreement 257129 (PoSecCo), by the Italian Ministry of Research within PRIN 2010-2011 project "GenData 2020" (2010RTFWBH), and by Google,

under the Google Research Award program. The work of Sushil Jajodia was partially supported by the US Air Force Office of Scientific Research under grant FA9550-09-1-0421.

References

[1] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, P. Samarati, Support for write privileges on outsourced data, in: Proc. of SEC 2012, Heraklion, Greece.

[2] E. Damiani, S. De Capitani di Vimercati, S. Jajodia, S. Paraboschi, P. Samarati, Balancing confidentiality and efficiency in untrusted relational DBMSs, in: Proc. of CCS 2003, Washington, DC, USA.

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

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

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

[6] M. Atallah, K. Frikken, M. Blanton, Dynamic and efficient key management for access hierarchies, in: Proc. of CCS 2005, Alexandria, VA, USA.

[7] M. Bellare, R. Canetti, H. Krawczyk, Keying hash functions for message authentication, in: Proc. of CRYPTO 1996, Santa Barbara, CA, USA.

[8] R. Jhawar, V. Piuri, P. Samarati, Supporting security requirements for resource management in cloud computing, in: Proc. of CSE 2012, CSE 2012, Paphos, Cyprus.

[9] V. P. R. Jhawar, M. Santambrogio, Fault tolerance management in cloud computing: A system-level perspective, IEEE Systems Journal (2012).

[10] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, P. Samarati, Private data indexes for selective access to outsourced data, in: Proc. of WPES 2011, Chicago, IL, USA.

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

[12] H. Wang, L. V. S. Lakshmanan, Efficient secure query evaluation over encrypted XML databases, in: Proc. of VLDB 2006, Seoul, Korea.

[13] C. Wang, N. Cao, K. Ren, W. Lou, Enabling secure and efficient ranked keyword search over outsourced cloud data, IEEE TPDS 23 (2012) 1467–1479.

[14] E. Mykletun, M. Narasimha, G. Tsudik, Authentication and integrity in outsourced databases, ACM TOS 2 (2006) 107–138.

[15] G. Ateniese, R. Burns, R. Curtmola, J. Herring, O. Khan, L. Kissner, Z. Peterson, D. Song, Remote data checking using provable data possession, ACM TISSEC 14 (2011) 12:1–12:34.

[16] C. Wang, S. Chow, Q. Wang, K. Ren, W. Lou, Privacy-preserving public auditing for secure cloud storage, IEEE TC 62 (2012) 362–375.

[17] G. Di Battista, B. Palazzi, Authenticated relational tables and authenticated skip lists, in: Proc. of DBSec 2007, Redondo Beach, CA, USA.

[18] R. Merkle, A certified digital signature, in: Proc. of CRYPTO 1989, Santa Barbara, CA, USA.

[19] H. Pang, A. Jain, K. Ramamritham, K. Tan, Verifying completeness of relational query results in data publishing, in: Proc. of SIGMOD 2005, Baltimore, MA, USA.

[20] R. Sion, Query execution assurance for outsourced databases, in: Proc. of VLDB 2005, Trondheim, Norway.

[21] H. Wang, J. Yin, C. Perng, P. Yu, Dual encryption for query integrity assurance, in: Proc. of CIKM 2008, Napa Valley, CA, USA.

[22] M. Xie, H. Wang, J. Yin, X. Meng, Integrity auditing of outsourced data, in: Proc. of VLDB 2007, Vienna, Austria.

[23] G. Miklau, D. Suciu, Controlling access to published data using cryptography, in: Proc. of VLDB 2003, Berlin, Germany.

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

[25] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, G. Pelosi, P. Samarati, Encryption-based policy enforcement for cloud storage, in: Proc. of SPCC 2010, Genova, Italy.

[26] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, G. Livraga, Enforcing subscription-based authorization policies in cloud scenarios, in: Proc. of DBSec 2012, Paris, France.

[27] M. Raykova, H. Zhao, S. Bellovin, Privacy enhanced access control for outsourced data sharing, in: Proc. of FC 2012, Bonaire.

[28] F. Zhao, T. Nishide, K. Sakurai, Realizing fine-grained and flexible access control to outsourced data with attribute-based cryptosystems, in: Proc. of ISPEC 2011, Guangzhou, China.

[29] S. Ruj, M. Stojmenovic, A. Nayak, Privacy preserving access control with authentication for securing data in clouds, in: Proc. of CCGrid 2012, Ottawa, Canada.

Appendix A. Proof of theorems

Theorem 3.1 (Correctness of procedure Define_Key_Derivation_Structure). *Let $\mathcal{U}$ be a set of users, $\mathcal{S}$ be an external server, $\mathtt{U}\subseteq 2^{\mathcal{U}}$ be a family of subsets of users in $\mathcal{U}$ such that $\forall u\in\mathcal{U}$, $\{u\}\in\mathtt{U}$, and $\mathtt{U}^s$ and $\mathtt{U}^a$ be two subsets of $\mathtt{U}$. Triple $\langle\mathcal{K},\mathcal{L},\mathcal{T}\rangle$ computed by function* **Define_Key_Derivation_Structure** *in Figure 2 is a key derivation structure (Definition 3.2).*

Proof. We prove that the triple $\langle\mathcal{K},\mathcal{L},\mathcal{T}\rangle$ computed by function **Define_Key_Derivation_Structure** satisfies all the conditions in Definition 3.2.

- Condition 1 is satisfied since in Step 1 the function generates a key for each set of users in $\mathtt{U}$ and inserts it into $\mathcal{K}'$ (lines 6–7). The set $\mathcal{K}'$ of keys resulting from Step 1 then corresponds to the set of keys of the set-based key derivation graph. In Step 2, the function generates a key $k_{\mathcal{S}}$ for the server (line 14) and inserts both key $k_{\mathcal{S}}$ and all the keys in $\mathcal{K}'$ into $\mathcal{K}$ (line 15). For each set U of users in $\mathtt{U}^s$, the function computes the key for $U\cup\{\mathcal{S}\}$ as $h^s(k_U)$ (lines 18–19), where h^s is a secure hash function, and inserts $k_{U\cup\{\mathcal{S}\}}$ into $\mathcal{K}$ (line 21). Similarly, for each set U of users in $\mathtt{U}^a$, the function computes the access key k^a_U as $h^a(k_U)$ (lines 24–25), where h^a is a secure hash function, and inserts k^a_U into $\mathcal{K}$ (line 27).

- Condition 2 is satisfied since in Step 1 the function defines a set $\mathcal{T}'$ of tokens that guarantees that each key k_U in $\mathcal{K}$ can be directly derived from a set $\{k_{U_1},\ldots,k_{U_n}\}$ of keys in $\mathcal{K}'$ such that $U_1\cup\ldots\cup U_n{=}U$ (lines 10–12). As proved in [5], this property is equivalent to Condition 2 in Definition 3.1. Therefore the set of tokens $\mathcal{T}'$ resulting from Step 1 corresponds to the set of tokens of the set-based key derivation graph. In Step 2 the function inserts into $\mathcal{T}$ all the tokens in $\mathcal{T}'$ (line 17) and, for each set U of users in $\mathtt{U}^s$, it defines and inserts into $\mathcal{T}$ token $t_{\mathcal{S},U\cup\{\mathcal{S}\}}$ that permits to derive $k_{U\cup\{\mathcal{S}\}}$ from $k_{\mathcal{S}}$ (line 23). □

Theorem 3.2 (Correct enforcement of authorizations). *Let $\mathcal{U}$ be a set of users, $\mathcal{S}$ be an external server, $\mathcal{O}$ be a set of resources such that $\forall o\in\mathcal{O}$ $r[o]$ and $w[o]$ are the read and write access lists of o, respectively. Our access control system satisfies the following conditions:*

1. $\forall u \in \mathcal{U}$ *and* $\forall o \in \mathcal{O}$, u *can decrypt* `encr_resource`$[o]$ *iff* $u \in r[o]$ (read authorization enforcement)*;*
2. $\forall u \in \mathcal{U}$ *and* $\forall o \in \mathcal{O}$, u *can decrypt* `encw_tag`$[o]$ *iff* $u \in w[o]$ (write authorization enforcement)*;*
3. $\forall o \in \mathcal{O}$, $\mathcal{S}$ *can decrypt* `encw_tag`$[o]$ (write control).

Proof. The proof is based on the fact that, by Theorem 3.1, triple $\langle \mathcal{K},\mathcal{L},\mathcal{T} \rangle$ computed by function **Define_Key_Derivation_Structure** is a key derivation structure. We first note that, by procedure **Initialize_System** in Figure 4, $\mathcal{K}$ includes a derivation key k_u for each user $u \in \mathcal{U}$, and a derivation key k_U for each set U of users representing a read or write access list of a resource $o \in \mathcal{O}$. In fact, function **Define_Key_Derivation_Structure** is called over $\mathcal{U}$, $\mathcal{S}$, `U`, $\mathtt{U}^s$, $\mathtt{U}^a$, h^s, and h^a, with $\mathtt{U}^s$ the set of write access lists, $\mathtt{U}^a$ the set of read access lists, and `U` the result of $\mathtt{U}^a \cup \mathtt{U}^s$ together with all the singleton sets $\{u\}$ of users in $\mathcal{U}$ (lines 2–5). We now prove that each condition in Theorem 3.2 holds.

1. u can decrypt `encr_resource`$[o] \Longrightarrow u \in r[o]$.
   Assume, by contradiction, that $u \notin r[o]$ can decrypt `encr_resource`$[o]$. Since `encr_resource`$[o]$ is computed by encrypting o with access key $k^a_{r[o]}$ (line 19), u can either compute or derive $k^a_{r[o]}$. $\mathcal{T}$ does not include any token that permits to derive access keys, therefore u needs to know derivation key $k_{r[o]} \in \mathcal{K}$ with which $k^a_{r[o]} \in \mathcal{K}$ has been computed. However, $\mathcal{T}$ includes a token (or a sequence thereof) from derivation key k_u of user u (lines 7–8) to derivation key k_U iff $u \in U$ (Condition 2 in Definition 3.1). This implies that $\{u\} \subseteq r[o]$, which contradicts our hypothesis.

   $u \in r[o] \Longrightarrow u$ can decrypt `encr_resource`$[o]$.
   By Condition 2 in Definition 3.1, there exists a token (or a sequence thereof) in $\mathcal{T}$ that permits to derive derivation key k_U from k_u iff $u \in U$. Therefore, if $u \in r[o]$, there exists a token (or a sequence thereof) in $\mathcal{T}$ from k_u to $k_{r[o]} \in \mathcal{K}$. Since u can derive $k_{r[o]}$ and h^a is public, she can also compute access key $k^a_{r[o]} = h^a(k_{r[o]})$ and decrypt `encr_resource`$[o]$.
2. u can decrypt `encw_tag`$[o] \Longrightarrow u \in w[o]$.
   Assume, by contradiction, that $u \notin w[o]$ can decrypt `encw_tag`$[o]$. Since `encw_tag`$[o]$ is computed by encrypting $tag[o]$ with key $k_{w[o] \cup \{\mathcal{S}\}}$ (line 18), u can compute or derive $k_{w[o] \cup \{\mathcal{S}\}}$. Since all tokens in $\mathcal{T}$ that permit to derive key $k_{w[o] \cup \{\mathcal{S}\}}$ shared with the server have $k_{\mathcal{S}}$ as starting

point (Condition 2 in Definition 3.2), u must know (or be able to derive) derivation $k_{w[o]}$. However, $\mathcal{T}$ includes a token (or a sequence thereof) from derivation key k_u of user u (lines 7–8) to derivation key $k_U \in \mathcal{K}$, iff $u \in U$ (Condition 2 in Definition 3.1). This implies that $\{u\} \subseteq w[o]$, which contradicts our hypothesis.

$u \in w[o] \implies u$ can decrypt `encw_tag`$[o]$.
By Condition 2 in Definition 3.1, there exists a token (or a sequence thereof) in $\mathcal{T}$ that permits to derive derivation key k_U from k_u iff $u \in U$. Therefore, if $u \in w[o]$, there exists a token (or a sequence thereof) in $\mathcal{T}$ from k_u to $k_{w[o]}$. Since u can derive $k_{w[o]}$ and h^s is public, she can also compute key $k_{w[o] \cup \{\mathcal{S}\}} = h^s(k_{w[o]})$ and decrypt `encw_tag`$[o]$.

3. $\mathcal{S}$ can decrypt `encw_tag`$[o]$.
   As noted above, `encw_tag`$[o]$ is computed by encrypting $tag[o]$ with key $k_{w[o] \cup \{\mathcal{S}\}}$. Since $\mathcal{S}$ knows key $k_{\mathcal{S}}$ and, for each key $k_{U \cup \{\mathcal{S}\}}$ in $\mathcal{K}$, $\mathcal{T}$ includes token $t_{\mathcal{S}, U \cup \{\mathcal{S}\}}$ (Condition 2 in Definition 3.2), $\mathcal{S}$ can derive $k_{w[o] \cup \{\mathcal{S}\}}$ and decrypt `encw_tag`$[o]$. □

Before proving Theorem 4.1, we show that both function **Get_Key** and function **Get_Shared_Key**, which possibly update the key derivation structure, do not compromise its correctness (Definition 3.2).

Lemma A.1 (Correctness of function Get_Key). *Let $\mathcal{U}$ be a set of users, U be a subset of $\mathcal{U}$, $\mathcal{S}$ be an external server, and $\langle \mathcal{K}, \mathcal{L}, \mathcal{T} \rangle$ be a key derivation structure. Triple $\langle \mathcal{K}', \mathcal{L}', \mathcal{T}' \rangle$ resulting from the execution of function* **Get_Key**(U) *in Figure 5 is a key derivation structure (Definition 3.2).*

Proof. Since we assume that $\langle \mathcal{K}, \mathcal{L}, \mathcal{T} \rangle$ is a key derivation structure when function **Get_Key** is called, we need to consider only the keys and tokens inserted, updated, or removed by the function.

If the key derivation structure already includes a key k_U known to all and only the users in U, the function does not modify $\langle \mathcal{K}, \mathcal{L}, \mathcal{T} \rangle$ and therefore the lemma holds (lines 1–2).

If, on the contrary, $k_U \notin \mathcal{K}$, function **Get_Key** inserts it into the key derivation structure. We then need to prove that such an insertion does not violate the conditions in Definition 3.2.

- Condition 1 is satisfied since function **Get_Key** generates a derivation k_U (and a label l_U) and computes the corresponding access key k_U^a (and

a label l^a_U). It then inserts k_U,k^a_U into $\mathcal{K}$ and l_U,l^a_U into $\mathcal{L}$ (lines 3–7). The function then inserts access key k^a_U into $\mathcal{K}$ only when the corresponding derivation key k_U has been inserted into $\mathcal{K}$.

- Condition 2 is satisfied as the set of tokens inserted into $\mathcal{T}$ by function **Get_Key** guarantees that k_U can be directly derived from a set $\{k_{U_1},\ldots,k_{U_n}\}$ of keys in $\mathcal{K}$ such that $U_1 \cup \ldots \cup U_n{=}U$ (lines 8–12). As proved in [5], this property is equivalent to Condition 2 in Definition 3.1.

Lemma A.2 (Correctness of function Get_Shared_Key). *Let $\mathcal{U}$ be a set of users, U be a subset of $\mathcal{U}$, $\mathcal{S}$ be an external server, and $\langle\mathcal{K},\mathcal{L},\mathcal{T}\rangle$ be a key derivation structure. Triple $\langle\mathcal{K}',\mathcal{L}',\mathcal{T}'\rangle$ resulting from the execution of function* **Get_Shared_Key**(U) *in Figure 5 is a key derivation structure (Definition 3.2).*

Proof. Since we assume that $\langle\mathcal{K},\mathcal{L},\mathcal{T}\rangle$ is a key derivation structure when function **Get_Shared_Key** is called, we need to consider only the keys and tokens inserted, updated, or removed by the function.

If the key derivation structure already includes a key $k_{U\cup\{\mathcal{S}\}}$ shared by the users in U and the external server, the function does not modify $\langle\mathcal{K},\mathcal{L},\mathcal{T}\rangle$ and therefore the lemma holds (lines 19–20).

If, on the contrary, $k_{U\cup\{\mathcal{S}\}}{\notin}\mathcal{K}$, function **Get_Shared_Key** inserts it into the key derivation structure. We then need to prove that such an insertion does not violate the conditions in Definition 3.2.

- Condition 1 is satisfied since function **Get_Shared_Key** computes key $k_{U\cup\{\mathcal{S}\}}$ as the result of hash function h^s over k_U (line 22) and it obtains k_U by calling function **Get_Key** (line 21), which does not compromise the correctness of the key derivation structure (as proved by Lemma A.1). Function **Get_Shared_Key** then inserts $k_{U\cup\{\mathcal{S}\}}$ and the corresponding label into $\mathcal{K}$ and $\mathcal{L}$, respectively (lines 24–25). The function then inserts key $k_{U\cup\{\mathcal{S}\}}$ into $\mathcal{K}$ only when derivation key k_U has been inserted into $\mathcal{K}$.

- Condition 2 is satisfied since function **Get_Shared_Key** inserts a token $t_{\mathcal{S},U\cup\{\mathcal{S}\}}$, which permits the server to derive $k_{U\cup\{\mathcal{S}\}}$ from $k_{\mathcal{S}}$ (line 26).□

Theorem 4.1 (Correct enforcement of policy updates). *Let $\mathcal{U}$ be a set of users, $\mathcal{S}$ be an external server, $\mathcal{O}$ be a set of resources with $r[o]$ and*

$w[o]$ the read and write access lists of o, respectively, and $\langle \mathcal{K},\mathcal{L},\mathcal{T} \rangle$ a key derivation structure. Procedures **Grant** *and* **Revoke** *in Figure 7 guarantee that the following conditions are satisfied:*

1. *$\forall u \in \mathcal{U}$ and $\forall o \in \mathcal{O}$, u can decrypt* `encr_resource`$[o]$ *iff* $u \in r[o]$ (read authorization enforcement)*;*
2. *$\forall u \in \mathcal{U}$ and $\forall o \in \mathcal{O}$, u can decrypt* `encw_tag`$[o]$ *iff* $u \in w[o]$ (write authorization enforcement)*;*
3. *$\forall o \in \mathcal{O}$, $\mathcal{S}$ can decrypt* `encw_tag`$[o]$ (write control)*.*

Proof. Since we assume that all the conditions are satisfied when procedure **Grant** (**Revoke**, respectively) is called, we need to consider only users and resources for which the policy changes. Also, Condition 1 is not affected by procedures **Grant** and **Revoke** as they neither modify the read access list of resources nor re-encrypt resources content.

- **Grant**(u,o). The procedure inserts u into $w[o]$ (line 2), therefore Condition 2 is satisfied iff u can decrypt `encw_tag`$[o]$. The write tag $tag[o]$ of resource o is encrypted by procedure **Encrypt_Tag** with the key k_{new} associated with label l_{new}. Since procedure **Grant** calls procedure **Encrypt_Tag** with $l_{w[o] \cup \{\mathcal{S}\}}$ as input, the server encrypts $tag[o]$ with key $k_{w[o] \cup \{\mathcal{S}\}}$ (line 5). This key belongs to the key derivation structure, since procedure **Grant** calls function **Get_Shared_Key** with $w[o]$ as input (line 3). By Lemma A.2, key $k_{w[o] \cup \{\mathcal{S}\}}$ can be derived by all and only users in $w[o]$ and by the server. Therefore, procedure **Grant** satisfies both Condition 2 and Condition 3.

- **Revoke**(u,o). The procedure removes u from $w[o]$ (line 6), therefore Condition 2 is satisfied iff u cannot decrypt `encw_tag`$[o]$. Procedure **Create_New_Tag** generates a new tag for o and encrypts it with the key k_{new} associated with label l_{new}. Since procedure **Revoke** calls procedure **Create_New_Tag** with $l_{w[o] \cup \{\mathcal{S}\}}$ as input, the server encrypts the new value of the tag with key $k_{w[o] \cup \{\mathcal{S}\}}$ (line 9). This key belongs to the key derivation structure, since procedure **Revoke** calls function **Get_Shared_Key** with $w[o]$ as input (line 7). By Lemma A.2, key $k_{w[o] \cup \{\mathcal{S}\}}$ can be derived by all and only users in $w[o]$ and by the server. Therefore, procedure **Revoke** satisfies both Condition 2 and Condition 3.

□