Protecting access confidentiality with data distribution and swapping

Sabrina De Capitani di Vimercati*, Sara Foresti*, Stefano Paraboschi†, Gerardo Pelosi‡, Pierangela Samarati*
**Università degli Studi di Milano, 26013 Crema - Italy Email: firstname.lastname@unimi.it*
†*Università degli Studi di Bergamo, 24044 Dalmine - Italy Email: parabosc@unibg.it*
‡*Politecnico di Milano, 20133 Milano - Italy Email: gerardo.pelosi@polimi.it*

***Abstract*—The protection of the confidentiality of outsourced data is an important problem. A critical aspect is the ability to efficiently access data that are stored in an encrypted format, without giving to the server managing access requests the ability to infer knowledge about the data content of the access executed by the clients. The approaches that have been proposed to solve this problem rely on a continuous rewriting and re-encryption of the accessed data, like the shuffle index that has recently been proposed. We here propose a different approach that uses three independent servers to manage the data structure. The use of three servers is motivated by the increased protection that derives from the use of independent servers compared to the use of a single server. The protection shows to increase in a significant way if a constraint is introduced that at every request an accessed node has to be moved to a different server. The use of three servers permits to keep the accessed data protected even when the servers collude. The protection is evaluated with a probabilistic model that estimates the loss of information that derives from the application of the technique.**

***Keywords*-Access confidentiality, Data distribution, Swapping**

I. Introduction

A recent trend and innovation in the IT scenario has been the increasing adoption of the cloud computing paradigm. Companies can rely on the cloud for data storage and management and then benefit from low costs and high availability. End users can benefit from cloud storage for enjoying availability of data anytime anywhere, even from mobile devices. Together with such a convenience comes however a loss of control of the data (stored and managed by the cloud). The problem of ensuring data confidentiality in data outsourcing and cloud scenarios has received considerable attention by the research and development communities in the last few years and several solutions have been proposed. A simple solution for guaranteeing data confidentiality consists in encrypting data. Modern cryptographic algorithms offer high efficiency and strong protection of data content. As noted by more recent works, simply protecting data content with an encryption layer does not fully solve the confidentiality problem, as *access confidentiality*, that is, the confidentiality of the specific accesses performed on the data, remains at risks. There are several reasons for which access confidentiality may be demanded [1], such as the fact that breaches in access confidentiality may leak information on access profiles of users and, in the end, even on the data themselves, therefore causing breaches in *data confidentiality*.

Several approaches have been recently proposed to protect access confidentiality [1], [2], [3]. While with different variations, such approaches share the common observation that the major problem to be tackled to provide access confidentiality is to break the static correspondence between data and the physical location. Among such proposals, the *shuffle index* [1] provides a key-based hierarchical organization of the data, supporting an efficient and effective access execution (e.g., including support of range operations). In this paper, we build on such an indexing structure and on the idea of dynamically changing, at every access, the physical location of data, and provide a new approach to access confidentiality based on a combination of *data distribution* and *swapping*. The idea of applying data distribution for confidentiality protection is in line with the evolution of the market, with an increasing number of providers offering computation and storage services, which represent an opportunity for providing better functionality and security. In particular, our approach relies on data distribution by allocating the data structure over three different servers, each of which will then see only a portion of the data blocks and will similarly have a limited visibility of the actual accesses on the data. Data swapping implies changing the physical location of accessed data by swapping them between the three involved servers. Swapping, in contrast to random shuffling, forces the requirement that whenever a block is accessed, the data retrieved from it (i.e., stored in the block before the access) *should not* be stored at the same block after the access. We illustrate in this paper how the use of three servers (for distributed data allocation) together with swapping (forcing data re-allocation across servers) provide nice protection guarantees, typically outperforming the use of a random shuffling assuming no collusion among servers, and maintaining sufficient protection guarantees even in the presence of collusions among two, or even all three, of the involved servers.

II. Basic concepts

A shuffle index is an *unchained* B+*-tree* such that: *i)* each node stores up to $F-1$ (with F the fan-out of the B+-tree) ordered values and has as many children as the number of values stored plus one; *ii)* the tree rooted at the j-th child of an internal node stores values included in the range $[v_{j-1}, v_j)$, where v_{j-1} and v_j are the $(j-1)$-th and j-th values in the node, respectively; and *iii)* all leaves, which store actual tuples, are at the same level of the tree, that is, they all have the same distance from the root node. Figure 1(a) illustrates an example of unchained $B+$-tree. In this figure, and in the remainder of the paper, for simplicity, we refer to the content of each node with a label (e.g., a), instead of explicitly reporting the values in it. In the example, root node r has six children $(a, \ldots, f)$ each with three to four children. For easy reference, we label the leaf nodes, descendants of a node, with the same label as the node concatenated with a progressive number (e.g., a_1, a_2, a_3 are the children of node a). At the logical level, each node is allocated to a logical identifier. Logical node identifiers are also used in internal nodes as pointers to their children. At the physical level, each node is translated into an encrypted chunk stored at a physical block. The encrypted chunk is obtained by encrypting the concatenation of the node identifier and its content (values and pointers to children). Encryption protects the confidentiality and integrity of each node as well as of the overall data structure.

Retrieval of a value in the tree requires walking the tree from the root to the target leaf, following at each level the pointer to the child in the path to the target leaf. Being the index stored in encrypted form, such an access requires an iterative process with the client downloading at each level the block of interest, decrypting it, and determining the next block (storing the child of interest) to be requested.

Although the data structure is encrypted, by observing a long enough sequence of accesses, the server (or other observers having access to it) could reconstruct the topology of the tree, identify repeated accesses, and possibly infer sensitive data content [4], [5]. To protect data and accesses from such inferences, the shuffle index uses of complementary techniques bringing confusion to the observer and destroying the static correspondence between nodes and blocks where they are stored. In particular: *i)* to provide confusion as to which block is the target of an access, more blocks (the target plus some covers) are requested at every access; *ii)* a cache is maintained with the most recently accessed paths; and *iii)* at every access, the nodes/blocks accessed and the ones in the cache are shuffled (randomly reassigning nodes to blocks, and performing a new encryption) and all the involved blocks rewritten back on the server.

III. Rationale of the approach

Our approach builds on the shuffle index by borrowing from it the base data structure (encrypted unchained $B+$-tree) and the idea of breaking the otherwise static correspondence between nodes and blocks at every access. It differs from the shuffle index in the management of the data structure, for storage and access (exploiting a distributed allocation), and in the way the node-block correspondence is modified, applying swapping instead of random shuffling, forcing the node involved in an access to change the server where it is stored (again exploiting the distributed allocation). Also, it departs from the cache, not requiring any storage at the client.

The basic idea of our approach is to randomly partition data among three independent servers, and, at every access, randomly move (*swap*) data retrieved from a server to any of the other two so that data retrieved from a server would not be at the same server after the access. Since nodes are randomly allocated to servers, the path from the root to the leaf target of an access can traverse nodes at different servers. Then, to provide uniform visibility at any access at every server (which should operate as if it was the only one serving the client), every time the node to be accessed at a given level belongs to one server, our approach also requests to access one additional block at the same level at each of the other servers.

The reader may wonder why we are distributing the shuffle index among *three* servers, and not two or four. The rationale behind the use of multiple servers is to provide limited visibility, at each of the servers, of the data structure and of the accesses to it. In this respect, even adopting two servers could work. However, an approach using only two servers would remain too exposed to collusion (between the two) that, by merging their knowledge, could reconstruct the node-block correspondence and compromise access and data confidentiality. The data swapping (in contrast to the random shuffling) we adopt, while providing better protection with respect to shuffling in general, implies deterministic reallocation in the case of two servers and could then cause exposure in case of collusion. The use of three servers provides instead considerable better protection. Swapping ensures that data are moved out from a server at every access, while still providing non determinism in data reallocation (as the data could have moved to any of the other two servers), even in presence of collusion among the three servers. While going from two servers to three servers provides considerably higher protection guarantees, further increasing the number of servers provides limited advantage, while instead increasing the complexity of the system.

IV. Data structure and three-server allocation

At the abstract level, our structure is essentially the same as the shuffle index, namely we consider an unchained $B+$-tree defined over candidate key K, with fan-out F, and storing data in its leaves. However, we consider the root to have three times the capacity of internal nodes. Since internal nodes and leaves will be distributed to three different servers, assuming a three times larger root allows us to conveniently split it among the servers (instead of replicating it) providing better access performance by potentially reducing the height of the tree. In fact, a $B+$-tree having at most $3F$ children for the root node can store up to three times the number

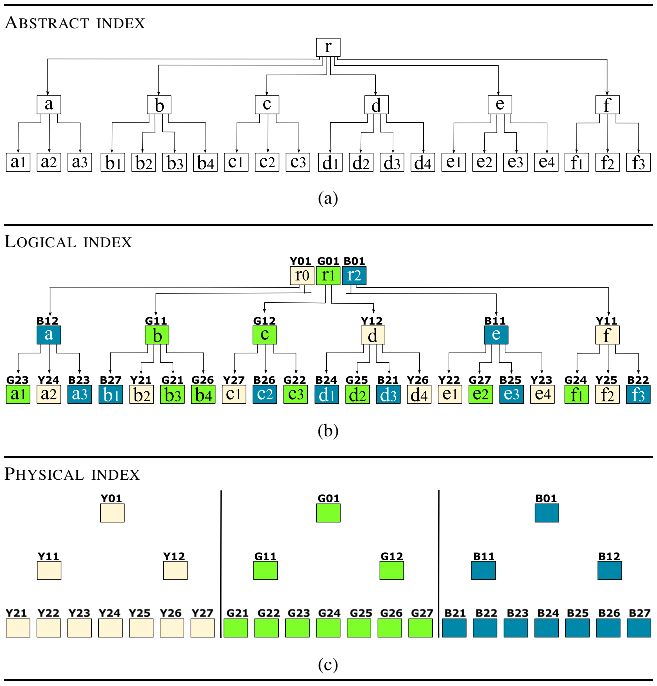
ABSTRACT INDEX
r
a b c d e f
a1 a2 a3 b1 b2 b3 b4 c1 c2 c3 d1 d2 d3 d4 e1 e2 e3 e4 f1 f2 f3
(a)
LOGICAL INDEX
Y01 G01 B01
r0 r1 r2
B12 G11 G12 Y12 B11 Y11
G23 Y24 B23 B27 Y21 G21 G26 Y27 B26 G22 B24 G25 B21 Y26 Y22 G27 B25 Y23 G24 Y25 B22
(b)
PHYSICAL INDEX
Y01 G01 B01
Y11 Y12 G11 G12 B11 B12
Y21 Y22 Y23 Y24 Y25 Y26 Y27 G21 G22 G23 G24 G25 G26 G27 B21 B22 B23 B24 B25 B26 B27
(c)

Figure 1: An example of abstract (a), logical (b), and physical (c) shuffle index distributed at three servers

of tuples/values stored in a traditional $B+$-tree of the same height. Formally, each internal abstract node n^a in the tree stores a list $v_1, \ldots, v_q$ of q values, with $\lceil \frac{F}{2} \rceil - 1 \leq q \leq F-1$ ($q \leq 3F-1$ for the root node), ordered from the smallest to the greatest, and has $q+1$ children. The i-th child of a node is the root of the subtree containing the values *val* with $v_{i-1} \leq val < v_i$, $i = 2, \ldots, q$; the first child is the root of the subtree with all values $val < v_1$, while the last child is the root of the subtree with all values $val \geq v_q$. Each leaf node stores a set of values, together with the tuples in the dataset having these values for attribute K. All the non-root nodes have to be at least 33% full. Figure 1(a) illustrates an example of our abstract data structure.

At the logical level, the abstract root node translates to three logical nodes, say r_0, r_1, r_2, each storing one third of the values and pointers to children of the abstract root node. More precisely, r_0 stores values $v_1, \ldots, v_i$, with $i = \lfloor \frac{q-2}{3} \rfloor$, and the corresponding pointers to children; r_1 stores values $v_{i+2}, \ldots, v_{2i+1}$, and the corresponding children; and r_2 stores the remaining values $v_{2i+3}, \ldots, v_q$, and the corresponding children. (Note that values v_{i+1} and v_{2i+2} are not necessary for the index definition and are then not explicitly stored in the obtained roots.) For instance, Figure 1(b) illustrates an example of logical index representing the abstract index in Figure 1(a) where the abstract root node r is represented by three logical nodes, r_0, r_1, r_2, each having two of the six children of the abstract root node r. Each (non-root) abstract node n^a translates to a logical node n and is allocated to a logical identifier $n.id$, used also to represent the pointer to n in its parent. To regulate data distribution at the different servers, we distinguish three subsets $\mathcal{ID}_i$, $i \in \{Y,G,B\}$, of logical identifiers corresponding to the physical addresses stored at each of the storage servers S_i, $i \in \{Y,G,B\}$. Allocation of abstract nodes to logical identifiers is defined through an allocation function, formally defined as follows.

Definition 4.1 (Distributed allocation): Let $\mathcal{N}^a$ be the set of abstract nodes in a shuffle index, S_Y, S_G, S_B be the servers storing it, and $\mathcal{ID}_Y$, $\mathcal{ID}_G$, $\mathcal{ID}_B$ be the set of logical identifiers at server S_Y, S_G, S_B, respectively. A *distributed allocation function* is a bijective function ϕ: $\mathcal{N}^a \rightarrow \mathcal{ID}_Y \cup \mathcal{ID}_G \cup \mathcal{ID}_B$ that associates a logical identifier with each abstract node.

Given an abstract node n^a, $\phi(n^a)$ determines the identifier of the logical node n where n^a is allocated, denoted $n.id$. In the following, we denote with $\sigma(id)$ the server at which the logical node with identifier *id* is stored. Note that the order of logical identifiers is independent from the node content. Also, the allocation of logical nodes to physical blocks and, more in general, to servers does not depend on the topology of the abstract structure. In other words, a node may be stored at a different server with respect to its parent and/or its siblings. An example of distribution of the index in Figure 1(a) is illustrated in Figure 1(b). For the sake of readability, logical identifiers are reported on the top of each node and blocks are color-coded (yellow for S_Y, green for S_G, and blue for S_B). For simplicity and easy reference, each logical identifier starts with a letter denoting the server where the corresponding block is stored (Y for S_Y, G for S_G, and B for S_B), and the first digit denotes its level in the tree. As an example, G_{24} is the identifier of a node at level 2 of the index and stored at server S_G. A distributed index $\mathcal{I}$ can be represented, at the logical level, as a pair $\langle \mathcal{N}, (S_Y, S_G, S_B) \rangle$, with $\mathcal{N}$ the set of logical nodes composing it, and S_Y, S_G, and S_B the servers where these nodes are physically stored. To guarantee distribution among the different servers (and provide uniform visibility at every server in access execution, as we will explain in the following section), the distributed allocation function guarantees that each non-root node in the index, as well as r_0, r_1, and r_2 together, has at least one child stored at each server. At starting time, we then assume the structure to be evenly distributed at the level of node, meaning that the children of each node are equally distributed among S_Y, S_G, and S_B (i.e., each server will be allocated one third ± 1 of the children of every node). We also assume the structure to be evenly distributed both globally and for each level in the tree. Figure 1(b) represents an example of logical index where the children of each node, the nodes in each level, and the nodes in the tree are evenly distributed to servers.

At the physical level, logical addresses are translated into physical addresses at the three servers. Node content is prefixed with a random salt and encrypted in CBC mode with a symmetric encryption function. The result of encryption is concatenated with the result of a MAC function applied to the encrypted node and its identifier, producing an encrypted block b allocated to a physical address. The presence of the node identifier in each block permits the client to assess the authenticity and integrity of the block content and, thanks to the identifiers of the children stored in each node, also of the whole index structure. Figure 1(c) illustrates the physical representation of the logical index in Figure 1(b). In the following, for simplicity and without loss of generality, we assume that the physical address of a block corresponds to the logical identifier of the node it stores. The view of each server S_i corresponds to the portion of the physical representation in Figure 1(c) allocated at S_i. Note that each server can see all and only the blocks allocated to it. We use the term node to refer to an abstract data content and block to refer to a specific memory slot in the logical/physical structure. When either terms can be used, we will use them interchangeably.

V. Working of the approach

We illustrate how access execution is performed adopting distributed covers and swapping to guarantee confidentiality of the accesses and of the data structure.

A. Distributed covers

Like in the shuffle index, retrieval of a key value (or more precisely the data indexed with that key value and stored in a leaf node) entails traversing the index starting from the root and following, at every node, the pointer to the child in the path to the leaf possibly containing the target value. Again, being data encrypted, such a process needs to be performed iteratively, starting from the root to the leaf, at every level decrypting (and checking for integrity) the retrieved node to determine the child to follow at the next level. Since our data structure is distributed among three servers and the allocation of nodes to servers is independent from the allocation of their ancestors and/or descendants, the path from the root to a target leaf may (and usually does) involve nodes stored at different servers. For instance, with reference to Figure 1, retrieval of a value d_1 entails traversing path $\langle r_1, d, d_1\rangle$ and hence accessing blocks G_{01}, Y_{12}, and B_{24} each stored at a different server. Retrieval of value a_3 entails traversing the path $\langle r_0, a, a_3\rangle$ and hence accessing blocks Y_{01}, B_{12}, and B_{23}, the first stored at S_Y and the last two stored at S_B. Since each server can observe different iterations and, after a long enough sequence of observations, also infer the levels associated with blocks, we aim at ensuring a uniform visibility at every server. In other words, we want every server to observe, for every search, the access to a block at each level, with each server then operating as if it was the only one serving the client. (Note that even if only one block is accessed at every level, no information is leaked to the server on the tree topology, since: *i)* the accessed blocks may not be actually in a parent-child relationship, and *ii)* the content of accessed blocks changes just after the access.) Our requirement of uniform visibility at each server is captured by the following property.

Property 5.1 (Uniform visibility): Let $\mathcal{I} = \langle \mathcal{N}, (S_Y,S_G,S_B)\rangle$ be a distributed index, and $N = \{n_1, \ldots, n_m\}$ be the set of logical nodes accessed by a search. The search satisfies *uniform visibility* iff for each S_i, $i{\in}\{Y,G,B\}$, and for each level l in $\mathcal{I}$, $\exists!\ n \in \mathcal{N}$ such that: *i)* $\sigma(n.id){=}S_i$; and *ii)* n is at level l in $\mathcal{I}$.

For instance, our two sample accesses above do not satisfy uniform visibility. To satisfy uniform visibility, we complement, at each level, the access required by the retrieval of the target value with two additional accesses at the servers that do not store the target block at that level. We call *covers* these additional accesses as they resemble cover searches of the shuffle index, although they have also many differences (e.g., they cannot be pre-determined as data allocation is unknown, they may not represent a path in the index, and they are not observed by the same server observing the target). Stressing their distributed nature, we term them distributed covers, defined as follows.

Definition 5.1 (Distributed cover): Let $\mathcal{I} = \langle \mathcal{N}, (S_Y,S_G,S_B)\rangle$ be a distributed index, and n be a node in $\mathcal{N}$. A set of *distributed covers* for n is a pair of nodes (n_i, n_j) in $\mathcal{N}$ such that the following conditions hold: *i)* n, n_i, n_j belong to the same level of the index; and *ii)* $\sigma(n.id) \neq \sigma(n_i.id)$, $\sigma(n.id) \neq \sigma(n_j.id)$, and $\sigma(n_i.id) \neq \sigma(n_j.id)$.

As stated by the definition above, distributed covers for a node n are a pair of nodes (n_i, n_j) that belong to the same level as n, and such that the three nodes are allocated at different servers. For instance, distributed covers for Y_{12} could be any of the following pairs: (B_{11}, G_{11}), (B_{11}, G_{12}), (B_{12}, G_{11}), (B_{12}, G_{12}). Similarly, at the leaf level, the distributed covers for B_{24} could be any pair of nodes (Y_{2*}, G_{2*}), with $*$ any value between 1 and 7 (e.g., (Y_{23}, G_{21})). The distributed covers of a root node are the roots at the other two servers (e.g., (G_{01},B_{01}) is the distributed cover pair for Y_{01}).

With the consideration of distributed covers, to guarantee uniform visibility at every server, access execution works as follows. Again, an iterative process is executed starting from the root to the leaf level. First, the client retrieves the root at all the three servers and decrypts them to determine the target root (i.e., the one going to the target value) and the target child node n to visit. It also chooses two distributed covers for n, in such a way that covers are indistinguishable from targets [1]. The client requests access to n and its distributed covers to the respective servers. It then decrypts the accessed nodes and iteratively performs the same process until the leaves (target and distributed covers) are reached. As an example, consider the data structure in Figure 1(b) and assume value d_1 is to be accessed. The nodes along the path to the target of the accesses are $\langle r_1, d, d_1\rangle$ entailing accesses to target blocks $\langle G_{01}, Y_{12}, B_{24}\rangle$. Assume that distributed covers (Y_{01}, B_{01}), (G_{11}, B_{11}), and (G_{21}, Y_{23}) are used for G_{01}, Y_{12}, and B_{24}, respectively. The nodes involved by the access, as observed by each server, are then Y_{01}, Y_{12}, Y_{23} for S_Y, G_{01}, G_{11}, G_{21} for S_G, and B_{01}, B_{11}, B_{24} for S_B. Note that each server simply observes a sequence of three accesses to three blocks, while it cannot see their content. In principle, according to Definition 5.1, every pair of nodes at the same level as n, but allocated at the other two servers, represents a pair of distributed covers for n. However, in the choice of distributed covers, we need to take into consideration the fact that accessed nodes are reallocated. In fact, when n is moved to a different block, the pointers to n in its parent must be updated to maintain consistency of the index. Therefore, the nodes involved in an access should always form a sub-tree, possibly including paths of different lengths. Each distributed cover at level l should then be child of the node along the path to the target at level $l-1$ or of one of its distributed covers. This is formally captured by the following definition of chained set of distributed covers.

Definition 5.2 (Chained distributed covers): Let $\mathcal{I}$=$\langle\mathcal{N}$,$(S_Y$,S_G,$S_B)\rangle$ be a distributed index, and $p = \langle n_0, \ldots, n_h\rangle$ be a path in $\mathcal{I}$. A *chained set* of distributed covers for p is a set $\mathcal{C}(p)$ of nodes in $\mathcal{N}$ s.t.: *i)* $p \subset \mathcal{C}(p)$; *ii)* $\forall n{\in}p$, $\exists$ $\{n'_i, n'_j\}{\subset}\mathcal{C}(p)$ with (n'_i, n'_j) distributed covers for n; and *iii)* $\forall n{\in}\mathcal{C}(p)$, either n is a root node or its parent belongs to $\mathcal{C}(p)$.

The distributed covers in the example above are chained as the covers at every level are children of a node accessed (either as target or cover) in the level above. Note that while in the example (for simplicity and readability of the figure) every accessed node has exactly one accessed child, such a condition is not needed. In fact, Definition 5.2 requires every node to have its parent in the access (so to enable update of pointers to the node in its parent), while a node can have no children in the access. For instance, Y_{26} (d_4) could have also been used instead of Y_{23} (e_4) as one of the covers for B_{24}, together with G_{21}. The resulting set would have still satisfied Definition 5.2.

B. Swapping

A desired requirement of our approach is that data retrieved (as target or cover) in an access are stored at a different server after the access. We capture such a requirement with a property of *continuous moving* as follows.

Property 5.2 (Continuous moving): Let $\mathcal{I} = \langle\mathcal{N}$, $(S_Y$,S_G,$S_B)\rangle$ be a distributed index, and $N = \{n_1, \ldots, n_m\}$ be the set of nodes in $\mathcal{N}$ accessed as target or distributed cover by a search. The search satisfies *continuous moving* iff, for each node $n \in N$, the server $\sigma(n.id)$ where n is stored before the access is different from the one where it is stored after the access.

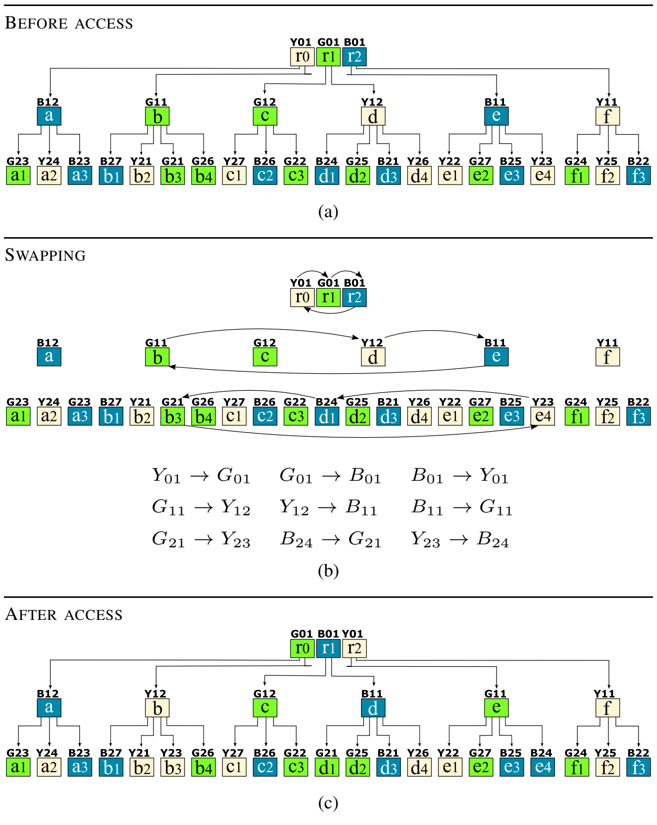
Before access
(a)
Swapping
$Y_{01} \to G_{01}$ $G_{01} \to B_{01}$ $B_{01} \to Y_{01}$
$G_{11} \to Y_{12}$ $Y_{12} \to B_{11}$ $B_{11} \to G_{11}$
$G_{21} \to Y_{23}$ $B_{24} \to G_{21}$ $Y_{23} \to B_{24}$
(b)
After access
(c)

Figure 2: Evolution of the distributed index for a search for value $d1$

Continuous moving prevents servers from building knowledge based on accesses they can observe as a node is immediately removed from a server after being accessed. For instance, servers will not be able to observe repeated accesses anymore. We guarantee satisfaction of this property by swapping the content of the blocks accessed at every level. Swapping is defined as follows.

Definition 5.3 (Swapping): Let *ID* be a set of logical identifiers. A *swapping* for *ID* is a random permutation π :*ID* $\to$ *ID* such that $\forall id{\in}ID$, $\sigma(id){\neq}\sigma(\pi(id))$.

Figure 2(b) illustrates a possible swapping among the nodes/blocks accessed searching for value d_1 over the index in Figure 1(a), assuming to adopt (Y_{01}, B_{01}), (G_{11}, B_{11}), and (G_{21}, Y_{23}) as distributed covers for for G_{01}, Y_{12}, and B_{24}. For instance, swap $Y_{01} \to G_{01}, G_{01} \to B_{01}, B_{01} \to Y_{01}$ causes r_0 to move to G_{01}, r_1 to move to B_{01}, and r_2 to move to Y_{01}. Figure 2(c) illustrates the effect of such a swapping on the data structure at the logical level. Note that before re-writing blocks at the servers, the content of the corresponding nodes is re-encrypted with a different random salt that changes at every access. The adoption of a different random salt in node encryption and the concatenation with a different node identifier guarantees to produce a different encrypted block, even if the content represents the same node. This makes it impossible for storage servers to track swapping operations. Given an index characterized by a distributed allocation function ϕ and a swapping function π over a subset *ID* of the identifiers in the index, the allocation function resulting from the swap is defined as: $\phi(n^a)$=$\pi(\phi(n^a))$ iff $\phi(n^a)\in$*ID*; $\phi(n^a)$=$\phi(n^a)$, otherwise. Note that the assignment function resulting from the application of a swap π still represents a distributed assignment function, since π is a permutation function. For instance, with reference to the example in Figure 2, we note that each node is associated with one identifier before and after the access, and vice versa each identifier is assigned to one node only both before and after the access.

Moving nodes among servers may reduce the number of children at a server for some nodes. In the worst case, a node may be left with no children on one of the servers. We note however that, since we initially define a balanced allocation and in traditional systems the fan-out of the tree is high (in the order of some hundreds), the probability that a node is left without children on one of the servers is extremely low, due to a natural regression to the mean that reduces the stochastic drift. To completely solve this risk we check that swapping does not create configurations where a server is not represented in the descendants of a node.

C. Access execution algorithm

Figure 3 illustrates the pseudocode of the algorithm, executed at the client-side, enforcing the search process over a distributed index, extended with our protection techniques. Given a request for searching *target_value*, the algorithm first downloads from each server the block storing a portion of the root node and swaps them according to a swapping function π (lines 1–3). The algorithm then visits the index and, for each level l=$1,\ldots,h$, determines the logical identifier *target_id* of the node at level l along the path to *target_value* (line 5), which is one of the children of the nodes in *Parents*. It then chooses a pair of distributed covers for *target_id* (line 6), that is, two nodes chosen among the children of nodes in *Parents*, and allocated at different servers, also with respect to *target_id* (Definition 5.1). The algorithm downloads from the storage servers the blocks of interest and decrypts their content retrieving the corresponding nodes (line 7). It randomly chooses a swapping function π (Definition 5.3) and reallocates accessed nodes accordingly; if the permutation causes a node in *Parents* not to have a descendant at each server, a new pair of covers is selected (lines 8–10). To guarantee the consistency of the tree structure, the algorithm updates the pointers to swapped nodes in their parents (i.e., nodes in *Parents*), which are then encrypted and sent back to the different servers for storage

```
/* $\mathcal{I}$=$\langle\mathcal{N}$,$(S_Y$,$S_G$,$S_B)\rangle$: distributed index with height $h$ */
INPUT   target_value : value to be searched in $\mathcal{I}$
OUTPUT  $n$ : leaf node that contains target_value
MAIN
1: Parents := download and decrypt block $Y_{01}$ from $S_Y$
          block $G_{01}$ from $S_G$ and block $B_{01}$ from $S_B$
2: let $\pi$ be a permutation of identifiers in Parents s.t. $\sigma(id)\neq\sigma(\pi(id))$
3: swap nodes in Parents according to $\pi$
4: for $l$:=1...$h$ do /* visit the index level by level*/
5:    target_id := identifier of the node at level $l$
                along the path to target_value
6:    randomly choose cover[1] and cover[2] s.t.
             they are children of Parents and
             $\sigma$(target_id)$\neq\sigma$(cover[1]), $\sigma$(target_id)$\neq\sigma$(cover[2]),
             $\sigma$(cover[1])$\neq\sigma$(cover[2])
7:    Read := download and decrypt each block with identifier
             id$\in${target_id,cover[1],cover[2]} from $\sigma(id)$
8:    let $\pi$ be a permutation of identifiers of nodes in Read s.t.
             $\sigma(id)\neq\sigma(\pi(id))$ and
             each $n\in$Parents has a child at $S_Y$,$S_G$,$S_B$
9:    if $\pi$ does not exist, goto 6
10:   swap nodes in Read according to $\pi$
11:   update pointers to children in Parents according to $\pi$
12:   encrypt and write each node $n\in$Parents at server $\sigma(n.id)$
13:   target_id := $\pi$(target_id)
14:   cover[1] := $\pi$(cover[1]), cover[2] := $\pi$(cover[2])
15:   Parents := Read
16: encrypt and write each node $n\in$Read at server $\sigma(n.id)$
17: return node $n\in$Read with $n.id$=target_id
```

Figure 3: Access algorithm

(lines 11–12). The algorithm also updates the identifier of the target and distributed covers according to π to preserve the correctness of the search process (lines 13–14). Once all the levels in the tree have been visited, the algorithm returns the leaf node where *target_value* is stored, if such a value belongs to the dataset (line 17).

The following theorem formally states and proves the correctness of the algorithm, and in particular the fact that it satisfies Properties 5.1 and 5.2 and maintains the correctness of the index structure. The proof of the Theorem has been omitted for space constraints.

Theorem 5.1: Let $\mathcal{I}$=$\langle\mathcal{N}$,$(S_Y$,S_G,$S_B)\rangle$ be a distributed index, and *target_value* be the target of an access. The algorithm in Figure 3:

1) satisfies Property 5.1 (*uniform visibility*);
2) satisfies Property 5.2 (*continuous moving*);
3) maintains unchanged the number of blocks stored at each server for each level $l = 0,\ldots,h$ (*distribution invariance*);
4) returns the unique node where *target_value* is, or should be, stored (*access correctness*);
5) maintains an index representing the original unchained $B+$-tree (*structure correctness*).

D. Protection analysis

We evaluate the protection of our approach with respect to guaranteeing confidentiality of the accesses against possible observers. We consider the servers as our observers as they have the most powerful view over the stored data and the accesses to them. The servers know (or can infer from their interactions with the client): the total number of blocks (nodes) and the height h of the index; the identifier of each block b and its level in the tree; the identifier of read and written blocks for each access operation. On the contrary, they do not know and cannot infer the content and the topology of the index (i.e., the pointers between parent and children), thanks to encryption of nodes. For simplicity, but without loss of generality, we focus our analysis only on leaf blocks/nodes since the high fan-out of the index ensures that internal nodes are involved in swapping operations more often than leaf nodes, resulting therefore more protected.

Guaranteeing access confidentiality means hiding to the servers the correspondence (as our distribution and swap aim to do) between nodes and the blocks where they are stored. We model the knowledge of an observer on the fact that a node n is stored at a block b as a probability value $P(b,n)$, expressing the confidence in such a knowledge, with $P(b,n)$=1 corresponding to certainty, and $P(b,n)=\frac{1}{|\mathcal{N}|}$ to complete absence of knowledge, with $\mathcal{N}$ the set of leaves in the index. The uncertainty over the block storing a node $n_i \in \mathcal{N}$ is measured through the entropy $H_{n_i} = -\sum_{j=1}^{|\mathcal{N}|} P(b_j, n_i) \log_2 P(b_j, n_i)$, applied on the probabilities $P(b_j, n_i)$ for all the blocks b_j in the index.

We evaluate the knowledge degradation of each server starting from the worst possible initial scenario, where each server knows the exact correspondence between nodes and blocks (i.e., $H_n = 0$, since $P(b,n)$=1 when n is allocated at block n, $P(b,n)$=0 otherwise). At every access request, the swapping performed by the client moves the content of each retrieved block to a server different from the one where it was initially stored. Hence, the entropy H_n of each accessed node evolves as a consequence of the access. Such evolution clearly depends on the server's ability to observe accessed blocks. In our base scenario (no collusion), S_Y (but the same applies to S_G and S_B) initially knows the node stored at each of its blocks, that is, $P(b,n)$=1 if n is allocated at block b of server S_Y; $P(b,n)=1/N$ if n is not at S_Y and b is at S_G or S_B, with N the number of blocks at S_G and S_B, as server S_Y does not have any knowledge of node-block allocation at the other servers; $P(b,n)$=0, otherwise. For each access, S_Y can observe the access to one block only, say b_y. After the access, the content of b_y is moved to a block b that is not stored at S_Y and on which it does not have any knowledge (i.e., it is moved to any of the blocks at S_G and S_B with equal probability). Also, the content of one of the blocks b stored at S_G or S_B is moved to b_y. In other words, the content of b_y after an access is, with equal probability, the content of any of the blocks at S_G and S_B. As an example, assuming that initially block b_y stores node n_y, after the first access $P(b_y,n_y)$ becomes 0 (from 1) and $P(b_j,n_y)$ becomes $1/N$ (from 0), with b_j any block at S_G or S_B. Also, $P(b_i,n_j)$ becomes $1/N$ (from 0), for each block b_i at server S_Y and for each node n_j initially stored at S_G or S_B. Then, for the accessed blocks, the information is immediately degraded near to the level of lowest information. Overall knowledge about the correspondence between nodes and blocks will be affected by a complete degradation as a sequence of accesses is executed. (These observations have been confirmed by a detailed analysis and experimental evaluation.)

We note that if two (or even three) servers collude, their initial knowledge as well as the ability to observe accesses to blocks improves. However, even in the worst case of full collusion (i.e., collusion among all servers), the knowledge of each server is progressively destroyed thanks to the uncertainty (among the two other servers) of the new allocation of the accessed nodes.

VI. Related work

The problem of protecting data in the cloud requires the investigation of different aspects (e.g., [6], [7], [8]). In particular, approaches supporting query execution consist in attaching to the encrypted data some indexes used for fine-grained information retrieval (e.g., [6], [9]), or in adopting specific cryptographic techniques for keyword-based searches (e.g., [10]). The main problem of these solutions is that they protect only the confidentiality of the data at rest.

Solutions for protecting access and pattern confidentiality are based on Private Information Retrieval (PIR) techniques. Such solutions, however, do not protect content confidentiality and suffer from high computational costs (e.g., [11]), even when different copies of the data are stored at multiple non-communicating servers (e.g., [12]). Recent approaches address the access and pattern confidentiality problems through the definition of techniques that dynamically change, at every access, the physical location of the data. Some proposals have investigated the adoption of the Oblivious RAM (ORAM) structure (e.g., [13]), in particular with recent proposals aimed at making ORAM more practical such as ObliviStore [2], Path ORAM [3], and Melbourne Shuffle [14]. ORAM has also been recently extended to operate in a distributed scenario [15], [16]. The goal of these solutions is to reduce communication costs for the client and then make ORAM-based approaches available also to clients using lightweight devices. The privacy guarantees provided by distributed ORAM approaches however rely on the fact that storage servers do not communicate or do not collude with each other. Our approach is instead more general and is specifically aimed at enhancing protection guarantees provided to the client. Alternative solutions are based on the adoption of a tree-based structure (e.g., [17], [18]) to preserve the content and access confidentiality.

The shuffle index has been first introduced in [1] and then adapted in [19], [20] to accommodate concurrent accesses on a shuffle index stored at one storage server or to operate in a distributed scenario with two storage providers. These solutions differ from the approach proposed in this paper since they rely on a traditional shuffling among accessed blocks (which do not impose the constraint of changing the server where nodes are allocated at each access). Furthermore, the proposal in [19] provides lower protection guarantees, as also demonstrated by our evaluation.

A different, although related, line of works is represented by fragmentation-based approaches for protecting data confidentiality (e.g., [7], [21]). These solutions are based on the idea of splitting sensitive data among different relations, possibly stored at different storage servers, to protect sensitive associations between attributes in the original relation. Although based on a similar principle, fragmentation-based approaches only protect content confidentiality, and are not concerned with access and pattern confidentiality.

VII. Conclusions

We have proposed an approach to protect both the confidentiality of data stored at external servers and the accesses to them. Our approach is based on the use of a key-based dynamically allocated data structure distributed over three independent servers. We have described our reference data structure and illustrated how our distributed allocation and swapping techniques operate at every access to ensure protection of access confidentiality.

Acknowledgment

This work was supported in part by: the EC within the 7FP under grant agreement 312797 (ABC4EU), and the Italian Ministry of Research within PRIN project "GenData 2020" (2010RTFWBH).

References

[1] 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, June 2011.

[2] E. Stefanov and E. Shi, "ObliviStore: High performance oblivious cloud storage," in *Proc. of IEEE S&P*, San Francisco, CA, May 2013.

[3] 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, Nov. 2013.

[4] M. Islam, M. Kuzu, and M. Kantarcioglu, "Inference attack against encrypted range queries on outsourced databases," in *Proc. of CODASPY*, San Antonio, TX, March 2014.

[5] H. Pang, J. Zhang, and K. Mouratidis, "Enhancing access privacy of range retrievals over $B+$-trees," *IEEE TKDE*, vol. 25, no. 7, pp. 1533–1547, 2013.

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

[7] V. Ciriani, S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati, "Combining fragmentation and encryption to protect privacy in data storage," *ACM TISSEC*, vol. 13, no. 3, pp. 22:1–22:33, July 2010.

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

[9] 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, June 2002.

[10] C. Wang, N. Cao, K. Ren, and W. Lou, "Enabling secure and efficient ranked keyword search over outsourced cloud data," *IEEE TPDS*, vol. 23, no. 8, pp. 1467–1479, 2012.

[11] R. Ostrovsky and W. E. Skeith, III, "A survey of single-database private information retrieval: Techniques and applications," in *Proc. of PKC*, Beijing, China, April 2007.

[12] C. Cachin, S. Micali, and M. Stadler, "Computationally private information retrieval with polylogarithmic communication," in *Proc. of EUROCRYPT*, Prague, Czech Republic, May 1999.

[13] 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, October 2008.

[14] O. Ohrimenko, M. Goodrich, R. Tamassia, and E. Upfal, "The Melbourne Shuffle: Improving oblivious storage in the cloud," in *Proc. of ICLAP*, Copenhagen, Denmark, July 2014.

[15] S. Lu and R. Ostrovsky, "Distributed Oblivious RAM for secure two-party computation," in *Proc. of TCC*, Tokyo, Japan, March 2013.

[16] E. Stefanov and E. Shi, "Multi-cloud oblivious storage," in *Proc. of ACM CCS*, Berlin, Germany, November 2013.

[17] P. Lin and K. Candan, "Hiding traversal of tree structured data from untrusted data stores," in *Proc. of WOSIS*, Porto, Portugal, April 2004.

[18] K. Yang, J. Zhang, W. Zhang, and D. Qiao, "A light-weight solution to preservation of access pattern privacy in un-trusted clouds," in *Proc. of ESORICS*, Leuven, Belgium, Sep. 2011.

[19] S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, G. Pelosi, and P. Samarati, "Distributed shuffling for preserving access confidentiality," in *Proc. of ESORICS*, Egham, UK, 2013.

[20] ——, "Supporting concurrency and multiple indexes in private access to outsourced data," *JCS*, vol. 21, no. 3, pp. 425–461, 2013.

[21] G. Aggarwal *et al.*, "Two can keep a secret: A distributed architecture for secure database services," in *Proc. of CIDR*, Asilomar, CA, January 2005.