DOI: 10.1109/TCC.2015.2449993

Three-server swapping for access confidentiality

Sabrina De Capitani di Vimercati, *Senior Member, IEEE,* Sara Foresti, *Member, IEEE,*
Stefano Paraboschi, *Member, IEEE,* Gerardo Pelosi, *Member, IEEE,*
and Pierangela Samarati, *Fellow, IEEE*

Abstract—We propose an approach to protect confidentiality of data and accesses to them when data are stored and managed by external providers, and hence not under direct control of their owner. Our approach is based on the use of distributed data allocation among three independent servers and on a dynamic re-allocation of data at every access. Dynamic re-allocation is enforced by swapping data involved in an access across the servers in such a way that accessing a given node implies re-allocating it to a different server, then destroying the ability of servers to build knowledge by observing accesses. The use of three servers provides uncertainty, to the eyes of the servers, of the result of the swapping operation, even in presence of collusion among them.

Index Terms—Shuffle index, content confidentiality, access confidentiality, pattern confidentiality, distributed swapping.

1 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 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 the data. Modern cryptographic algorithms offer high efficiency and strong protection of data content. Simply protecting data content with an encryption layer does not fully solve the confidentiality problem, as *access confidentiality*, namely the confidentiality of the specific accesses performed on the data remains at risk. There are several reasons for which access confidentiality may be demanded [1], among which 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. 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 where they are stored. Among such proposals, the *shuffle index* [1] provides a key-based hierarchical organization of the data, supporting then 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 accesses to the data. Data swapping implies changing the physical location of accessed data by swapping them among 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 (as it is to be expected) 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.

The remainder of the paper is organized as follows. Section 2 recalls the basic concepts of the shuffle index. Section 3 introduces the rationale of our approach. Section 4 describes our index structure working on three servers. Section 5 presents the working of our approach, discussing protection techniques and data access. Section 6 analyzes protection guarantees. Section 7 discusses the motivations behind our choice of swapping and of three as the number of servers to be used, and provides some performance and economic considerations for our approach. Section 8 illustrates related works. Finally, Section 9 concludes the paper.

- *S. De Capitani di Vimercati, S. Foresti, and P. Samarati are with the Università degli Studi di Milano, Italy.*
  E-mail: firstname.lastname@*unimi.it*
- *G. Pelosi is with Politecnico di Milano, Italy.*
  E-mail: gerardo.pelosi@polimi.it
- *S. Paraboschi is with the Università degli Studi di Bergamo, Italy.*
  E-mail: parabosc@unibg.it

2 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 children of a non-root 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 of nodes content and of the tree structure. Also, it provides integrity of each node (as tampering would be detected) and of the structure overall (being the node identifier and the pointers to children also encrypted in the block).

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 (starting from the root) the block of interest, decrypting it, and determining the next block 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 also infer sensitive data content [2], [3]. To protect data and accesses from such inferences, the shuffle index makes use of complementary techniques bringing confusion to the observer and destroying the static correspondence between nodes and blocks where they are stored. In particular, in the original shuffle index proposal: *i)* to provide confusion as to which block is the actual 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.

3 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 physical blocks at every access. It differs from the shuffle index in the management of the data structure, for both storage and access (which exploit 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 block where it is stored (again exploiting the distributed allocation). Also, it departs from the cache, then not requiring any storage at the client side.

The basic idea of our approach is to randomly partition data among three independent storage 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 allocated 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 (distributed cover) at the same level at each of the other servers.

The reader may wonder why we are distributing the index structure 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. Also, the data swapping 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 considerably better protection. Swapping ensures that data are moved out from a server at every access, 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 (see Section 7).

4 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 different 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 of tuples/values stored in a traditional $B+$-tree of the same height. Formally, each

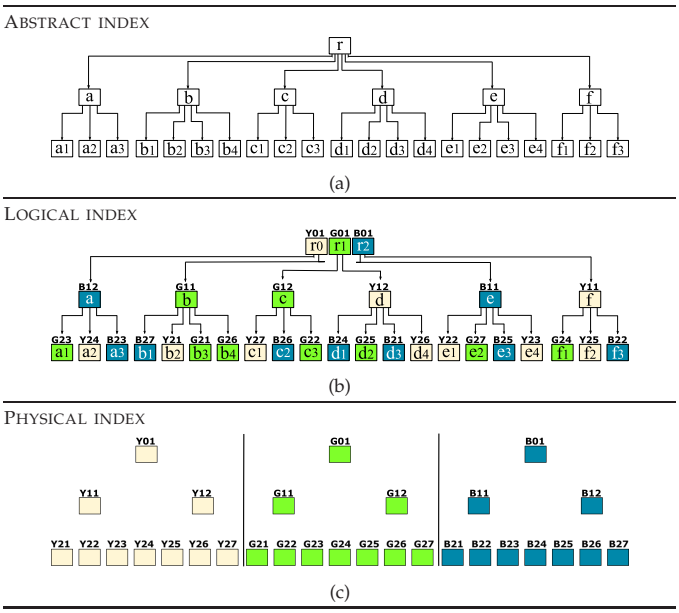


Fig. 1: An example of abstract (a), logical (b), and physical (c) index distributed at three servers

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 pointers to children; and r_2 stores the remaining values $v_{2i+3}, \ldots, v_q$, and the corresponding pointers to 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.) Figure 1(b) illustrates an example of logical index representing the abstract index in Figure 1(a) where the abstract root r is represented by three logical nodes, r_0, r_1, r_2, each having two of the six children of 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 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 distributed index $\mathcal{I}$, S_Y, S_G, S_B be the servers storing $\mathcal{I}$, and $\mathcal{ID}_Y$, $\mathcal{ID}_G$, $\mathcal{ID}_B$ be the sets 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, corresponding to light, medium, and dark gray in b/w printout). For simplicity and easy reference, logical identifiers start 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 their first digit denotes its level in the tree. For instance, G_{24} is the logical identifier of a node at level 2 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 storage servers where these nodes are physically stored. To guarantee distribution among the different servers (and provide uniform visibility at every server in access execution), the distributed allocation function guarantees that at level 1 (children of the root) there is at least one node stored at each storage server, and that each non-root node in the index 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 the three servers (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 distributed 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. Formally, the block b representing a node n is the concatenation $\mathcal{E}||\mathcal{T}$ of two strings obtained as follows: $\mathcal{E}=E_{k_1}(salt||n)$ and $\mathcal{T}=MAC_{k_2}(id||\mathcal{E})$, with E a symmetric encryption function, k_1 the encryption key, *MAC* a strongly unforgeable keyed cryptographic hash function with key k_2, and $salt$ a randomly chosen value. The presence of the node identifier in each block enables 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, with $i \in \{Y, G, B\}$, 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. In the following, 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.

5 Working of the approach

In this section, we illustrate how access execution is performed adopting *distributed covers* and *swapping* protection techniques to guarantee data and access confidentiality.

5.1 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 integrity of) 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 topology of the index structure, 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 server S_Y and the last two stored at server 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 and at each access. In other words, we want every server to observe, for every search, the access to one block at each level, with each server then operating as if it was the only one serving the client. This approach guarantees that each server has uniform visibility over every access, independently from the allocation of the target of the search. (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: *1)* $\sigma(n.id) = S_i$; and *2)* n is at level l in $\mathcal{I}$.

In other words, for each access, one and only one node per level should be accessed at every server. To illustrate, our two sample accesses above do not satisfy uniform visibility. For instance, in the first access S_G is accessed for level 0 (G_{01}), but not for levels 1 and 2. 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 predetermined as data allocation is unknown, they may not represent a path in the distributed 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: *1)* n, n_i, n_j belong to the same level of $\mathcal{I}$; and *2)* $\sigma(n.id) \neq \sigma(n_i.id)$, $\sigma(n.id) \neq \sigma(n_j.id)$, 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 l of the structure 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}) are the distributed covers 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 roots 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 randomly chooses two distributed covers for n. 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 node d_1 to be the target. 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. Figure 2(a) illustrates the nodes involved in the access, either as target (denoted with a bullet) or as distributed covers, at each level also indicating the parent-child relationship among them at the abstract level. Figure 2(b) provides the same information distinguishing the nodes accessed at every server. Note that each server simply observes a sequence of three accesses to three blocks, while the node content (reported in the figure for clarity) is not visible to the servers.

Even if any pair of nodes at the same level as n, but allocated at the other two servers, can work as distributed

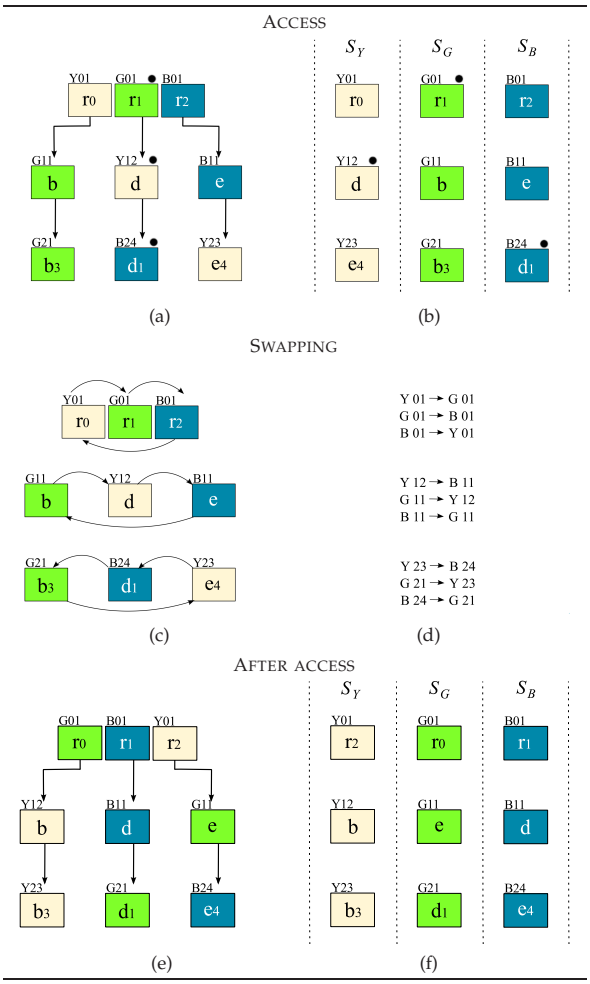


Fig. 2: Logical (a) and physical (b) view of the nodes/blocks accessed searching for value d_1; swapping among accessed nodes/blocks (c-d); logical (e) and physical (f) view of the effects of the swapping.
Target nodes/blocks are denoted with •

covers for n, 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 structure. 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.

Definition 5.2 (Chained distributed covers). Let $p = \{n_0, \ldots, n_h\}$ be the path (sequence of nodes) to the target, and $\mathcal{C}(p) = \langle (n_{0i}, n_{0j}), \ldots (n_{hi}, n_{hj}) \rangle$ be the sequence of distributed covers for the nodes in the path. $\mathcal{C}(p)$ is *chained* if $\forall x = 1, \ldots, h$, n_{xi} and n_{xj} are children of one of the nodes in $\{n_{x-1}, n_{(x-1)i}, n_{(x-1)j}\}$.

In other words, every node in $\mathcal{C}(p)$, but the roots, must have its parent in $\mathcal{C}(p)$. The distributed covers in Figure 2(a) 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 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. The reason for such a choice is twofold: it provides better protection to the parent-child relationship among accessed nodes, and it permits to find more easily distributed covers for target nodes. 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} still satisfying Definition 5.2.

5.2 Swapping

A desired requirement of our approach is that data retrieved (either as target or as cover) in an access are stored after the access at a different server. 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 covers 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.

Continuous moving prevents servers from building knowledge based on accesses they can observe, since 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 $\pi : ID \rightarrow ID$ such that $\forall id \in ID$, $\sigma(id) \neq \sigma(\pi(id))$.

Figure 2(d) illustrates a possible swapping among the nodes/blocks accessed at each server by the search in Figure 2(a-b), resulting in swapping content among them as depicted in Figure 2(c). For instance, swap $Y_{01} \rightarrow G_{01}$, $G_{01} \rightarrow B_{01}$, $B_{01} \rightarrow Y_{01}$ 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(e) illustrates the effect of such a swapping on the data structure at the logical level. Figure 2(f) shows the changes in the content of blocks stored at each server. 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 distributed 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

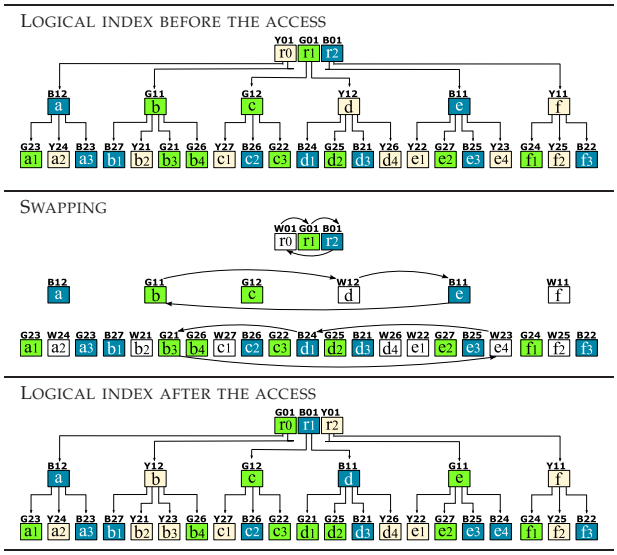


Fig. 3: Evolution of the logical index in Figure 1 for the search of d_1 illustrated in Figure 2

reference to the example in Figure 2, we note that each node is associated with exactly one identifier and vice versa before and after the access. Figure 3 illustrates the logical distributed index before and after the access searching for d_1 and the swapping among accessed nodes, which preserves the correctness of the allocation function (Definition 4.1).

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 three 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, as illustrated in the next section, we check that swapping does not create configurations where a server is not represented in the descendants of a node.

5.3 Access execution algorithm

Figure 4 illustrates the pseudocode of the algorithm, executed at the client-side, searching for a value in our approach. The algorithm visits the index level by level, starting from the root. At each level l the algorithm chooses two distributed covers for the node along the path to the target, accesses the target and cover blocks, decrypt them, and swap their content. To guarantee consistency of the index, swapping is also reported in the parents of accessed nodes, which are then re-encrypted and re-written at the storage servers. Finally, the algorithm returns the leaf node storing the target value.

Theorem A.1 in the Appendix formally states and proves the correctness of the algorithm.

6 Protection Analysis

We evaluate the protection of our approach with respect to guaranteeing confidentiality of the accesses against possible

/* $\mathcal{I}$=$\langle \mathcal{N}$, (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 π be a permutation of identifiers of nodes
in *Parents* s.t. $\sigma(id)\neq\sigma(\pi(id))$
3: swap nodes in *Parents* according to π
4: **for** l:=1. . .h **do** /* visit the index level by level*/
5: *target_id* := id 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 σ(*target_id*)$\neq\sigma$(*cover*[1]),
σ(*target_id*)$\neq\sigma$(*cover*[2]), and σ(*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 π be a permutation of identifiers of nodes in *Read*
s.t. $\sigma(id)\neq\sigma(\pi(id))$ and
each $n\in$*Parents* has at least one child at S_Y,S_G,S_B
9: **if** π does not exist, **then** goto 6
10: swap nodes in *Read* according to π
11: update pointers to children in *Parents* according to π
12: encrypt and write each node $n\in$*Parents* at server $\sigma(n.id)$
13: *target_id* := π(*target_id*)
14: *cover*[1] := π(*cover*[1]), *cover*[2] := π(*cover*[2])
15: *Parents* := *Read*
16: encrypt and write each node $n\in$*Parents* at server $\sigma(n.id)$
17: return node $n\in$*Read* with $n.id$=*target_id*

Fig. 4: Access algorithm

observers. In particular, we consider the servers as our observers as they have the most powerful view over the stored data as well as of the accesses to them (Section 6.1). Guaranteeing confidentiality of the accesses means hiding to the servers the correspondence (as our distribution and swap aim to do) between nodes and blocks where they are stored. We perform our analysis considering two different, opposite, starting scenarios. The first one represents a worst case scenario where, at initialization time, each server knows the node-block correspondence exactly (Section 6.2). We then illustrate how our approach is able to quickly destroy the knowledge of the servers at every access, even in presence of collusion among them. The second scenario considers instead the case where the servers do not have any knowledge at initialization time about node-block correspondence (Section 6.3). We then illustrate how our approach prevents the servers from building knowledge on the node-block correspondence based on their knowledge on the accesses being performed.

6.1 Modeling knowledge

The storage servers know (or can infer from their interactions with the client) the following information: the total number of blocks (nodes) in the distributed index; the height h of the tree structure; 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 nor can infer the content and the topology of the index (i.e., the pointers between parent and children), thanks to the fact that nodes are encrypted. For simplicity, but without loss of generality, we focus our analysis only on leaf blocks/nodes, since leaves are considerably more exposed than internal nodes. Internal nodes are more protected since they are

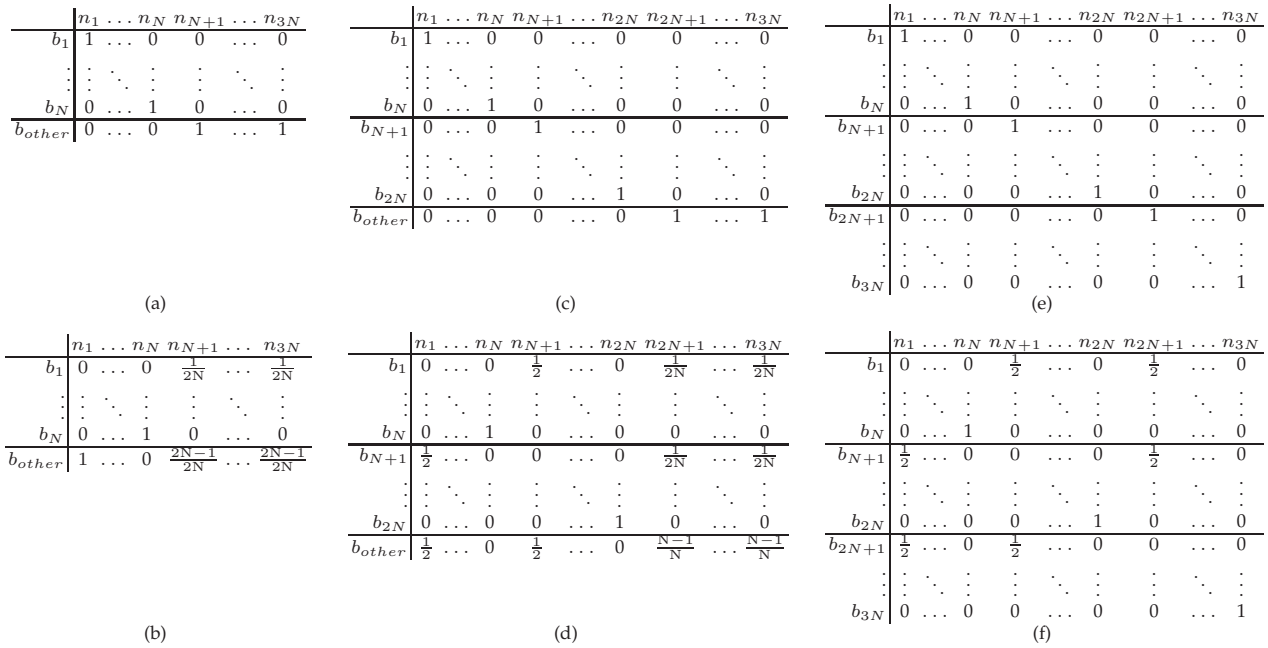


Fig. 5: Probability matrices in the worst case scenario at initialization (a,c,e) and after the first access (b,d,f) to blocks b_1, b_{N+1}, b_{2N+1} with: no collusion (a,b), collusion among two servers (c,d), and full collusion (e,f)

accessed, and hence involved in swapping operations, more often than leaf nodes.

Let $\mathcal{N}$ be the set of logical leaf nodes in the unchained $B+$-tree, and $\mathcal{B}$ be the set of blocks storing them at any of the servers. We assume data to be equally distributed among storage servers S_Y, S_G, and S_B (i.e., the number N of nodes stored at each server is equal to $\frac{|\mathcal{N}|}{3}$). For concreteness and simplicity of notation, we assume that $|\mathcal{N}|$ is a multiple of 3, and that blocks $b_1, \ldots, b_N$ are at S_Y, blocks $b_{N+1}, \ldots, b_{2N}$ are at S_G, and blocks $b_{2N+1}, \ldots, b_{3N}$ are at S_B.

The knowledge of a server S_x, with $x \in \{Y, G, B\}$, on the fact that a node n is stored at a block b can be expressed as the probability $P_x(b,n)$ that S_x knows that node n is stored at block b. Probability $P_x(b,n)$ has value 1 if the server knows with certainty that n is stored at b, and value $\frac{1}{|\mathcal{N}|}$ for every n if the server does not have any knowledge on node-block allocation (i.e., the block could contain n or any other node in $\mathcal{N}$). The overall degree of uncertainty of a server S_x about the block containing a node n can be represented as the entropy H_n^x, computed on the non-zero probabilities $P_x(b_i, n)$, for all $b_i \in \mathcal{B}$, that is, $H_n^x = -\sum_{i=1}^{|\mathcal{B}|} P_x(b_i, n) \log_2 P_x(b_i, n)$. Note that $H_n^x = 0$ means that the server knows exactly the block storing n. In fact, in this case $P_x(b_i,n)=1$ for each block b_i and then $H_n^x = -\sum_{i=1}^{|\mathcal{B}|} 1 \log_2 1 = 0$. On the contrary, $H_n^x = \log_2 |\mathcal{N}|$ means that the server has complete uncertainty about such a correspondence. In fact, $P_x(b_i,n)=\frac{1}{|\mathcal{N}|}$ for each block b_i and then $H_n^x = -\sum_{i=1}^{|\mathcal{B}|} \frac{1}{|\mathcal{N}|} \log_2 \frac{1}{|\mathcal{N}|} = \log_2 |\mathcal{N}|$.

6.2 Knowledge degradation

In the worst case initialization scenario, every server S_x, with $x \in \{Y, G, B\}$, initially knows the exact correspondence between nodes and its blocks (i.e., $H_n^x = 0$ since $P_x(b,n)=1$ if n is allocated at b, $P_x(b,n)=0$ otherwise, with n a node in $\mathcal{N}$ and b one of the blocks stored at S_x). We show, for each node n, the evolution of entropy H_n^x (i.e., knowledge degradation) for each server as a consequence of a random sequence of accesses. For concreteness, we refer the discussion to server S_Y (but the same applies to S_G and S_B) and write $P(b,n)$ instead of $P_Y(b,n)$ and H_n instead of H_n^Y.

No collusion. We first consider the natural configuration where servers do not collude, that is, each server has knowledge of the overall sets of nodes but observes only the encrypted content and accesses to the blocks it stores. At initialization time, S_Y has complete knowledge on the node-block correspondence for the blocks it stores, while it does not have any information on the allocation of nodes to blocks stored at the other two servers. For simplicity, we assume each node n_i to be initially allocated at block b_i (i.e., $P(b_i,n_i)=1$ and $P(b_i,n_j)=0$). Figure 5(a) illustrates the probability values for the different nodes and blocks: each cell $[b_i, n_j]$ in the matrix reports the value of $P(b_i, n_j)$. Consistently with the fact that S_Y does not have any information on blocks at the other servers, all such blocks are summarized in a single row b_{other} in the matrix, which reports the probability that n_j is not at S_Y (i.e., b_{other} is the sum of the probabilities $P(b_i, n_j)$ with b_i a block not at S_Y). We now illustrate how such probability values (and then the entropy) evolve as accesses are executed. In the discussion, we use $\mathbf{b}_i$ to denote the whole row in the probability matrix associated with block b_i, that is, the vector over cells $[b_i, n_j]$, with $j=1, \ldots, 3N$. Operations over rows are to be interpreted to operate cell-wise.

Consider the first access observed by S_Y and let b_y (storing n_y) be the block accessed. Because of swapping, after the access, block b_y will certainly not contain anymore node n_y since the node has moved to one of the blocks at the other two servers (with equal probability among the

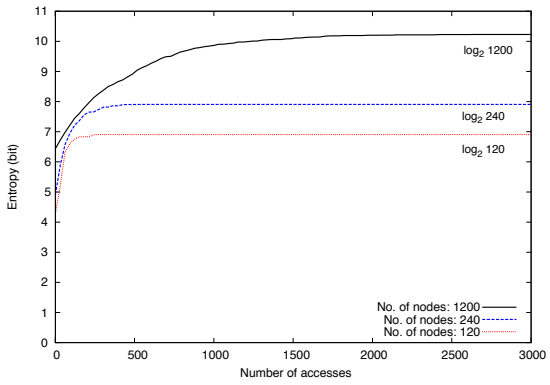


Fig. 6: Entropy evolution varying the number of leaf nodes

blocks, as S_Y does not have any knowledge over them). Hence, $P(b_y, n_y)$ will change to 0 (from 1 before the access) while $P(b_{other}, n_y)$=1 (from 0 before the access). Similarly, b_y could contain after the access, with equal probability, any of the other $2N$ nodes previously stored at one of the other servers. For any n_j with $j>N$, $P(b_y, n_j)$ will change to $\frac{1}{2N}$ (from 0 before the access) and $P(b_{other}, n_j)$ will change to $\frac{2N-1}{2N}$ (from 1 before the access). Assuming, as an example, that the first access is for block b_1, Figure 5(b) illustrates the probability matrix after the access execution. Extending the reasoning to a sequence of accesses, we can formalize the changes to the probability matrix due to the observation of the access to a block b_y as follows (the values in the right side of the formulas are those before the access):

- $\mathbf{b}_y := \frac{1}{2N}\mathbf{b}_{other}$;
- $\mathbf{b}_{other} := \mathbf{b}_y + \frac{2N-1}{2N}\mathbf{b}_{other}$;
- $\mathbf{b}_i := \mathbf{b}_i$, with $i \neq y$ and $i \neq other$.

Note that the first access described above is indeed an instantiation of these formulas. In fact, with reference to our example, $P(b_1, n_1)$ changes from 1 to $\frac{1}{2N} \cdot 0 = 0$ after the access.

To evaluate the increase of entropy due to changes in allocation probabilities, we performed a series of experiments with indexes of different sizes. In the simulations, we considered both uniform and non-uniform distributions of the logical access requests and this aspect did not have a detectable impact on the results. Figure 6 shows the results of the experiments and confirm that the initial knowledge of the server suffers rapid degradation, independently from the index size. While large indexes show a less steep increase in the entropy trend, this is balanced by the greater uncertainty on the node-block correspondences due to their large number of leaf nodes/blocks.

Collusion between two servers. We now evaluate protection (i.e., destruction of knowledge for this scenario) in presence of collusion between two servers. For concreteness and easy notation, let us assume the two colluding servers to be S_Y and S_G. By colluding, S_Y and S_G combine their knowledge of the initial node-block correspondence, producing the initial probability matrix illustrated in Figure 5(c), where b_{other} now refers to the blocks at server S_B, over which S_Y and S_G have no knowledge. Also, S_Y and S_G can combine their observations on the accesses. Assume then the initial configuration and a first observation on the access of block b_y (storing n_y at S_Y) and block b_g (storing n_g at S_G). Let us first consider block b_y and node n_y, as the same (just substituting g for y and vice versa) applies to b_g and n_g. Because of swapping, after the access, block b_y will certainly not contain anymore node n_y since the node content has been moved to either b_g (with probability $\frac{1}{2}$) or to any other (equiprobable) block at S_B (again with probability $\frac{1}{2}$). Hence, $P(b_y, n_y)$ will change to 0 (from 1 before the access) while $P(b_g, n_y)$=$P(b_{other}, n_y)$=$\frac{1}{2}$ (from 0 before the access). Similarly, b_y could contain after the access, either n_g (with probability $\frac{1}{2}$) or any of the nodes previously stored in b_{others} (each with equal probability $\frac{1}{2N}$). Formally, $P(b_y, n_g)$=$\frac{1}{2}$ (from 0 before the access) while for any n_j with j>$2N$, $P(b_y, n_j)$, will change to $\frac{1}{2N}$ (from 0 before the access) and $P(b_{other}, n_j)$ will change to $\frac{N-1}{N}$ (from 1 before the access). Assume, as an example, that the first access is for block b_1 at S_Y and for block b_{N+1} at S_G, Figure 5(d) illustrates the probability matrix after the access execution. Extending the reasoning to a sequence of accesses, we can formalize the changes to the probability matrix due to the observation of the access to blocks b_y and b_g as follows (the values in the right side of the formulas are those before the access):

- $\mathbf{b}_y := \frac{1}{2}(\mathbf{b}_g + \frac{1}{N}\mathbf{b}_{other})$;
- $\mathbf{b}_g := \frac{1}{2}(\mathbf{b}_y + \frac{1}{N}\mathbf{b}_{other})$;
- $\mathbf{b}_{other} := \frac{1}{2}(\mathbf{b}_y + \mathbf{b}_g) + \frac{N-1}{N}\mathbf{b}_{other}$;
- $\mathbf{b}_i := \mathbf{b}_i$, with $i \neq y$, $i \neq g$, and $i \neq other$.

We performed some experiments evaluating the evolution of entropy in presence of collusion between two servers comparing it with the base case where no collusion exists. Figure 7(a) illustrates the results of such experiments for a data structure with $|\mathcal{N}|$=1200, where the solid black line corresponds to the base case of no collusion and the dotted blue line (darker in b/w printout) to the case where two servers collude. We note that even if the entropy shows a less steep increase than in the no collusion scenario (as it is to be expected given the combined knowledge of the two servers), our approach still provides considerable degradation in the knowledge of colluding servers. In fact, even colluding, the two servers still cannot detect whether an accessed node has been allocated to one of them or to any of the other blocks not under their control.

Full collusion. We now evaluate protection (i.e., destruction of knowledge) in presence of collusion among all the three servers. By colluding, the servers can share information on the initial node-block correspondence, which would then be completely known to them as shown in Figure 5(e), and on the block accessed. Assume then the initial configuration and a first observation on the access to blocks b_y (storing n_y at S_Y), b_g (storing n_g at S_G), and b_b (storing n_b at S_B). Again, let us first consider block b_y and node n_y, as the same applies to b_g and n_g, and to b_b and n_b. Because of swapping, after the access block b_y will certainly not contain anymore node n_y as the node has moved either to b_g or to b_b, each with probability $\frac{1}{2}$. Hence, $P(b_y, n_y)$ will change to 0 (from 1 before the access) while $P(b_g, n_y)$=$P(b_b, n_y)$ becomes $\frac{1}{2}$ (from 0 before the access). Similarly, b_y could contain after the access, either n_g or n_b each with probability $\frac{1}{2}$. Such

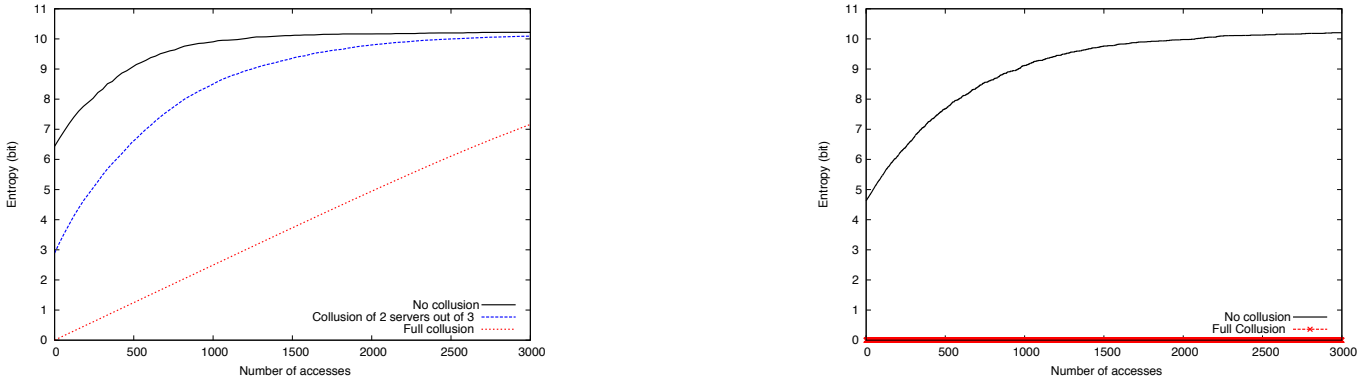


Fig. 7: Entropy evolution for an index with 1200 leaf nodes distributed among three (a) and two (b) storage servers

changes in the probability are illustrated in Figure 5(f), which assumes, as an example, that the first access is for block b_1 at S_Y, b_{N+1} at S_G, and b_{2N+1} at S_B. Extending the reasoning to a sequence of accesses, we can formalize the changes to the probability matrix due to the observation of the access to blocks b_y, b_g, and b_b as follows (the values in the right side of the formulas are those before the access):

- $\mathbf{b}_y := \frac{1}{2}(\mathbf{b}_g + \mathbf{b}_b)$;
- $\mathbf{b}_g := \frac{1}{2}(\mathbf{b}_y + \mathbf{b}_b)$;
- $\mathbf{b}_b := \frac{1}{2}(\mathbf{b}_y + \mathbf{b}_g)$;
- $\mathbf{b}_i := \mathbf{b}_i$, with $i \neq y$, $i \neq g$, and $i \neq b$.

In Figure 7(a), entropy evolution in case of full collusion is represented by the dotted red line (lighter in b/w printout). The increase of entropy (i.e., the knowledge degradation) is clearly lower than in the case of no or partial (between two servers) collusion. However, the knowledge of each server is progressively destroyed thanks to the uncertainty (among the two other servers) of the new allocation of the accessed node.

6.3 Knowledge gain

We now evaluate, with a similar approach, a different initialization scenario, where the servers have no knowledge on the node-block correspondence, and assume that a storage server (or more of them in case of collusion) knows the content of the node n^* target of one every T accesses. However, it does not know the corresponding distributed covers, which are randomly chosen. To model the knowledge of the servers on the content of the target node (one every T access requests), in the analysis of the no collusion, collusion between two servers, and full collusion scenario, we select n^* uniformly at random among the leaves of the abstract index. The block storing n^* is, in turn, chosen following the discrete probability mass function (pmf) specified by the column in the probability matrix associated with the target node n^*. The two blocks representing the distributed covers of n^* are chosen among the blocks stored at the other two servers. In the following, we show for each node n, the evolution of entropy H_n (i.e., knowledge gain) for each server given the execution of a random sequence of accesses.

No collusion. We first consider the natural configuration where servers do not collude. At initialization time, S_Y has no knowledge on the node-block correspondence. Therefore, the initial configuration of the probability matrix is uniform (see Figure 8(a)). Consider the first access observed by S_Y to a known target node n^*, and let b_y be the block accessed at S_Y, and b_i any other block stored at S_Y. Since the target of the access is known to S_Y, it knows for sure that n^* is allocated (both before and after swapping) at one among the three accessed blocks with equal probability. Hence, the probability $P(b_y, n^*)$ will change to $\frac{1}{3}$ (from $\frac{1}{3N}$ before the access), while the probability that node n^* is allocated at a block b_i stored at S_Y different from b_y (and then not accessed) changes to $P(b_i, n^*)=0$. The probability that n^* is allocated at a block stored at a server different from S_Y instead remains unchanged $P(b_{other}, n^*)=\frac{2}{3}$. The values of the probabilities $P(b_y, n_j)$ that each non-target node $n_j \neq n^*$ is allocated at b_y before the access are uniformly redistributed over the whole set of $3N$ blocks, that is, $\frac{1}{3N}P(b_y, n_j)$ is added to $P(b_i, n_j)$, with $n_j \neq n^*$ and $i \neq y$. (In fact, it is more likely for blocks different from b_y to store nodes different from n^*.) Similar changes apply to the probability values for blocks that are stored at S_G and S_B. Hence, $2N \cdot \frac{1}{3N}P(b_y, n_j)=\frac{2}{3}P(b_y, n_j)$ is added to $P(b_{other}, n_j)$. Analogously, $P(b_y, n_j)$ is reduced to be $\frac{1}{3N}$ of its original value, that is $\frac{1}{3N}P(b_y, n_j)$. Figure 8(b) illustrates the probability matrix after the execution of an access with target n^* that accesses block b_1. Extending the reasoning to a sequence of accesses, we can formalize the changes to the probability matrix due to the observation of an access with known target n^* as follows (the values in the right side of the formulas are those before the access):

- $\mathbf{b}_y := \frac{1}{3N}\mathbf{b}_y$ and $\mathbf{b}_y[n^*] := \frac{1}{3}$;
- $\mathbf{b}_{other} := \mathbf{b}_{other} + \frac{2}{3}\mathbf{b}_y$ and $\mathbf{b}_{other}[n^*] := \frac{2}{3}$;
- $\mathbf{b}_i := \mathbf{b}_i + \frac{1}{3N}\mathbf{b}_y$ and $\mathbf{b}_i[n^*] := 0$, with $i \neq y$ and $i \neq other$.

Note that for accesses whose target is not known to S_Y, the formulas illustrated in the case of no collusion among the servers in Section 6.2 apply.

To evaluate entropy evolution, similarly to what done in the previous scenario, we performed a series of experiments considering an index with $|\mathcal{N}|=1200$ leaf nodes,

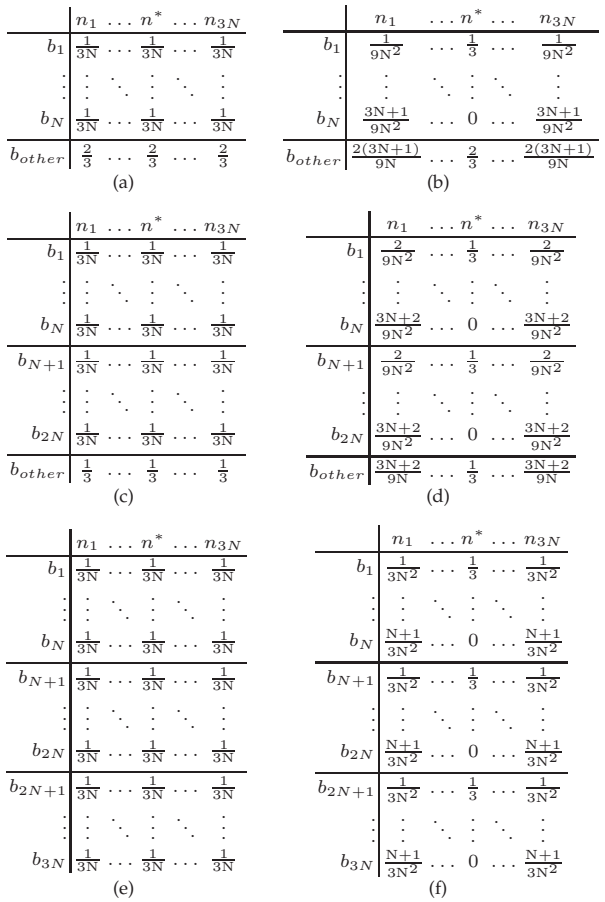


Fig. 8: Probability matrices at initialization (a,c,e) and after the first known access (b,d,f) to blocks b_1, b_{N+1}, b_{2N+1} with: no collusion (a,b), collusion among two servers (c,d), and full collusion (e,f)

and varying the frequency of known accesses to be one every $T\in\{16,8,4,2,1\}$ requests. At initialization time, the probability matrix is uniform and then the initial value of the average entropy is the maximum theoretical one (i.e., $\log_2 1200$). Figure 9(a) shows that the system reaches an equilibrium between the information gain acquired by observing known accesses and the destruction of information caused by swapping. Note that entropy is higher when the frequency of accesses with known target is lower. However, the minimum entropy value reached at equilibrium is high even when the server is supposed to know all the accesses ($T = 1$ in Figure 9(a)). This confirms the effectiveness of the design of our distributed index structure, showing how it is able to guarantee access and pattern confidentiality.

Collusion between two servers. We now evaluate the entropy evolution when two servers (S_Y and S_G for concreteness) collude. Consider the initial uniform configuration (see Figure 8(c)) and assume that the first access has n^* as target and accesses blocks b_y at S_Y and b_g at S_G. The probability matrix evolves similarly to the case in which the servers do not collude. Because of swapping, the target n^* of the access can be allocated to any of the accessed blocks (i.e., b_y, b_g, or a block at S_B) with the same probability. Hence, $P(b_y,n^*)=P(b_g,n^*)=P(b_{other},n^*)=\frac{1}{3}$, while $P(b_i,n^*)=0$ with b_i a block stored at S_Y or S_G different from b_y and b_g. The values of probabilities $P(b_y,n_j)$ and $P(b_g,n_j)$, with $n_j\neq n^*$, before the access are uniformly redistributed over the whole set of $3N$ blocks, including b_y, b_g, and the blocks at S_B. Assuming, as an example, that the first access is known to be for n^* and it accesses block b_1 at S_Y and block b_{N+1} at S_G, Figure 8(d) illustrates the probability matrix after the access execution. Extending the reasoning to a sequence of accesses, we can formalize the changes to the probability matrix due to the observation of an access with known target n^* as follows (the values in the right side of the formulas are those before the access):

- $\mathbf{b}_y := \frac{1}{3N}(\mathbf{b}_y+\mathbf{b}_g)$ and $\mathbf{b}_y[n^*] := \frac{1}{3}$;
- $\mathbf{b}_g := \frac{1}{3N}(\mathbf{b}_y+\mathbf{b}_g)$ and $\mathbf{b}_g[n^*] := \frac{1}{3}$;
- $\mathbf{b}_{other} := \mathbf{b}_{other}+\frac{1}{3}(\mathbf{b}_y+\mathbf{b}_g)$ and $\mathbf{b}_{other}[n^*] := \frac{1}{3}$;
- $\mathbf{b}_i := \mathbf{b}_i+\frac{1}{3N}(\mathbf{b}_y+\mathbf{b}_g)$ and $\mathbf{b}_i[n^*] := 0$, with $i\neq y$, $i\neq g$, and $i\neq other$.

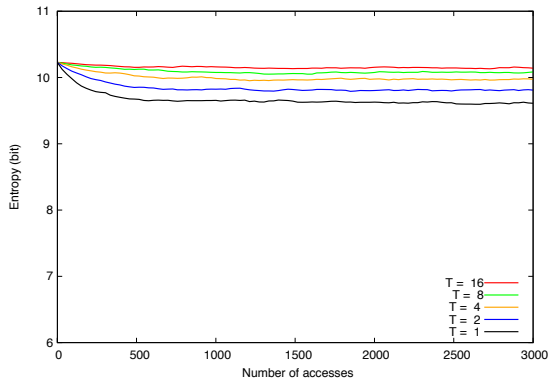
Again, for accesses whose target is not known to S_Y, the formulas illustrated in the case of collusion between two servers in Section 6.2 apply.

Figure 9(b) shows the protection (i.e., the average entropy value) offered by an index with $|\mathcal{N}|$=1200 leaf nodes distributed among three storage servers that know the content of the node retrieved by the client one every $T\in\{16,8,4,2,1\}$ access requests. As expected, the equilibrium is reached at a lower entropy value compared with the case in which there is no collusion. However, the degree of protection exhibited by our solution still guarantees pattern confidentiality.

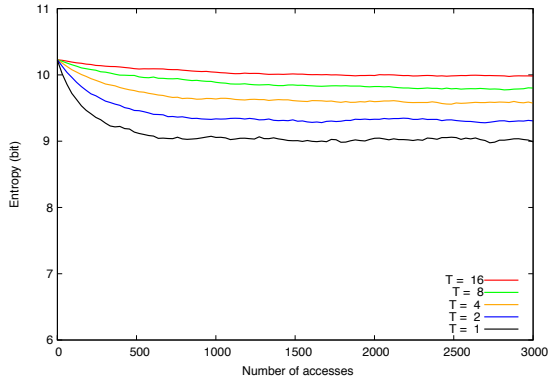
Full collusion. We now evaluate the knowledge gain (i.e., how entropy decreases) in presence of collusion among all the three servers. Consider an initial configuration of absence of knowledge in the node-block allocation (Figure 8(e)). Assume now that the first observation is for an access with n^* as target and over b_y at S_Y, b_g at S_G, and b_b at S_B. By colluding, the servers can share information on the leaf block accessed at each server, therefore they know for sure that n^* is allocated at one among b_y, b_g, and b_b with equal probability. Hence, $P(b_y,n^*)=P(b_g,n^*)=P(b_b,n^*)=\frac{1}{3}$, while $P(b_i,n^*)=0$ for each b_i with $i\neq y$, $i\neq g$, and $i\neq b$. Again, the values of probabilities $P(b_y,n_j)$, $P(b_g,n_j)$, and $P(b_b,n_j)$, with $n_j\neq n^*$, before the access are uniformly redistributed over the whole set of $3N$ blocks, including the accessed blocks. These changes in the probability matrix are illustrated in Figure 8(f), which assumes, as an example, that the known target of the first access is n^* and that blocks b_1 at S_Y, b_{N+1} at S_G, and b_{2N+1} at S_B are accessed.

For each access request, we can formalize the changes to the probability matrix due to the observation of the access to blocks b_y, b_g, and b_b like for the case of full collusion described in Section 6.2, except for the T-th access whose target is known, for which the update of the probability matrix is as follows (the values in the right side of the formulas are those before the access):

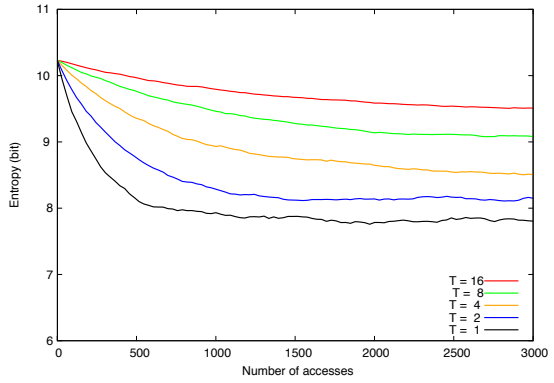
- $\mathbf{b}_y := \frac{1}{3N}(\mathbf{b}_y+\mathbf{b}_g+\mathbf{b}_b)$ and $\mathbf{b}_y[n^*] := \frac{1}{3}$;
- $\mathbf{b}_g := \frac{1}{3N}(\mathbf{b}_y+\mathbf{b}_g+\mathbf{b}_b)$ and $\mathbf{b}_g[n^*] := \frac{1}{3}$;
- $\mathbf{b}_b := \frac{1}{3N}(\mathbf{b}_y+\mathbf{b}_g+\mathbf{b}_b)$ and $\mathbf{b}_b[n^*] := \frac{1}{3}$;



(a)



(b)



(c)

Fig. 9: Evolution of the entropy for a distributed index with 1200 leaf nodes distributed among three storage servers starting from a no-knowledge scenario in a no collusion (a), collusion between two servers (b), and full collusion (c) scenario

- $\mathbf{b}_i := \mathbf{b}_i + \frac{1}{3N}(\mathbf{b}_y + \mathbf{b}_g + \mathbf{b}_b)$ and $\mathbf{b}_i[n^*] := 0$, with $i \neq y$, $i \neq g$, and $i \neq b$.

Figure 9(c) shows the protection (i.e., the average entropy value) offered by an index with $|\mathcal{N}|$=1200 leaf nodes distributed among three storage servers that know the content of the node retrieved by the client one every $T \in \{16, 8, 4, 2, 1\}$ access requests. As expected, the equilibrium is reached at a lower entropy value compared with the cases of no or partial (between two servers) collusion. However, it still presents a significant level of obfuscation of the correlation between nodes and blocks, demonstrating the effectiveness of the swapping technique.

	Shuffling	Swapping
Two servers	2.5	8.8
Three servers	5.8	9.2

TABLE 1: Comparison among the rates of entropy increase (bits/10^4 accesses) for shuffling and swapping with two and three servers, assuming no collusion.

7 Discussion

The main characteristics of our approach are the use of swapping (in contrast to shuffling) and the use of three servers for managing the data structure. In this section we provide the motivation behind these choices. We close the section with a comment on the performance and cost of our approach.

Why swapping. Swapping forces an accessed node to be reallocated to a block at a different server from the one where the node was stored before the access. By contrast, with random shuffling, the node can remain with the server (i.e., under its control) with probability $\frac{1}{S}$, where S is the number of servers (three in our approach). Considering the worst case scenario illustrated in Section 6.2, in absence of collusion (as the system is expected to work) swapping provides a much higher increase of entropy (i.e., a much higher degradation of the knowledge) at each server. Table 1 reports the rate of entropy increase after the first access starting from the full knowledge initial configuration, comparing shuffling and swapping techniques combined with the use of two and three servers (we will discuss on the number of servers next), and assuming no collusion. As it is clear from the table, swapping outperforms shuffling, with entropy increase being three times as much in the case of two servers and twice as much in the case of three servers. The analysis in Section 6.3 further confirms this observation. In fact, even if entropy initially decreases, it never goes below a minimum threshold (i.e., maximum knowledge), which is higher if the servers do not collude.

Why three servers. Collusion, while unlikely, cannot be completely ruled out (or in any case some protection must be provided against it). When only two servers are used, the requirement of swapping to move the accessed node out of its original block implies a deterministic reallocation of the node (to the other server). In case of collusion between the two servers this discloses the block to which the node is re-allocated. Assuming the worst initial configuration in Section 6.2, the determinism of the swapping operation would provide then no increase in entropy in presence of collusion. Figure 7(b) illustrates the entropy evolution for an index with 1200 leaf nodes with the use of two servers. As the figure shows while two servers provide an increase in entropy (as also reported in Table 1) in case of no collusion (solid black line), the entropy would show no increase in the case of collusion (dotted red line). By contrast, as already discussed in the Section 6.2 and visible in Figure 7(a), the use of three servers shows an increase of entropy (and hence protection due to the degradation of the knowledge of the

servers) even when all servers collude. This is also testified by the lower decrease of the entropy obtained when the servers do not have any initial knowledge, but they know the target of each access. In fact, as visible in Figure 9, the minimum entropy reached when the servers do not collude is 7.3% higher than the case of two colluding servers, and 18.7% higher than the case of all the servers colluding.

System Performance. Among the several factors contributing to the response time, our evaluation showed that the latency of the network is the factor with greatest impact in a large-bandwidth WAN scenario (which is the most interesting and natural environment for data outsourcing applications). We considered a data set of 2 GiB stored in the leaves of a $B+$-tree built on a numerical candidate key of fixed-length, with h=3, F=512, and nodes of 8 KiB. The index is distributed among three servers. The client machine runs the algorithm in Figure 4 on an Intel Core i5-2520M CPU at 2.5 GHz with 8 GiB RAM. Each server is equipped with an Intel Core i7-920 CPU at 2.6 GHz with 12 GiB RAM, and 120 GiB SSD disk SATA III with read throughput 240 MiB/s, and write throughput 220 MiB/s. Client and servers run an Ubuntu-OS with the ext4 file system. To configure the network environment, we adopted a professional-grade tool suite (i.e., Traffic Control and Network Emulation, for Linux systems) and we chose a representative continental WAN configuration suitable for interactive traffic with LAN-like bandwidth (i.e., 100Mbps), and round-trip time (RTT) modeled as a normal distribution with mean of 30ms and standard deviation of 2.5ms.Our experiments showed an average overall response time equal to nearly 96ms, which derives from: *i)* the latencies for the execution of three exchanges between the client and the servers, *ii)* the execution at the servers side of accesses to the blocks on the storage devices, and *iii)* the execution at the client side of the algorithm, with the decryption of the received blocks, the analysis of their content, the swapping and the re-encryption of the blocks. The overall response time is then dominated by the network latency (*i*), represented three interactions (one per server) of the maximum among three (one per server) RTTs that follow the above distribution with a 30ms average. The fraction of time (*ii*) employed to execute random read/write accesses on current flash memories is largely $<$0.5ms, which is negligible compared to the previous component. Thanks to the efficiency of symmetric encryption algorithms on modern CPUs, the decryption, analysis, swapping, and re-encryption time (*iii*) on the client has an even lower impact, introducing a delay roughly equal to 0.1ms.

In terms of scalability, the only parameter influencing the performance of the system is the overall size of the outsourced data set as the number of network interactions between the client and each server is proportional to the number of levels of the index (i.e., response time $\propto(h)\times$RTT). We note that the number of servers has a limited (or no) impact on response times, as servers are simultaneously accessed.

Economic considerations. One may wonder how the involvement of three servers (in contrast to one or two) impacts the overall costs of the system. In this section, we provide some economic considerations on the approach. The price lists of most cloud servers present three cost components (we take the March 2015 prices of Amazon S3 as a reference; similar pricing schemes are used by the other providers): *1)* monthly amount of stored data (US$ 30 per month per TB); *2)* number of access requests (US$ 5 per million PUT requests, and US$ 0.4 per million GET requests); and *3)* amount of data transferred out of the server (roughly US$ 80 per TB; data sent to the server is free of charge). The second parameter dominates the third one when requests transfer on average less than 50KB, which is the case for our index (the node size we typically used in experiments is a few KB). A simple analysis shows that, for an index distributed at three servers, the storage and access costs are comparable when the system has to manage around 10K index access requests per day over a 1TB data collection. More precisely, when the access frequency is lower, the storage costs dominate; when the access frequency is higher, it is the cost of upload (PUT) requests that dominates.[1] However, for systems with a low ratio between access frequency and storage size the costs for a solution involving three servers is comparable with those incurred for a solution with a single server. This is due to the fact that the overall memory needed for our distributed index structure is independent on the number of servers on which the index is stored, and the access cost will linearly increase with the number of servers used.

8 Related work

The problem of protecting data in the cloud requires the investigation of different aspects (e.g., [4], [5], [6], [7]). Approaches supporting query execution in data outsourcing scenarios consist in attaching to the encrypted data some metadata (indexes) used for fine-grained information retrieval (e.g., [4], [8]), or in adopting specific cryptographic techniques for keyword-based searches (e.g., [9]). These solutions however 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., [10]), even when different copies of the data are stored at multiple non-communicating servers (e.g., [11]). 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., [12]), in particular with recent proposals aimed at making ORAM more practical such as ObliviStore [13], Path ORAM [14], and Melbourne Shuffle [15]. ORAM has also been recently extended to operate in a distributed scenario [16], [17]. 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

1. PUT requests are 12.5 times more expensive than GET requests; in our distributed index PUT and GET requests occur with similar frequency, then the cost of PUT requests dominates that of GET requests.

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., [18], [19]) to preserve content and access confidentiality.

The shuffle index has been first introduced in [1] and then adapted in [20], [21] 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 [20] provides lower protection guarantees, as also demonstrated by our evaluation. The distribution of the shuffle index among three servers, together with the adoption of swapping, for protecting access and pattern confidentiality has been first proposed in [22]. This paper considerably extends our prior work by providing a comprehensive analysis and an experimental evaluation of the protection guarantees provided by the adoption of our techniques.

A different, although related, line of works is represented by fragmentation-based approaches for protecting data confidentiality (e.g., [5], [23]). 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.

9 Conclusions

We have proposed an approach that protects both the confidentiality of data stored at external servers and the accesses to them. This 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. Our analysis illustrates the protection offered by our approach considering two representative scenarios. We considered first a worst-case scenario where servers start with a complete knowledge of the data they store, showing how swapping quickly brings to a degradation of such a knowledge. We also analyzed a scenario where the servers do not have initial knowledge, but know the individual accesses, and show how our approach prevents knowledge accumulation. Our analysis confirms that distributed allocation and swapping provide nice protection guarantees, typically outperforming traditional shuffling, even in presence of collusion.

Appendix

Theorem A.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 4:

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 a distributed index representing the original unchained $B+$-tree (*structure correctness*).

Proof: (SKETCH). We separately prove each of the conditions in the theorem.

1) Uniform visibility. At the root level, the algorithm accesses the three root nodes (line 1). If the root nodes are stored at three different servers before the access (as it is by definition of distributed index), Property 5.1 holds for level 0. For each level l=1, . . . , h the algorithm accesses the target node (line 5) and two cover nodes (Definition 5.1), which are nodes at level l stored at different servers (line 6). Since the distributed index is stored at three servers, Property 5.1 holds also for every level l=1, . . . , h. Note that, given a block n, there always exists a pair of distributed covers for it. In fact, it is sufficient for the node along the path to the target to have a child at each server as the distributed covers of n can be its siblings. Since the index is initially distributed in a balanced way to the servers and, at each access, the algorithm guarantees that each node still have a child at each server (lines 8-9), each node always has at least a child at each server.

2) Continuous moving. At the root level, the algorithm moves the content of the three accessed blocks according to a permutation function π that satisfies Definition 5.3 (lines 2-3). Such a permutation always exists since the three root nodes are initially represented by three blocks stored at three different servers. Hence, the three roots are stored at a different server after the access, satisfying Property 5.2 for level 0. For each level l=1, . . . , h the algorithm moves the content of the target node and of its two distributed covers according to a permutation function π that satisfies Definition 5.3 (lines 8-10). In fact, a node and its distributed covers are three blocks stored at three different servers, for which a permutation π that satisfies Definition 5.3 always exists. Property 5.2 is then satisfied also for every level l=1, . . . , h.

3) Distribution invariance. Since function π, used to move the content of nodes to different blocks at each access, is a permutation function (line 2, line 8), the set of logical node identifiers (i.e., $\mathcal{ID}_Y$, $\mathcal{ID}_G$, $\mathcal{ID}_B$) does not change and then also their allocation to the servers.

4) Access correctness. The algorithm first accesses all the logical root nodes. Hence, it certainly accesses also the node at level 0 along the path to the target value. For each level l, the algorithm identifies, among the children of the nodes accessed at level $l-1$, the node along the path to *target_value*. Since the node at level l along the path to the target is the child of the node at level $l-1$ along the same path, the algorithm will find it. Note that swapping does not affect the correctness of the algorithm since variables *target_id*, *cover*[1], and *cover*[2] (used to keep track of the node along the path to the target and its distributed covers) are updated according to π before passing to the next level in the tree.

5) Structure correctness. The only operation that could compromise the consistency of the abstract data structure is swapping. However, every time a non-root node n is moved to $\pi(n.id)$, the reference to n in its parent is updated according to π (line 11). Such an update is made permanent by writing back at the servers the parent node (line 12). The parent of each accessed node at level $l>0$ is a node in variable *Parents*, which stores the set of nodes accessed at level $l-1$. In fact, initially *Parents* is set to the three root nodes. At the end of each iteration of the **for** loop, it is set (for the next iteration) to the set *Read* of nodes accessed at level l. Since the nodes accessed at level $l>0$ are direct descendants of the ones accessed at level $l-1$ (lines 5-6), the direct ancestor of each node in *Read* is in *Parents*. □

ACKNOWLEDGMENTS

This work was supported in part by: the EC under grant agreements 312797 (ABC4EU) and 644579 (ESCUDO-CLOUD), and the Italian MIUR within 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] 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.
[3] 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.
[4] P. Samarati and S. De Capitani di Vimercati, "Data protection in outsourcing scenarios: Issues and directions," in *Proc. of ASIACCS*, Beijing, China, April 2010.
[5] 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, 2010.
[6] R. Jhawar, V. Piuri, and P. Samarati, "Supporting security requirements for resource management in cloud computing," in *Proc. of CSE*, Paphos, Cyprus, December 2012.
[7] F. Hu, M. Qiu, J. Li, T. Grant, D. Tylor, S. McCaleb, L. Butler, and R. Hamner, "A review on cloud computing: Design challenges in architecture and security," *CIT*, vol. 19, no. 1, pp. 25–55, 2011.
[8] 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.
[9] 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.
[10] 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.
[11] C. Cachin, S. Micali, and M. Stadler, "Computationally private information retrieval with polylogarithmic communication," in *Proc. of EUROCRYPT*, Prague, Czech Republic, May 1999.
[12] 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.
[13] E. Stefanov and E. Shi, "ObliviStore: High performance oblivious cloud storage," in *Proc. of IEEE S&P*, San Francisco, CA, May 2013.
[14] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: An extremely simple Oblivious RAM protocol," in *Proc. of CCS*, Berlin, Germany, November 2013.
[15] 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.
[16] S. Lu and R. Ostrovsky, "Distributed Oblivious RAM for secure two-party computation," in *Proc. of TCC*, Tokyo, Japan, March 2013.
[17] E. Stefanov and E. Shi, "Multi-cloud oblivious storage," in *Proc. of CCS*, Berlin, Germany, November 2013.
[18] P. Lin and K. Candan, "Hiding traversal of tree structured data from untrusted data stores," in *Proc. of WOSIS*, Porto, Portugal, April 2004.
[19] 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, September 2011.
[20] 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.
[21] ——, "Supporting concurrency and multiple indexes in private access to outsourced data," *JCS*, vol. 21, no. 3, pp. 425–461, 2013.
[22] ——, "Protecting access confidentiality with data distribution and swapping," in *Proc. of BDCloud*, Sydney, Australia, December 2014.
[23] G. Aggarwal *et al.*, "Two can keep a secret: A distributed architecture for secure database services," in *Proc. of CIDR 2005*, Asilomar, CA, January 2005.

Sabrina De Capitani di Vimercati is a professor at the Computer Science Department, Università degli Studi di Milano, Italy. Her research interests are in the area of security, privacy, and data protection. She has been a visiting researcher at SRI International, CA (USA), and George Mason University, VA (USA). She chairs the IFIP WG 11.3 on Data and Application Security and Privacy.
http://www.di.unimi.it/decapita

Sara Foresti is an associate professor at the Computer Science Department, Università degli Studi di Milano, Italy. Her research interests are in the area of data security and privacy. Her PhD thesis received the ERCIM STM WG 2010 award. She has been a visiting researcher at George Mason University, VA (USA). She has been serving as PC chair and member of several conferences.
http://www.di.unimi.it/foresti

Stefano Paraboschi is a professor and deputy-chair at the Dipartimento di Ingegneria of the Università degli Studi di Bergamo, Italy. He has been a visiting researcher at Stanford University and IBM Almaden, CA (USA), and George Mason University, VA (USA). His research focuses on information security and privacy, Web technology for data intensive applications, XML, information systems, and database technology.
http://cs.unibg.it/parabosc

Gerardo Pelosi is an assistant professor at the Department of Electronics, Information and Bioengineering, Politecnico di Milano, Italy. His research interests focus on computer security and privacy, secure storage and data management, and applied aspects of cryptography. He has been serving as a PC member of several conferences. He is inventor of ten granted patents on hardware design of cryptographic systems.
http://home.deib.polimi.it/pelosi

Pierangela Samarati is a professor at the Computer Science Department, Università degli Studi di Milano, Italy. Her main research interests are in data protection, security, and privacy. She has published more than 240 papers in journals, conference proceedings, and books. She has received several awards. She has been named ACM Distinguished Scientist (2009) and IEEE Fellow (2012).
http://www.di.unimi.it/samarati