Access Control for the Shuffle Index

Sabrina de Capitani Di Vimercati, Sara Foresti, Stefano Paraboschi, Gerardo Pelosi, Pierangela Samarati

Access Control for the Shuffle Index

Sabrina De Capitani di Vimercati[1], Sara Foresti[1], Stefano Paraboschi[2], Gerardo Pelosi[3], and Pierangela Samarati[1]

[1] Università degli Studi di Milano, Italy – *firstname.lastname*@unimi.it
[2] Università degli Studi di Bergamo, Italy – parabosc@unibg.it
[3] Politecnico di Milano, Italy – gerardo.pelosi@polimi.it

Abstract. The shuffle index provides confidentiality guarantees for accesses to externally outsourced data. In this paper, we extend the shuffle index with support for access control, that is, for enforcing authorizations on data. Our approach bases on the use of selective encryption and on the organization of data and authorizations in two shuffle indexes. Our proposal enables owners to regulate access to their data supporting authorizations allowing different users access to different portions of the data, while at the same time guaranteeing confidentiality of access.

1 Introduction

The rapid advancement in ICT and the increasing adoption of cloud computing paradigms 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., [11]), 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., [12,15]). The research and industrial community have recognized these issues and investigated different aspects of the problem, with considerable attention paid to the need to maintain information confidential to the providers themselves that, even if trustworthy to provide the service, should not be allowed visibility over the stored data. In addition to the need to protect confidentiality of the stored data (*content confidentiality*), recent proposals have been devoting attention to the need to protect 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 (this latter also referred to as pattern confidentiality). There are several reasons for which access confidentiality should be protected, including the simple 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 itself. Among the recent proposals specifically considering the access confidentiality problem in database management scenarios (and therefore with attention to efficiency and functionality guarantees that should be provided) is the *shuffle index* [5]. The shuffle index provides an index-based hierarchical organization of the data

supporting efficient and effective access execution and provides 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 [6], 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 confidentiality (of content and access) 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* [2,3]) have emerged, they cannot be simply applied in conjunction with the shuffle index, given the specific characteristics of the index and its access execution, as well as the need to ensure access confidentiality guarantees.

In this paper, we provide an approach to support access control over the shuffle index (Section 2) 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 (Section 3 and Section 4). We show that our proposal correctly enforces the access control policy established by the data owner and has limited performance overhead (Section 5).

2 Shuffle Index

The *shuffle index* [5] 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 node stores up to $F - 1$ ordered values $v_1, v_2, \ldots, v_q$, 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 $v \geq v_q$; its i-the child ($i = 2, \ldots, q$) is the root of the subtree including all values $v_{i-1} \leq v < v_i$. 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 storing a node corresponds to the logical identifier of the node itself. 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 nodes content 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 the abstract (c), logical (d), and physical (e) level, respectively, of a shuffle index storing the 19 tuples in Figure 1(a), indexed according to the values of attribute I. Actual tuples are stored in the leaves of the index structure, where, for simplicity, we however 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 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),

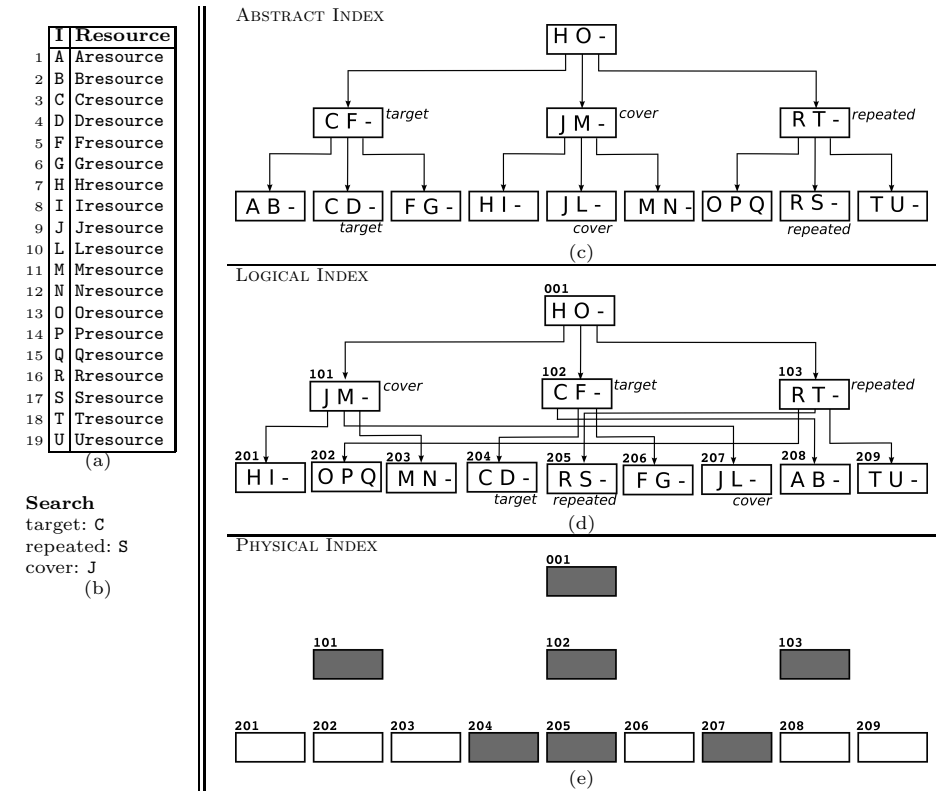

Fig. 1. An example of a relation (a), an access over it (b), and of abstract (c), logical (d) and physical (e) shuffle index

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 the block content or on the roles (target, repeated, cover) of the blocks. 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 the fact that the content of node X is moved to Y).

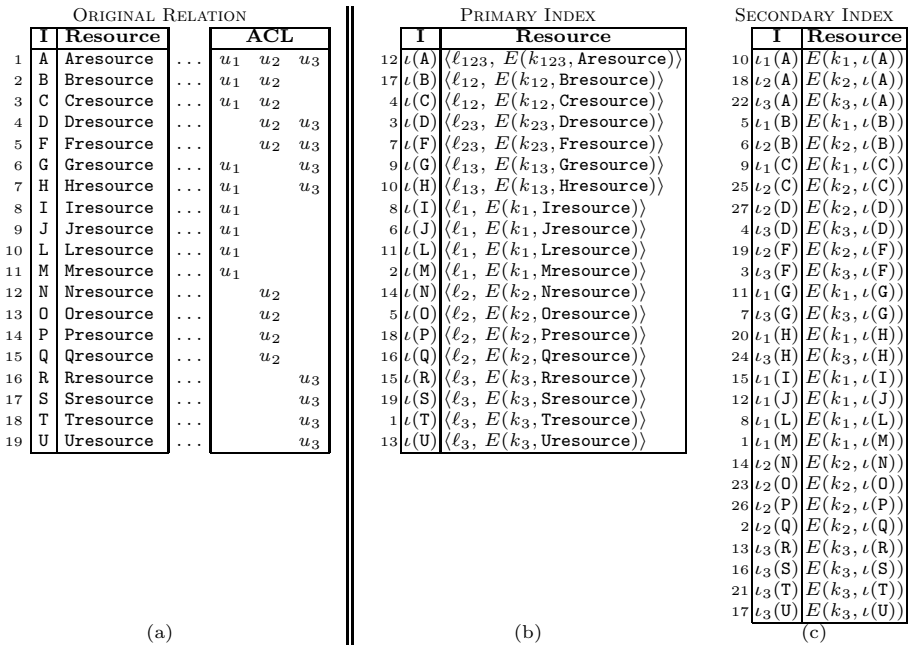

Fig. 2. Relation of Figure 1(a) with *acl*s associated with its resources (a), relation for the primary index (b), relation for the secondary index (c)

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 of Figure 1(a), considering three users (u_1, u_2, u_3). The figure reports, for each tuple r in the dataset, the corresponding $acl(r)$, that is the set of users authorized to access it. (Note that authorizations do not explicitly report the access privileges, which is considered to be 'read', since we assume access by users to be read-only, with write operations reserved to the owner.) When clear from the context, with a slight abuse of notation, in the following 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, $acl(\texttt{A})=\{u_1,u_2,u_3\}$, while $acl(\texttt{B})=\{u_1,u_2\}$.

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 would be created for index value A and the corresponding resource

`Aresource`, 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 user sets (i.e., *acl*s), with a user u_i knowing her encryption key k_i as well as those of the *acl*s 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), `Aresource` would be encrypted with key k_{123} known to all users while `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 with 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. 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 essentially 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 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;* 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 encryption of each resource with a key known only to authorized users. To apply selective encryption, we then define a key set for the encryption policy as follows.

Definition 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 $\mathcal{K}=\{k_i \mid u_i \in \mathcal{U}\} \cup \{k_{i_1,\ldots,i_n} \mid \exists r \in \mathcal{R}, \{u_{i_1},\ldots,u_{i_n}\} = acl(r)\}$ 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 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 [3].

We are now ready to define the first index used by our approach. This first 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 (Primary Index – Data). *Let $\mathcal{R}(I,Resource)$ be a relation, I be the indexing attribute, ι be an encoding function for I computable only by the data owner, 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 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 $r[Resource]$ 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 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 $hash(v,k_o)$). 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).

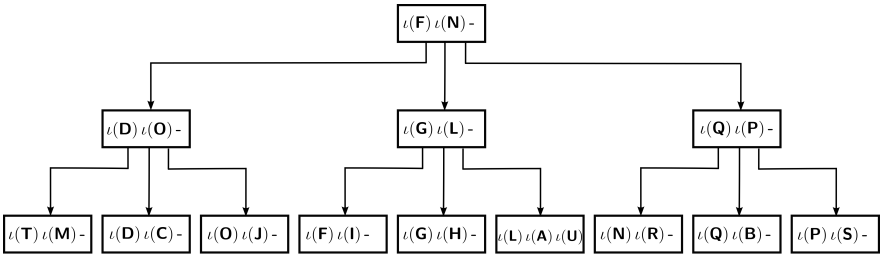

Fig. 3. Primary shuffle index for the relation in Figure 2(b)

Building the index on the encoded values provides protection of the original index values, and their order relationship, against users and storing server that observe the index on the encoded values. In fact, the encoding is non-invertible (hence the encoded values do not leak any information on the original values), and destroys the original ordering (hence 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 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 knowledge is provided to each user u_i encrypted with the user key k_i (so to make it non intelligible to other users and to the server) and is indexed with a user-based encoding, so to provide a distinct mapping for every user u_i, which can be computed only by u_i. The second index of our approach is therefore a *secondary* index providing user-based mapping as follows.

Definition 4 (Secondary Index – User-based Mapping). *Let $\mathcal{R}(I,Resource)$ be a relation, I be the indexing attribute, $\mathcal{P}$ be a primary index for $\mathcal{R}$ over I with encoding function ι, $\mathcal{U}$ be a set of users, $\{\iota_i \mid u_i \in \mathcal{U}\}$ be a set of encoding functions for I such that ι_i is computable only by user u_i and by the data owner, 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 over relation $\mathcal{S}(I,Resource)$ having a*

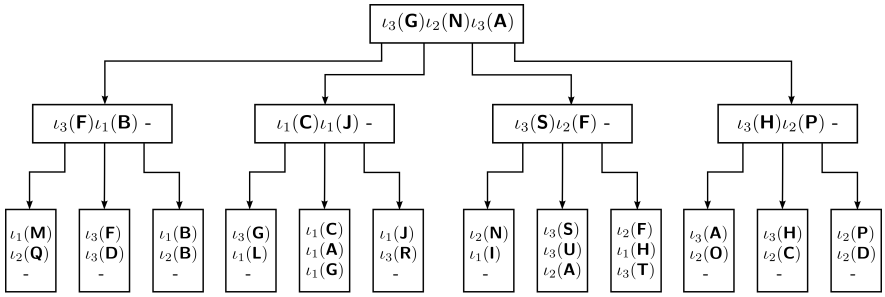

Fig. 4. Secondary shuffle index for the relation in Figure 2(c)

tuple s for each pair $\langle r,u_i\rangle$, $r\in\mathcal{R}$ 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., $\iota_i(v)=hash(v,k_i)$). 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. Notice how, once again, the encoding does not convey any information on the ordering of the original index values. Note that the secondary index has a larger number of tuples than the original index, since the encoding of an original index value is encrypted as many times as the number of users who can access it. For instance, in our example, there are three instances of $\iota(\mathtt{A})$. Figure 4 illustrates the tree structure for the index in Figure 2(c). We note however that the secondary index is very slim as the resources are simply the encryption, with the key of a user, of the owner encoding. While in our examples, for simplicity, we maintain the same topology, the structure of the secondary index is independent from the structure of the primary index, meaning that they may have different fan-out and height.

Note that 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 allowed to access. Figure 5(a-b) illustrates a possible logical organization for the primary and secondary index of our example, 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 which remain non intelligible to u_1 as they are encoded with the mapping of other users (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 her. For instance, consider the primary index in Figure 5(b). User u_1, being authorized for `B` will know that ι(`B`) is the corresponding encoding. At the same time, however, ι(`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 Access Execution

We now illustrate how the two indexes described in the previous section are jointly used for accessing a tuple of interest. To retrieve a tuple in $\mathcal{R}$ with value v for I, a user u_i would need to perform the following steps:

1. compute the user-based mapping $\iota_i(v)$=$hash(v,k_i)$;
2. search $\iota_i(v)$ in the secondary index $\mathcal{S}$, retrieving the corresponding encoded value $\iota(v)$;
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 ι_1(`C`)=$hash$(`C`,k_1) and then searches it in the secondary index in Figure 5(a). The search returns block S205, from which ι(`C`) is retrieved. Hence, u_1 searches ι(`C`) in the primary index in Figure 5(b). The search returns block P202, from which u_1 can retrieve resource `Cresource`.

Note that the steps above assume the searched value to be present in the index. If the value is not present in the secondary index, its user-based mapping 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 present in the dataset but for which the searching user is not authorized, present to the searching user the same observable as the search for a missing value (hence not disclosing anything to the user about values she is not authorized to access).

The steps above simply illustrate how to retrieve a target value. However, both the primary and the secondary index 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 completely independent in the two indexes. The only dependency among the two indexes is the fact that - clearly - the target to be searched in the primary index is the tuple retrieved by the search on the secondary index.

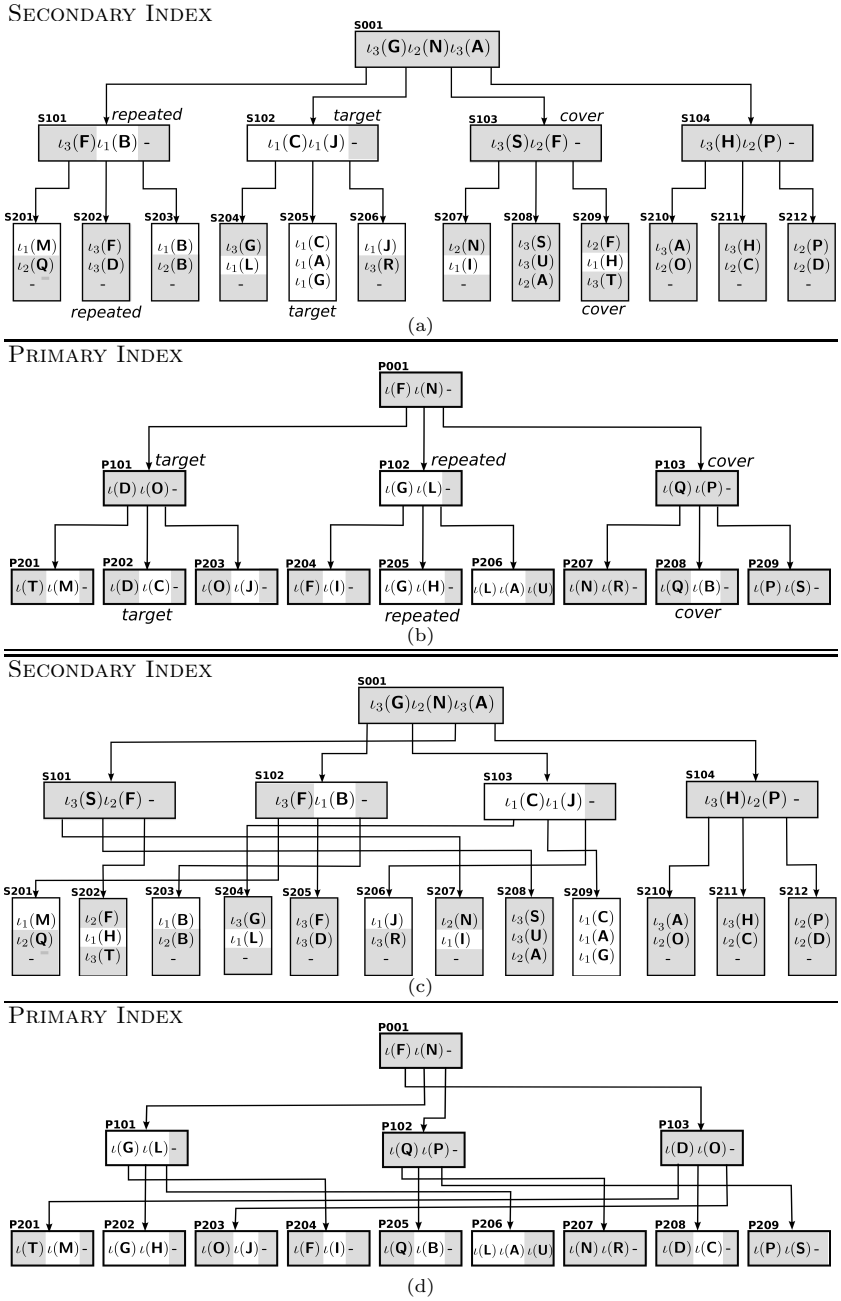

Fig. 5. Secondary and primary index before (a-b) and after (c-d) the access by u_1 over `C`. Secondary index: i) cover: $\iota_2(\mathtt{F})$, ii) repeated access: [S001,S101,S202], iii) shuffling: S101→S102, S102→S103, S103→S101, S202→S205, S205→S209, S209→S202. Primary index: i) cover: $\iota(\mathtt{Q})$, 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

Covers, repeated searches, and shuffling on the primary and secondary index 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, as follows.

- *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., 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 the previous user. Although such a leakage would be only on encoded values, we avoid it simply by 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 hers without revealing to the user the target of the previous search (which might have been performed by others).
- *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 correct execution of the shuffling operation.

The pseudocode of the algorithm accessing and managing the primary and the secondary index is reported in Appendix.

Figure 5(a-b) illustrates an example of access execution for search of value `C` by user u_1, assuming $\iota_2(\texttt{F})$ as cover and path [S001,S101,S202] as repeated access for the secondary index, and $\iota(\texttt{Q})$ as cover and path [P001,P102,P205] as a 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, and S209→S202; for the primary index as P101→P103, P102→P101, P103→P102, P202→P208, P205→P202, P208→P205.

5 Analysis

We discuss the protection guarantees (i.e., the correct enforcement of authorizations and the protection of access and pattern confidentiality) and the performance of our approach.

Access control enforcement. To demonstrate that the primary and secondary indexes described in Section 3 guarantee the correct enforcement of the access control policy, we need to prove that each user u_i can access all and only the resources and index values in $\mathcal{R}$ she is authorized to access. Formally, $\forall u_i \in \mathcal{U}$: *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 $u_i \in acl(r)$.

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 $hash(r[I], k_i)$ and u_i knows key k_i, by Definition 2. Tuple s exists and belongs to $\mathcal{S}$ by Definition 4. Tuple p exists and belongs to $\mathcal{P}$ by Definition 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 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 nodes content to enable shuffling, and the co-domain of the encoding functions.

Consider now a user u_i s.t. $acl(r)=\{u_{i_1}, \ldots, u_{i_n}\}$ and $u_i \notin \{u_{i_1}, \ldots, u_{i_n}\}$. 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 the tuple since it is encrypted with a key k_X (Definition 3) that u_i does not know. In fact, by Definition 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 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 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).

Access confidentiality. 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 [7]. 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 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.
We now consider a user as our observer. A user can observe the blocks accessed by another user in a previous access (for repeated accesses), but she cannot identify the target of the access. In fact, this set of blocks includes the target, covers, and repeated accesses. Furthermore, each leaf stores multiple encoded values, which correspond to index values that are not close to each other since the encoding function is not order-preserving. A user can also possibly trace shuffling operations, but this would require her to download the whole index at each access.

Performance evaluation. The performance of the system is assessed as the average response time experienced by an authorized client when submitting an access request. 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. Moreover, 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. Our experiments show that the latency of the network is the factor with the greatest impact in a large-bandwidth LAN/WAN scenario. To assess the performance of our 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 of fixed-length to allow the indexing of more than 200K different values. The size of the blocks (nodes) of each index was 8KiB. The hardware used in the experiments included a client machine with an Intel Core i5-2520M CPU at 2.5GHz, L33MiB, 8GiB RAM DDR3 1066, running an Arch Linux OS. The server machine run an Intel Core i7-920 CPU at 2.6GHz, L38MiB, 12GiB, RAM DDR3 1066, 120GiB 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 100ms and

standard deviation of 2.5ms. The performance figures obtained for accessing the secondary and the primary index exhibit an average value equal to 750ms, which compares favorably with the response time of 630ms experienced by the client when accessing two plain encrypted indexes (i.e., without shuffling).

6 Related Work

Classical works 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) or through specific cryptographic techniques that support keyword-based searches (e.g., [10,18]). 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., [1,5,6,7,8,13,14,16,17,19]). PIR solutions are computationally expensive and do not protect content confidentiality (e.g., [1,14]). The Oblivious RAM (ORAM) dynamic structure, which has been extensively studied, guarantees content, access, and pattern confidentiality (e.g., [19]). While preliminary proposals suffer from high computational and communication overheads, recent attempt to make ORAM more practical in real-world scenarios (e.g., ObliviStore [16] and Path ORAM [17]). Besides ORAM structure, also tree-based dynamically allocated structures have been studied that provide a good trade-off between privacy and performance (e.g., [5,6,7,8,13]). In particular, the shuffle index has first been proposed in [5] and then extended to support concurrent accesses by different users [6], to operate in a distributed scenario characterized by the presence of multiple (three) storage servers [8], and to support insertion and removal of tuples in the outsourced relation [7]. All these solutions, however, are based on the implicit assumption that a user can access either all the tuples in the leaves of the shuffle index or none of them.

A related line of work 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: selective encryption (e.g., [3]), and attribute-based encryption (e.g., [9]). Our work extends selective encryption proposals since we combine the shuffle index with selective encryption to enable efficient access to the data through a tree-based index, while not revealing to users index values they are not authorized to access [4].

7 Conclusions

We have presented an approach to enrich the shuffle index with access control. The enriched 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 overhead.

Acknowledgements. This work was supported in part by the EC within the 7FP under grant agreement 312797 (ABC4EU) and within the H2020 under grant agreement 644579 (ESCUDO-CLOUD).

References

1. Cachin, C., Micali, S., Stadler, M.: Computationally private information retrieval with polylogarithmic communication. In: Proc. of EUROCRYPT. Prague, Czech Republic (May 1999)
2. De Capitani di Vimercati, S., Foresti, S., Jajodia, S., Paraboschi, S., Samarati, P.: Over-encryption: Management of access control evolution on outsourced data. In: Proc. of VLDB. Vienna, Austria (September 2007)
3. De Capitani di Vimercati, S., Foresti, S., Jajodia, S., Paraboschi, S., Samarati, P.: Encryption policies for regulating access to outsourced data. ACM TODS 35(2), 12:1–12:46 (April 2010)
4. De Capitani di Vimercati, S., Foresti, S., Jajodia, S., Paraboschi, S., Samarati, P.: Private data indexes for selective access to outsourced data. In: Proc. of WPES 2011. Chicago, IL (October 2011)
5. De Capitani di Vimercati, S., Foresti, S., Paraboschi, S., Pelosi, G., Samarati, P.: Efficient and private access to outsourced data. In: Proc. of ICDCS. Minneapolis, MN (June 2011)
6. De Capitani di Vimercati, S., Foresti, S., Paraboschi, S., Pelosi, G., Samarati, P.: Supporting concurrency and multiple indexes in private access to outsourced data. JCS 21(3), 425–461 (2013)
7. De Capitani di Vimercati, S., Foresti, S., Paraboschi, S., Pelosi, G., Samarati, P.: Shuffle index: Efficient and private access to outsourced data. ACM TOS 11(4), 19:1–19:55 (October 2015)
8. De Capitani di Vimercati, S., Foresti, S., Paraboschi, S., Pelosi, G., Samarati, P.: Three-server swapping for access confidentiality. IEEE TCC (2016), pre-print
9. Goyal, V., Pandey, O., Sahai, A., Waters, B.: Attribute-based encryption for fine-grained access control of encrypted data. In: Proc. of CCS. Alexandria, VA (October-November 2006)
10. Hacigümüs, H., Iyer, B., Mehrotra, S., Li, C.: Executing SQL over encrypted data in the database-service-provider model. In: Proc. of SIGMOD. Madison, WI (June 2002)
11. Jhawar, R., Piuri, V.: Fault tolerance management in iaas clouds. In: Proc. of ESTEL. Rome, Italy (October 2012)
12. Jhawar, R., Piuri, V., Samarati, P.: Supporting security requirements for resource management in cloud computing. In: Proc. of CSE. Paphos, Cyprus (December 2012)
13. Lin, P., Candan, K.: Hiding traversal of tree structured data from untrusted data stores. In: Proc. of WOSIS. Porto, Portugal (April 2004)
14. Ostrovsky, R., Skeith, III, W.E.: A survey of single-database private information retrieval: Techniques and applications. In: Proc. of PKC. Beijing, China (April 2007)
15. Samarati, P., De Capitani di Vimercati, S.: Cloud security: Issues and concerns. In: Murugesan, S., Bojanova, I. (eds.) Encyclopedia on Cloud Computing. Wiley (2016)

16. Stefanov, E., Shi, E.: ObliviStore: High performance oblivious cloud storage. In: Proc. of IEEE S&P. San Francisco, CA (May 2013)
17. Stefanov, E., van Dijk, M., Shi, E., Fletcher, C., Ren, L., Yu, X., Devadas, S.: Path ORAM: An extremely simple Oblivious RAM protocol. In: Proc. of CCS. Berlin, Germany (November 2013)
18. Wang, C., Cao, N., Ren, K., Lou, W.: Enabling secure and efficient ranked keyword search over outsourced cloud data. IEEE TPDS 23(8), 1467–1479 (2012)
19. Williams, P., Sion, R., Carbunar, B.: Building castles out of mud: Practical access pattern privacy and correctness on untrusted storage. In: Proc. of CCS. Alexandria, VA (October 2008)

A Access Execution Algorithm

Figure 6 illustrates the algorithm, executed at the client side, searching for a value in the primary and secondary index. The algorithm operates as discussed in Section 4 and relies on function **Search** to access the primary and secondary index structures.

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 randomly chooses *num_cover*+1 values in the domain of the (primary or secondary) index and it downloads from the server the identifiers of the blocks visited by the previous search (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 (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*), 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).

Given the request by user u_i to search for value *target_value*, the algorithm computes the user-based mapping $\iota_i(target\_value)$ and invokes function **Search** to search for such a value in the secondary index (lines 1–4). It decrypts the tuple retrieved by function **Search**, obtaining the encoded value $\iota(target\_value)$ for *target_value* (line 5). If such a value is not NULL (meaning that there is a tuple that u_i can access with index value equal to *target_value*), the algorithm invokes function **Search** over the primary index, looking for $\iota(target\_value)$. It then computes/retrieves the encryption key necessary to decrypt the retrieved resource and decrypts it. It returns the plaintext resource to the user (lines 7–11).

If the result of function **Search** over the secondary index is NULL, the algorithm runs a fake search over the primary index (not to disclose any information to other users and to the server about u_i's privileges) and returns an empty resource to the user (lines 12-14).

```
/* P, S : primary and secondary index */
/* num_cover : number of cover searches */
/* u_i,k_i : user performing the access and her key */
/* hash : non-invertible cryptographic hash function */
INPUT    target_value : value to be searched in the shuffle index
OUTPUT   resource with index value target_value
MAIN
 1: /* Phase 1: compute the user-based mapping ι_i(target_value) */
 2: target_idx := hash(target_value, k_i)
 3: /* Phase 2: search ι_i(target_value) in the secondary index */
 4: s := Search(S,target_idx,num_cover)
 5: target_idx := decrypt s[Resource] with k_i /* encoded value ι(target_value) */
 6: /* Phase 3: search ι(target_value) in the primary index */
 7: if target_idx ≠ NULL then
 8:    p := Search(P,target_idx,num_cover)
 9:    k := retrieve key k with label ℓ, where p[Resource]=⟨ℓ,content⟩
10:    result := decrypt content with k
11:    return(result)
12: else target_idx := randomly choose a value for ι(target_value)
13:    Search(P,target_idx,num_cover)
14:    return(NULL)
SEARCH(T,target_value,num_cover)
 1: repeated_search[0,...,T.height] := download and decrypt the blocks of accesses for T
 2: randomly choose cover_value[1...num_cover+1] for target_value in the co-domain of hash
 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] := id 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
10:       repeated := target, 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 π be a permutation of the identifiers of nodes in Nodes
16:    shuffle nodes in Nodes according to π
17:    update pointers to children of the parents of nodes in Nodes according to π
18:    encrypt and write at the server nodes accessed at iteration level − 1
19:    target := π(target)
20:    cover[i] := π(cover[i]), i=1...num_cover+1
21:    repeated := π(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)
```

Fig. 6. Shuffle index access algorithm