2016 IEEE 8th International Conference on Cloud Computing Technology and Science

A dynamic tree-based data structure for access privacy in the cloud

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

***Abstract*—We present a novel approach for guaranteeing access privacy to data stored at an external cloud provider. Our solution relies on the grouping of resources into buckets then organized with a binary search tree. The tree is built on an index computed in a non-invertible non-order preserving way, and supports efficient key-based retrieval. Our approach to provide access privacy builds on this data organization providing uniform observability to the server in access execution and dynamically changing not only the physical storage allocation, but also the logical structure itself. Our analysis and experimental evaluation show the effectiveness of our approach.**

***Keywords*—Access privacy, dynamic data structure, key-based retrieval, binary search tree**

I. Introduction

The fast and considerable advancements in ICT solutions and available services have brought to an ever increasing adoption of and reliance on external services for storing, sharing, and accessing data. Together with convenience, involvement of external services brings also the natural worry of ensuring protection of confidentiality of possibly sensitive information. The research and industrial communities have shown considerable interest and attention to the problem of protecting the confidentiality of data stored in the cloud, investigating different aspects of the problem (e.g., [1], [2]). In general, data are assumed to be encrypted before outsourcing and to remain non intelligible to the server providing access to them. If ensuring protection of the data in storage is well understood and can rely today on a fine range of solutions, the problem of guaranteeing confidentiality of the access itself (access privacy), while recognized as important, is still in its early days. Protecting access confidentiality requires maintaining confidential the fact that an access request aims at a specific piece of information or that two requests aim at the same target. Besides desirable by itself, confidentiality of access strengthens storage confidentiality (as a breach of access confidentiality could leak information on the actual data content behind the encrypted storage).

Traditionally addressed within the line of work of Private Information Retrieval (PIR) [3], [4] (known to suffer from high computational complexity), access confidentiality has been recently addressed by several researchers aiming at more practical solutions, limiting computational overhead and providing effective key-based retrieval capabilities. Among them, there are the more recent ORAM-based solutions and the shuffle index [5], [6], [7]. A common aspect of these approaches is the idea of breaking the otherwise static correspondence between data and the physical locations where they are stored.

978-1-5090-1445-3/16$31.00 © 2016 IEEE (CloudCom'16)

In this paper, we propose a novel approach to provide access privacy. Our solution groups resources in buckets according to a randomly and non-invertible mapping and associates non-order preserving indexes with buckets, organizing then bucket indexes with a binary search tree. Such bucketization and indexing provide fine support for key-based retrieval while protecting confidentiality of original index values and their relationships. Like previous works, our approach dynamically changes the allocation of buckets (i.e., nodes of the tree) to physical blocks at every access, so to destroy the correspondence between data and physical locations. In addition to this, our approach protects confidentiality by making accesses all look alike from the point of view of the server, and continuously changing the logical organization of data themselves.

The main advantage of our approach is that it does not require to store data at the client. Both the shuffle index [5] and ORAM-based solutions [6], [7] require instead to maintain a local cache and a local map and stash, respectively. We note that while ORAM-based solutions allow also the storage of the local map and stash at the server side, this solution yields a bandwidth blowup of at least two order of magnitude compared with an unprotected solution [8]. Besides not requiring the client to commit storage, being stateless for the client, our approach supports access by multiple clients. Compared with the shuffle index, in addition to dynamically changing physical location of data (as the shuffling does), we also change the logical structure, adding a further level of confusion with respect to observables by the server. Compared with ORAM [6], [7], [8], in addition to providing good reliability guarantees (being resilient to client failures, as all resources are always stored at the server in a complete and consistent way), we enjoy satisfactory performance figures. Considering that the main service provided by data outsourcing applications is the durable and reliable storage of data, our approach keeps the state of the system safely stored on the remote server. Hence, our solution fits well within the replication, backup, and migration mechanisms adopted by any storage back-end application to cope with software or hardware failures during the system lifetime.

II. Overview of the approach

Our goal is to protect access privacy against any possible observer. Since the most powerful observer is the storing server itself, without loss of generality, we assume the server as our observer. Our approach to provide access privacy is based on

2330-2186/16 $31.00 © 2016 IEEE
DOI 10.1109/CloudCom.2016.65

a combination of techniques that avoids causing, in access execution, observables that can be exploited by the server to learn information about the access. A first level of protection is therefore represented by our storage organization, which provides key-based retrieval functionality, while leaving content not intelligible to the server. For storage, data are clustered in buckets, which are then indexed which a key and organized in a binary search tree for it to support efficient retrieval without exposing any information on the original values. Content is encrypted client-side before upload. Hence, the server receives from the data owner a set of encrypted blocks to store, and serves requests accessing them. The application of encryption and the fixed size of the blocks ensure confidentiality in storage of the blocks *content* with respect to the server.

Like other works in this area [5], our approach makes the data structure dynamic (re-allocating nodes in blocks at every access) to destroy the otherwise static correspondence between nodes and blocks where they are stored. In addition to this, we also re-arrange the tree structure itself, thus introducing a further level of protection, and make every access to the data structure uniform (independently of where the actual target of a search is located in the tree). The building blocks of our solution are as follows.

Uniform accesses (Section IV). All accesses download from the server the same (constant) number of blocks, regardless of where the target is located. Blocks non pertaining to the target path are not recognizable as such. This permits to hide the block storing the target among all the accessed blocks.

Target bubbling (Section V). At every access, the target node is moved up in the data structure by means of rotations, causing also a re-arrangement of the data structure. The main motivation for bringing the target up in the tree with rearrangement of the data structure is that a subsequent (or close) search for the same node will not follow the same path in the tree.

Speculative rotations (Section VI). At every access, possible rearrangements (rotations) at the logical level making (sub-)trees shorter can be enforced. Reason for this is to maintain the height of the tree to be at most twice the height of the balanced tree, that is, $2\lfloor\log(|N|)\rfloor$. While not a protection technique per se, rotations also bring protection benefits, since they cause a change in the topology of the tree structure.

Physical re-allocation (Section VII). At every access, all the accessed nodes are allocated to different physical blocks. Reallocation also entails re-encrypting nodes with a nonce (i.e., an always distinct random salt) so to make the nodes not recognizable and re-allocation not traceable.

III. Data organization and storage

We assume data outsourced to be generic resources identified by an index value for the search. For instance, with reference to a relational database, resources are tuples in a relation and the index is the primary key of the relation. For outsourcing, we organize data with a binary search tree. To support efficient key-based (traversal and) retrieval while protecting the ordering among index values: *i)* each node is a *bucket* containing up to Z real resources, and *ii)* index values are mapped into *bucket indexes* (used in the organization of the tree) in a *non-order preserving* way. Bucketization permits

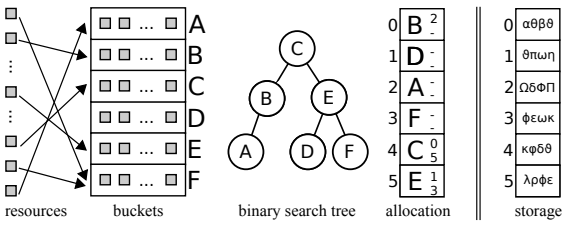


Figure 1. Data structure construction and its physical representation

to limit the height of the tree, and therefore the length of search paths. Non-order preserving mapping protects the order relationship among the content of the nodes involved in a tree traversal (which could otherwise be leaked). We do not make any assumption on the mapping of original indexes to bucket indexes as long as such mapping: *i)* is *non invertible* (to avoid reconstructing the original index knowing the bucket index), *ii)* does *not* require storing any *explicit map* at the client (i.e., it is simply a function that can be computed at run-time), *iii)* is *not order-preserving* (so to protect the order relationship among original indexes), and *iv)* resources are *well distributed* among the nodes (to have all nodes indistinguishably of the same size, nodes with fewer resources are padded with dummies, whose occurrences should be kept limited). A simple approach to provide such a mapping consists in applying a pseudo-random function on the original index values and mapping to the same bucket the pseudo-random values with the same value for a given number of most significant bits.

At the physical level, each node is stored in a physical block in encrypted form. Allocation function $\phi : N \rightarrow ID$ randomly maps each node to the identifier of the physical block where it is stored. Pointers between nodes are represented, at the physical level, by storing in each internal node the identifiers of the blocks storing its children. The content of a block storing a node is obtained by first encrypting the concatenation of the node's content with a random salt to destroy plaintext distinguishability, and then concatenating the result with the output of a MAC (Message Authentication Code) function applied to the encrypted node and the block identifier. Formally, the block content is computed as *b=Enc||Token*, with *Enc=E(salt||n,*k_e*)* and *Token=MAC(id||Enc,*k_m*)* where E is a symmetric encryption function with key k_e, *salt* a randomly chosen salt, and *MAC* a strongly unforgeable keyed cryptographic hash function with key k_m. In this way, the client can assess the authenticity of the node returned by the server as well as of the whole data structure, thanks to the presence of pointers to children in each internal node.

Figure 1 summarizes the data structure construction (bucketization, tree definition, allocation) and its physical representation. At initialization time, we assume the tree to be balanced, and hence with height $\lfloor\log(|N|)\rfloor$. 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 physical structure. When either terms can be used, we will use them interchangeably. Having noted that each node in the binary search tree contains several resources and the ordering of indexes of the tree does not leak any information on the ordering among original index values, from now on we will explain our techniques with reference to the binary search tree and its index.

IV. UNIFORM ACCESSES

Without the application of any protection technique, access execution and server's observations would be as follows. To access the node (which from now on we will call *target*) containing a sought value, the client performs an iterative process, retrieving first the block containing the root node, and then iteratively determining the child to retrieve at the next level until the target is reached. The observation of the server would then be a sequence of requests of block downloads. Serving an access, the server can observe the blocks in the path to the target and the block storing the target (which is the last one downloaded). Since node indexes and their parent-child relationship do not convey any information on the original index values (or their relationship), a path observation does not cause a problem per se. However, accumulating exact knowledge on target nodes and observing multiple searches, the server could observe or infer possible access patterns, as well as - combining observations with possible knowledge of frequencies of accesses to real values - eventually breach access (and even content) confidentiality.

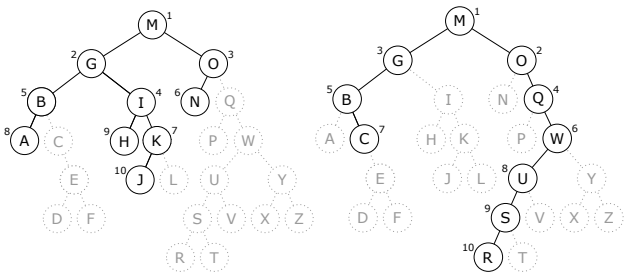
The first level of our protection aims at preventing the server to observe (and accumulate) exact information on the target of a search. To this end, we make accesses all look alike from the point of view of the server, and perform searches in the tree always accessing the same number of nodes (and hence blocks at the server), regardless of where the target is located in the tree, be it the root or the deepest leaf. Setting the (constant) number of nodes to be accessed at every search, we need to ensure that it is sufficient to reach any target, that is, it can cover the longest path in the tree. In a search tree, the number of nodes in the longest path can go from a minimum of $\lfloor \log(|N|) \rfloor + 1$ (balanced tree) to a maximum of $|N|$ (tree degenerated in a list). Aiming at balancing some degree of freedom in the data structure (which we dynamically re-arrange at every access) while avoiding degeneration, we set a limit on the height of the tree to be at most $2\lfloor \log(|N|) \rfloor$ (Section VI illustrates enforcement of such a limit), which is a well recognized performance trade-off between the height of a perfectly balanced tree ($\lfloor \log(|N|) \rfloor$) and the amortized height of an unbalanced tree with the target bubbling mechanism in place ($3\lfloor \log(|N|) \rfloor - 2$) [9]. The longest path in our data structure has at most $2\lfloor \log(|N|) \rfloor + 1$ nodes. Also, we assume the children of the root to always be read. Hence, we set the constant number of nodes to be read at every access to $2\lfloor \log(|N|) \rfloor + 2$. If, as it will typically be the case, the number of nodes in the path to the target plus the other child of the root (meaning the one not in the path to the target) do not reach $2\lfloor \log(|N|) \rfloor + 2$, we complement the access with other nodes, which we call *fillers*. (The reason for assuming both children of the root to be always read is to accommodate flexibility in the choice of indistinguishable fillers.) Every access request will always be translated into a sequence $A = \langle n_1, \ldots, n_m \rangle$, with $m = 2\lfloor \log(|N|) \rfloor + 2$, of accesses to nodes (corresponding to blocks for the server).

In choosing fillers, we need to ensure their indistinguishability from nodes in a target path. In this case, from the point of view of the server, any of the m nodes accessed could correspond to the actual target of the search, others being nodes in the path to the target or fillers, all indistinguishable one from the other. In this respect, choosing fillers just at random at any place in the data structure would not provide such a characteristic, as being completely unrelated in the structure, they could be recognizable as fillers. In fact, while (as we will see later on) we prevent the server from accumulating topological information across accesses, the server can observe a sequence of blocks accessed where a sequence of never-downloaded blocks is followed by a sequence of blocks intersecting with a previous search. This situation would expose the blocks in the intersection as fillers (a path is always connected, hence their occurrence after the never-downloaded blocks implies that they cannot belong to the path). A possible natural choice of selecting fillers at random wherever in the tree could then make them, or others following them in the sequence, recognizable as fillers. To avoid exposing accesses to such intersection attacks, we (randomly) choose fillers in such a way that they are connected to the paths (either target or fillers) being followed in the tree (i.e., a node can be accessed only if its parent has been) and always proceed forward in levels in the tree (i.e., the node visited next in the sequence cannot have a level lower than the one visited before it). Selecting fillers so that they are connected to nodes already accessed (path continuity) and with monotonically non-decreasing levels (forward visit) avoids possible intersection attacks from the server, guaranteeing fillers to be indistinguishable from a genuine path to a target.

Definition 4.1 (Uniform access): Let T be the data structure. A sequence $A = \langle n_1, \ldots, n_m \rangle$ of nodes in T is said to be a *uniform access* iff: *1)* $m = 2\lfloor \log(|N|) \rfloor + 2$ (constant number); *2)* $\forall n_i \in A : n_j \in path(n_i, T) \implies n_j \in A$ (path continuity); *3)* $level(n_i, T) \leq level(n_{i+1}, T)$, $i = 1, \ldots, m-1$ (forward visit).

Ensuring forward visit requires to 'think ahead' for the need of fillers, to avoid being blocked in a situation where there is no node that can be accessed at a level equal to or higher than the last one visited, but fewer than $m = 2\lfloor \log(|N|) \rfloor + 2$ nodes have been accessed. An easy way to avoid ending in such a situation consists in keeping track, in each node, of the number of nodes in the longest path of its children (i.e., for each of them, their height plus one), and, when performing searches, of the number of nodes to be still read to reach the fixed number $2\lfloor \log(|N|) \rfloor + 2$. Searches can then be performed level by level (forward visit). After having read the root and its children, we choose, in addition to the node to the target, one or more filler nodes, children of a node read at the previous level (path continuity), such that the sum of the number of nodes in their longest path is greater than the difference between the number of nodes to be still read and the maximum length of the path to the target. If a target is retrieved at level l, the search (at that and subsequent levels) continues with filler nodes only. The nodes to be accessed at each level in the tree are downloaded in sequence and in random order, to prevent the server from identifying how many blocks are accessed at each level and which of them is along the path to the target.

Figure 2 illustrates two possible accesses on a sample data structure with 26 nodes, which then requires to visit 10 nodes at each access. Nodes involved in the access are circled with solid lines and the numbers at their side represent the order in the sequence of requests to the server. In both accesses, any of the accessed nodes could be the actual target or a filler. Also, the two accesses, while visiting different nodes, could actually correspond to a search for the same target (e.g., B).



(a) (b)

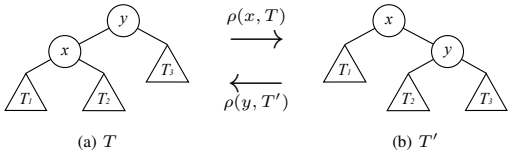
Figure 2. Two sample accesses

V. TARGET BUBBLING

Our second protection technique aims at hiding from the server subsequent (or close) searches for the same target. Even with fillers, such searches would necessarily contain the same path, and this situation could be easily observed by the server that would see access to the common sequence of corresponding blocks. For instance, any search for R in the tree in Figure 2 will visit nodes M, O, Q, W, U, S, R and 3 filler nodes (one of which will always be G), accessing the corresponding blocks. Observing accesses that visit 8 common blocks, the server can reasonably infer that the accesses are for the same target with high probability. The longer the common sequence, the higher the probability that the target of the two accesses is the same. To protect against such intersection attacks, our second technique simply dictates to bubble the target of a search up in the tree, so that at the end of the access, the target appears at the top of the tree (regardless of where it was before the search). A subsequent search for the same target (repeated search) would find the target high in the tree, then randomly proceeding following filler nodes. This would result in an access retrieving a set of blocks different from the previous one, hence appearing to the server as a search for a different target. A repeated search would then not be recognizable as such by the server.

In choosing where to move the target up in the tree, we note that placing the target in the root would seem the best choice for protecting repeated subsequent searches (as any search always accesses the root anyway). This would possibly expose recurrences of the same target at a fixed distance. In fact, after a sequence of all different searches (each aiming at a different target), the target of the m-last search would be at level m in the tree (having first been placed in the root and then moved down m times to accommodate the bubbling of the subsequent targets), and the server could exploit such a knowledge to make inferences on the target of the access. While noting that, due to the synergy with the other techniques of our approach, this situation would not be that deterministic, to avoid any determinism in the first place, we move the target up in the tree choosing the (high) level at which to place it at random. We assume a level *top* (which we expect to be typically 1 to 3) above which the target should be placed. At every access, the new tree level at which the target should be placed is randomly chosen between 1 and the minimum between *top* and the current level of the target (a target is never moved down).

Moving the target up in the tree at the wished level is realized by applying classical single tree rotations of binary search trees. A rotation essentially swaps the child-parent relationship between a node and its parent, placing the node at the level of its parent and making the parent a child of the node (right if the node was the left child of its parent and vice versa). We denote the rotation of a node n in a tree T as $\rho(n, T)$. Figure 3 illustrates the result of such rotations, where tree T' in Figure 3(b) is the result of rotation $\rho(x, T)$ and tree T in Figure 3(a) is the result of $\rho(y, T')$. In the figure, we single-out only nodes directly involved (x and y), representing the remaining ones as sub-trees (T_1, T_2, T_3). Formally, a tree where the target has bubbled up is defined as follows.



(a) T (b) T'

Figure 3. Tree rotations

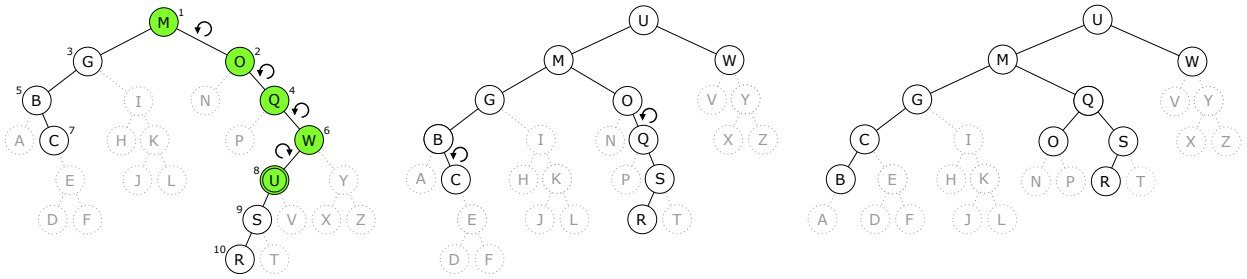
Definition 5.1 (Bubbled target): Let T be the data structure, n be a target node, l=*level*(n, T) be its level before the access, and $l' \leq l$ be the new level at which the target should be placed. The data structure T' equivalent to T where the target has *bubbled up* is $T' = \rho(n, (\rho(n, \ldots (\rho(n, T)))))$, with *level*$(n, T') = l'$, obtained by recursively applying a sequence of $l' - l$ rotations.

For instance, Figure 4(a) illustrates the nodes accessed by a search for U. The target is double circled, nodes in the path to the target are colored, and filler nodes are denoted with solid lines. Curved arrows on arcs show the rotations to bubble the target to the root level, swapping U with (in sequence) W, Q, O, and M. Figure 4(b) shows the resulting tree.

As already noted, since at each access the target is moved up in the tree, the targets of recent accesses will be located high in the tree (close to the root), while nodes that have not been accessed since long time will be at deeper levels in the tree (close to leaves). This is due to the fact that rotations that bubble up the target change the level of the other nodes in the top levels of at most one (up or down), and therefore it takes a few accesses for a high (or raised high) node to move down in the tree (e.g., the root node in Figure 4(a) becomes the left child of the root in Figure 4(b)). We also note that repeated accesses to a same target keep it in the top levels of the tree. This provides protection of repeated searches, since all such accesses, following random filler nodes in the tree, will look all different. We also note that bubbling the target with a recursive sequence of rotations causes changes in the topology of the tree, adding confusion to the server.

VI. SPECULATIVE ROTATIONS

Bubbling up the target after each access causes a natural reorganization of the tree. Because of rotations, at each access the height of the tree can increase (or decrease) by one. A long sequence of accesses can then potentially unbalance the tree structure. To ensure that any node can be reached via a uniform access, we need to maintain the height of the tree to be at most $2\lfloor \log(|N|) \rfloor$. To this end, at every access, we consider nodes involved in the access in decreasing order of level in the tree, and, for each node, we evaluate whether its



(a) accessed nodes and target bubbling (b) speculative rotations (c) resulting tree

Figure 4. Nodes downloaded to access U and rotations performed to bubble up the target (a), tree with the target bubbled to the root and speculative rotations (b), and the resulting tree (c)

rotation can shorten some paths in the tree. Intuitively, rotation $\rho(n_i, T)$ decreases by one the length of the path reaching one of the children of n_i, while increasing by one the length of the path reaching n_i's sibling, say n_j. For instance, with reference to Figure 3, rotation $\rho(x, T)$ shortens of one the length of all paths ending in T_1 and increases by one the length of all paths ending in T_3 (the contrary happens for rotation $\rho(y, T')$). It is then easy to see that rotating n_i is potentially beneficial to shorten (or maintain limited) the height of the tree every time the height of the subtree rooted at n_i is greater than the height of the subtree rooted at its sibling n_j of at least two.

Definition 6.1 (Beneficial rotation): Let T be the data structure, n_i be a node in T, and n_j be its sibling. Rotation $\rho(n_i, T)$ is *beneficial* to possibly keep the height of T limited iff *height*(n_i, T)>*height*$(n_j, T) + 1$.

For instance, the beneficial rotations that are performed over the tree in Figure 4(b), resulting when bubbling up the target, are ρ(C,T) and ρ(Q,T). Enforcing them results in the tree in Figure 4(c). Note how rotating C decreases the length of the paths to D and F, decreasing also by one the height of the tree itself. Note also how the topology of the tree has changed with respect to the tree before the access (Figure 4(a)).

A beneficial rotation does not guarantee to reduce the height of the tree, but it can shorten sub-trees, hence avoiding later degeneration of the structure. At every access, we evaluate whether rotating accessed nodes would be beneficial and, if so, we perform such speculative rotations. This regardless of the height of the tree, to also try to avoid reaching a length close to our limit of $2\lfloor\log(|N|)\rfloor$. As only exception, we never perform rotations on direct children of the target (or of one of its ancestor) as this would decrease the level of the target (which should instead remain at the level where it was bubbled up). For instance, considering the tree in Figure 4(b), even if rotation ρ(M,T) is beneficial, it would move the target to a lower level. In our example, the height of the tree resulting from the application of speculative rotations (Figure 4(c)) is 5 while the height of the tree before access (Figure 4(a)) was 6.

Typically, simply performing beneficial rotations at every access allows maintaining the height of the tree within our aimed maximum of $2\lfloor\log(|N|)\rfloor$. In the unlikely case (one over 3,000 in our experiments) where after such speculative rotations the tree height is $2\lfloor\log(|N|)\rfloor + 1$ (given our control it can certainly never go higher than that at any access), a further pass of rotations can be performed.

An additional advantage of our speculative rotations is related to the protection of access privacy. In fact, a rotation swaps the parent-child relationship between the rotated node and its parent, also changing the parent of one of its children (see Figure 3). Therefore, rotations change the topology of the tree, modifying search paths for some nodes. Since the topology changes at every access, the server cannot accumulate knowledge on it. We note that accommodating different topological structures is also the reason why we do not aim at maintaining the tree perfectly balanced (as the structure would have less degrees of freedom), and set instead - was a trade-off - our height limit to be twice the height of the perfectly balanced tree.

VII. Physical re-allocation

Enforcing rotations to bring up the target and shorten paths changes the logical organization of the tree. As noted, such a change in topology provides some protection since it changes the location of nodes and therefore paths to be followed to reach them. Still, since the tree is a search tree, even if topology changes, strong commonalities can remain even after rotations. For instance, a rotation reverses a parent-child relationship between two nodes n_i and n_j, but still the two will be connected. A sequence of rotations can bring more changes, but still common sub-paths may remain. The fact that the server can infer the path followed to reach a given node is not an issue per se since, as already noted, the index on which the tree is organized does not convey any information on the original index values and their relationships. However, if the server can maintain such a knowledge across the accesses, it can potentially reconstruct the topology of the tree and observe paths in common between different accesses, hence possibly learning information on an access.

We note that the server only observes accesses to blocks (not nodes) and that the parent-child relationship is (partially) known to the server since the access is iterative: a block will be child of one of the blocks accessed before. The uncertainty of the parent-child relationship comes from the fact that more nodes can be accessed at any level of the tree (since in addition to the target, also filler nodes will be followed). The $(i-1)$-th block accessed in an access sequence could be a parent, uncle, brother or even not be in a direct relationship with the i-th accessed block. For instance, in the sequence of nodes A=⟨M,O,G,Q,B, W,C,U,S,R⟩ (Figure 4(a)) accessed to retrieve U, M is parent of O, O is sibling of G, G is uncle of Q, Q is not in relationship with B. However, such uncertainty

cannot provide protection from a server observing common blocks among sequences of accesses. To prevent the server from accumulating information on the topology of the tree, we destroy such information by re-allocating all nodes involved in an access changing their physical location (i.e., changing the blocks where they are stored). At the physical level, and therefore from the point of view of the server, topological information is destroyed. A block id_i that contained the child of another block id_j before an access can now contain a node appearing in a completely unrelated path that might even have only the root in common with the path to id_j, or be the root itself. A subsequent access visiting the same block id_i might (and most probably will) pertain to a completely different path in the tree. In other words, with re-allocation the (even uncertain) information on relationships among blocks that can be observed in an access will not hold anymore after the access is completed, preventing knowledge accumulation by the server. The physical re-allocation of nodes is formally defined as follows.

Definition 7.1 (Re-allocation): Let T be the data structure and $\forall n_i \in T$, $id_i = \phi(n_i)$ be the identifier of the physical block storing n_i before the access. Let A be the nodes involved in an access execution and $\pi : \mathit{ID}_A \rightarrow \mathit{ID}_A$ be a random permutation of $\mathit{ID}_A = \{\phi(n) : n \in A\}$. *Re-allocation* changes the allocation function ϕ for all $n_i \in A$ to be $\phi(n_i) = \pi(id_i)$.

Re-allocation entails moving a node to a different physical block (or leaving it at the same if so dictated by the permutation). Re-allocation requires to re-encrypt the node with a different random salt. All blocks accessed will be rewritten and will all look different from any read block. The server will then not be able to learn any information on the re-allocation process and, in particular, will not be able to trace where the former content of a block might have been re-allocated. We note that, at the physical level, re-allocation also requires to update the parents of the re-allocated nodes, to guarantee the correct representation of pointers to children (and then the correctness of the tree structure). This is not an issue since the path continuity guaranteed by the access (Definition 4.1) ensures that the parent of every node involved in the re-allocation is also involved in the re-allocation (and therefore it is available to the client for content update and re-writing). Figure 5(a) illustrates the original content of the blocks accessed by the search in Figure 4, an example of their physical re-allocation, and their content after re-allocation. In the figure, we report in each node its index value and the identifier of the blocks storing its children (symbol $-$ denotes the absence of the child). The block identifier is reported on the left of each block. Figures 5(b-c) illustrate the server view before (b) and after (c) the access, where blocks downloaded/uploaded are colored.

Thanks to the fact that every node is moved to a different untraceable physical block every time it is accessed, re-allocation prevents the server from determining whether two accesses visited a same node (or sub-path). Hence, the server will not be able to reconstruct the frequency of accesses to nodes by observing accesses to physical blocks. Indeed, accesses that aim at the same target (or visit the same path in T) will access a different set of physical blocks. Furthermore, since re-allocated nodes belong to different paths and are located at different levels in the tree, re-allocation also destroys information the server could have gained on the topology of the tree by observing the sequence of accessed blocks/nodes. In fact, a block storing a node at level i in the tree might contain after the access a node in a completely different path and at a completely different level.

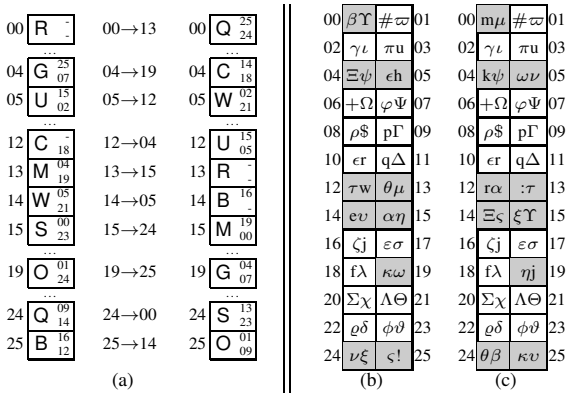


Figure 5. An example of physical re-allocation (a) and of view of the server before (b) and after (c) the access in Figure 4

VIII. Analysis and experimental evaluation

To assess the access privacy provided by our approach we need to evaluate the indistinguishability of accesses or - put another way - the degree of confusion on the accesses to the server. To this end, we start noting that the physical re-allocation employed by our approach can be compared with the physical re-allocation of the shuffle index [5] (it is actually stronger). In particular, the shuffle index re-allocates the logical nodes accessed on disjoint physical paths of its tree structure on a per-level basis. The entropy-based analysis used to show the soundness of such a mechanism in obfuscating the mapping between nodes and blocks applies also to our approach. In addition, our approach enjoys even stronger guarantees than the ones proved in [5]. Indeed, the shuffle index changes physical location of only a limited set of nodes and operates only within level of the logical structure, while our approach changes the allocation of all nodes involved in an access, operating also across levels, hence producing a complete re-allocation of the whole set of accessed nodes. With respect to short-term observations (protected by the cache in the shuffle index) our approach, bubbling the target at the top, is clearly protected since repeated accesses are indistinguishable, as already noted. Enjoying such theoretical analysis and observations, which apply also to our solution, we then performed an experimental analysis on our approach. We evaluated how and to what extent our proposal hides to the server the correspondence between nodes and blocks where they are stored. We implemented our approach in Java and evaluated: *i)* how the *height of the data structure* can vary; *ii)* the *effectiveness of rotations* in protecting access privacy; *iii)* the degree of *obfuscation of the actual paths* observed by the server at every access request due to the physical re-allocation.

In our analysis, we used a data structure with 256 nodes and a height ranging from 8 to 16. We simulated different access profiles by synthetically generating a sequence of target index values that follow a self-similar probability distribution with

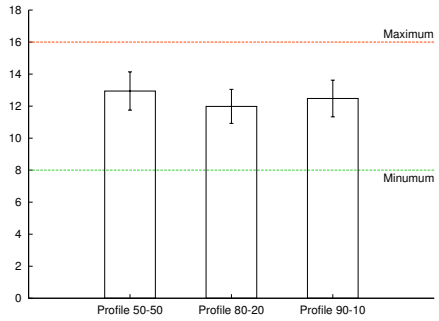


Figure 6. Average height of a tree with 256 nodes, considering $500,000$ accesses

skewness γ in the range [0, 0.5][1]. A value of γ=0.5 generates a sequence of values that follows a uniform probability density function. The results of our experimental evaluation have been obtained executing $500,000$ accesses for target values drawn from three self-similar distributions [10] with γ=0.5 (50-50 rule), γ=0.20 (80-20 rule), and γ=0.10 (90-10 rule), respectively.

Data structure. Figure 6 shows the average height of the tree for the different access profiles. As visible from the figure, the average is around $1.5h$ (with h=$\lfloor \log(|N|) \rfloor$ as a baseline), with a sample standard deviation of 1. Hence, the data structure maintains itself within the set limit, also nicely providing rooms for fillers in the search. Since the height of the tree dictates the number of client-side interactions that would be needed to access the data structure (following a path in the tree), we note that, requesting a constant number of accesses, our solution exhibits a $\times 2$ overhead with respect to an encrypted binary tree (i.e., an encrypted binary tree that still requires the client to visit the tree level-by-level), which however would provide no access privacy protection at all.

Effectiveness of rotations. To evaluate the effectiveness of rotations (target bubbling and speculative) for protecting access privacy, we analyzed the average length of the common prefix paths to a common target in sequences of subsequent accesses. Figure 7(a) shows the results of such an analysis, where the x-axis reports the node identifiers in ascending order, and the y-axis reports the average length of the common prefix. It is interesting to note that the reported average value for the maximum common prefix of the logical paths aimed at the same target is around one third the average height of the tree (Figure 6), implying that rotations largely change the topology of the data structure. Also, if the statistical distribution of the target values is highly skewed, only a few values will be accessed for serving most of the access requests. The two spikes in Figure 7(a) confirm that our approach keeps as near as possible to the root the most recently (and frequently) accessed nodes, thus being effective in making subsequent accesses to the same target indistinguishable from random ones.

Path obfuscation. The experimental evaluation validates the ability of physical re-allocation involving all accessed nodes to provide indistinguishability of the profiles of the accesses to the data structure.

Figure 7(b) shows the rank/frequency distribution of block identifiers observed by the server when only physical re-allocation is applied. The figure shows that the physical re-allocation alone is already able to make skewed frequency distributions of the accesses to the blocks quite close to the one corresponding to a flat access profile.

Combined protection. The small differences among the curves in Figure 7(b) are a consequence of the information leakage coming from the observations of the blocks shared by different access requests. Such differences disappear thanks to the contribution of rotations. This is visible in Figure 7(c), showing the rank/frequency distribution of block identifiers observed by the server during the execution of access requests when all our protection techniques are applied. The figure validates our approach to preserve access privacy as it shows how the proposed techniques make skewed frequency distributions of accesses to the blocks statistically indistinguishable from the one produced by a uniform access profile.

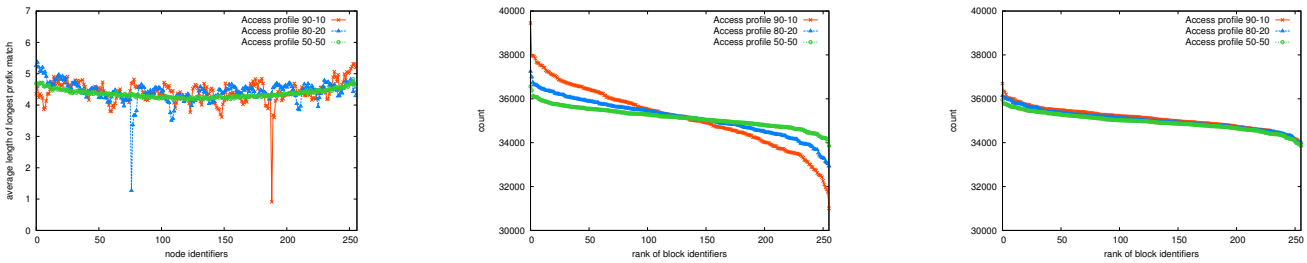
IX. Related work

With the increasing interest in data outsourcing, many proposals have first been devoted to the protection (of the confidentiality) of data in storage (e.g., [11], [12]). Recently, significant attention has been given to the problem of protecting confidentiality of accesses. Current proposals are based on Private Information Retrieval (PIR) techniques or on dynamically allocated data structures, which change the physical location where data are stored at each access (e.g., [3], [4], [5], [6], [7], [13], [14], [15], [16], [17], [18]). PIR solutions are computationally expensive and do not protect content confidentiality (e.g., [3], [4]). Dynamic data structures rely on the Oblivious RAM (ORAM) for protecting content, access, and pattern confidentiality (e.g., [6], [7], [16], [18]), or on tree-based structures (e.g., [5], [13], [14], [15], [17]). While preliminary ORAM-based proposals suffer from high computational and communication overheads, recent attempts (e.g., ObliviStore [6] and Path ORAM [7]) make ORAM more practical in real-world scenarios [19].

Path ORAM based solutions [7] store data both at the server side and in a local cache (stash) at the client side. The client also stores a position map (with size proportional to the number of data blocks) that keeps track of where the data are physically stored. To reduce to one block the storage at the client, recent proposals move the stash from client to server and store the position map recursively on the server in smaller ORAMs. These approaches, however, cause an increase in response time and a bandwidth blowup of over two orders of magnitude in data exchange between client and server [8]. To reach a constant bandwidth blowup, an additively homomorphic encryption construction can be used to perform server computations (i.e., $\widetilde{\omega}(\log^4 N)$, where N is the number of outsourced data blocks), but at the cost of an increased computational effort for the client (i.e., $\widetilde{\omega}(log^2 N)$) [20]. Our approach applies only efficient symmetric encryption primitives and has limited bandwidth blowup and client storage capacity ($\omega(logN)$ blocks) as well as lower computational requirements ($O(\log N)$) at the client side.

Solutions that rely on tree-based data structures provide

[1]Given an index domain of cardinality d, a self-similar distribution with skewness γ provides a probability of $1-\gamma$ of choosing one of the first γd domain values; the same proportion holds when considering any sub-range of the domain values.



(a) (b) (c)

Figure 7. Average length of the maximum common prefix among the paths reaching the same target (a) and rank/frequency distributions of the block identifiers corresponding to self-similar access profiles with $\gamma \in \{0.5, 0.2, 0.1\}$ when only the physical re-allocation (b), and when all protection techniques are applied (c)

a good trade-off between privacy and performance (e.g., [5], [13], [14], [15], [17]). Among them, the shuffle index has first been proposed in [13]. A shuffle index is a dynamically allocated B+-tree offering access and pattern confidentiality, while supporting efficient key-based data organization and retrieval. The B+-tree is stored at the server side in encrypted form and jointly uses cover searches (fake searches indistinguishable from actual searches, executed in parallel), cache (most recent visited target paths), and shuffling for protecting the confidentiality of accesses (corresponding to physical re-allocation). The shuffle index has then been extended to support access control [21] and concurrent accesses by different users [14], to operate in a distributed scenario characterized by the presence of multiple (three) storage servers [15], and to support insertion and removal of tuples in the outsourced relation [5]. The main differences between the shuffle index and our solution is that they are based on different protection techniques and, in particular, the shuffle index does not change the logical tree structure but relies mainly on shuffling. Also, our proposal does not require any client-side storage.

X. Conclusions

We presented a dynamic tree-based data structure for storing resources at an external server and guaranteeing access privacy. Our approach does not require to maintain any storage at the client side. The advantage of being stateless, besides not requiring the client to commit resources, also consists in accommodating multiple clients and providing resilience of the structure against failures or non availability of the client. The dynamically restructuring of the tree at both logical and physical levels provides access privacy, making the frequency distribution of accesses to the physical blocks indistinguishable from the one produced by a uniform access profile.

Acknowledgments

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

References

[1] S. De Capitani di Vimercati, S. Foresti, and P. Samarati, "Managing and accessing data in the cloud: Privacy risks and approaches," in *Proc. of CRiSIS*, Cork, Ireland, October 2012.

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

[3] 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.

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

[5] S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, G. Pelosi, and P. Samarati, "Shuffle index: Efficient and private access to outsourced data," *ACM TOS*, vol. 11, no. 4, pp. 19:1–19:55, October 2015.

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

[7] 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.

[8] L. Ren, C. W. Fletcher, X. Yu, A. Kwon, M. van Dijk, and S. Devadas, "Unified oblivious-RAM: Improving recursive ORAM with locality and pseudorandomness," *IACR Cryptology ePrint Archive*, vol. 205, 2014.

[9] D. Sleator and R. Tarjan, "Self-adjusting binary search trees," *J. ACM*, vol. 32, no. 3, pp. 652–686, Jul. 1985.

[10] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. Weinberger, "Quickly generating billion-record synthetic databases," in *Proc. of SIGMOD*, Minneapolis, MN, 1994.

[11] 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.

[12] 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.

[13] 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.

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

[15] ——, "Three-server swapping for access confidentiality," *IEEE TCC*, 2016, pre-print.

[16] J. Dautrich and C. Ravishankar, "Tunably-oblivious memory: Generalizing ORAM to enable privacy-efficiency tradeoffs," in *Proc. of CODASPY*, San Antonio, TX, March 2015.

[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] 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.

[19] V. Bindschaedler, M. Naveed, X. Pan, X. Wang, and Y. Huang, "Practicing oblivious access on cloud storage: The gap, the fallacy, and the new way forward," in *Proc. of CCS*, Denver, CO, October 2015.

[20] S. Devadas, M. van Dijk, C. W. Fletcher, L. Ren, E. Shi, and D. Wichs, "Onion ORAM: A constant bandwidth blowup oblivious RAM," in *Proc. of TCC*, Tel Aviv, Israel, January 2016.

[21] S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, G. Pelosi, and P. Samarati, "Access control for the shuffle index," in *Proc. of DBSec*, Trento, Italy, July 2016.