Journal of Computer Security 0 (0) 1
IOS Press

Enforcing authorizations while protecting access confidentiality[1]

Sabrina De Capitani di Vimercati [a,*], Sara Foresti [a], Stefano Paraboschi [b], Gerardo Pelosi [c], and Pierangela Samarati [a]

[a] *Dipartimento di Informatica, Università degli Studi di Milano, Italy*
E-mails: sabrina.decapitani@unimi.it, sara.foresti@unimi.it, pierangela.samarati@unimi.it
[b] *Dipartimento di Ingegneria Gestionale, dell'Informazione e della Produzione, Università degli Studi di Bergamo, Italy*
E-mail: parabosc@unibg.it
[c] *Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Italy*
E-mail: gerardo.pelosi@polimi.it

Abstract. Cloud computing is the reference paradigm to provide data storage and management in a convenient and scalable manner. However, moving data to the cloud raises several issues, including the confidentiality of data and of accesses that are no more under the direct control of the data owner. The shuffle index has been proposed as a solution for addressing these issues when data are stored at an external third party.
In this paper, we extend the shuffle index with support for access control, that is, for enforcing authorizations on data. Our approach is based on the use of selective encryption and on the organization of data and authorizations in two shuffle indexes. Owners regulate access to their data through authorizations that allow different users to access different portions of the data, while, at the same time, the confidentiality of accesses is guaranteed. The proposed approach also supports update operations over the outsourced data collection (i.e., insertion, removal, and update) as well as of the access control policy (i.e., grant and revoke). Also, our approach protects the nature of each access operation, making revoke operations and resource removal operations indistinguishable by the storing server and/or observing users.

Keywords: Shuffle index, Access control, Data confidentiality, Access confidentiality

1. Introduction

The rapid advancement in Information and Communication Technology (ICT) and the growing adoption of the cloud computing paradigm have produced an ever increasing reliance on external parties for storing and processing data. Together with the clear benefits in term of low cost and high availability (e.g., [2]), the involvement of external providers for storing data and providing services raises also issues of ensuring proper protection of information against the providers themselves (e.g., [3–5]). The research and industrial community has recognized these issues and investigated different aspects of the problem, with considerable attention paid to the need of maintaining information confidential to the providers themselves that, even if trustworthy to provide the service, should not be allowed visibility over the

[1] A preliminary version of this paper appeared under the title "Access Control for the Shuffle Index," in *Proc. of the 30th Annual IFIP WG 11.3 Conference on Data and Applications Security and Privacy* (DBSec 2016), Trento, Italy, July 2016 [1].

[*] Corresponding author. E-mail: sabrina.decapitani@unimi.it.

stored data [5]. In addition to the need to protect confidentiality of the stored data (*content confidentiality*), other proposals (e.g., [6–16]) have been devoting attention to the need of protecting confidentiality of the accesses executed on the data (*access confidentiality*), that is, protecting confidentiality on the fact that an access aims at a specific piece of information or that two accesses aim at the same target (which is also referred to as *pattern confidentiality*). There are several reasons for which access confidentiality should be guaranteed, including the fact that breaches to access confidentiality may leak information on access profiles, and, in the end, even on the data themselves, therefore breaking data confidentiality [17]. In fact, if the frequency distribution of accesses to the outsourced data is known (external knowledge), a cloud provider can keep track of the frequency with which the encrypted data are accessed and can then reconstruct the content of the encrypted dataset. Collected information about accesses can then possibly enable the observing cloud provider to infer sensitive information about the content of the outsourced data collection as well as sensitive information about its users. For instance, a cloud provider observing a search by a user on a medical database can infer the disease of interest for the user and can then also infer that the requesting user (or a person close to her) suffers from that disease, thus breaching her privacy. The protection of content and access confidentiality is therefore fundamental to also protect the privacy of the users accessing data.

Among the recent proposals specifically considering the access confidentiality problem in database management scenarios (with attention to efficiency and functionality guarantees that should be provided) there is the *shuffle index* [7]. The shuffle index is an index-based hierarchical organization of the data supporting efficient and effective access execution and providing access confidentiality with limited (compared to classical solutions) performance overhead. The key idea to provide access confidentiality is a dynamic re-allocation of data at every access so to breach the otherwise static correspondence between data and physical blocks in which they are stored. The shuffle index, while supporting accesses by multiple users [8], assumes all users to be entitled to access the complete data structure: data are encrypted with a key shared between the data owner and all users, and all users can retrieve and decrypt these data, hence accessing the plaintext content. Encryption is applied only to provide (content and access) confidentiality with respect to the storing server. However, in many situations, access privileges may need to be granted selectively, that is, different users should be authorized to view only a portion of the stored data. While existing solutions for enforcing authorizations in data outsourcing context in presence of honest-but-curious providers (e.g., *selective encryption* [18, 19]) have emerged, they cannot be simply applied in conjunction with the shuffle index, given the specific characteristics of the index and its access execution procedure.

In this paper, we present an approach to support access control over the shuffle index to ensure that access to the data be granted only in respect of authorizations specified by the data owner. Our approach leverages the availability of selective encryption to provide a self-enforcing layer of protection over the data themselves. To allow for authorizations enforcement while maintaining access confidentiality guarantees, our approach makes use of two shuffle indexes: a primary index, storing and providing access to selectively encrypted data, and a secondary index, enabling enforcement of access control. A preliminary version of this work appeared in [1]. Here, we extend our earlier proposal by supporting the evaluation of range queries and the definition of indexes over non-key attributes. Also, we illustrate how the proposal in [1] can be extended to manage the insertion, deletion, and update of outsourced resources as well as the grant and revoke of privileges to users. We show that our proposal correctly enforces the access control policy defined by the data owner and has limited performance and economic overhead.

The remainder of this paper is organized as follows. Section 2 summarizes the shuffle index approach. Section 3 introduces the primary and secondary index structures for access control enforcement. Sec-

tion 4 focuses on the support of range queries in presence of selective access restrictions. Section 5 illustrates an approach for the definition of primary and secondary indexes over non-key attributes. Section 6 describes the working of access operations over the primary and secondary indexes. Section 7 presents an approach for the management of data and policy updates. Section 8 discusses the correctness, security guarantees, performance advantages, and economic costs of our approach. Section 9 presents related works. Finally, Section 10 concludes the paper.

2. Shuffle Index

The *shuffle index* [7] is a dynamically allocated data structure offering access and pattern confidentiality while supporting efficient key-based data organization and retrieval. A data collection organized in a shuffle index is a set of pairs $\langle$*index_value*, *resource*$\rangle$ with *index_value* a candidate key for the collection (i.e., no two resources share the same value for *index_value*) used for index definition, and *resource* the corresponding resource associated with the index value. For simplicity, we assume the data collection to be a relational table $\mathcal{R}$ defined over a simplified schema $\mathcal{R}$(*I*,*Resource*), with *I* the indexed attribute and *Resource* the resource content. At the *abstract* level, a shuffle index for $\mathcal{R}$ over *I* is an *unchained B+-tree* (i.e., there are no links between the leaves) with fan-out F defined over attribute I, storing the tuples in $\mathcal{R}$ in its leaves. Each non-root node stores up to q ordered values $v_1, v_2, \ldots, v_q$, with $\lceil F/2 \rceil \leqslant q \leqslant F-1$, and has as many children as the number of values stored plus one. The first child of a node is the root of the subtree including all values $v < v_1$; its last child is the root of the subtree including all values $v \geqslant v_q$; its i-th child ($i = 2, \ldots, q$) is the root of the subtree including all values $v_{i-1} \leqslant v < v_i$. Actual resources are stored in the leaves of the tree in association with their index value. At the *logical* level, each node is associated with a logical identifier. Logical identifiers are used in internal nodes as pointers to their children and do not reflect the order relationship among the values stored in the nodes. At the *physical* level, each node is stored in *encrypted form* in a physical block and logical identifiers are translated into physical addresses at the storing server. For the sake of simplicity, we assume that the physical address of a block corresponds to the logical identifier of the node stored in the block. The encrypted node is obtained by encrypting the concatenation of the node identifier, its content (values and pointers to children or resources), and a randomly generated nonce (*salt*). Formally, block b storing node n is defined as $E(k, salt||id||n)$, where E is a symmetric encryption function with key k and *id* is the identifier of node n. Encryption protects the confidentiality of the content of nodes and the structure of the tree, as well as the integrity of each node and of the structure overall. Figure 1(c-e) illustrates an example of the shuffle index storing the relation in Figure 1(a), indexed according to the values of attribute *I*, at the abstract (c), logical (d), and physical (e) level. Actual tuples are stored in the leaves of the index structure, where, for simplicity, we report only the index values.

To retrieve the tuple with a given index value in the shuffle index, the tree is traversed from the root following the pointers to the children until a leaf is reached. Since the shuffle index is stored at the server in encrypted form, such a process is iterative, with the client retrieving from the server (and decrypting) one node at a time to determine the child node to be read at the next level. To protect access and pattern confidentiality, in addition to storing nodes in encrypted form at the server, the shuffle index uses the following three techniques in access execution.

- *Cover searches*: in addition to the target value, additional values, called *covers*, are requested. Covers, chosen in such a way to be indistinguishable from the target and to operate on disjoint paths in

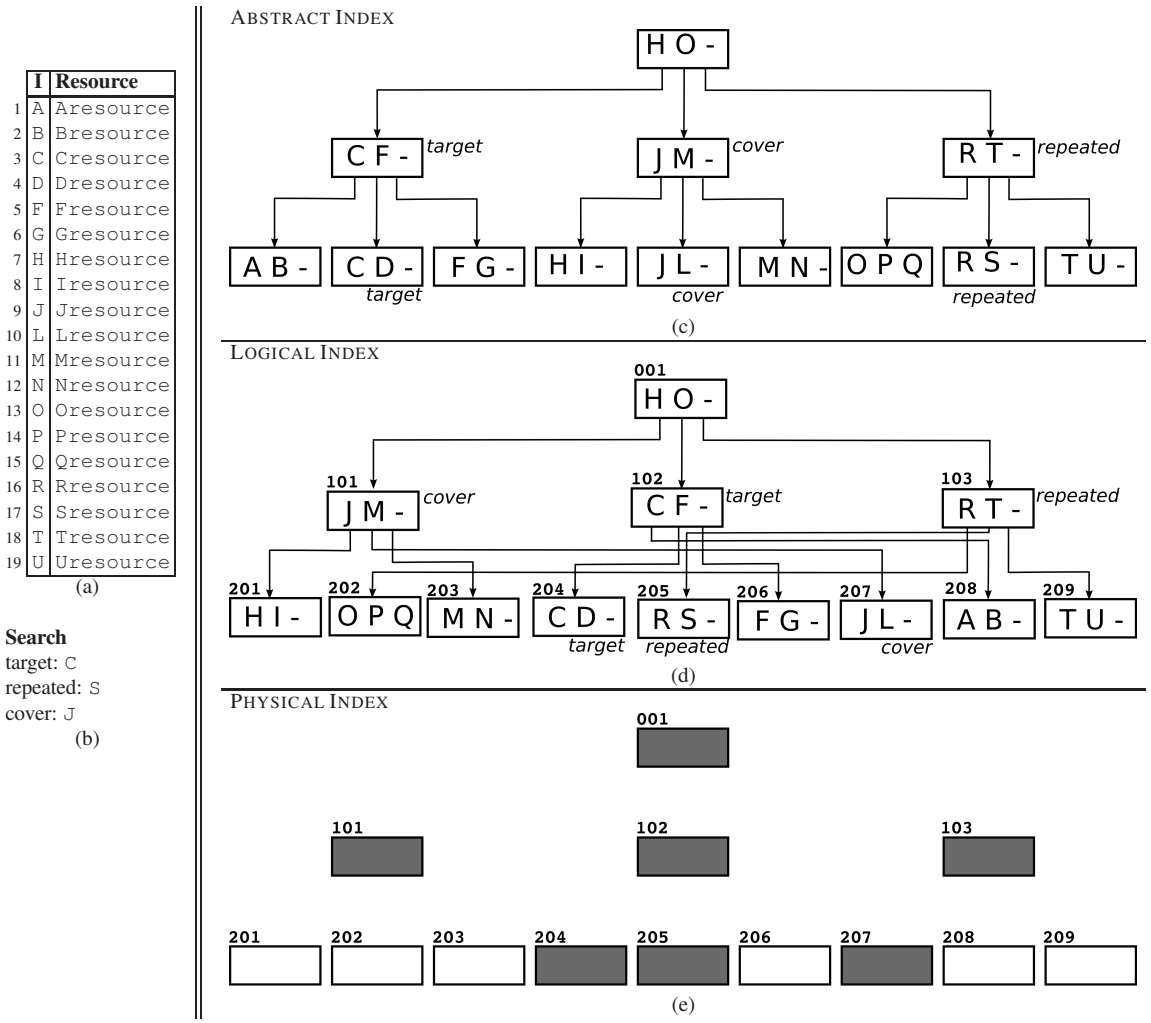

Figure 1. An example of a relation (a), an access over it (b), and of abstract (c), logical (d), and physical (e) shuffle index

the tree (also disjoint from the path of the target), provide uncertainty to the server on the actual target. If *num_cover* searches are used, the server will observe access to *num_cover*$+1$ distinct paths and corresponding leaf blocks, any of which could be the actual target.

- *Repeated access*: to avoid the server learning when two accesses refer to the same target since they would have a path in common, the shuffle index always produces such an observable by choosing, as one of the covers for an access, one of the values of the access just before it (if the current access is for the same target as the previous access, a new cover is used). In this way, the server always observes a repeated access, regardless of whether the two accesses refer to the same or to a different target.
- *Shuffling*: at every access, the nodes involved in the access are shuffled (i.e., allocated to different logical identifiers and corresponding physical blocks), re-encrypted (with a different random salt and including the new identifier of the block) and re-stored at the server. Shuffling provides dynamic reallocation of all the accessed nodes, thus destroying the otherwise static correspondence between

physical blocks and their content. This prevents the server from accumulating knowledge on the data allocation as at any access such an allocation is refreshed.

To illustrate, consider the shuffle index in Figure 1(c-e) and the search in Figure 1(b) for the tuple with index value C, assuming S as repeated access and J as fresh cover. The access entails reading (i.e., retrieving from the server) the nodes annotated in the figure, with the server only observing downloads of the corresponding encrypted blocks in Figure 1(e) but not able to learn anything on their content or role (target, repeated, cover). Shuffling could produce, after the access, a re-allocation of the accessed nodes. For instance, 205→204, 204→207, 207→205 (where X→Y denotes that the content of node X is moved to node Y).

3. Primary and Secondary Indexes for Access Control

Providing access control means enabling data owners to regulate access to their data and selectively authorize different users with different views over the data. Figure 2(a) illustrates possible authorizations on the data in Figure 1(a), considering three users u_1, u_2, and u_3. The figure reports, for each tuple r in the dataset, the corresponding $acl(r)$, that is, the set of users authorized to read it. Note that we assume access by users to be read-only, and write operations reserved to the owner. When clear from the context, with a slight abuse of notation, we will denote the access control list of a tuple r as either $acl(r)$ or $acl(r[I])$, with $r[I]$ its index value. For instance, acl(A)=$\{u_1,u_2,u_3\}$, and acl(B)=$\{u_1,u_2\}$.

Before diving into our solution, we note that there could be two natural and straightforward approaches to enforce authorizations in the shuffle index, each of which would have however limitations and drawbacks. A first natural approach would be to simply associate a key k_i with each user u_i and produce different replicas of the data. Each tuple would be replicated as many times as the number of users authorized to access it. Each copy would be encrypted with the key of the user for which it is produced. For instance, with reference to Figure 2(a) three copies of resource Aresource would be created and encrypted with keys k_1, k_2, and k_3, respectively. Different shuffle indexes would then be defined, one for each user, organizing and supporting accesses to the tuples that the user is authorized to access. Such an approach, besides bearing obvious data management problems (as replicas would need to be maintained consistent) would affect the protection offered by the shuffle index. In fact, it would organize each shuffle index only on a limited portion of the data (for each user, only those tuples that she can access, that is, less than half of the original tuples for each user in our example) with consequent limitations in the choice of covers. An alternative solution could then be to maintain the shuffle index as a single structure (so to build it on the complete dataset), and avoid replicas by producing only one encrypted copy for each tuple. Replicas can be avoided by considering different encryption keys not only for individual users but also for set of users (i.e., *acls*), with a user u_i knowing her encryption key k_i as well as those of the *acls* in which she is included. Each resource would then be encrypted only once and the encryption key with which it is encrypted known only to its authorized users. For instance, with reference to Figure 2(a), resource Aresource would be encrypted with key k_{123} known to all users while resource Bresource would be encrypted with key k_{12} known to u_1 and u_2 only. While such selective encryption correctly enforces access to the encrypted resources, it leaves the problem of ensuring protection (and controlling the possible exposure) of the index values on which the shuffle index is organized. As a matter of fact, on one hand, leaving such index values accessible to all users for traversing the tree would disclose to every user the complete set of index values, even those of the tuples she is not authorized to access.

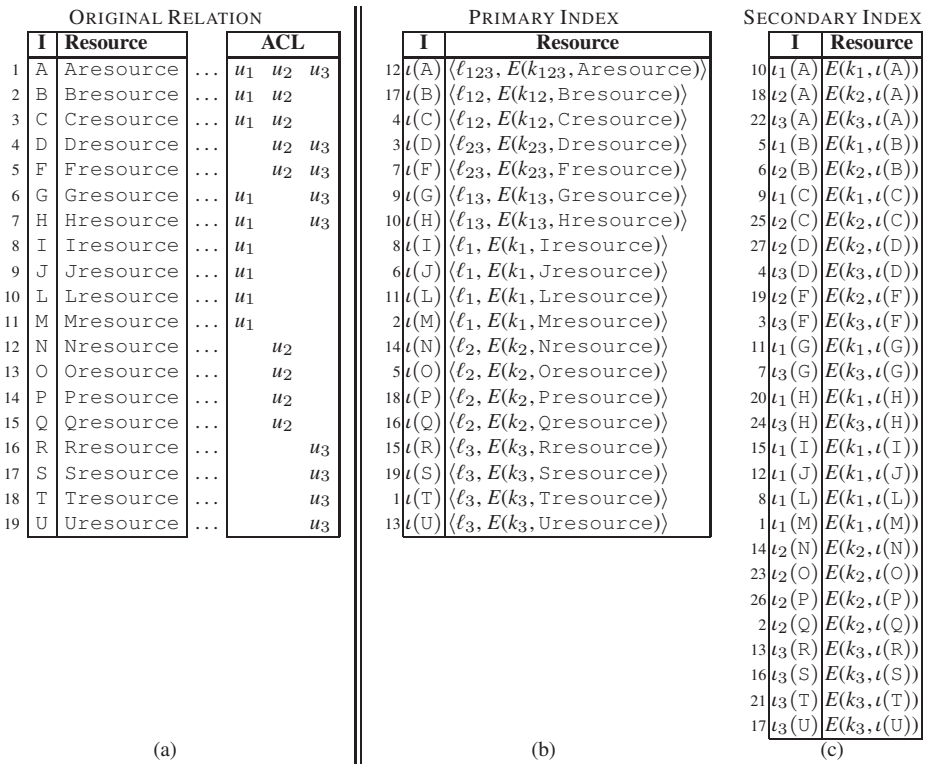

Figure 2. Relation of Figure 1(a) with *acls* associated with its resources (a), relation for the primary index (b), and relation for the secondary index (c)

On the other hand, such index values cannot be encrypted with the same encryption key used for the corresponding resources, as otherwise the ability to traverse the tree by users would be affected.

Starting from these observations, we build our approach providing selective encryption while protecting index values themselves against unauthorized users without affecting their ability to retrieve those tuples they are authorized to access. Our approach is based on the definition of two different indexes. A *primary index*, defined over an encoded version of the original index values, and a *secondary index*, providing a mapping enabling users to retrieve the value to look for in the primary index. Both indexes make use of an encoding of the values to be indexed to make them intelligible only to authorized users. We then start by defining an encoding function as follows.

Definition 3.1 (Encoding Function)**.** *Let $\mathcal{R}$(I,Resource) be a relation with I defined over domain $\mathcal{D}$. A function $\iota : \mathcal{D} \rightarrow \mathcal{E}$ is an* encoding function *for I iff ι is:* i) *non-invertible;* ii) *non order-preserving; and* iii) *injective.*

Intuitively, an encoding function maps the domain of index values I onto another domain of values $\mathcal{E}$, avoiding collisions (i.e., $\forall v_x, v_y \in I$ with $v_x \neq v_y$, $\iota(v_x) \neq \iota(v_y)$), and in such a way that the original ordering among values is destroyed. Also, non-invertibility ensures the impossibility of deriving the inverse function (from encoded to original values). For instance, an encoding function can be realized as a keyed cryptographic hash function operating on the domain of attribute I.

The second building block of our solution is the application of selective encryption, namely the encryption of each resource with a key known only to authorized users. To apply selective encryption, we then define a set of keys for the encryption policy as follows.

Definition 3.2 (Encryption Policy Keys)**.** *Let $\mathcal{R}(I,Resource)$ be a relation, $\mathcal{U}$ be a set of users, and, $\forall r \in \mathcal{R}$, $acl(r) \subseteq \mathcal{U}$ be the acl of r. The set $\mathcal{K}$ of* encryption policy keys *for $\mathcal{R}$ is a set $\mathcal{K} = \{k_i \mid u_i \in \mathcal{U}\} \cup \{k_{i_1,\ldots,i_n} \mid \exists r \in \mathcal{R}$, with $acl(r)=\{u_{i_1},\ldots,u_{i_n}\}\}$ of encryption keys. Each key $k_X \in \mathcal{K}$ has a* public label ℓ_X. *Each user $u_i \in \mathcal{U}$ knows the set $\mathcal{K}_i = \{k_i\} \cup \{k_X \mid k_X \in \mathcal{K} \wedge i \in X\}$ of keys.*

Definition 3.2 defines all the keys needed (and the knowledge of users on them) to apply selective encryption, meaning to encrypt the data selectively so that only authorized users can access them while optimizing key management and avoiding data replication. The public label associated with a key allows referring to the key without disclosing its value. Note that knowledge by a user of all the keys of the access control lists to which she belongs does not require direct distribution of the keys to the user, since hierarchical organization of keys and use of publicly available tokens enabling key derivation can provide such a knowledge to the user [19].

We are now ready to define the first index used by our approach. This index, called *primary*, is the one storing the actual data on which accesses should operate (i.e., tuples in $\mathcal{R}$). To provide selective access as well as enable all users to traverse the index without leaking to them information (index values and resources) they are not authorized to access, the index combines value encoding and selective encryption. Formally, the primary index is defined as follows.

Definition 3.3 (Primary Index – Data)**.** *Let $\mathcal{R}(I,Resource)$ be a relation with indexing attribute I, ι be an encoding function for I, and $\mathcal{K}$ be the set of encryption policy keys for $\mathcal{R}$. A* primary index *for $\mathcal{R}$ over I is a shuffle index for relation $\mathcal{P}(I,Resource)$ over I having a tuple p for each tuple $r \in \mathcal{R}$ such that $p[I] = \iota(r[I])$ and $p[Resource] = \langle \ell_{i_1,\ldots,i_n}, E(k_{i_1,\ldots,i_n}, r[Resource])\rangle$, with E a symmetric encryption function, $acl(r) = \{u_{i_1},\ldots,u_{i_n}\}$, and $k_{i_1,\ldots,i_n} \in \mathcal{K}$.*

The primary index stores original data in encrypted form, encrypting each tuple with the key corresponding to its *acl* (i.e., known only to the users authorized to read the tuple). The inclusion in $p[Resource]$ of the label enables authorized users to know the key to be used for the decryption of the resource. The primary index is built on encoded values computable only by the data owner. For instance, the encoding function can be implemented through a keyed cryptographic hash function, using a key k_o known only to the data owner (i.e., the encoded value $\iota(v)$ for a tuple r with index value v can be computed as $h(k_o,v)$). Note that, although each resource singularly taken appears encrypted in the leaves of the primary index, all the nodes are (also) encrypted with a key k known to every user in the system. This second encryption layer is necessary to enable shuffling (Section 2).

Building the primary index on the encoded values provides protection of the original index values and of their order relationship against users and storing server observing the index. In fact, the non-invertibility of the encoding function ensures that the encoded values do not leak any information on the original values. Also, since the encoding function is non order-preserving, the order relationship between encoded value does not leak anything on the order relationship among the original values.

Figure 2(b) illustrates a primary index $\mathcal{P}$ for our running example. The ordering among the encoded values is reported with numbers on the left of the table. Figure 3 illustrates the tree structure for such primary index. Note how the different order among the values to be indexed causes a different content

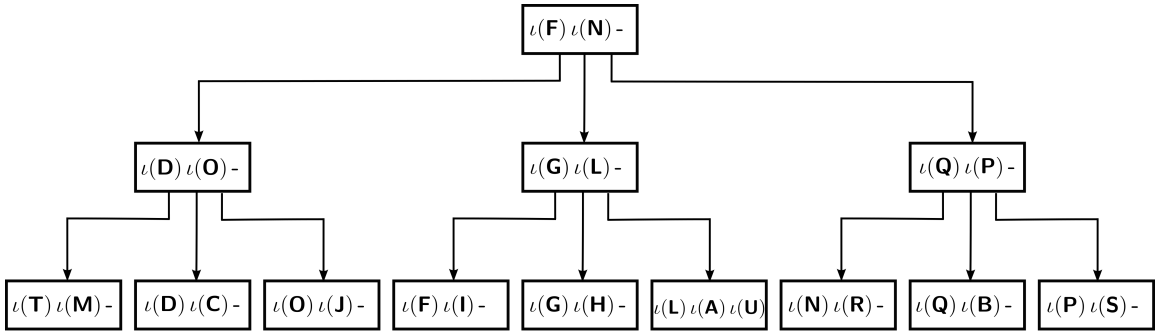

Figure 3. Primary shuffle index for the relation in Figure 2(b)

within the leaves and a different ordering among them with respect to the shuffle index in Figure 1(a) built over the original (non-encoded) index values.

While the index on the encoded values provides the ability to traverse the tree to look for the resource associated with an encoded value, to retrieve a given resource (i.e., the resource corresponding to an original value for the indexing attribute) one would need to know the encoding of such value. For instance, resource `Aresource` would be stored in association with index value $\iota(\mathtt{A})$. The encoding (i.e., the fact that $\iota(\mathtt{A})$ corresponds to `A`) is however known only to the data owner.

The second index of our approach allows the data owner to selectively disclose to users the mapping of encoding ι, releasing to every user the mapping for (*all and only*) those values she is authorized to access. Such a mapping is provided to each user u_i by encrypting all encoded values accessible by u_i with her key k_i (so to make them non intelligible to other users and to the server) and by using a user-based encoding function ι_i for indexing, so to provide a distinct mapping for every user u_i, which can be computed only by u_i and by the data owner. The second index of our approach is therefore a *secondary* index providing user-based mapping as follows.

Definition 3.4 (Secondary Index – User-based Mapping)**.** *Let $\mathcal{R}$(I,Resource) be a relation with indexing attribute I, ι be the encoding function used in the primary index $\mathcal{P}$, $\mathcal{U} = \{u_1, \ldots, u_n\}$ be a set of users with encoding function $\iota_i, i = 1, \ldots, n$, and $\mathcal{K}$ be the set of encryption policy keys for $\mathcal{R}$. A* secondary index *for $\mathcal{R}$ and $\mathcal{P}$ is a shuffle index for relation $\mathcal{S}$(I,Resource) over I having a tuple s for each pair $\langle r,u_i\rangle$, with $r\in\mathcal{R}$ and $u_i\in acl(r)$, such that $s[I]= \iota_i(r[I])$ and $s[Resource] = E(k_i, \iota(r[I]))$, with E a symmetric encryption function and $k_i \in \mathcal{K}$.*

For instance, the encoding function of each user u_i can be implemented as a cryptographic hash function, using a key k_i known to user u_i only (i.e., $\iota_i(v)$=$h(k_i,v)$). Figure 2(c) illustrates a secondary index for our running example. Again, the number on the left of the table is the ordering among the index values of the secondary index. Note that, once again, the encoding does not convey any information on the ordering of the original index values. Also, while the secondary index has a larger number of tuples than the original index because the encoding of an original index value is encrypted as many times as the number of users who can access it, the index is very slim as the resources are simply the encryption, with the key of a user, of the owner encoding. For instance, in our example, there are three instances of $\iota(\mathtt{A})$. Figure 4 illustrates the tree structure for the secondary index in Figure 2(c), where, for simplicity, we maintain the same topology as the primary index. However, the structure of the secondary index is

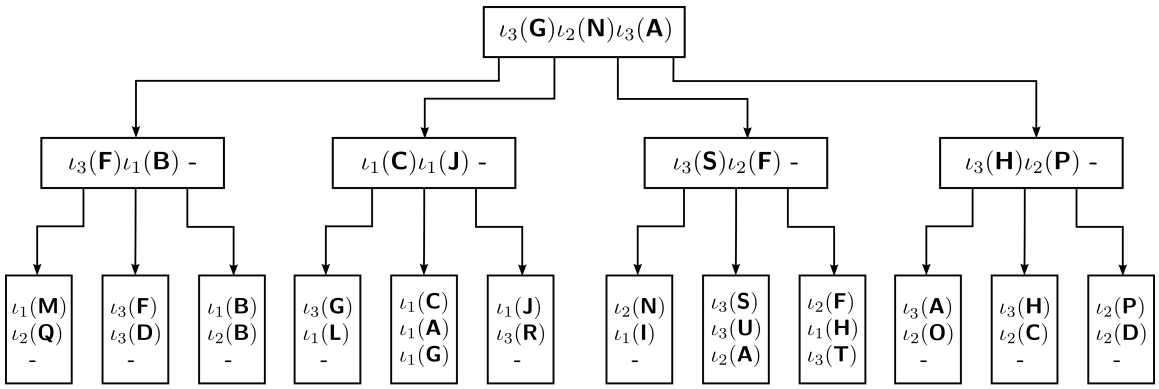

Figure 4. Secondary shuffle index for the relation in Figure 2(c)

independent from the structure of the primary index, meaning that they may have different fan-out and height.

The property of the encoding function of destroying the ordering among original index values is particularly important to guarantee protection. In fact, users will know all encoded values computed by the data owner (i.e., the co-domain of function ι), but will know the actual mapping (i.e., the actual value v corresponding to $\iota(v)$) only for the values they are authorized to access. Figure 5(a-b) illustrates a possible logical organization for the primary and secondary index of our example for user u_1, where, for simplicity of illustration, we assume the logical organization to reflect (at this initial time) the abstract organization of the tree. We distinguish blocks of the primary and secondary index by adding prefix P and S, respectively, to their identifiers. The coloring represents the visibility of users u_1. Encoded values with grey background are those that remain non intelligible to u_1 as they are encoded with the function of another user (for the secondary index) or their owner encoding is not disclosed to u_1 (for the primary index).

Since encoding does not preserve ordering, encoded values non intelligible to a user will remain protected, as no inference can be drawn on them from their presence or order relationships with respect to other encoded values which are intelligible to the user. For instance, consider the primary index in Figure 5(b). User u_1, being authorized for `B` will know that $\iota(\texttt{B})$ is the corresponding encoding. At the same time, however, $\iota(\texttt{Q})$, stored in the same node, remains non intelligible to her. User u_1 simply observes the presence of another encoded value but will be able to infer neither its corresponding original value nor its order relationship with respect to `B`.

4. Support for Range Queries

Our primary and secondary indexes enable authorized users to efficiently evaluate equality queries characterized by a condition of the form $I = v$, with v a value in the domain of I that the user is authorized to access. However, users may also need to evaluate *range queries* aimed to retrieve all resources whose index value is between a lower and an upper bound (i.e., within a range [*lower_bound*, *upper_bound*]). In absence of access control restrictions, the shuffle index supports range queries by translating each of them into an equivalent set of equality queries [9]. The idea is to use the logical organization of the data in the shuffle index. By construction, the first value stored in each leaf node, apart from the first leaf,

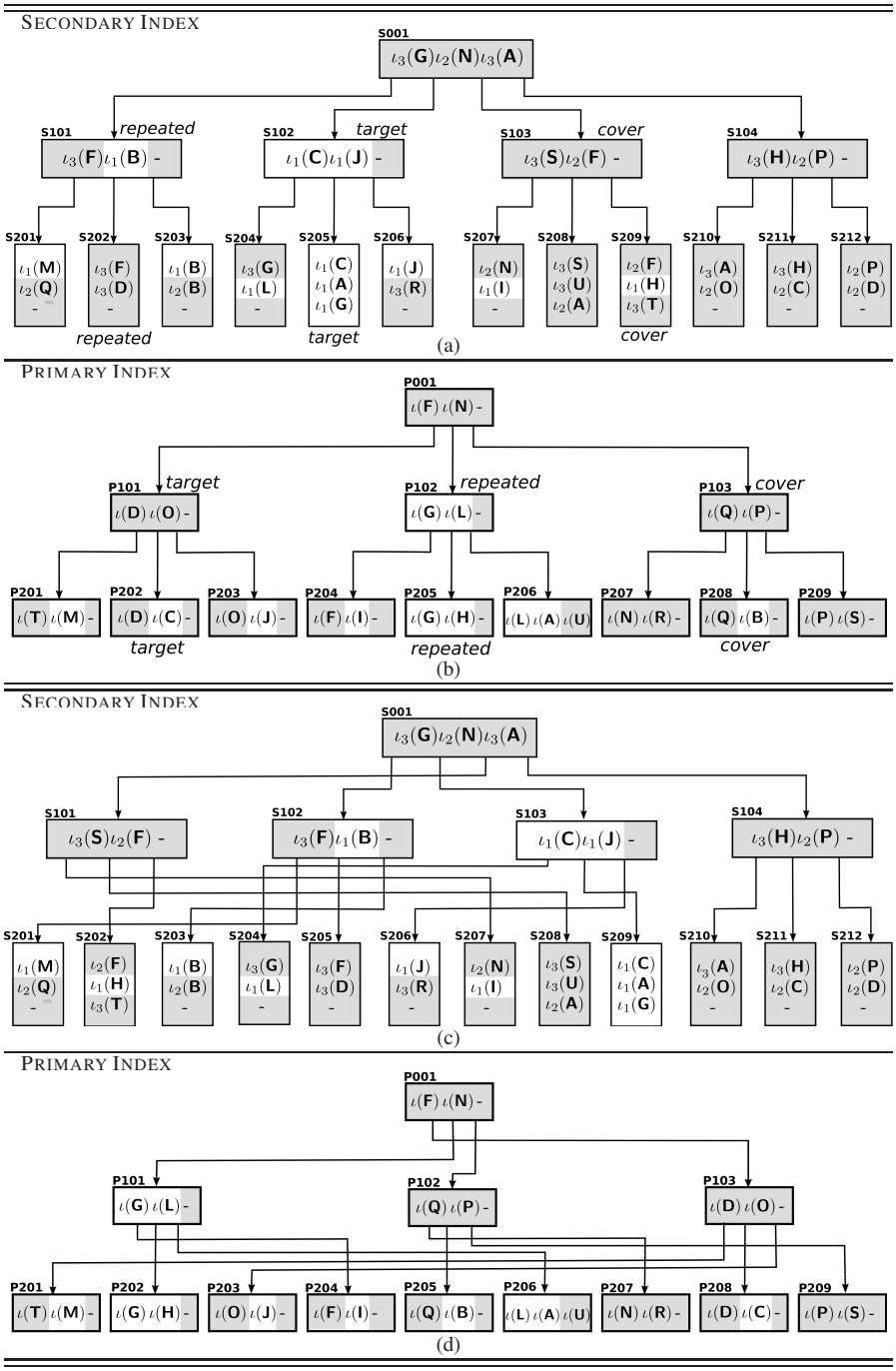

Figure 5. Secondary and primary index before (a-b) and after (c-d) the access by u_1 over C. Secondary index: i) cover: $\iota_2(\mathtt{F})$, ii) repeated access: [S001,S101,S202], iii) shuffling: S101→S102, S102→S103, S103→S101, S202→S205, S205→S209, S209→S202. Primary index: i) cover: $\iota(\mathtt{Q})$, ii) repeated access: [P001,P102,P205], iii) shuffling: P101→P103, P102→P101, P103→P102, P202→P208, P205→P202, P208→P205. The gray background denotes encoded values non intelligible to u_1

is also represented in an internal node. In particular, considering two contiguous leaf nodes, n_i and n_j, the smallest value in n_j is also represented in the deepest common ancestor of n_i and n_j For instance, considering leaf nodes [F,G,-] and [H,I,-] in the shuffle index in Figure 1(c), value H is also stored in the root node. While visiting the shuffle index searching for a value in n_i, the user can identify and keep track of the first value in n_j (i.e., for each node along the path to the target, the process keeps track of the successor - if any - of the target value). A range query is then evaluated by first searching for *I=lower_bound*, to retrieve the first leaf node in the range as well as the next index value in the sequence. The evaluation of range condition [*lower_bound*, *upper_bound*] iteratively visits contiguous leaves (in the abstract index) until the range has been completely covered.

This approach cannot be adopted when accesses to the outsourced data are regulated by an access control policy since the encoding functions adopted in the definition of the primary index and of the secondary index is non-order preserving (Definition 3.1), meaning that contiguous leaves in the abstract primary and secondary indexes do not store resources with contiguous original index values. For instance, in the primary index in Figure 3, A appears in a leaf together with L and U, while B is in a different (not contiguous) leaf.

Consider a user u_i who is authorized to access a sequence $\langle v_1, \ldots, v_l \rangle$ of index values, with $v_1 < v_2 < \ldots < v_l$. To efficiently support the evaluation of range queries also in presence of access control restrictions, each value $\iota_i(v_j)$ in the leaves of the secondary index, with $j = 1, \ldots, l$, is coupled with the next index value v_{j+1} in the sequence. As an example, considering the secondary index in Figure 4, ι_3(A) is coupled with D since, according to the access control policy in Figure 2(a), user u_3 can access A followed by D. Intuitively, our approach provides the same information as the linked list among the leaves of a $B+$-tree, but it reveals to each user the relative order among only the index values that she is authorized to access. Given a resource r with $r[I] = v_j$ that u_i is authorized to access, the tuple s in the secondary index for the pair $\langle r,u_i \rangle$ has then content: $s[I] = \iota_i(v_j)$ and $s[Resource] = E(k_i, \iota(v_j)||v_{j+1})$, as formally defined in the following.

Definition 4.1 (Secondary Index - Chain)**.** *Let $\mathcal{R}$(I,Resource) be a relation with indexing attribute I, ι be the data owner encoding function used in the primary index $\mathcal{P}$, $\mathcal{U} = \{u_1, \ldots, u_n\}$ be a set of users with encoding function $\iota_i, i = 1, \ldots, n$, $\langle r_{i_1}, \ldots, r_{i_m} \rangle \subseteq \mathcal{R}$ be the set of resources that u_i can access, in ascending ordered by I, and $\mathcal{K}$ be the set of encryption policy keys for $\mathcal{R}$. A* secondary index *for $\mathcal{R}$ and $\mathcal{P}$ is a shuffle index for relation $\mathcal{S}$(I,Resource) over I having a tuple s for each pair $\langle r_{i_j}, u_i \rangle$, with $r_{i_j} \in \mathcal{R}$ and $u_i \in acl(r_{i_j})$, such that $s[I] = \iota_i(r_{i_j}[I])$ and $s[Resource] = E(k_i, \iota(r_{i_j}[I]) \;||\; r_{i_{j+1}}[I])$, with E a symmetric encryption function, $k_i \in \mathcal{K}$, and $r_{i_{j+1}}[I]$ = – when j=m.*

According to this definition, the next value $r_{i_{j+1}}[I]$ in the authorized sequence for user u_i is not stored in its encoded form (i.e., we store the original index values) because in this way user u_i can verify whether the evaluation of a range query must be terminated, which happens when $r_{i_{j+1}}[I]$ is greater than *upper_bound* or is equal to –. Also, user u_i can always compute the user encoding $\iota_i(r_{i_{j+1}}[I])$ and retrieve from the secondary index the owner encoding $\iota(r_{i_{j+1}}[I])$. Figure 6 illustrates the relation of the secondary index in Figure 2(c), extended to support range queries. The evaluation of a range query then starts with the execution of an equality query with condition *I=lower_bound*, thus retrieving the resource associated with the *lower_bound* index value as well as the next index value in the sequence that the user can access. The next index value is iteratively used in an equality query to retrieve the corresponding resource and the subsequent index value. The process terminates when the whole range has been covered.

Note that we materialize the chain between encoded values in the secondary index and not in the primary index because the materialization of the sequence of original index values in the primary index

	I	Resource
10	ι_1(A)	$E(k_1, \iota$(A)‖B)
18	ι_2(A)	$E(k_2, \iota$(A)‖B)
22	ι_3(A)	$E(k_3, \iota$(A)‖D)
5	ι_1(B)	$E(k_1, \iota$(B)‖C)
6	ι_2(B)	$E(k_2, \iota$(B)‖C)
9	ι_1(C)	$E(k_1, \iota$(C)‖G)
25	ι_2(C)	$E(k_2, \iota$(C)‖G)
27	ι_2(D)	$E(k_2, \iota$(D)‖F)
4	ι_3(D)	$E(k_3, \iota$(D)‖F)
19	ι_2(F)	$E(k_2, \iota$(F)‖N)
3	ι_3(F)	$E(k_3, \iota$(F)‖G)
11	ι_1(G)	$E(k_1, \iota$(G)‖H)
7	ι_3(G)	$E(k_3, \iota$(G)‖H)
20	ι_1(H)	$E(k_1, \iota$(H)‖I)
24	ι_3(H)	$E(k_3, \iota$(H)‖R)
15	ι_1(I)	$E(k_1, \iota$(I)‖J)
12	ι_1(J)	$E(k_1, \iota$(J)‖L)
8	ι_1(L)	$E(k_1, \iota$(L)‖M)
1	ι_1(M)	$E(k_1, \iota$(M)‖−)
14	ι_2(N)	$E(k_2, \iota$(N)‖O)
23	ι_2(O)	$E(k_2, \iota$(O)‖P)
26	ι_2(P)	$E(k_2, \iota$(P)‖Q)
2	ι_2(Q)	$E(k_2, \iota$(Q)‖−)
13	ι_3(R)	$E(k_3, \iota$(R)‖S)
16	ι_3(S)	$E(k_3, \iota$(S)‖T)
21	ι_3(T)	$E(k_3, \iota$(T)‖U)
17	ι_3(U)	$E(k_3, \iota$(U)‖−)

Figure 6. Relation of the secondary index in Figure 2(c), extended to support range queries

would reveal to users the relative order among some of the encoded values, possibly also among values that they cannot access. This clearly nullifies the advantage of using a non-order preserving encoding function.

5. Indexes over Non-Key Attributes

Being the shuffle index structure (and hence also our primary and secondary indexes) a $B+$-tree, it must be defined over a candidate key I for relation $\mathcal{R}$, meaning that $\mathcal{R}$ does not include two distinct resources with the same index value. This restriction, however, can be relaxed, and our indexes can be constructed over a *non-key attribute* (i.e., more resources may have the same index value). In this case, a search for a value v of the index attribute should return *all* (and only) the resources having index value equal to v that the requesting user is authorized to access. Also, the result returned to the user should not reveal anything about the existence of other resources with the same index value but that the user cannot access. As an example, consider user u_1 and the relation in Figure 7(a) where attribute I is a non-key attribute. According to the access control policy in Figure 7(b), a search by user u_1 for value `a` of attribute I should return `Aresource`, `Bresource`, `Gresource`, and `Hresource`.

Our idea for using the shuffle index over a non-key attribute consists in making different the multiple occurrences of the same index values. In other words, we propose to associate different encodings (for both the primary and secondary indexes) with different occurrences of the same original index value. Intuitively, such encodings can be obtained by combining each occurrence with a different random salt. The same random salts can instead be used with different index values since being the index values

ORIGINAL RELATION

	K	I	Resource		ACL		
1	A	a	Aresource	...	u_1	u_2	u_3
2	B	a	Bresource	...	u_1	u_2	
3	C	b	Cresource	...	u_1	u_2	
4	D	b	Dresource	...		u_2	u_3
5	F	c	Fresource	...		u_2	u_3
6	G	a	Gresource	...	u_1		u_3
7	H	a	Hresource	...	u_1		u_3
8	I	b	Iresource	...	u_1		
9	J	c	Jresource	...	u_1		
10	L	c	Lresource	...	u_1		
11	M	d	Mresource	...	u_1		
12	N	a	Nresource	...		u_2	
13	O	a	Oresource	...		u_2	
14	P	b	Presource	...		u_2	
15	Q	d	Qresource	...		u_2	
16	R	a	Rresource	...			u_3
17	S	b	Sresource	...			u_3
18	T	c	Tresource	...			u_3
19	U	d	Uresource	...			u_3

(a)

PRIMARY INDEX

	I	Resource
10	$\iota(\mathtt{a})$	$\langle \ell_{123}, E(k_{123}, \mathtt{Aresource})\rangle$
14	$\iota(\mathtt{a} \oplus \mathfrak{p}_1)$	$\langle \ell_{12}, E(k_{12}, \mathtt{Bresource})\rangle$
3	$\iota(\mathtt{b})$	$\langle \ell_{12}, E(k_{12}, \mathtt{Cresource})\rangle$
11	$\iota(\mathtt{b} \oplus \mathfrak{p}_1)$	$\langle \ell_{23}, E(k_{23}, \mathtt{Dresource})\rangle$
4	$\iota(\mathtt{c})$	$\langle \ell_{23}, E(k_{23}, \mathtt{Fresource})\rangle$
13	$\iota(\mathtt{a} \oplus \mathfrak{p}_2)$	$\langle \ell_{13}, E(k_{13}, \mathtt{Gresource})\rangle$
12	$\iota(\mathtt{a} \oplus \mathfrak{p}_3)$	$\langle \ell_{13}, E(k_{13}, \mathtt{Hresource})\rangle$
19	$\iota(\mathtt{b} \oplus \mathfrak{p}_2)$	$\langle \ell_1, E(k_1, \mathtt{Iresource})\rangle$
1	$\iota(\mathtt{c} \oplus \mathfrak{p}_1)$	$\langle \ell_1, E(k_1, \mathtt{Jresource})\rangle$
15	$\iota(\mathtt{c} \oplus \mathfrak{p}_2)$	$\langle \ell_1, E(k_1, \mathtt{Lresource})\rangle$
5	$\iota(\mathtt{d})$	$\langle \ell_1, E(k_1, \mathtt{Mresource})\rangle$
6	$\iota(\mathtt{a} \oplus \mathfrak{p}_4)$	$\langle \ell_2, E(k_2, \mathtt{Nresource})\rangle$
17	$\iota(\mathtt{a} \oplus \mathfrak{p}_5)$	$\langle \ell_2, E(k_2, \mathtt{Oresource})\rangle$
2	$\iota(\mathtt{b} \oplus \mathfrak{p}_3)$	$\langle \ell_2, E(k_2, \mathtt{Presource})\rangle$
9	$\iota(\mathtt{d} \oplus \mathfrak{p}_1)$	$\langle \ell_2, E(k_2, \mathtt{Qresource})\rangle$
16	$\iota(\mathtt{a} \oplus \mathfrak{p}_6)$	$\langle \ell_3, E(k_3, \mathtt{Rresource})\rangle$
7	$\iota(\mathtt{b} \oplus \mathfrak{p}_4)$	$\langle \ell_3, E(k_3, \mathtt{Sresource})\rangle$
18	$\iota(\mathtt{c} \oplus \mathfrak{p}_3)$	$\langle \ell_3, E(k_3, \mathtt{Tresource})\rangle$
8	$\iota(\mathtt{d} \oplus \mathfrak{p}_2)$	$\langle \ell_3, E(k_3, \mathtt{Uresource})\rangle$

(b)

SECONDARY INDEX

	I	Resource
18	$\iota_1(\mathtt{a})$	$E(k_1, \iota(\mathtt{a})\|\|(\mathtt{a},\mathfrak{s}_1^1))$
7	$\iota_2(\mathtt{a})$	$E(k_2, \iota(\mathtt{a})\|\|(\mathtt{a},\mathfrak{s}_1^2))$
16	$\iota_3(\mathtt{a})$	$E(k_3, \iota(\mathtt{a})\|\|(\mathtt{a},\mathfrak{s}_1^3))$
25	$\iota_1(\mathtt{a},\mathfrak{s}_1^1)$	$E(k_1, \iota(\mathtt{a} \oplus \mathfrak{p}_1)\|\|(\mathtt{a},\mathfrak{s}_2^1))$
17	$\iota_2(\mathtt{a},\mathfrak{s}_1^2)$	$E(k_2, \iota(\mathtt{a} \oplus \mathfrak{p}_1)\|\|(\mathtt{a},\mathfrak{s}_2^2))$
6	$\iota_1(\mathtt{b})$	$E(k_1, \iota(\mathtt{b})\|\|(\mathtt{b},\mathfrak{s}_1^1))$
26	$\iota_2(\mathtt{b})$	$E(k_2, \iota(\mathtt{b})\|\|(\mathtt{b},\mathfrak{s}_1^2))$
15	$\iota_2(\mathtt{b},\mathfrak{s}_1^2)$	$E(k_2, \iota(\mathtt{b} \oplus \mathfrak{p}_1)\|\|(\mathtt{b},\mathfrak{s}_2^2))$
19	$\iota_3(\mathtt{b})$	$E(k_3, \iota(\mathtt{b})\|\|(\mathtt{b},\mathfrak{s}_1^3))$
1	$\iota_2(\mathtt{c})$	$E(k_2, \iota(\mathtt{c})\|\|\text{-})$
24	$\iota_3(\mathtt{c})$	$E(k_3, \iota(\mathtt{c})\|\|(\mathtt{c},\mathfrak{s}_1^3))$
8	$\iota_1(\mathtt{a},\mathfrak{s}_2^1)$	$E(k_1, \iota(\mathtt{a} \oplus \mathfrak{p}_2)\|\|(\mathtt{a},\mathfrak{s}_1^1))$
14	$\iota_3(\mathtt{a},\mathfrak{s}_1^3)$	$E(k_3, \iota(\mathtt{a} \oplus \mathfrak{p}_2)\|\|(\mathtt{a},\mathfrak{s}_2^3))$
2	$\iota_1(\mathtt{a},\mathfrak{s}_2^1)$	$E(k_1, \iota(\mathtt{a} \oplus \mathfrak{p}_3)\|\|\text{-})$
13	$\iota_3(\mathtt{a},\mathfrak{s}_2^3)$	$E(k_3, \iota(\mathtt{a} \oplus \mathfrak{p}_3)\|\|(\mathtt{a},\mathfrak{s}_3^3))$
20	$\iota_1(\mathtt{b},\mathfrak{s}_1^1)$	$E(k_1, \iota(\mathtt{b} \oplus \mathfrak{p}_2)\|\|(\mathtt{a},\mathfrak{s}_3^1))$
4	$\iota_1(\mathtt{c})$	$E(k_1, \iota(\mathtt{c} \oplus \mathfrak{p}_1)\|\|(\mathtt{c},\mathfrak{s}_1^1))$
21	$\iota_1(\mathtt{c},\mathfrak{s}_1^1)$	$E(k_1, \iota(\mathtt{c} \oplus \mathfrak{p}_2)\|\|\text{-})$
9	$\iota_1(\mathtt{d})$	$E(k_1, \iota(\mathtt{d})\|\|\text{-})$
3	$\iota_2(\mathtt{a},\mathfrak{s}_2^2)$	$E(k_2, \iota(\mathtt{a} \oplus \mathfrak{p}_4)\|\|(\mathtt{a},\mathfrak{s}_3^2))$
12	$\iota_2(\mathtt{a},\mathfrak{s}_3^2)$	$E(k_2, \iota(\mathtt{a} \oplus \mathfrak{p}_5)\|\|\text{-})$
23	$\iota_2(\mathtt{b},\mathfrak{s}_2^2)$	$E(k_2, \iota(\mathtt{b} \oplus \mathfrak{p}_3)\|\|\text{-})$
10	$\iota_2(\mathtt{d})$	$E(k_2, \iota(\mathtt{d} \oplus \mathfrak{p}_1)\|\|\text{-})$
27	$\iota_3(\mathtt{a},\mathfrak{s}_2^3)$	$E(k_3, \iota(\mathtt{a} \oplus \mathfrak{p}_6)\|\|\text{-})$
5	$\iota_3(\mathtt{b},\mathfrak{s}_1^3)$	$E(k_3, \iota(\mathtt{b} \oplus \mathfrak{p}_4)\|\|\text{-})$
22	$\iota_3(\mathtt{c},\mathfrak{s}_1^3)$	$E(k_3, \iota(\mathtt{c} \oplus \mathfrak{p}_3)\|\|\text{-})$
11	$\iota_3(\mathtt{d})$	$E(k_3, \iota(\mathtt{d} \oplus \mathfrak{p}_2)\|\|\text{-})$

(c)

Figure 7. Relation of Figure 1(a) with non-key attribute I and *acl*s associated with its resources (a), relation for the primary index (b), and relation for the secondary index (c) defined over I

different, also their combination with salts already used for other index values would be in any case different. Salts are associated with resources through a *salt function* that is formally defined as follows.

Definition 5.1 (Salt function)**.** *Let $\mathcal{R}(I,Resource)$ be a relation with a non-key indexing attribute I. A* salt function *is a function $\mathfrak{S} : \mathcal{R} \rightarrow$* Salt*, with* Salt *a set of salts generated through a pseudo-random function, that associates a random salt in* Salt *with each resource r in $\mathcal{R}$ in such a way that $\forall r_j, r_l \in \mathcal{R}$, with $r_j[I] = r_l[I]$, $\mathfrak{S}(r_j) \neq \mathfrak{S}(r_l)$.*

Given a salt function $\mathfrak{S}$, the encoded value of the non-key index attribute value of a resource r is then computed by applying the encoding function ι on the combination (e.g., xor) between the original index value $r[I]$ and the associated salt $\mathfrak{S}(r)$ (i.e., $\iota(r[I] \oplus \mathfrak{S}(r))$). For confidentiality reasons, the pseudo-random generation function at the basis of the definition of the salt function (Definition 5.1) is known to the data owner only. This guarantees that a user cannot search for (or infer the existence of) a resource she is not authorized to access since she cannot reconstruct the sequence of salts used by the data owner. Resources having the same original index value are then stored and organized in the primary index as resources having different index values (Definition 3.3). In this way, neither the server nor non-authorized users can infer the presence of multiple resources with the same original index value and their number. Figure 7(b) illustrates the primary index $\mathcal{P}$ defined over attribute I for the relation in Figure 7(a), where $\mathfrak{p}_1, \ldots, \mathfrak{p}_6$ is the sequence of salts used in the encoding of the index values. Note also that for each value v of the original non-key indexing attribute, the encoded value for one of the

occurrences of v could be computed without combining it with a salt. In fact, the corresponding encoded value would be different from all the other encoded values of v. In the figure, we assume that the first occurrence of each index value is not combined with any salt.

Similarly to the primary index, also the construction of the secondary index needs to be revised due to the presence of multiple occurrences of the same index values. Again, such multiple occurrences are made distinguishable by combining them with different salts. Like for the primary index, the multiple occurrences of the same index value are combined with different salts, while different index values can be combined with salts already used for other index values. Salts are generated by the data owner with a salt function that *may* or *may not* be known to the users.

If the salt function is unknown to the users, we need a solution for allowing them to retrieve from the secondary index all tuples related to the resources with the same index value that she can access. Our approach consists in storing (in encrypted form) the sequence of salts adopted for computing the encodings of the multiple occurrences of the same index value. We assume that the different occurrences are ordered (note that any arbitrary order would be fine) and that the encoded value of the first occurrence is computed by applying only the encoding function of the users. This encoded value can be computed by the authorized users and allows them to retrieve from the secondary index the first tuple related to the target index value. The content of such tuple includes both the encoded value to be searched in the primary index and the salt used to compute the encoding of the next occurrence of the target index value to be searched in the secondary index. The definition of the secondary index is then slightly adjusted as follows.

Definition 5.2 (Secondary Index – Non-Key Attribute)**.** *Let $\mathcal{R}$(I,Resource) be a relation with a non-key indexing attribute I, ι be the data owner encoding function used in the primary index $\mathcal{P}$, $\mathcal{U} = \{u_1, \ldots, u_n\}$ be a set of users with encoding function $\iota_i, i = 1, \ldots, n$, $\langle r_{i_1}, \ldots, r_{i_m} \rangle \subseteq \mathcal{R}$ be the set of resources that u_i can access, with $r_{i_1}[I] = r_{i_2}[I] = \ldots = r_{i_m}[I]$, $\mathfrak{S}_p$ and $\mathfrak{S}_s$ be two salt functions used for the primary and secondary index, respectively, and $\mathcal{K}$ be the set of encryption policy keys for $\mathcal{R}$. A* secondary index *for $\mathcal{R}$ and $\mathcal{P}$ is a shuffle index for relation $\mathcal{S}$(I,Resource) over I having a tuple s for each pair $\langle r_{i_j}, u_i \rangle$, with $r_{i_j} \in \mathcal{R}$ and $u_i \in acl(r_{i_j})$, such that $s[I] = \iota_i(r_{i_j}[I] \oplus \mathfrak{S}_s(r_{i_j}))$ and $s[Resource] = E(k_i, \iota(r_{i_j}[I] \oplus \mathfrak{S}_p(r_{i_j})) \;||\; next)$ with $next = \langle r_{i_j}[I], \mathfrak{S}_s(r_{i_{j+1}}) \rangle$ if $j + 1 < m$; –, otherwise.*

As an example, consider the secondary index in Figure 7(c) defined over attribute I for the primary index in Figure 7(b), and the access control policy and the original relation in Figure 7(a). Suppose that user u_1 is interested in retrieving all the resources with index value `a`. User u_1 computes the encoding of `a`, that is, ι_1(`a`), and searches the corresponding tuple in the secondary index. From the returned tuple, user u_1 retrieves ι(`a`), which is needed to u_1 to retrieve `Aresource` from the primary index, and the pair $\langle$`a`,$\mathfrak{s}_1^1\rangle$, which is needed to compute the next encoding for `a` that u_1 searches by using $\mathfrak{s}_1^1$ as salt.

If the salt function is known to the users, the data owner can decide to use a different salt function $\mathfrak{S}_i$ for each user u_i. In this case, each user can autonomously generate the sequence of salts adopted by the data owner for making different the multiple occurrences of the same index values. The only additional information that a user needs is the number of occurrences of each index value v that the user can access. Using this information, u_i is able to generate all the salts used to encode the occurrences of v she is authorized to access. Note that u_i can decide to search for the different occurrences of v in a random order, further strengthening protection guarantees. Figure 8 illustrates the primary and secondary

ORIGINAL RELATION

	K	I	Resource		ACL		
1	A	a	Aresource	...	u_1	u_2	u_3
2	B	a	Bresource	...	u_1	u_2	
3	C	b	Cresource	...	u_1	u_2	
4	D	b	Dresource	...		u_2	u_3
5	F	c	Fresource	...		u_2	u_3
6	G	a	Gresource	...	u_1		u_3
7	H	a	Hresource	...	u_1		u_3
8	I	b	Iresource	...	u_1		
9	J	c	Jresource	...	u_1		
10	L	c	Lresource	...	u_1		
11	M	d	Mresource	...	u_1		
12	N	a	Nresource	...		u_2	
13	O	a	Oresource	...		u_2	
14	P	b	Presource	...		u_2	
15	Q	d	Qresource	...		u_2	
16	R	a	Rresource	...			u_3
17	S	b	Sresource	...			u_3
18	T	c	Tresource	...			u_3
19	U	d	Uresource	...			u_3

(a)

PRIMARY INDEX

	I	Resource
10	$\iota(\mathtt{a})$	$\langle \ell_{123}, E(k_{123}, \mathtt{Aresource})\rangle$
14	$\iota(\mathtt{a} \oplus \mathfrak{p}_1)$	$\langle \ell_{12}, E(k_{12}, \mathtt{Bresource})\rangle$
3	$\iota(\mathtt{b})$	$\langle \ell_{12}, E(k_{12}, \mathtt{Cresource})\rangle$
11	$\iota(\mathtt{b} \oplus \mathfrak{p}_1)$	$\langle \ell_{23}, E(k_{23}, \mathtt{Dresource})\rangle$
4	$\iota(\mathtt{c})$	$\langle \ell_{23}, E(k_{23}, \mathtt{Fresource})\rangle$
13	$\iota(\mathtt{a} \oplus \mathfrak{p}_2)$	$\langle \ell_{13}, E(k_{13}, \mathtt{Gresource})\rangle$
12	$\iota(\mathtt{a} \oplus \mathfrak{p}_3)$	$\langle \ell_{13}, E(k_{13}, \mathtt{Hresource})\rangle$
19	$\iota(\mathtt{b} \oplus \mathfrak{p}_2)$	$\langle \ell_1, E(k_1, \mathtt{Iresource})\rangle$
1	$\iota(\mathtt{c} \oplus \mathfrak{p}_1)$	$\langle \ell_1, E(k_1, \mathtt{Jresource})\rangle$
15	$\iota(\mathtt{c} \oplus \mathfrak{p}_2)$	$\langle \ell_1, E(k_1, \mathtt{Lresource})\rangle$
5	$\iota(\mathtt{d})$	$\langle \ell_1, E(k_1, \mathtt{Mresource})\rangle$
6	$\iota(\mathtt{a} \oplus \mathfrak{p}_4)$	$\langle \ell_2, E(k_2, \mathtt{Nresource})\rangle$
17	$\iota(\mathtt{a} \oplus \mathfrak{p}_5)$	$\langle \ell_2, E(k_2, \mathtt{Oresource})\rangle$
2	$\iota(\mathtt{b} \oplus \mathfrak{p}_3)$	$\langle \ell_2, E(k_2, \mathtt{Presource})\rangle$
9	$\iota(\mathtt{d} \oplus \mathfrak{p}_1)$	$\langle \ell_2, E(k_2, \mathtt{Qresource})\rangle$
16	$\iota(\mathtt{a} \oplus \mathfrak{p}_6)$	$\langle \ell_3, E(k_3, \mathtt{Rresource})\rangle$
7	$\iota(\mathtt{b} \oplus \mathfrak{p}_4)$	$\langle \ell_3, E(k_3, \mathtt{Sresource})\rangle$
18	$\iota(\mathtt{c} \oplus \mathfrak{p}_3)$	$\langle \ell_3, E(k_3, \mathtt{Tresource})\rangle$
8	$\iota(\mathtt{d} \oplus \mathfrak{p}_2)$	$\langle \ell_3, E(k_3, \mathtt{Uresource})\rangle$

(b)

SECONDARY INDEX

	I	Resource
18	$\iota_1(\mathtt{a})$	$E(k_1, \iota(\mathtt{a})\|\|4)$
7	$\iota_2(\mathtt{a})$	$E(k_2, \iota(\mathtt{a})\|\|4)$
16	$\iota_3(\mathtt{a})$	$E(k_3, \iota(\mathtt{a})\|\|4)$
25	$\iota_1(\mathtt{a}, \mathfrak{s}_1^1)$	$E(k_1, \iota(\mathtt{a} \oplus \mathfrak{p}_1))$
17	$\iota_2(\mathtt{a}, \mathfrak{s}_1^2)$	$E(k_2, \iota(\mathtt{a} \oplus \mathfrak{p}_1))$
6	$\iota_1(\mathtt{b})$	$E(k_1, \iota(\mathtt{b})\|\|2)$
26	$\iota_2(\mathtt{b})$	$E(k_2, \iota(\mathtt{b})\|\|3)$
15	$\iota_2(\mathtt{b}, \mathfrak{s}_1^2)$	$E(k_2, \iota(\mathtt{b} \oplus \mathfrak{p}_1))$
19	$\iota_3(\mathtt{b})$	$E(k_3, \iota(\mathtt{b})\|\|2)$
1	$\iota_2(\mathtt{c})$	$E(k_2, \iota(\mathtt{c})\|\|1)$
24	$\iota_3(\mathtt{c})$	$E(k_3, \iota(\mathtt{c})\|\|2)$
8	$\iota_1(\mathtt{a}, \mathfrak{s}_2^1)$	$E(k_1, \iota(\mathtt{a} \oplus \mathfrak{p}_2))$
14	$\iota_3(\mathtt{a}, \mathfrak{s}_1^3)$	$E(k_3, \iota(\mathtt{a} \oplus \mathfrak{p}_2))$
2	$\iota_1(\mathtt{a}, \mathfrak{s}_3^1)$	$E(k_1, \iota(\mathtt{a} \oplus \mathfrak{p}_3))$
13	$\iota_3(\mathtt{a}, \mathfrak{s}_2^3)$	$E(k_3, \iota(\mathtt{a} \oplus \mathfrak{p}_3))$
20	$\iota_1(\mathtt{b}, \mathfrak{s}_1^1)$	$E(k_1, \iota(\mathtt{b} \oplus \mathfrak{p}_2))$
4	$\iota_1(\mathtt{c})$	$E(k_1, \iota(\mathtt{c} \oplus \mathfrak{p}_1))$
21	$\iota_1(\mathtt{c}, \mathfrak{s}_1^1)$	$E(k_1, \iota(\mathtt{c} \oplus \mathfrak{p}_2))$
9	$\iota_1(\mathtt{d})$	$E(k_1, \iota(\mathtt{d})\|\|1)$
3	$\iota_2(\mathtt{a}, \mathfrak{s}_2^2)$	$E(k_2, \iota(\mathtt{a} \oplus \mathfrak{p}_4))$
12	$\iota_2(\mathtt{a}, \mathfrak{s}_3^2)$	$E(k_2, \iota(\mathtt{a} \oplus \mathfrak{p}_5))$
23	$\iota_2(\mathtt{b}, \mathfrak{s}_2^2)$	$E(k_2, \iota(\mathtt{b} \oplus \mathfrak{p}_3))$
10	$\iota_2(\mathtt{d})$	$E(k_2, \iota(\mathtt{d} \oplus \mathfrak{p}_1))$
27	$\iota_3(\mathtt{a}, \mathfrak{s}_3^3)$	$E(k_3, \iota(\mathtt{a} \oplus \mathfrak{p}_6))$
5	$\iota_3(\mathtt{b}, \mathfrak{s}_1^3)$	$E(k_3, \iota(\mathtt{b} \oplus \mathfrak{p}_4))$
22	$\iota_3(\mathtt{c}, \mathfrak{s}_1^3)$	$E(k_3, \iota(\mathtt{c} \oplus \mathfrak{p}_3))$
11	$\iota_3(\mathtt{d})$	$E(k_3, \iota(\mathtt{d} \oplus \mathfrak{p}_2))$

(c)

Figure 8. Primary and secondary indexes of Figure 7 defined over a non-key attribute assuming that users know the salt function

indexes in Figure 7 obtained assuming that each user shares a random generation function with the data owner. As an example, consider again user u_1 who searches for the tuples with value a for attribute I. User u_1 will first search the encoded value $\iota_1(\mathtt{a})$. The search for $\iota_1(\mathtt{a})$ over the secondary index returns to the user both $\iota(\mathtt{a})$, which is needed to u_1 to retrieve Aresource from the primary index, and value 4, which is the number of occurrences of a that u_1 can access. User u_1 will then generate, using her own function $\mathfrak{S}_1$, salts $\mathfrak{s}_1^1$, $\mathfrak{s}_2^1$, and $\mathfrak{s}_3^1$ that u_1 uses to compute the encoded values corresponding to the other three occurrences of a that she can access.

6. Access Execution

We now illustrate how the primary and secondary indexes are jointly used for accessing a resource of interest. To retrieve a resource in $\mathcal{R}$ with target value v for I, a user u_i would need to perform the following three steps:

Step 1) compute the user-based encoding $\iota_i(v)=h(v,k_i)$;
Step 2) search $\iota_i(v)$ in the secondary index $\mathcal{S}$, retrieving the corresponding encoded value $\iota(v)$;
Step 3) search $\iota(v)$ in the primary index $\mathcal{P}$, retrieving the corresponding target tuple.

As an example, consider the indexes in Figure 5(a-b) and suppose that user u_1 searches index value C. User u_1 computes $\iota_1(\mathtt{C})=h(k_1,\mathtt{C})$ and then searches it in the secondary index in Figure 5(a). The search returns block S205, from which $\iota(\mathtt{C})$ is retrieved. Hence, u_1 searches $\iota(\mathtt{C})$ in the primary index in Figure 5(b). The search returns block P202, from which u_1 can retrieve resource Cresource.

If user u_i is interested in all the resources with a value for the indexing attribute I that falls in the range [*lower_bound*, *upper_bound*], she will first start searching *lower_bound*, following the steps illustrated above. Such a search will reveal her the next value in the sequence that she can access (Section 4). The iterative search process terminates when the next value in the sequence is greater than or equal to *upper_bound*. Note that accesses to an index structure defined over a non-key attribute operate in a similar way.

If the target value is not present in the secondary index, its user-based encoding does not appear in the block returned by *Step 2*. In such a case, the process will continue providing a random value for the search in *Step 3*, so to provide to the server the same observation as a successful search. Note also that the search for a value that is in the dataset but that the requesting user is not authorized to access appears to the requesting user as the search for a missing value (hence, the access process does not disclose anything to the user about values she is not authorized to access). Also, when evaluating a range condition, it is important to interleave an access to the secondary index with an access to the primary index. In principle, the user could search for all the values of interest in the secondary index, before searching for the first value in the range in the primary index. This practice, however, would reveal to the server the nature of the search operation.

The steps above illustrate how to retrieve a target value. However, both the primary and the secondary indexes are shuffle indexes and accesses should not simply aim at the target value but should also be protected with the techniques (cover, repeated searches, and shuffling) devoted to protect access confidentiality. The application of these techniques on the two indexes is completely independent, meaning that the choice of covers, repeated searches, and shuffling can be different for the two indexes. The only dependency between the two indexes is the fact that - clearly - the target to be searched in the primary index is the value retrieved by the search on the secondary index.

Covers, repeated searches, and shuffling on the primary and secondary indexes work essentially in the same way as they work in the shuffle index in absence of authorizations (Section 2). However, the nature of these indexes requires minor adjustments in their application, which we describe in the following.

- *Cover searches*. For both the secondary and the primary indexes, cover searches should be chosen from the set of *encoded values*, in contrast to the set of original values. The reason for this is that every user has limited knowledge on the set of original index values while she can have complete knowledge of the encoded values in the indexes (i.e., knowledge of the complete co-domains of all the encodings of all the users and the complete co-domain of the encoding of the owner). Since the encoding is non-invertible, this knowledge does not leak any information and allows the widest possible choice to the user.
- *Repeated accesses*. Repeated accesses for the primary and secondary indexes should refer to blocks, instead of specific values. The reason for this is that two subsequent accesses can be performed by two different users and therefore considering repeated searches referred to values would leak to the second user the target of the search of another user. Although such a leakage would be only on encoded values, we avoid it by simply assuming repeated accesses to be referred to blocks (and not to values) and to consider all accessed blocks, not only the target. At every access, we then store at the server the identifiers of the blocks (target, covers, or repeated accesses) accessed during the last search. The knowledge of such identifiers is sufficient for a user to repeat an access to one of the paths visited by the search just before her, without revealing to the user the target of the previous search.
- *Shuffling*. Shuffling works just like in the original proposal. We note that when shuffling, a user may move also content which is not intelligible to her. However, she will not be able to change the

content for which she is not authorized (since she would not know the encryption key and tampering would be detected). Note that since all physical blocks stored at the server are encrypted (with a key shared between all users and the data owner) and encryption of the block as a whole is refreshed at every shuffle, the server cannot detect whether the content of a block (or part of it) has changed or not. Hence, the fact that a user can operate only on a portion of the block does not prevent the correct execution of the shuffling operation.

Figure 9 illustrates function **Access**, executed at the client side, for accessing the primary and secondary indexes when searching for a value or a range of values. This function operates as discussed above and relies on function **Search**.

Function **Access** takes as input a range [*target_value_lower*,*target_value_upper*] of index values to be searched and a boolean variable *range*, and returns set *Resources* that contains the retrieved resources. Note that when searching for a value, variables *target_value_lower* and *target_value_upper* are both equal to the target value and variable *range* is set to false; variable *range* is set to true, otherwise. Function **Access** first initializes set *Resources* to the empty set and variable *target_value* to the first value to be searched (lines 1-2). Then, the function computes the user-based encoding $\iota_i(target\_value)$ and invokes function **Search** to search for such a value in the secondary index (lines 5-7). It decrypts the tuple retrieved by function **Search**, obtaining the encoded value $\iota(target\_value)$ for *target_value*, and the next original index value that user u_i is authorized to access (line 8). Two cases can now occur. In the first case, $\iota(target\_value)$ is not NULL, meaning that there is a tuple that the requesting user can access with index value equal to *target_value*. In this case, the function invokes **Search** over the primary index, looking for $\iota(target\_value)$. It then computes/retrieves the encryption key necessary to decrypt the retrieved resource, decrypts it, and adds it to *Resources*. If variable *range* is false or the *next* value in the sequence of values accessible by the requesting user is greater than *target_value_upper*, the search process is terminated by setting variable *range* to false. Otherwise, variable *target_value* is set to *next* and the search process continues (lines 15-17). In the second case, the result of function **Search** over the secondary index is NULL, that is, $\iota(target\_value)$ is null. Function **Access** then performs a fake search over the primary index to avoid the disclosure of any information to other users and to the server about the privileges of the requesting user (lines 18-19).

Function **Search** receives as input the shuffle index $\mathcal{T}$ on which it should operate, the index value *target_value* target of the access, and the number *num_cover* of covers to be adopted. It returns the tuple *r* with index value *target_value* (if any). The function downloads from the server the identifiers of the blocks visited by the previous search and randomly chooses $num\_cover+1$ values in the domain of the (primary or secondary) index (lines 1-3). It then visits the shuffle index level by level, starting from the root. At each level *level*, the function determines the identifiers of the nodes along the path to the target, covers, and repeated access (lines 5-8). If the block along the path to the target has been accessed by the previous search, it is repeated (in this case, an additional cover is used). The function downloads from the server and decrypts the blocks of interest (line 13) and shuffles their content (line 16). To guarantee the correctness of the search and of the index structure, the function updates the references to children of the nodes accessed at level $level-1$ (which are the parents of the nodes shuffled at level *level*), and variables *target*, *repeated*, and $cover[1,\ldots,num\_cover]$ (lines 17-21). The nodes at level $level-1$ are then encrypted and written at the server. The identifiers of the nodes accessed at level *level* are then used to update $repeated\_search[level]$ (line 23). Once the leaf node where *target_value* is possibly stored has been reached, the function extracts and returns the tuple with index value equal to *target_value* (lines 25-27).

```
/* P, S : primary and secondary index */
/* num_cover : number of cover searches */
/* u_i,k_i : user performing the access and her key */
/* h : non-invertible cryptographic hash function */

Access(target_value_lower,target_value_upper,range)/* target_value_lower and target_value_upper are both equal to target_value
 1: Resources := ∅                                    when searching a single value target_value and range is false */
 2: target_value:=target_value_lower
 3: repeat
 4:   /* Phase 1: compute the user-based encoding ι_i(target_value) */
 5:   target_idx := h(k_i, target_value)
 6:   /* Phase 2: search ι_i(target_value) in the secondary index */
 7:   s := Search(S,target_idx,num_cover)
 8:   ⟨target_idx,next⟩ := decrypt s[Resource] with k_i /* encoded value ι(target_value) and next original index value */
 9:   /* Phase 3: search ι(target_value) in the primary index */
10:   if target_idx ≠ NULL then
11:     p := Search(P,target_idx,num_cover)
12:     Let p[Resource]=⟨ℓ,content⟩; retrieve key k with label ℓ
13:     result := decrypt content with k
14:     Resources := Resources ∪ {result}
15:     if range then
16:       if next > target_value_upper then range:=false
17:       else target_value:=next
18:   else target_idx := randomly choose a value for ι(target_value)
19:        Search(P,target_idx,num_cover)
20: until range
21: return(Resources)

Search(T,target_value,num_cover) /* Function that searches for target_value in T */
 1: repeated_search[0,...,T.height] := download and decrypt the identifiers of blocks for repeated accesses to index T
 2: randomly choose cover_value[1...num_cover+1] for target_value in the co-domain of h
 3: repeated := repeated_search[0] /* identifier of the root block */
 4: for level:=1...T.height do
 5:   /* identify the blocks to read from the server */
 6:   target := identifier of the node at level level along the path to target_value
 7:   cover[i] := identifier of the node at level level along the path to cover_value[i], i=1...num_cover+1
 8:   repeated := block identifier in repeated_search[level] that is a descendant of repeated
 9:   if target is the identifier of a node in repeated_search[level] then repeated := target
10:   else num_cover := num_cover−1
11:   ToGet := {target,repeated} ∪ cover[1...num_cover] /* ids of the blocks to be downloaded */
12:   /* read blocks */
13:   Nodes := download and decrypt the blocks with identifier in ToGet
14:   /* shuffle nodes */
15:   let ℘ be a permutation of the identifiers of nodes in Nodes
16:   shuffle nodes in Nodes according to ℘
17:   update pointers to children of the parents of nodes in Nodes according to ℘
18:   encrypt and write at the server nodes accessed at iteration level − 1
19:   target := ℘(target)
20:   cover[i] := ℘(cover[i]), i=1...num_cover+1
21:   repeated := ℘(repeated)
22:   /* update the repeated search at level level */
23:   repeated_search[level] := ToGet
24: encrypt and write at the server nodes accessed at iteration T.height and repeated_search
25: let n∈Nodes the node with n.id=target
26: let r∈n be the tuple such that r[I]=target_value
27: return(r)
```

Figure 9. Shuffle index access algorithm

Figure 5(a-b) illustrates an example of access execution for searching value C by user u_1. We assume that $\iota_2(\mathtt{F})$ is the cover and path [S001,S101,S202] is the repeated access for the secondary index, and that $\iota(\mathtt{Q})$ is the cover and path [P001,P102,P205] is the repeated access for the primary index. Accessed nodes are, besides the root, those annotated (as target, cover, or repeated) in the figure. Figure 5(c-d) illustrates the new structure of the indexes that would result assuming shuffling: for the secondary index as S101→S102, S102→S103, S103→S101, S202→S205, S205→S209, S209→S202; for the primary index as P101→P103, P102→P101, P103→P102, P202→P208, P205→P202, P208→P205. Consider now a search for range [C,G] by user u_1. The search illustrated above returns, besides Cresource, also index value G. Function **Access** will then search for $\iota_1(\mathtt{G})$ in the secondary index, and for $\iota(\mathtt{G})$ in the primary index, retrieving resource Gresource and index value H. Since the range of interest has been completely covered, the search process terminates and returns to user u_1 Cresource and Gresource.

7. Data and Policy Updates

We now describe how possible changes in the authorization policy (which are regulated by the data owner) can be supported. Changes in the authorization policy can be of three types: *1)* insertion/deletion/update of a resource; *2)* insertion/deletion of a user; and *3)* grant/revoke of an authorization. We note that the insertion/deletion of a user has an impact on the policy only when the user is involved in authorizations. In the following, we then focus on the insertion, removal, and update of resources and on grant and revoke of authorizations. For simplicity, we assume that each operation refers to a single resource r and a single user u (extensions to sets of tuples and users are immediate). The examples refer to the primary index in Figure 3 and the secondary index in Figure 4.

7.1. Removal, insertion, and update of a resource

An observer can recognize operations that remove, insert, or update a resource from read-only accesses whenever they require a change in the structure of the primary and/or secondary indexes. To make them indistinguishable from read-accesses, we adopt the approach in [9]. This solution prevents the removal of nodes from the structure by marking removed tuples as non valid, while it adopts probabilistic splits to make the insertion of new tuples indistinguishable from read accesses. Intuitively, to prevent an observer from discriminating insert operations when a node is split, nodes are possibly split also when visited by a read access, according to the result of a random function.

- *Removal*. The removal of a resource r requests the removal of the encoded value $\iota(r[I])$ as well as all values $\iota_i(r)$, $\forall u_i \in acl(r)$ from the leaves of the primary and secondary index, respectively. These values are therefore searched in the primary and secondary index and the corresponding resources are marked as 'non-valid' (e.g., encrypted with a key known to the data owner only, or overwritten with a dummy content) [9]. For instance, the removal of resource Cresource with index value C requires to set as non-valid the resources with index values $\iota(\mathtt{C})$ (primary index) and $\iota_1(\mathtt{C})$, $\iota_2(\mathtt{C})$ (secondary index). Note that, for the working of the system, the index values in the secondary index do not need to be removed or set as non-valid. Indeed, the user will discover the absence of the resource of interest when visiting the primary index.
- *Insertion*. The insertion of a new resource r with $acl(r)$ requires the insertion of a new tuple p in the primary index, having $p[I]=\iota(r[I])$ and containing the original resource in encrypted form,

that is, $p[Resource]=\langle \ell_{i_1,\ldots,i_n}, E(k_{i_1,\ldots,i_n}, r[Resource])\rangle$, with $acl(r)=\{u_{i_1},\ldots,u_{i_n}\}$. Analogously, for each user u_i in $acl(r)$ a new tuple s is inserted in the secondary index, having $s[I] = \iota_i(r[I])$ and $s[Resource] = E(k_i, \iota(r[I]))$. For instance, the insertion of a new resource Zresource with index value Z and $acl(\mathtt{Z})=\{u_2,u_3\}$ requires the insertion of tuple p with $p[Resource]=\langle \ell_{23}, E(k_{23},\mathtt{Zresource})\rangle$ and index value $p[I]=\iota(\mathtt{Z})$ in the primary index, and of two tuples s_1 and s_2 in the secondary index, with $s_1[I]=\iota_2(\mathtt{Z})$, $s_1[Resource]=\langle E(k_2,\iota(\mathtt{Z}))\rangle$, and $s_2[I]=\iota_3(\mathtt{Z})$, $s_2[Resource]=\langle E(k_3,\iota(\mathtt{Z}))\rangle$.

- *Update*. The update of a resource may have an impact on the primary and secondary indexes only when it requires a change in the index value. In fact, if the index value does not change, it is sufficient to search for the resource to be updated, and to modify its encrypted representation in the primary index during the access operation. On the contrary, when the index value associated with the resource needs to be updated, it is necessary to modify the primary index to move the resource to the correct leaf, and the secondary index to enable users to retrieve the resource when searching for its new index value. Such an update can be seen (and realized) as the removal of a resource followed by the insertion of the same resource with a new value for the index attribute.

7.2. Grant and revoke

Grant and revoke operations require the insertion and removal of tuples in the primary and/or secondary index, which are again performed according to the approach described in [9].

- *Grant*. A request to grant user u_i access to resource r requires a change only in the secondary index to allow user u to retrieve the encoded value $\iota(r[I])$. The data owner then inserts a new tuple s in the secondary index with $s[I]=\iota_i(r[I])$ and $s[Resource]=E(k_i,\iota(r[I]))$. Also, the data owner re-encrypts resource r in the primary index, using an encryption key that also u_i can derive (i.e., the key associated with the new access control list $acl(r)$ of the resource). For instance, to grant user u_3 access to resource Cresource, the data owner inserts a tuple with index value $\iota_3(\mathtt{C})$ and content $\langle E(k_3,\iota(\mathtt{C}))\rangle$ in the secondary index. Also, she re-encrypts the content of the tuple with index value $\iota(\mathtt{C})$ in the primary index, using key k_{123}.
- *Revoke*. To prevent users from distinguishing between the removal of a resource that she is authorized to access and the revoke of her access privilege for the same resource, the encoding $\iota(r[I])$ of the index value of the revoked resource must be changed. The primary and the secondary indexes must then be updated accordingly. The data owner then computes a new encoded value $\iota(r[I], salt)$ for the resource by concatenating a random *salt* with the original index value $\iota(r[I], salt)=h(k_o,r[I]||salt)$. The data owner removes from the primary index the tuple p_{old} storing the resource before the policy update, and inserts a new tuple p_{new} storing the resource after the policy update. The new tuple p_{new} has index value $p_{new}[I]=\iota(r[I], salt)$ and stores resource r, encrypted with a key known to authorized users only (i.e., all users X in $acl(r)$ but u_i), that is, $p_{new}[Resource]=\langle \ell_X, E(k_X,r[Resource])\rangle$. Since the encoded value associated with the revoked resource has been updated, the data owner must modify the secondary index, to enable authorized users to retrieve the resource. For each authorized user u_j, the data owner updates the tuple s in the secondary index with $s[I]=\iota_j(r[I])$, setting its content to $s[Resource]=E(k_j,\iota(r[I], salt))$. For instance, assume that user u_1 is revoked access to resource Cresource. The data owner first re-computes the encoded value for C, $\iota(\mathtt{C}, salt)$, removes from the primary index tuple p_{old} with $p_{old}[I]=\iota(\mathtt{C})$, and inserts a new tuple p_{new} with the same content as the removed one, but encrypted with k_2, $p_{new}[Resource]=\langle \ell_2, E(k_2, \mathtt{Cresource})\rangle$ and index value $p_{new}[I]=\iota(\mathtt{C}, salt)$. The data

owner then updates the secondary index: she removes tuple s_1 with index value $s_1[I]{=}\iota_1(\mathtt{C})$, and updates the content of tuple s_2 with index value $s_2[I]{=}\iota_2(\mathtt{C})$, setting $s_2[Resource]{=}E(k_2,\iota(\mathtt{C},salt))$.

We observe that, to guarantee that the server cannot recognize read accesses from modifications to the dataset or the access policy, every access to the primary index must be preceded by an access to the secondary index (and every access to the secondary index must be followed by an access to the primary index). Whenever the access to the primary (or secondary) index is not necessary for an update operation, the data owner performs a read access searching for a random value.

7.3. Token management for efficient revocation

The solution described above for the management of revoke operations, although effective, implies a high communication cost for the data owner, especially if the revoked resource can be accessed by many users. Indeed, the data owner needs to access the secondary index as many times as the number of users in the *acl* of the revoked resource. An alternative approach that limits the overhead of revoke management consists in using a different encryption key for each resource. Each resource is then associated with a set of tokens, enabling authorized users to derive the encryption key [20]. To revoke a user access to a resource, the data owner re-encrypts the resource and modifies its tokens, without the need to update the secondary index.

Formally, each resource r_i is encrypted with an encryption key k_i randomly chosen by the data owner and used exclusively for it. The set of keys used for the encryption policy is then defined as follows.

Definition 7.1 (Encryption Policy Keys with Tokens)**.** *Let $\mathcal{R}(I,Resource)$ be a relation, $\mathcal{U}$ be a set of users, and, $\forall r{\in}\mathcal{R}$, $acl(r){\subseteq}\mathcal{U}$ be the acl of r. The set $\mathcal{K}$ of* encryption policy keys *for $\mathcal{R}$ is a set $\mathcal{K}{=}\{k_i \mid u_i \in \mathcal{U}\} \cup \{k_j \mid r_j \in \mathcal{R}\}$ of encryption keys. Each key $k_i{\in}\mathcal{K}$ has a* public label ℓ_i*. Each user $u_i{\in}\mathcal{U}$ knows the set $\mathcal{K}_i = \{k_i\} \cup \{k_j \mid k_j \in \mathcal{K} \wedge u_i \in acl(r_j)\}$ of keys.*

Each resource r_i is complemented, in the primary index, with a *token* $t_{j,i}$ for each user u_j. Token $t_{j,i}$ enables u_j to derive k_i from her key k_j, if $u_j{\in}acl(r_i)$; $t_{j,i}$ is a random string of the same length as real tokens, otherwise. Token $t_{j,i}$ enabling key derivation is defined as $t_{j,i}{=}k_i{\oplus}h(k_j,\ell_i)$, where ℓ_i is a publicly available label associated with k_i, $\oplus$ is the bitwise xor operator, and h is a deterministic cryptographic function. Given the set $\mathcal{U} = \{u_1,\ldots,u_n\}$ of users, the tuple p_i representing a resource r_i in the primary index has content $p_i[Resource]{=}\langle \ell_i, E(k_i,r[Resource]),t_{1,i},\ldots,t_{n,i}\rangle$.

The primary index structure is then formally defined as follows.

Definition 7.2 (Primary Index with Tokens)**.** *Let $\mathcal{R}(I,Resource)$ be a relation, I be the indexing attribute, ι be an encoding function for I, $\mathcal{U} = \{u_1,\ldots,u_n\}$ be a set of users, and $\mathcal{K}$ be the set of encryption policy keys for $\mathcal{R}$. A* primary index *for $\mathcal{R}$ over I is a shuffle index over relation $\mathcal{P}(I,Resource)$ having a tuple p_i for each tuple $r_i{\in}\mathcal{R}$ such that $p_i[I] = \iota(r_i[I])$ and $p_i[Resource]{=}\langle \ell_i, E(k_i, r_i[Resource])t_{1,i},\ldots,t_{n,i}\rangle$ such that if $u_j{\in}acl(r_i)$ $t_{j,i}{=}k_i{\oplus}h(k_j,\ell_i)$; $t_{j,i}$ is a fake token, otherwise, for each $j = 1,\ldots,n$.*

Figure 10 illustrates the primary index, with tokens stored together with resources, for the relation and access control policy in Figure 1(a). Note that the secondary index is not affected by this approach for token management.

This revised structure of the primary index has an impact on the operations aimed at updating the outsourced resource collection and the authorization policy. While tuple insertion, tuple update, and

ORIGINAL RELATION

	I	Resource		ACL		
1	A	Aresource	...	u_1	u_2	u_3
2	B	Bresource	...	u_1	u_2	
3	C	Cresource	...	u_1	u_2	
4	D	Dresource	...		u_2	u_3
5	F	Fresource	...		u_2	u_3
6	G	Gresource	...	u_1		u_3
7	H	Hresource	...	u_1		u_3
8	I	Iresource	...	u_1		
9	J	Jresource	...	u_1		
10	L	Lresource	...	u_1		
11	M	Mresource	...	u_1		
12	N	Nresource	...		u_2	
13	O	Oresource	...		u_2	
14	P	Presource	...		u_2	
15	Q	Qresource	...		u_2	
16	R	Rresource	...			u_3
17	S	Sresource	...			u_3
18	T	Tresource	...			u_3
19	U	Uresource	...			u_3

(a)

PRIMARY INDEX

	I	Resource			
12	ι(A)	$\langle \ell_A$	$E(k_A,$Aresource$)$	$k_A \oplus h(k_1,\ell_A)$	$k_A \oplus h(k_2,\ell_A)$	$k_A \oplus h(k_3,\ell_A)\rangle$
17	ι(B)	$\langle \ell_B$	$E(k_B,$Bresource$)$	$k_B \oplus h(k_1,\ell_B)$	$k_B \oplus h(k_2,\ell_B)$	$fake_{3,B}\rangle$
4	ι(C)	$\langle \ell_C$	$E(k_C,$Cresource$)$	$k_C \oplus h(k_1,\ell_C)$	$k_C \oplus h(k_2,\ell_C)$	$fake_{3,C}\rangle$
3	ι(D)	$\langle \ell_D$	$E(k_D,$Dresource$)$	$fake_{1,D}$	$k_D \oplus h(k_2,\ell_D)$	$k_D \oplus h(k_3,\ell_D)\rangle$
7	ι(F)	$\langle \ell_F$	$E(k_F,$Fresource$)$	$fake_{1,F}$	$k_F \oplus h(k_2,\ell_F)$	$k_F \oplus h(k_3,\ell_F)\rangle$
9	ι(G)	$\langle \ell_G$	$E(k_G,$Gresource$)$	$k_G \oplus h(k_1,\ell_G)$	$fake_{2,G}$	$k_G \oplus h(k_3,\ell_G)\rangle$
10	ι(H)	$\langle \ell_H$	$E(k_H,$Hresource$)$	$k_H \oplus h(k_1,\ell_H)$	$fake_{2,H}$	$k_H \oplus h(k_3,\ell_H)\rangle$
8	ι(I)	$\langle \ell_I$	$E(k_I,$Iresource$)$	$k_I \oplus h(k_1,\ell_I)$	$fake_{2,I}$	$fake_{3,I}\rangle$
6	ι(J)	$\langle \ell_J$	$E(k_J,$Jresource$)$	$k_J \oplus h(k_1,\ell_J)$	$fake_{2,J}$	$fake_{3,J}\rangle$
11	ι(L)	$\langle \ell_L$	$E(k_L,$Lresource$)$	$k_L \oplus h(k_1,\ell_L)$	$fake_{2,L}$	$fake_{3,L}\rangle$
2	ι(M)	$\langle \ell_M$	$E(k_M,$Mresource$)$	$k_M \oplus h(k_1,\ell_M)$	$fake_{2,M}$,	$fake_{3,M}\rangle$
14	ι(N)	$\langle \ell_N$	$E(k_N,$Nresource$)$	$fake_{1,N}$	$k_N \oplus h(k_2,\ell_N)$	$fake_{3,N}\rangle$
5	ι(O)	$\langle \ell_O$	$E(k_O,$Oresource$)$	$fake_{1,O}$	$k_O \oplus h(k_2,\ell_O)$	$fake_{3,O}\rangle$
18	ι(P)	$\langle \ell_P$	$E(k_P,$Presource$)$	$fake_{1,P}$	$k_P \oplus h(k_2,\ell_P)$	$fake_{3,P}\rangle$
16	ι(Q)	$\langle \ell_Q$	$E(k_Q,$Qresource$)$	$fake_{1,Q}$	$k_Q \oplus h(k_2,\ell_Q)$	$fake_{3,Q}\rangle$
15	ι(R)	$\langle \ell_R$	$E(k_R,$Rresource$)$	$fake_{1,R}$	$fake_{2,R}$	$k_R \oplus h(k_3,\ell_R)\rangle$
19	ι(S)	$\langle \ell_S$	$E(k_S,$Sresource$)$	$fake_{1,S}$	$fake_{2,S}$	$k_S \oplus h(k_3,\ell_S)\rangle$
1	ι(T)	$\langle \ell_T$	$E(k_T,$Tresource$)$	$fake_{1,T}$	$fake_{2,T}$	$k_T \oplus h(k_3,\ell_T)\rangle$
13	ι(U)	$\langle \ell_U$	$E(k_U,$Uresource$)$	$fake_{1,U}$	$fake_{2,U}$	$k_U \oplus h(k_3,\ell_U)\rangle$

(b)

Figure 10. Relation of Figure 1(a) with *acl*s associated with its resources (a) and relation for the primary index (b) when tokens are stored in leaf nodes

grant operations are marginally affected by the change in the primary index structure and operate as illustrated in Sections 7.1 and 7.2, the removal of tuples and the revoke of authorizations need to be revised as follows.

- *Removal.* To remove resource r, the data owner substitutes the encrypted representation of r in the primary index with a random string of the same length and invalidates all the tokens, substituting each of them with a fake token. When user u searches for the index value $r[I]$ of a removed resource r, she will not be able to use the corresponding token. Using this approach, the data owner does not need to visit the secondary index to invalidate the encoded representation of $r[I]$ for each user u in $acl(r)$. For instance, considering the example in Figure 10, to remove resource Cresource, the data owner only needs to search for ι(C) in the primary index, download the corresponding tuple, generate three fake tokens (one for each user) and a random string of the same length as the encrypted resource, and upload the new tuple.
- *Revoke.* To revoke user u_i access to resource r_j, the data owner randomly generates a new encryption key k_{new} for r_j and re-encrypts the resource with the new key. The data owner generates a token for each user authorized for r_j (i.e., $u \in acl(r_j)$), enabling her to compute the new encryption key k_{new}, and a fake token for the other users. The content of the tuple in the primary index storing r_j is then updated to $p[Resource]=\langle \ell_{new}, E(k_{new}, r_j[Resource]), t_{1,new}, \ldots, t_{n,new}\rangle$, where $t_{i,new}$ is a token enabling u_i to derive k_{new} from k_i if $u_i \in acl(r_j)$; it is a fake token, otherwise. Note that the data owner does not need to update any of the tuples in the secondary index (only the primary index is modified for the enforcement of revoke operations). For instance, with reference to the example in Figure 10, assume that the data owner wants to revoke to user u_1 access to Cresource. First, the data owner searches for ι(C) in the primary index, and downloads the corresponding tuple $\langle \ell_C, E(k_C,$Cresource$), k_C \oplus h(k_1,\ell_C), k_C \oplus h(k_2,\ell_C), fake_{3,C}\rangle$. She then decrypts $E(k_C,$Cresource$)$ using key k_C, obtaining plaintext resource Cresource. She generates a new encryption key k'_C with label ℓ'_C, and re-encrypts Cresource with k'_C. The data owner computes a token $t'_{2,C}$ enabling user u_2 to derive k'_C from her own key k_2, and generates two fake tokens for u_1 and u_3. The data owner

finally updates the tuple in the primary index as $\langle \ell'_{\mathrm{C}}, E(k'_{\mathrm{C}},\texttt{Cresource}), fake_{1,\mathrm{C}}, k'_{\mathrm{C}} \oplus h(k_2, \ell'_{\mathrm{C}}), fake_{3,\mathrm{C}}\rangle$. This approach guarantees that a user cannot distinguish between the removal of a resource r she is authorized to access and the revocation of her privilege over it. In fact, after the update of the tuple representing r in the primary index, she will be associated with a fake token that cannot be used for decryption.

Our solution for token management has two advantages. First, it permits the data owner to manage revoke operations in a simpler and less expensive way. In fact, she does not need to modify the secondary index, but only the primary index is affected. Second, this solution facilitates key and token management: each resource can be encrypted with a fresh new key (which is not shared with other resources), and tokens are stored together with resources. Hence, searches over the primary/secondary index do not require to also access a token catalog, which should be properly protected. Also, key derivation requires only one derivation step.

8. Analysis

In this section, we demonstrate the correct enforcement of the (dynamic) authorization policy defined by the data owner (Section 8.1), we discuss the protection of access and pattern confidentiality provided by our approach (Section 8.2), and we analyze the performance and economic overhead it causes (Section 8.3).

8.1. Correctness

The primary and secondary indexes described in Section 3 guarantee the correct enforcement of the access control policy if each user u_i can access all and only the resources and index values in $\mathcal{R}$ she is authorized to access, as formally stated by the following theorem.

Theorem 8.1. *Let $\mathcal{R}(I,Resource)$ be a relation, $\mathcal{U}$ be a set of users, $acl(r)\subseteq\mathcal{U}$ be the acl of r, $\forall r\in\mathcal{R}$. The encryption policy keys (Definition 3.2), the primary index $\mathcal{P}(I,Resource)$ for $\mathcal{R}$ over I (Definition 3.3), and the secondary index $\mathcal{S}(I,Resource)$ for $\mathcal{R}$ and $\mathcal{P}$ (Definition 3.4)* correctly enforce *$acl(r)$, $\forall r\in\mathcal{R}$, iff $\forall u_i \in \mathcal{U}$, the following conditions hold:* i) *u_i can access resource r*[Resource] *iff $u_i\in acl(r)$;* ii) *u_i can see an index value v iff $\exists r\in\mathcal{R}$ s.t. $r[I]=v$ and $u_i\in acl(r)$.*
The insertion, removal, and update of the tuples and the grant and revoke of authorization (Sections 7.1 and 7.2) preserve the correctness of policy enforcement.

Proof. Consider a user u_i s.t. $acl(r)=\{u_{i_1},\ldots,u_{i_n}\}$ and $u_i\in\{u_{i_1},\ldots,u_{i_n}\}$. We need to show that u_i can retrieve the plaintext content of tuple r. A user u_i can retrieve and decrypt r iff: *i)* u_i can compute $\iota_i(r[I])$; *ii)* $\exists! s \in \mathcal{S}$ s.t. $s[I]=\iota_i(r[I])$ and $s[Resource]=E(k_i,\iota(r[I]))$; *iii)* $\exists! p \in \mathcal{P}$ s.t. $p[I]=\iota(r[I])$ and $p[Resource]=\langle \ell_{i_1,\ldots,i_n}, E(k_{i_1,\ldots,i_n}, r[Resource])\rangle$; and *iv)* u_i can visit $\mathcal{S}$ and $\mathcal{P}$.
User u_i can compute $\iota_i(r[I])$ since it is defined as $h(k_i, r[I])$ and u_i knows key k_i, by Definition 3.2. Tuple s exists and belongs to $\mathcal{S}$ by Definition 3.4. Tuple p exists and belongs to $\mathcal{P}$ by Definition 3.3. User u_i can decrypt the content of $s[Resource]$ as she knows $k_i\in\mathcal{K}_i$, and the content of $p[Resource]$ as she knows $k_{i_1,\ldots,i_n}\in\mathcal{K}_i$ because $u_i\in acl(r)$, by Definition 3.2. Any authorized user, including u_i, can visit both $\mathcal{S}$ and $\mathcal{P}$ since she knows both the encryption key k used by the data owner to encrypt the content of nodes to enable shuffling, and the co-domain of the encoding functions.

Note that the observations above hold also when a new resource r is inserted into the data collection. Indeed, as illustrated in Section 7, the data owner inserts in the primary and in the secondary index the same tuples that she would have inserted at initialization time for r.
The ability of user u_i to retrieve and decrypt r is also not affected by grant and revoke operations. When user u_i is granted access to r, the data owner inserts a new tuple s in the secondary index having $s[I]{=}\iota_i(r[I])$ and $s[Resource]{=}E(k_i,\iota(r[I]))$, thus enabling the user to retrieve $\iota(r[I])$. Since the data owner also re-encrypts the content of tuple p in the primary index having $p[I]{=}\iota(r[I])$ with the key of the new $acl(r)$, which also includes u_i, user u_i can decrypt $p[Resource]$ and access r.
Let us now consider the case where another user u_j, $j \neq i$, is granted access to a resource r that u_i can access. This grant operation does not affect the ability of u_i to retrieve and decrypt r. Indeed, no tuple is removed from the primary and secondary index, and u_i can derive the key used to re-encrypt r since $u_i{\in}acl(r)$. Similarly, if u_j, $j \neq i$, is revoked access to a resource r that u_i can access, the ability of u_i to access r is not affected. In fact, tuple s with $s[I]{=}\iota_i(r[I])$ is not modified or removed from the secondary index. Also, the key used to re-encrypt r can still be derived by u_i, since $u_i{\in}acl(r)$ also after the revoke operation.
Consider now a user u_i s.t. $acl(r){=}\{u_{i_j},\ldots,u_{i_n}\}$ and $u_i{\notin}\{u_{i_j},\ldots,u_{i_n}\}$. We need to show that u_i can access neither the plaintext content of $r[Resources]$, nor index value $r[I]$. It is immediate to see that u_i cannot access the plaintext content of $r[Resources]$ since it is encrypted with a key k_X (Definition 3.3) that u_i does not know. In fact, by Definition 3.3, since u_i does not belong to $acl(r)$, she does not know the corresponding encryption key. User u_i cannot compute or guess index value $r[I]$ because $r[I]$ is never represented in internal or leaf nodes of the primary and secondary indexes; it is instead represented via its encoded value (i.e., $\iota(r[I])$ in the primary index and $\iota_j(r[I])$, $\forall u_j{\in}acl(r)$, in the secondary index). Since the encoding function is, by Definition 3.1, non-invertible, u_i cannot exploit her knowledge of encoded values to retrieve the corresponding original index values. Also, the traversal of the primary (and secondary) index does not reveal u_i anything about the original index values. In fact, by Definition 3.1, the encoding function does not preserve the order relationship among values. Hence, similar encoded values (e.g., represented in the same leaf) may not correspond to similar original values (and vice versa).
Let us now analyze the case where user u_i is revoked access to r. After the revoke operation, u_i can access neither the plaintext content of r, nor index value $r[I]$ anymore. Since the resource is re-encrypted with a key k_X that u_i does not know, she cannot access the plaintext content of the resource. In fact, after the revoke operation, u_i does not belong to $acl(r)$ anymore. Furthermore, she cannot identify the new encoded value $\iota(r[I],salt)$ of the resource, since tuple s with $s[I]{=}\iota_i(r[I])$ has been removed from the secondary index and *salt* is a random nonce. □

Also the alternative approach for token management discussed in Section 7.3 guarantees the correct enforcement of the access control policy, as stated by the following theorem.

Theorem 8.2. *Let $\mathcal{R}(I,Resource)$ be a relation, $\mathcal{U}$ be a set of users, $acl(r){\subseteq}\mathcal{U}$ be the acl of r, $\forall r{\in}\mathcal{R}$. The encryption policy keys (Definition 7.1), the primary index $\mathcal{P}(I,Resource)$ for $\mathcal{R}$ over I (Definition 7.2), and the secondary index $\mathcal{S}(I,Resource)$ for $\mathcal{R}$ and $\mathcal{P}$ (Definition 3.4)* correctly enforce *$acl(r)$, $\forall r{\in}\mathcal{R}$, iff $\forall u_i \in \mathcal{U}$, the following conditions hold:* i) *u_i can access resource r*[Resource] *iff $u_i{\in}acl(r)$;* ii) *u_i can see an index value v iff $\exists r{\in}\mathcal{R}$ s.t. $r[I]{=}v$ and $u_i{\in}acl(r)$.*
The insertion, removal, and update of the tuples and the grant and revoke of authorizations (Section 7.3) preserve the correctness of policy enforcement.

Proof. Let us first consider the initial configuration outsourced by the data owner. The main difference with respect to the base scenario discussed above consists in key management. Indeed, each resource r_j is encrypted with a different key k_j. In the primary index, the tuple p_j storing r_j also includes a token $t_{i,j}$ for each user u_i. Such a token either enables u_i to derive k_j, if $u_i \in acl(r_j)$. It is a fake token, which does not enable any key derivation, otherwise. Hence, user u_i s.t. $u_i \in acl(r_j)$ can retrieve and decrypt r_j, while user $u_i \notin acl(r_j)$ can access neither the plaintext content $r_j[Resources]$, nor the index value $r_j[I]$ of the resource.
We need to show that the insertion and removal of tuples and authorizations does not affect the correctness of policy enforcement. The insertion of a new resource r maintains the correct enforcement of the access control policy because it implies the insertion in the primary and secondary index, of the same tuples that would have been inserted for r at initialization time. Similarly, the removal of a resource r does not affect the enforcement of the access control policy. In fact, it implies the substitution of the encrypted representation of r in the primary index with a random string of the same length, and of each token with a fake one.
When user u_i is granted access to r_j, the data owner inserts a new tuple s in the secondary index having $s[I]=\iota_i(r[I])$ and $s[Resource]=E(k_i,\iota(r_j[I]))$, thus enabling the user to retrieve $\iota(r_j[I])$. Since the data owner also modifies the token $t_{i,j}$ in the primary index to enable user u_i to compute the encryption key k_j used to protect $p_j[Resource]$, grant operations do not affect the correct enforcement of the access control policy.
When user u_i is revoked access to r_j, the data owner modifies the tuple p_j in the primary index storing it, substituting $t_{i,j}$ with a fake token. The data owner also re-encrypts r_j with a new key k_{new} and updates the tokens $t_{k,j}$ of non-revoked users u_k, enabling them to derive k_{new}. Therefore, any user $u \in acl(r_j)$ can still access r_j in plaintext. On the contrary, the token $t_{z,j}$ for a non-authorized user u_z (including u_i) is fake. Then, non-authorized users cannot decrypt the content of $p_j[Resource]$. □

8.2. Access confidentiality

We now discuss the confidentiality guarantees provided by the proposed approach. To analyze access and pattern confidentiality, we consider two possible observers: the storing server and a user of the system. Indeed, the storing server is the party with the highest potential for observations among the parties that are not authorized to access the content of resources, since all accesses are executed by it. Authorized users have instead plaintext visibility over a subset of the resources.

Server. We first consider the storing server as our observer and analyze the protection offered by our proposal for the novel aspects introduced with respect to the shuffle index proposal in [9]. In our analysis, we assume that the server knows: the number of blocks (nodes) in the primary and secondary index; the height of the two tree structures; the identifier of each block and its level in the tree; and the identifier of read and written blocks for each access operation. (This knowledge can be acquired by observing a sufficiently long sequence of accesses to the indexes.) Despite the ability of the server to observe all the accesses by the users and its knowledge of the shuffle index, it cannot identify the resource target of an access (access confidentiality) and it cannot infer whether two searcher aim at the same or at a different resource (patter confidentiality).

Like in the original proposal, we focus the analysis on the leaves of the shuffle index. In fact, nodes at a higher level are subject to a greater number of accesses, due to the multiple paths that pass through them, and are then involved in a larger number of shuffling operations, which increase their protection.

A search or an insert operation on the primary and secondary index operates as in the original proposal. Hence, it enjoys the protection guarantees given by the combined adoption of covers, repeated searches, and shuffling. In the considered scenario, however, we operate with two indexes and each search for a value entails an access to the secondary index followed by an access to the primary index. The targets of the two accesses are related as they are the encoding of the same original index value. However, both indexes protect the target of accesses (as well as patterns thereof) and the covers and repeated searches adopted for the two indexes are different. This practice prevents the server from identifying any correspondence between the values in the leaves of the two indexes.

Differently from searches aiming at a single value, searches for a range of values do not operate in the same way as in the original proposal. However, our approach for supporting range queries provides the same protection guarantees. Indeed, a range query appears to an observer as a sequence of searches for a single value. Since our approach, based on the combined adoption of covers, repeated searches, and shuffling, provides access confidentiality guarantees and our encoding functions are not order preserving, the server observing a sequence of accesses cannot determine whether such a sequence corresponds to a range query or to a sequence of searches for independent values. Search operations over a shuffle index defined on a non-key attribute operate in a similar way as range queries, and hence enjoy the same protection guarantees.

User. We now consider a user as our observer, who has knowledge of: the plaintext content of a subset of the resources, the encryption key used by the data owner for shuffling (and hence has potential visibility of the encrypted content of all the blocks at the server), and the information necessary for executing a search over the primary and secondary indexes (i.e., her encoding function, the identifier of the blocks visited by the most recent access). However, an authorized user cannot identify the target of searches performed by other users and she cannot infer the changes in the access control policy over resources that she cannot access.

To infer the target of a search operation executed by a different user, the attacker could exploit her knowledge on the identifiers of the blocks visited by a previous access. However, for repeated accesses, we keep track of the identifiers of the blocks visited along the path to the target, to covers, and of repeated accesses. Furthermore, each leaf node stores multiple encoded values, which correspond to index values that are not close to each other since the encoding function is not order-preserving. Hence, the attacker cannot gain any information about the target of the last access. To reconstruct the content of the outsourced relation, and hence identify the target of accesses performed by other users, the attacker could also exploit her knowledge of the encryption key used by the data owner to wrap blocks. In fact, by downloading all the accessed blocks for each access, she could nullify their shuffling. However, this would require the user to download the whole (primary and secondary) index after each access, which seems impracticable.

Since the removal of a resources (or of a privilege) does not cause the deletion of the corresponding tuple in the primary and secondary indexes, but these tuples are re-encrypted with a new key, only users authorized for the resource can detect the change. However, a user cannot distinguish between the removal of a resource that she is authorized to access and the revoke of her privilege over the same resource (as discussed in Section 7). Similarly, thanks to the adoption of probabilistic splits, the insertion of a new resource or the grant of a privilege for a resource can be detected only by users authorized for the resource. Also in this case, however, a user cannot distinguish between the insertion of a new resource for which she is authorized and the grant of her privilege for the same resource. We note however that a user u_i who can access a resource r that is subject to a grant or a revoke operation for a different user

u_j can detect such a policy change, even if she cannot identify u_j. Indeed, r is re-encrypted to enforce the grant or the revoke operation. We note that a user can identify these changes in the set of outsourced resources and in the access control policy only if she keeps a copy of the resources that she is authorized to access and compares her copy with the one stored at the server.

8.3. Performance and economic evaluation

To analyze the overhead caused by the adoption of our approach, we evaluated the performance and economic costs of the adoption of our protection techniques.

Performance evaluation. The performance of the system is measured as the average response time experienced by an authorized client when submitting an access request. To assess the performance of our access algorithm, we configured the primary index and the secondary index as 3-layer unchained $B+$-trees with fan-out 512, both of them built on a numerical candidate key. We did not vary the fan-out of our indexes since system configurations providing a primary index and a secondary index with fixed heights and different fan-outs exhibit similar average response times for the client request. Also, varying the number of authorized users and the size of the access control lists do not significantly influence the performance of the system, as long as the fan-out of the secondary index is chosen to be reasonably large. We set the size of the internal and leaf blocks (nodes) to 8 KiB and 16 KiB, respectively, for both the primary and the secondary index and we fixed *num_cover* to 1 (i.e., two additional searches are executed for each access request, one is the cover and one is a repeated search [9]). The hardware used in the experiments included a client machine with an Intel Core i5–2520M CPU at 2.5 GHz, L3–3 MiB, 8 GiB RAM DDR3 1066, running an Arch Linux OS. The server machine runs an Intel Core i7–920 CPU at 2.6 GHz, L3–8 MiB, 12 GiB, RAM DDR3 1066, 120 GB SSD disk running an Ubuntu OS. The network environment was configured through the `NetEm` suite for Linux operating systems to emulate a typical WAN interactive traffic with a round-trip time modeled as a normal distribution with mean of 100 ms and standard deviation of 2.5 ms. Our experiments show that the latency of the network is the factor with the greatest impact in a large-bandwidth LAN/WAN scenario. This result confirms the performance analysis in [9], where we also showed the cost of CPU and disk. The performance figures obtained for accessing the secondary and the primary indexes looking for a value exhibit an average value equal to 750 ms, which compares favorably with the response time of 630 ms experienced by the client when accessing two plain encrypted indexes (i.e., without shuffling). These results are coherent with the fact that, at coarse-level, our approach is based on two consecutive accesses to two shuffle indexes (i.e., the overall response time of our solution is comparable to the response time experienced by two accesses to two shuffle indexes [9] with the same height and size of the blocks). We can then conclude that the support for access control does not add significant overhead and does not affect the performance of the shuffle index.

The response time we obtained for accessing the secondary and primary index during the execution of a range query is between 752 ms and $L{\cdot}750$ ms (on average), where L is the number of resources that the requesting user is authorized to access in the range specified in her query. This is due to the fact that the user needs to search for each value that she can access in the range of interest, because the encoding functions used for the secondary and primary indexes distribute the original values uniformly among the leaves of the data structures.

To analyze the performance of our primary and secondary index structures also in case the data collection and/or users privileges change, we implemented the approach proposed in [9] to make the insertion

	Primary index leaf block size (KiB)						
Cost (USD)	**16**	**32**	**64**	**128**	**256**	**512**	**1024**
$Cost_{\texttt{storage}}$	73.60	78.20	87.40	105.80	141.40	211.80	352.60
$Cost_{\texttt{access}}$	97.20	97.20	97.20	97.20	97.20	97.20	97.20
$Cost_{\texttt{out}}$	16.09	20.11	28.16	44.25	76.44	140.80	269.50
$Cost_{\texttt{total}}$	*186.89*	*195.51*	*212.76*	*247.25*	*315.04*	*449.81*	*719.36*

Figure 11. Costs (USD) of data storage, access, and transfer per month, depending on the size of the primary index leaf blocks and of the fixed size of the secondary index (30TiB)

and removal of tuples in a shuffle index indistinguishable from read accesses. The performance overhead caused by the support of insertion and removal operations is $\approx$2.7 ms$\cdot$(*num_cover*+1)=5.4 ms on average for each (read, insert, delete) access operation.

Economic evaluation. The price lists of most cloud service providers present three cost components (we take the February 2017 prices of Amazon S3 as a reference; similar pricing schemes are used by most providers in the cloud market): *1)* $Cost_{\texttt{storage}}$, monthly cost of the stored data (2.3 USD/TB per month, for less than 50 TB, 2.2 USD/TB per month, for between 50 TB and 500 TB, and 2.1 USD/TB per month, for more than 500 TB); *2)* $Cost_{\texttt{access}}$, cost of the access requests ($Cost_{\texttt{PUT}}$=5 USD per million `PUT` requests, and $Cost_{\texttt{GET}}$=0.4 USD per million `GET` requests); and *3)* $Cost_{\texttt{out}}$, cost of the data transferred out of the server ($Cost_{\texttt{perTib}}$=90 USD/TB, for transferring up to 10 TB; $Cost_{\texttt{perTib}}$=85 USD/TB for transferring between 11 TB and 50 TB; $Cost_{\texttt{perTib}}$=70 USD/TB, for transferring between 50 TB and 150 TB; $Cost_{\texttt{perTib}}$=50 USD/TB, for transferring between 150 TB and 500 TB). Sending data to the server or deleting data is free of charge.

The total monthly cost $Cost_{\texttt{total}}$ for outsourcing the management of a set of resources using our shuffle index is computed as $Cost_{\texttt{total}} = Cost_{\texttt{storage}} + Cost_{\texttt{access}} + Cost_{\texttt{out}}$. In particular, the storage cost ($Cost_{\texttt{storage}}$) and the access cost ($Cost_{\texttt{out}}$) will split up in several tiers depending on the amount of storage used during a month and the total number of accesses to the secondary and the primary indexes (as this entails the amount of bandwidth used for every network data transfers out of the cloud provider). We now present an example to better detail how these three cost components are computed.

Consider a dataset organized in a secondary and a primary index, both with height *height*=3, fan-out F=512, and *num_cover*=1. Assume that the set $\mathcal{U}$ of users includes 50 subjects and that access control lists include (on average) $0.3\cdot|\mathcal{U}|$ users each. We assume internal nodes to be stored in blocks of 8 KiB, leaf nodes of the secondary index to be stored in blocks of 16 KiB, and leaf nodes of the primary index to be stored in blocks of size varying in the set $\{16\,\text{KiB}, 32\,\text{KiB}, 64\,\text{KiB}, 128\,\text{KiB}, 256\,\text{KiB}, 512\,\text{KiB}, 1\,\text{MiB}\}$ (to better accommodate resources). The size of the primary index will therefore depend on the size of its leaf blocks and will vary in the set $\{2\text{TiB}, 4\text{TiB}, 8\,\text{TiB}, 16\,\text{TiB}, 32\,\text{TiB}, 64\,\text{TiB}, 128\,\text{TiB}\}$. The size of the secondary index will depend on the number and cardinality of the access control lists and, in our running example, will amount to approximately 30 TiB.

The monthly cost $Cost_{\texttt{storage}}$ for the usage of cloud storage crosses more than one tier. Figure 11 shows that such a cost varies from 73.6 USD to 352.6 USD, depending on the size of the primary index leaf blocks and the fixed size of the secondary index.

The cost $Cost_{\texttt{access}}$ of the accesses takes into account the number and type of operations requested to the cloud provider and is computed as the product of the number η of accesses by the cost $C_{\texttt{a}}$ of a single access , that is, $Cost_{\texttt{access}}$=$\eta\cdot C_{\texttt{a}}$, where $C_{\texttt{a}} = 2\cdot height\cdot(num\_cover+2)\cdot(Cost_{\texttt{GET}} + Cost_{\texttt{PUT}})$ USD per millions of accesses. Indeed, for each search operation on the secondary and primary index,

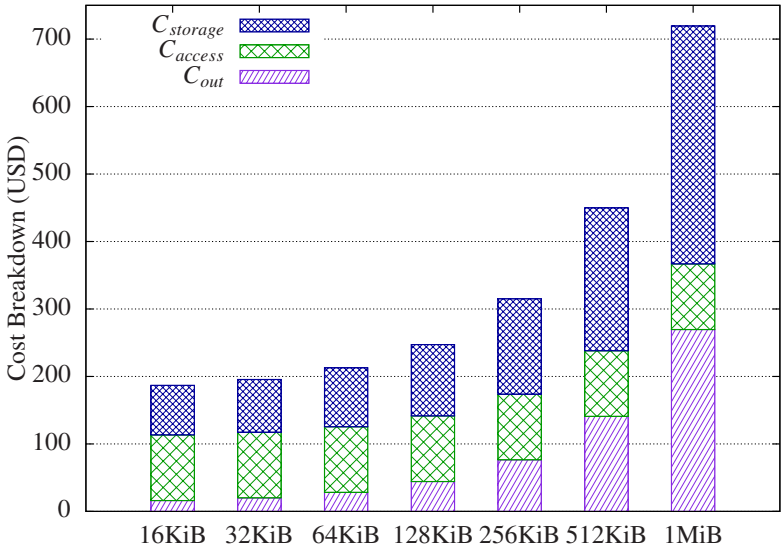

Figure 12. Total monthly cost (USD) varying the size of the primary index leaf blocks

we access $2{\cdot}height{\cdot}(num\_cover{+}2)$ blocks because we visit the target path, *num_cover* cover paths, and one repeated access. Each path includes *height* nodes, both in the primary and in the secondary index. Each block is first downloaded and then uploaded after shuffling, hence we pay a `GET` and a `PUT` request for each accessed block. In our example, $C_{\texttt{access}}{=}97.2$ USD per millions of accesses.

The cost $Cost_{\texttt{out}}$ of bandwidth usage is computed as the product of the number η of accesses by the cost $C_{\texttt{b}}$ of transferring out of the cloud server the volume of data implied by each access, that is, $Cost_{\texttt{out}}{=}\eta{\cdot}C_{\texttt{b}}$, where $C_{\texttt{b}}{=}\ (num\_cover+2){\cdot}(\ (2{\cdot}\ height-2)\cdot \texttt{InternalBlockSize} + \texttt{PrimLeafSize}{+}\texttt{SecLeafSize})\cdot Cost_{\texttt{per TiB}})\cdot 10^6$ USD per million of accesses. In fact, $C_{\texttt{b}}$ depends on the number and size of downloaded/uploaded blocks and can cross one or more tiers ($Cost_{\texttt{perTiB}}$), depending on the actual number of access requests. In our example, assuming the highest cost value (i.e., $Cost_{\texttt{per TiB}}{=}90$ USD), the size of the primary index leaf block will mostly influence the amount of $Cost_{\texttt{out}}$. Figure 11 shows that $Cost_{\texttt{out}}$ varies from 16.09 USD to 269.56 USD, per millions of accesses.

Figure 11 illustrates the total monthly cost $Cost_{\texttt{total}}$ of our example, which varies from 186.89 *USD* to 719.36 *USD*. The total monthly cost as well as its components are graphically illustrated in Figure 12. It is interesting to note that $Cost_{\texttt{access}}$ (which is constant as it does not depend on the size of blocks, but only on the height of the shuffle index) dominates $Cost_{\texttt{out}}$ when the size of the leaf blocks of the primary index is less than 512 KiB, while the storage cost $Cost_{\texttt{storage}}$ remains the main contributor to the aggregate cost when the leaf blocks of the primary index has size up to 1 MiB.

The additional cost of a solution featuring the access control mechanism described in this paper compared with the adoption of one shuffle index with no access control restrictions, quickly decreases as the ratio between the size of leaf blocks on the primary index and the size of any other block increases. Considering the configuration illustrated above, the overhead of our solution ranges from 70%, when internal blocks have size 8 KiB and leaf blocks on the primary and secondary index have size 16 KB, down to 11%, when leaf blocks on the primary index have size 1 MiB.

As a concluding remark, we note that our approach for updating the access control policy and for inserting/removing data into/from the primary and secondary index never releases any physical storage on the cloud provider. This implies that the overall cost of our approach may include a cost of $\sim$2 USD (from 2.1 to 2.3 USD) per month for storing each TiB of data that is no more needed by the data owner. This additional cost highly depends on the kind of data stored at the storing server since it influences the size of the leaf blocks of the primary index. We note however that the impact of this additional cost of storage on the total monthly cost quickly decreases as the size of the primary index or the number of accesses increases. Indeed, it represents from 1.07%, in case of leaf blocks of 16 KiB, to 0.28%, in case of leaf blocks of 1024 KiB, of the total cost.

9. Related Work

Several proposals on data outsourcing protect data (content) confidentiality by wrapping a layer of encryption around them, and support query evaluation through indexes (i.e., metadata complementing the outsourced encrypted dataset [5, 21]) or specific cryptographic techniques that allow keyword-based searches (e.g., [22]).

Solutions for protecting access and pattern confidentiality are based on Private Information Retrieval (PIR) techniques or on dynamically allocated data structures, which change the physical location where data are stored at each access (e.g., [7–15, 23]). PIR solutions are computationally expensive and do not protect content confidentiality (e.g., [12, 23]). The Oblivious RAM (ORAM) dynamic structure, originally proposed in [24, 25], has been extensively studied and guarantees content, access, and pattern confidentiality [6, 13–16]. According to these solutions, data are stored in a pyramid-shaped database layout and, to enable data retrieval, each level of the structure is associated with a Bloom filter and a hash function. Access confidentiality is obtained by caching searches and reorganizing the ORAM structure every time the cache becomes full. Such a reorganization entails a significant performance overhead. To make ORAM more practical in real-world scenarios, recent approaches limit the bandwidth overhead paid when accessing the remote storage (e.g., ObliviStore [13], Path ORAM [14], Ring ORAM [16], Onion ORAM [6]). Such an achievement provides considerable benefits when the computational cost of the operations executed by the client on the retrieved data is dominated by the transfer time of the data from the server to the client (and vice versa). Significant bandwidth savings are achieved at the cost of trusting the storage provider for performing a limited amount of computation over encrypted data (exclusive ORs in [16] and computations on encrypted data by means of an additively homomorphic encryption scheme in [6]) to limit the amount of data to be sent to the client at each access. An effective alternative to ORAM-based structures is represented by tree-based, dynamically allocated structures, which were proven to provide a good trade-off between access privacy and performance (e.g., [7–11]). In particular, the shuffle index has first been proposed in [7] and subsequently extended to support concurrent accesses from different users [8], to operate in a distributed scenario characterized by the presence of multiple (three) storage servers [10], and to support insertion and removal of tuples in the outsourced relation [9]. A proper tuning of the shuffle index parameters permits to obtain communication overheads comparable to (or even better than) ORAM-based solutions, thus providing a valid alternative to them even when taking into account communication costs. An advantage of the shuffle index proposal is represented by the fact that its bandwidth overhead depends only on the height of the tree structure, while it is independent from the size of the indexed data, or the number of accesses performed. Also, the shuffle index requires only a minimal storage at the client-side (i.e., the block address of the root node of

the tree). All these solutions rely on the implicit assumption that any user is either authorized to access all the tuples stored in the data structure or none of them.

A line of research related to our proposal addresses the problem of enforcing access control restrictions over outsourced data. These solutions are based on the idea that the data themselves should enforce the access control policy. Current approaches follow two different strategies: attribute-based encryption (e.g., [26–34]) and selective encryption (e.g., [19, 35]). Attribute-Based Encryption (ABE) [32] provides fine-grained access control over encrypted data, enforcing an authorization policy defined on attributes associated with resources or with users. Depending on wether attributes and policies are associated with resources or users, we distinguish between Ciphertext-Policy ABE (CP-ABE) [26] and Key-Policy ABE (KP-ABE) [28] systems. According to CP-ABE, each user is associated with a set of descriptive attributes, which are also embedded in her private key. Each resource is instead associated with an authorization policy, which is embedded in its encrypted representation. Only users whose attributes satisfy the authorization policy can decrypt the resource. In KP-ABE the private key of each user embeds her authorization policy, while the encrypted representation of each resource embeds a set of attributes. Each user can decrypt only the resources that satisfy her access policy. Different ABE access structures (e.g., tree-based [26–28, 31] and matrix-based [29, 30, 33] structures) provide flexibility in the definition of access control policies. The security of ABE constructions depends on a variety of mathematical problems. Also, the ciphertext size and encryption/decryption time increase with the complexity of the access structure. To mitigate these drawbacks, in [33, 34] the authors investigate the idea of delegating to external cloud providers the most demanding portion of the required computations and the storage of large ciphertexts. However, modern ABE-schemes do not support the management of write privileges, and cause considerable overheads for the enforcement of revoke operations. They seem not suited for a combined adoption with data structures aimed at protecting access confidentiality. A different line of works aimed at enforcing access control over encrypted outsourced data relies on selective encryption and proper key management [19, 35, 36]. These schemes translate the authorization policy of the data owner into an equivalent encryption policy, managing the generation and distribution of encryption keys to users and the use of keys for resource encryption in such a way to enable each user to decrypt all and only the resources she is authorized to read [18, 19]. This approach has then been extended to operate in a scenario characterized by the presence of multiple data owners [37] and to support the enforcement of write privileges [35]. The problem of efficiently managing revoke operations has also been addressed, proposing a solution that does not require the complete re-encryption of the revoked resource, but only of a small portion of the same [38].

The enforcement of access control restrictions over the data stored in a shuffle index has been first proposed in [1], exploiting the ideas and the design principles of selective encryption [18, 19]. In this paper, we considerably extended this prior work by providing support for data and policy updates, as well as for range queries and indexes defined over non-key attributes.

10. Conclusions

We have presented an approach to extend the shuffle index with access control. Our extended shuffle index provides guarantees of access confidentiality while enabling data owners to regulate access to their data selectively granting visibility to users. Also, like the original proposal, it has limited performance and economic overhead. We have shown how the proposed approach can efficiently support the evaluation of range queries and, even if it naturally manages resources collection organized according

to a key attribute, how it can operate when index values have multiple occurrences. Finally, we have discussed how to manage the evolution of both the policy (i.e., grant and revoke operations), and the set of resources stored in the shuffle index (i.e., the insertion, removal, and update of resources).

Acknowledgements

This work was supported in part by: the EC within the FP7 under grant agreement 312797 (ABC4EU) and within the H2020 under grant agreement 644579 (ESCUDO-CLOUD).

References

[1] S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, G. Pelosi and P. Samarati, Access control for the shuffle index, in: *Proc. of DBSec*, Trento, Italy, 2016.
[2] R. Jhawar and V. Piuri, Fault tolerance management in IaaS clouds, in: *Proc. of ESTEL*, Rome, Italy, 2012.
[3] R. Jhawar and V. Piuri, Fault tolerance and resilience in cloud computing environments, in: *Computer and Information Security Handbook, 2nd Edition*, J. Vacca ed, Morgan Kaufmann, 2013.
[4] R. Jhawar, V. Piuri and P. Samarati, Supporting security requirements for resource management in cloud computing, in: *Proc. of CSE*, Paphos, Cyprus, 2012.
[5] P. Samarati and S. De Capitani di Vimercati, Cloud security: Issues and concerns, in: *Encyclopedia on Cloud Computing*, S. Murugesan and I. Bojanova, eds, Wiley, 2016.
[6] S. Devadas, M. van Dijk, C.W. Fletcher, L. Ren, E. Shi and D. Wichs, Onion ORAM: A constant bandwidth blowup oblivious RAM, in: *Proc. of TCC*, Tel Aviv, Israel, 2016.
[7] S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, G. Pelosi and P. Samarati, Efficient and private access to outsourced data, in: *Proc. of ICDCS*, Minneapolis, MN, 2011.
[8] S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, G. Pelosi and P. Samarati, Supporting concurrency and multiple indexes in private access to outsourced data, *JCS* **21**(3) (2013), 425–461.
[9] S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, G. Pelosi and P. Samarati, Shuffle index: Efficient and private access to outsourced data, *ACM TOS* **11**(4) (2015), 1–55.
[10] S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, G. Pelosi and P. Samarati, Three-server swapping for access confidentiality, *IEEE TCC* (2016), pre-print.
[11] P. Lin and K. Candan, Hiding traversal of tree structured data from untrusted data stores, in: *Proc. of WOSIS*, Porto, Portugal, 2004.
[12] R. Ostrovsky and W.E. Skeith, III, A survey of single-database private information retrieval: Techniques and applications, in: *Proc. of PKC*, Beijing, China, 2007.
[13] E. Stefanov and E. Shi, ObliviStore: High performance oblivious cloud storage, in: *Proc. of IEEE S&P*, San Francisco, CA, 2013.
[14] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu and S. Devadas, Path ORAM: An extremely simple Oblivious RAM protocol, in: *Proc. of CCS*, Berlin, Germany, 2013.
[15] P. Williams, R. Sion and B. Carbunar, Building castles out of mud: Practical access pattern privacy and correctness on untrusted storage, in: *Proc. of CCS*, Alexandria, VA, 2008.
[16] L. Ren, C. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. Van Dijk and S. Devadas, Constants count: Practical improvements to Oblivious RAM, in: *Proc. of Usenix*, Washington, DC, 2015.
[17] M.S. Islam, M. Kuzu and M. Kantarcioglu, Inference attack against encrypted range queries on outsourced databases, in: *Proc. of Codaspy*, San Antonio, TX, USA, 2014.
[18] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi and P. Samarati, Over-encryption: Management of access control evolution on outsourced data, in: *Proc. of VLDB*, Vienna, Austria, 2007.
[19] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi and P. Samarati, Encryption policies for regulating access to outsourced data, *ACM TODS* **35**(2) (2010), 1–46.
[20] M. Atallah, M. Blanton, N. Fazio and K. Frikken, Dynamic and efficient key management for access hierarchies, *ACM TISSEC* **12**(3) (2009), 1–43.
[21] H. Hacigümüs, B. Iyer, S. Mehrotra and C. Li, Executing SQL over encrypted data in the database-service-provider model, in: *Proc. of SIGMOD*, Madison, WI, 2002.
[22] C. Wang, N. Cao, K. Ren and W. Lou, Enabling secure and efficient ranked keyword search over outsourced cloud data, *IEEE TPDS* **23**(8) (2012), 1467–1479.

[23] C. Cachin, S. Micali and M. Stadler, Computationally private information retrieval with polylogarithmic communication, in: *Proc. of EUROCRYPT*, Prague, Czech Republic, 1999.
[24] O. Goldreich, Towards a theory of software protection and simulation by oblivious RAMs, in: *Proc. of STOC*, New York, NY, 1987.
[25] R. Ostrovsky, Efficient computation on oblivious RAMs, in: *Proc. of STOC*, Baltimore, MD, 1990.
[26] J. Bethencourt, A. Sahai and B. Waters, Ciphertext-policy Attribute-Based Encryption, in: *Proc. of IEEE S&P*, Oakland, CA, 2007.
[27] V. Goyal, A. Jain, O. Pandey and A. Sahai, Bounded ciphertext policy attribute based encryption, in: *Proc. of ICALP*, Reykjavik, Iceland, 2008.
[28] V. Goyal, O. Pandey, A. Sahai and B. Waters, Attribute-based encryption for fine-grained access control of encrypted data, in: *Proc. of CCS*, Alexandria, VA, 2006.
[29] M. Green, S. Hohenberger and B. Waters, Outsourcing the decryption of ABE ciphertexts, in: *Proc. of USENIX*, San Francisco, CA, 2011.
[30] A.B. Lewko, T. Okamoto, A. Sahai, K. Takashima and B. Waters, Fully secure functional encryption: Attribute-based encryption and (hierarchical) inner product encryption, in: *Proc. of EUROCRYPT*, French Riviera, France, 2010.
[31] R. Ostrovsky, A. Sahai and B. Waters, Attribute-based encryption with non-monotonic access structures, in: *Proc. of CCS*, Alexandria, VA, 2007.
[32] A. Sahai and B. Waters, Fuzzy identity-based encryption, in: *Proc. of EUROCRYPT*, Aarhus, Denmark, 2005.
[33] B. Waters, Ciphertext-policy attribute-based encryption: An expressive, efficient, and provably secure realization, in: *Proc. of PKC*, Taormina, Italy, 2011.
[34] K. Yang, X. Jia, K. Ren, B. Zhang and R. Xie, DAC-MACS: effective data access control for multiauthority cloud storage systems, *IEEE TIFS* **8**(11) (2013), 1790–1801.
[35] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, G. Livraga, S. Paraboschi and P. Samarati, Enforcing dynamic write privileges in data outsourcing, *Computers & Security* **39** (2013), 47–63.
[36] M. Raykova, H. Zhao and S.M. Bellovin, Privacy enhanced access control for outsourced data sharing, in: *Proc. of FC*, Kralendijk, Bonaire, 2012.
[37] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, G. Pelosi and P. Samarati, Encryption-based policy enforcement for cloud storage, in: *Proc. of SPCC*, Genova, Italy, 2010.
[38] E. Bacis, S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, M. Rosa and P. Samarati, Mix&Slice: Efficient access revocation in the cloud, in: *Proc. of CCS*, Vienna, Austria, 2016.