Access Privacy in the Cloud

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

[1] Università degli Studi di Milano, 26013 Crema, Italy
firstname.lastname@unimi.it
[2] Università degli Studi di Bergamo, 24044 Dalmine, Italy
parabosc@unibg.it
[3] Politecnico di Milano, 20133 Milan, Italy
gerardo.pelosi@polimi.it

Abstract. Moving data to the cloud represents today a growing trend as it provides considerable advantages, both in terms of economy of scale and flexibility/elasticity for data owners. In such a scenario, there is however a clear need for solutions aimed at protecting the confidentiality of (sensitive) data and accesses. In this chapter, we illustrate some solutions proposed in the literature for protecting access confidentiality and classify them, depending on the underlying data structure used for data storage and support for access operations, in two classes: *i)* ORAM-based approaches, and *ii)* dynamically allocated data structures.

1 Introduction

The increasingly growing adoption of cloud technologies demands for solutions able to guarantee an efficient and secure use of outsourced storage services. The benefits brought by such services range from improved scalability and accessibility of data to decreased management costs, providing a flexible alternative to expensive, locally-implemented solutions. However, moving possibly sensitive data to the cloud exposes them to new privacy threats, arising specifically from the fact that they are kept out of the data owner's premises [5, 23]. Indeed, the cloud provider storing the data is trusted to properly provide its service (e.g., to store data and protect them against outside attacks). However, it is not fully trusted to access the plaintext content of the (possibly sensitive) data it stores.

Encryption techniques are a necessary component to ensure the confidentiality of data managed by a cloud provider. The adoption of encryption at the client side guarantees that only (authorized) users, who legitimately know (or can compute) the encryption keys used to protect confidential data, are able to access the plaintext data content. Although encryption provides protection guarantee of confidentiality of data at rest, it falls short in scenarios where data stored at an external cloud provider are accessed (read and/or written). Indeed, observing accesses to an outsourced data collection may reveal sensitive information about the user performing the search operation as well as about the data collection itself [15, 16, 18]. Consider, as an example, a publicly available medical database. Disclosing the fact that *Alice* is looking for the treatments for a

rare disease reveals to an observer the fact that she (or a person close to her) suffers from such a disease, with a clear privacy violation. Similarly, disclosing the fact that two accesses aim at the same target encrypted data item permits an observer to keep track of the frequency of accesses to data items and, exploiting external knowledge on the frequency of accesses to the corresponding plaintext data, reveals her the sensitive content of the outsourced dataset. Different techniques have then been proposed to protect both *access confidentiality* (i.e., confidentiality of the target of each access request) and *pattern confidentiality* (i.e., confidentiality of the fact that two accesses aim at the same target). The first line of works that addressed this problem is based on Private Information Retrieval (PIR, e.g., [20]). However, PIR-based approaches implicitly assume that the accessed data collection is not sensitive, and that only access operations need to be protected. Also, these solutions suffer from high computational costs that limit their applicability in real world scenarios. In this chapter, we will specifically focus on two recent classes of approaches aimed at protecting data, access, and pattern confidentiality while reducing computational cost with respect to PIR-based solutions. The first class is based on the adoption of ORAM (Oblivious Random Access Memory) data structure, which is a layered structure that supports equality search operations while hiding the target of the access to the eyes of the storage server. ORAM-based solutions are based on the idea that data are re-allocated to the top level of the layered structure after each access. These solutions, although effective, suffer from the fact that ORAM data structure does not preserve the natural ordering among data items. Hence, as an example, it does not support range searches. The second class of solutions overcomes this drawback by adopting dynamically allocated data structures for protecting access confidentiality. These solutions organize data in well known data structures traditionally used to support efficient access to the data (e.g., $B+$-trees) and change the allocation of accessed data to memory slots at each access, to prevent an observer from identifying repeated accesses by observing read and write operations at the memory level.

In the remainder of this chapter, we first illustrate some approaches based on the adoption of Oblivious RAM structure (Section 2), and then describe two dynamically allocated data structures (Section 3). Finally, we present our conclusions (Section 4).

2 Oblivious RAM Data Structures

One of the most widely known class of approaches adopted to protect access and pattern confidentiality is based on the adoption of ORAM (*Oblivious RAM*) data structures.

ORAM has first been proposed by Goldreich and Ostrovsky in [13, 14, 19] to the aim of concealing the memory access patterns of a software program running on a microprocessor, to safeguard the software from illegitimate duplication and consequent redistribution. To this purpose, ORAM acts as an interface between the microprocessor and the memory subsystem, in such a way to make mem-

ory access patterns indistinguishable. During a program execution, the ORAM interface makes the probability distribution of a sequence of memory addresses independent from the input values of the program and dependent only from the length of the program. Any ORAM requires $\Omega(\log N)$ bandwidth overhead to conceal an access pattern from a storage space including N items [13, 14, 19]. Also, the best ORAM implementation [14] requires $O(N \log N)$ server storage and implies an amortized communication overhead of $O(\log^3 N)$ ($O(N \log^2 N)$, resp.) in the average case (worst case, resp.).

ORAM structure has recently been adopted for the definition of approaches aimed at protecting the confidentiality of accesses to data stored at a remote server. In fact, the problem of protecting memory access pattern generated by a software program is very similar to the problem of privately retrieving data from a remote storage server. Indeed, even if from a practical perspective the two problems present some differences (e.g., different costs of read and write operations, storage capacity on the client side, latency of network communications compared with the one between a microprocessor and its memory subsystem), from a theoretical point of view the two problems can be modeled in the same way as both aim at protecting the confidentiality of access operations to the eyes of the party in charge of its execution (i.e., the processor and the storage server, respectively). Considering a simplified scenario characterized by one client and one storage server, client's data are individually encrypted using an encryption key known only to the client, and the resulting *blocks* are stored in a ORAM-based structure at the server side. Intuitively, ORAM-based structures conceal from the storage server the exact memory location where the block containing the target data item is stored by retrieving more than one block at a time (i.e., the target block and some additional blocks). The client then changes the allocation of data items to memory locations and writes re-encrypted blocks back at the server, according to the new allocation strategy. The strategy used for the traversal of the data structure makes the accesses to different data items indistinguishable. In particular, repeated accesses become indistinguishable from accesses to different target data items.

In the remainder of this section, we will first describe the original hierarchical ORAM structure [14], and then illustrate more recent variations over the original architecture, Path ORAM [22] and Ring ORAM [21], aimed at reducing its computational overhead.

2.1 Hierarchical ORAM

Consider a set of N data items, uniquely identified through an identifier $id \in ID$, that should be stored in a hierarchical ORAM [13, 14, 19] structure. Each data item is individually encrypted, using a semantically-secure cipher and a key known only to the client, before being stored in the ORAM structure. This guarantees that no information about the plaintext content of the data item can be leaked from its encrypted representation. In the following, we illustrate the structure of hierarchical ORAM, and the working of access operations.

Structure. Hierarchical ORAM is a pyramid-shaped data structure composed of $\lceil \log N \rceil$ *levels*, which can be used to store client's data items. Each level l in the ORAM structure includes 2^l *buckets*, with a storage capacity of $k\lceil \log N \rceil$ *slots* each, $k \geq 1$. Each slot in a bucket can store either an encrypted data item (*real* block) or an encrypted dummy/empty item (*dummy* block). Thanks to the adoption of a semantically secure cipher, real blocks and dummy blocks are indistinguishable to the eyes of the storing server.

Each level l except the first one ($2 \leq l \leq \lceil \log N \rceil$) in the ORAM structure has a *hash function* $h_l : ID \rightarrow \{1, \ldots, 2^l\}$ that associates the identifier of a data item, $id \in ID$, with the unique position of the bucket on level l where the data item might be stored. Figure 1(a) illustrates a 3-level hierarchical ORAM structure, where each bucket stores up to 4 blocks. In the figure, we report on the top of each bucket its position in the level; real blocks are gray and dummy blocks are white.

At initialization time, the N real blocks obtained encrypting client's data items are stored in the last and largest level (i.e., $l = \lceil \log N \rceil$) of the ORAM structure. Hence, each real block is stored in the slot identified by the hash function associated with the last level in the structure. All the other blocks in the last level, as well as any block in all the other levels of the ORAM structure, are filled with dummy blocks.

Read Access. Access operations to data stored in a hierarchical ORAM structure require to maintain two invariants to protect access and pattern confidentiality: *i)* access operations do not reveal to the server the level where the target block is stored (guaranteed by always accessing one bucket at each level of the ORAM structure); and *ii)* access operations never retrieve a block in the same bucket more than once (even when repeating access to the same data item).

Let us consider an access request for the data item identified by *id*. The client starts visiting the ORAM structure from its top level and retrieves, for each level l, the bucket where the target data item could be stored at level l. To this purpose, the client computes $h_l(id)$, $l = 2, \ldots, \lceil \log N \rceil$. Note that the client always retrieves both the buckets on the top level (i.e, $l = 1$) of the ORAM structure, which does not have any hash function. To prevent leaking to the storage server the level where the target data item is stored, the client always ends her visit of the ORAM structure at the bottom level $l = \lceil \log N \rceil$ of the structure. In fact, stopping the access process at a different level would inevitably reveal to the storage server that the target data item was stored at the last accessed level. Consider, as an example, the search for value C over the hierarchical ORAM in Figure 1(a). The clients iteratively downloads the buckets denoted with a bold blue fence in the figure. In fact, the client first downloads the two buckets at level $l = 1$. Then, it computes $h_2(C) = 4$ and downloads the 4th bucket at level 2. Even if C belongs to the downloaded bucket at level 2, the client computes $h_3(C) = 7$ and downloads the 7th bucket at level 3.

The client decrypts each bucket downloaded from the server and locally stores its plaintext representation. Once the client has completed her visit of the ORAM structure (i.e., she has downloaded the bucket at level $l = \lceil \log N \rceil$), she moves

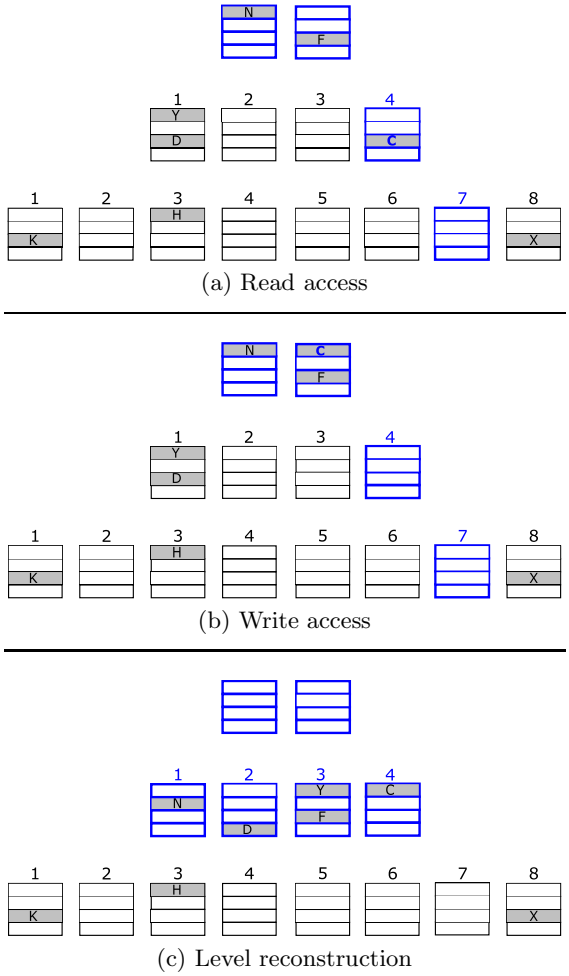


Fig. 1. An example of an access searching for C in a hierarchical ORAM structure (a,b) and of reconstruction of the first level of the ORAM structure (c)

the target data item in one of the two buckets at level $l = 1$. The client then removes the target block from the bucket where it was stored before the access, substitutes it with a fresh dummy block, re-encrypts the target data item, and inserts the resulting encrypted block in one of the two buckets at level $l = 1$. Since the top level is not associated with any hash function, the choice of the bucket where to insert the target block depends on the sequence number of the current access request (odd or even). The client then re-encrypts all the accessed blocks and writes the downloaded buckets back at the server, following the same order as read accesses (i.e., starting from the top of the structure). Considering the search for C in the ORAM structure in Figure 1(a), the client moves the block storing C to one of the two buckets at level 1 (the second one in the example)

and re-encrypts all the accessed blocks. The client then rewrites, in the order, the accessed buckets at the server starting from the top of the structure. Figure 1(b) illustrates the status of the ORAM structure after the access searching for C.

Even if each bucket in the ORAM structure stores up to $k\lceil \log N \rceil$, after $2k\lceil \log N \rceil$ access operations the two buckets at level 1 will be full. Hence, to guarantee that the second invariant is satisfied (i.e., no block is retrieved more than once in the same bucket), it is necessary to reconstruct the first level of the ORAM structure. To this aim, the blocks in the first level are obliviously transferred to the second level. To obliviously transfer blocks, the client changes the hash function h_2 of the second level of the ORAM structure and reorganizes all the (real) blocks that were stored in the buckets on level 1 and on level 2, accordingly. Clearly, this implies downloading, decrypting, re-encrypting, and rewriting back at the server all the buckets at level 1 and at level 2. In general, after 2^l access operations, some buckets at level l ($1 \leq l \leq \lceil \log N \rceil$) will be full and it will be necessary to obliviously transfer all the data blocks at level l to level $l+1$, changing the hash function at level $l+1$ and applying a $O(N \log N)$ *oblivious sorting algorithm*. Note that after $2^{\lceil \log N \rceil} k \lceil \log N \rceil$ accesses it is necessary to change the hash function of the bottom level of the ORAM structure, which implies downloading, decrypting, re-encrypting, and rewriting back at the server the whole data collection. For instance, assuming that the first level in the ORAM structure in Figure 1(b) needs to be reconstructed, the client moves data items N, C, and F to the second level and re-defines h_2. As visible in Figure 1(c), this implies re-writing both the buckets at level 1 and the buckets at level 2, since all the data items in these two levels can be allocated at any of the buckets in level 2. Indeed, in the considered example, Y moves from the 1st to the 3rd bucket.

Write Access. Since every read access operation by the client implies re-writing all the accessed buckets, client operations consisting of access requests to read, write, insert, or delete a block are indistinguishable from the point of view of the storing server. Indeed, they all present the same access pattern, thanks to the adoption of an encryption function that obfuscates whether the item inserted in the top level before rewriting buckets back at the server contains an actual data item already stored in the ORAM structure, a new data item, or a dummy item.

2.2 Path ORAM

Building on the original hierarchical ORAM structure, a considerable research effort has been spent to make ORAM schemes more practical and efficient. Path ORAM [22] is a recent ORAM-based approach that does not require expensive periodic level reconstruction.

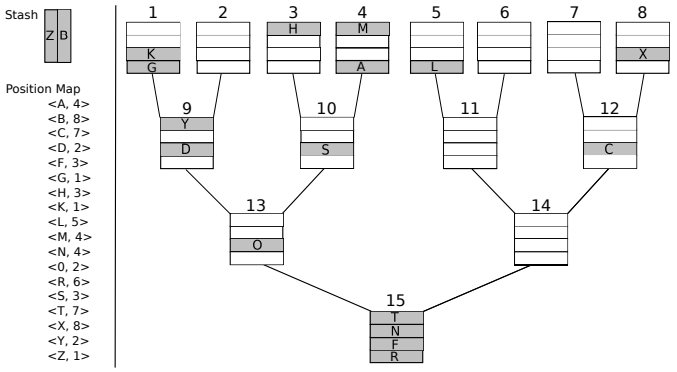
Structure. Path ORAM is a binary tree with height $h = \lceil logN \rceil$ and N leaves, where N is the number of data items in the data collection. Each node in the Path ORAM structure is a *bucket* that can store up to $Z \geq 1$ (real or dummy) blocks each. Each data item is associated with a leaf in the Path ORAM structure, uniquely identifying a set of buckets (those along the path to the leaf node)

where the data item can be stored. The client keeps track of data-leaf association by locally storing a *position map*, which is a set of pairs of the form $\langle id, pos \rangle$, where *id* is the identifier of a data item and *pos* is the position identifying the corresponding leaf in the tree. The size of the position map is $O(N \log N/B)$, where B is the node size.

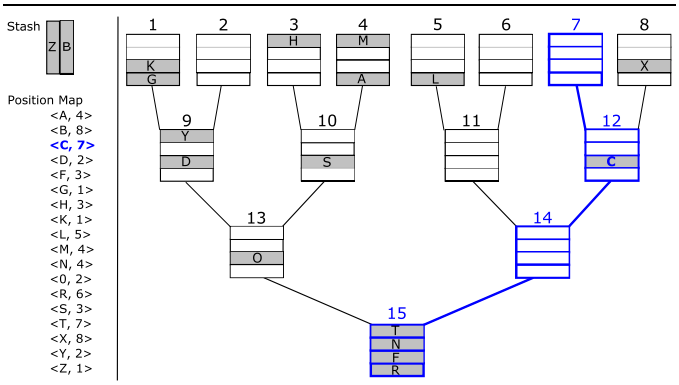
Besides the position map, the client also locally stores a portion of the data collection in a local *stash* having size $O(\log N)$. The local stash is necessary to properly manage access operations, as illustrated in the following, by guaranteeing that each data item is always stored either in a bucket along the path as per the position map or in the local stash. Figure 2(a) illustrates, on the right an example of a Path ORAM structure with 8 leaves and height equal to 3, where each bucket stores up to $Z = 4$ blocks. In the figure, node identifiers are reported on top of nodes, real blocks are gray, while dummy blocks are white. The figure also illustrates, on the left, the local stash and the position map stored at the client.

Read Access. To retrieve the data item with identifier equal to *id*, the client first retrieves from the local map the position *pos* of the corresponding leaf node. The client then sends a request to the storing server, and downloads the $h + 1$ buckets along the path from the root of the tree to the leaf node in position *pos*. Indeed, if not in the local stash, the data item of interest is stored in one of these buckets. The client decrypts the downloaded $Z(h+1)$ blocks and inserts the corresponding data items in the local stash. To guarantee that future searches for the same target data item do not visit the same path, the client assigns a new randomly chosen position (i.e., a new leaf) to the target data item and updates the local position map accordingly. Consider, as an example, a search for value C in the Path ORAM structure in Figure 2(a). The client first downloads the buckets along the path to node 7, that is, 15, 14, 12, and 7 (see Figure 2(b), where accessed nodes are denoted with a bold blue fence). It decrypts the five downloaded real blocks and inserts them into the local stash, which included values Z and B before the access. It then randomly assigns a new position to C, 6 in the example.

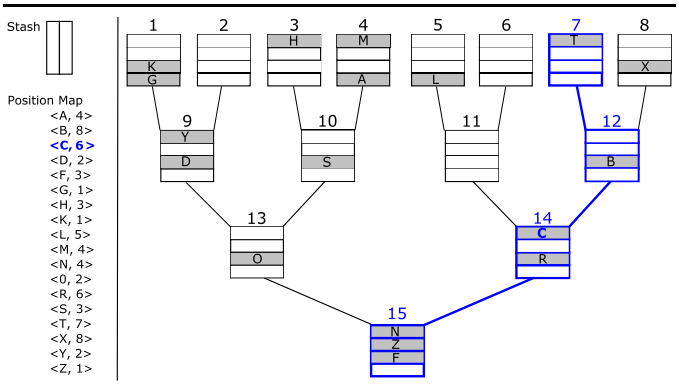
The client then rewrites the downloaded blocks back at the server, after having possibly changed their content. In particular, the client inserts into the buckets to be rewritten back all the data items in the local stash that are associated with a leaf whose path intersects the visited/downloaded path. In such a bucket reorganization, the client moves data items as close as possible to leaf nodes. To prevent the server from tracking eviction operations, all the accessed data items are re-encrypted and, once the buckets along the visited path have been updated, written back at the storing server. Considering the search for value C illustrated in Figure 2, the client inserts C into node 14, which is the deepest node along the common sub-paths to 7 and 6. Also, the client can evict Z and B from the stash, inserting them into buckets 15 and 12, respectively. The client will also push T to node 7 and R to node 14, while N and F remain in the root node. The client then re-encrypts real and dummy blocks and rewrites the updated content of buckets 15, 14, 12, and 7 at the server (see Figure 2(c),



(a) Path ORAM structure



(b) Read access



(c) Write access

Fig. 2. An example of Path ORAM structure (a) and of the path read (a) and written (b) by an access operation searching fro C

where written nodes are denoted with a bold blue fence). Clearly, even if any data item in the stash could be inserted into the root node, due to capacity constraints, the remaining data items are stored in the local stash. On the contrary, if after the eviction from the stash a bucket along the visited path is not full, it is completed with dummy blocks.

The size of the local stash as well as the size of buckets need to be carefully chosen to avoid overflows. Indeed, as demonstrated in [22], if the size of buckets is lower than 4 (i.e., $Z < 4$), buckets close to the root tend to become congested and cause the stash to grow indefinitely, with the non-negligible probability of having a number of data items associated with a leaf node greater than the capacity of the corresponding path. On the contrary, if the number of blocks per bucket is greater than or equal to 4 (i.e., $Z \geq 4$), a stash with size $\mathrm{O}(Z(h+1))$ guarantees a negligible probability of stash overflow.

Path ORAM causes $2Z\lceil \log N \rceil$ access overhead, $\mathrm{O}(N)$ server storage overhead, and $\mathrm{O}(\log N)\omega(1) + \mathrm{O}(N \log N/B)$ client storage overhead. The storage overhead at the client side is due to the need of locally accommodating the stash, $\mathrm{O}(\log N)\omega(1)$, and the position map, $O(N \log N/B)$. To reduce the client storage overhead, an alternate version of the Path ORAM design proposes to recursively outsource the position map in a sequence of smaller Path ORAM structures [22]. This permits to reduce the client storage overhead to $\mathrm{O}(\log N)\omega(1)$, at the cost of increasing the access overhead to $\mathrm{O}(\log^2 N/\log B)$ and the number of communication rounds per operation between the client and the server to $\mathrm{O}(\log N/\log B)$.

2.3 Ring ORAM

A further improvement of the hierarchical ORAM structure is represented by Ring ORAM [21], which is a recent ORAM-based approach aimed at reducing the bandwidth overhead of Path ORAM. Indeed, Ring ORAM reduces access overhead to $\mathrm{O}(1)$ and the overall bandwidth to $\sim 2.5 \log(N)$, assuming that the storage server can perform computations. We note, however, that ORAM schemes requiring server-side computations are not compatible with basic cloud-storage services (e.g., Amazon S3) [2].

Structure. Ring ORAM adopts the same server-side structure as Path ORAM, with the only difference that each node in the tree is complemented with additional *metadata*. The metadata associated with a node include a set of S additional dummy blocks, a randomly chosen permutation map that associates the positions of blocks in a bucket with their identifiers, and a counter of accesses to the bucket. Figure 3 represents an example of a Ring ORAM structure, together with the local stash and position map stored at the client.

Read Access. Ring ORAM adopts an approach similar to Path ORAM to retrieve the data item with identifier equal to *id*. The client first retrieves from the local map the position *pos* of the corresponding leaf node and downloads from the server the metadata of the nodes along the path to *pos*. Note that the metadata size is much less than the node size. Based on the information in the downloaded metadata, the client selects one block for each node along the

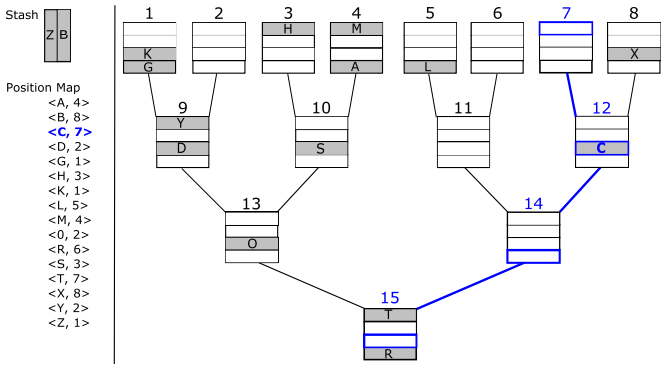


Fig. 3. An example of Ring ORAM structure and the blocks downloaded by an access operation searching for C

path to the target leaf. In particular, for each node along the path, the client selects: the target block, if it is stored in the node; an unread dummy block, otherwise. Indeed, by reading only metadata, the client can determine whether the requested block is present in the bucket, identify its position using the offsets map, or choose an unread dummy block using the counter of accesses.

Since only one of the $O(\log N)$ blocks downloaded from the server is a real block (i.e., the block of interest), Ring ORAM can guarantee O(1) online bandwidth in access execution, by requiring some server-side computation. Indeed, if dummy blocks have a fixed content (e.g., $d_i = 0$), and the server computes the *xor* of all encrypted blocks selected along the target path, the client can easily retrieve the content of the only real block downloaded (i.e., the target block). The server then computes $E(x, r) \oplus E(d_1, r_1) \oplus \ldots \oplus E(d_n, r_n)$ where x is the target block, d_i is a dummy block, and r_i is a random nonce employed by the client when encrypting the block and picked from a pseudo-random number generator seeded with a value obtained from the position of the block in the node and the level in the tree of the considered node. By computing $E(d_1, r_1) \oplus \ldots \oplus E(d_n, r_n)$ the client can then retrieve the target block and, by decrypting it, the target data item. Consider, as an example, the structure in Figure 3 and a search for value C, which is associated with leaf 7 in the position map. The client will download from the server the metadata along the path $15 \rightarrow 14 \rightarrow 12 \rightarrow 7$ (denoted with a bold blue line in the figure) from the server. Assuming that, based on the metadata, the client discovers that C is stored in bucket 12, she identifies an unread dummy block in buckets 7 (d_7), 14 (d_{14}), and 15 (d_{15}) and asks the server to compute $E(C, r_{12}) \oplus E(d_7, r_7) \oplus E(d_{14}, r_{14}) \oplus E(d_{15}, r_{15})$. The client will then compute $E(d_7, r_7) \oplus E(d_{14}, r_{14}) \oplus E(d_{15}, r_{15})$ to retrieve the encrypted block $E(C, r_{12})$, and then extract the plaintext target data item.

To guarantee access and pattern confidentiality, Path ORAM requires that each block in a bucket, be it dummy or real, is read at most once. If a bucket is accessed many times, there is the possibility for dummy blocks to be exhausted.

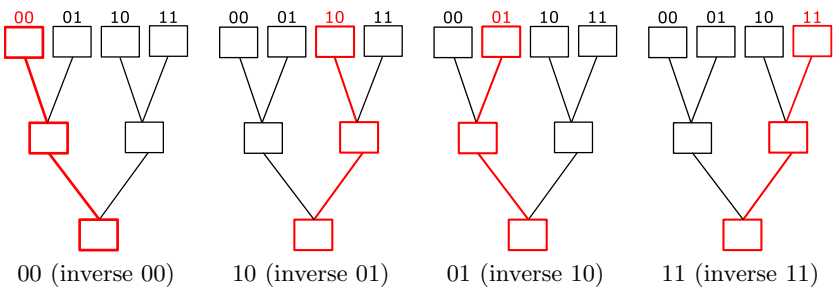


Fig. 4. Order in which paths are written in a Ring ORAM structure

To overcome this problem, Ring ORAM adopts an *early reshuffle* approach to reshuffle a bucket after it has been accessed by S read operations.

To optimize the cost of access operations, differently from Path ORAM, Ring ORAM does not rewrite back accessed buckets at each read operation. On the contrary, it performs write operations periodically (once every A read accesses), evicting as many data items from the stash as possible. Write operations are performed in a specific (inverse lexicographic) order to minimize overlap between consecutive write paths and hence maximize the effectiveness of the eviction strategy. Consider, as an example, a Ring ORAM structure with four leaves. As visible from Figure 4, writing the paths to the leaves in inverse lexicographic order minimizes intersection between subsequent accesses. Note that in the figure, for simplicity, we report only the identifier of leaf nodes and do not represent buckets content.

3 Dynamically Allocated Data Structures

An alternative class of approaches aimed at protecting and access and pattern confidentiality is represented by *dynamically allocated data structures* (e.g., [1, 3, 4, 6–12, 17]). Intuitively, these techniques are based on the idea that traditional data structures used to efficiently store and retrieve data (e.g., binary search trees, B+-trees, hash tables) can be profitably used to enforce access and pattern confidentiality, by dynamically reallocating accessed data at each read operation. This class of solutions has the advantage over ORAM-based solutions that data are organized in the data structure according to the value of an index attribute (or identifier). Hence, they naturally offer support for range queries and easily accommodate changes in the number of data items stored in the structure. On the contrary, the structures illustrated in Section 2 do not reflect, in their organization, the logical order among identifiers. Indeed, the parent-child relationships among nodes does not depend on the value of the identifiers of the data items they store. Hence, they do not offer support for range queries.

In the remainder of this section, we first describe the shuffle index [11], which is a dynamically allocated data structure based on the organization of data in a B+-tree, and a self-balancing binary tree data structure [7].

3.1 Shuffle Index

The shuffle index [8] is a dynamically allocated data structure that logically organizes data in a $B+$-tree, to enable efficient data retrieval while protecting access and pattern confidentiality. In the following, we illustrate the structure of the shuffle index, and the working of access operations.

Structure. The shuffle index, at the abstract level, is an unchained $B+$-tree (i.e., a $B+$-tree with no connection between contiguous leaves, not to reveal to the storing server their relative order) defined over a candidate key for the set of outsourced data items. Given the fan-out F of the index structure, each internal node of the shuffle index stores an ordered sequence of $q-1$ values $v_1 \leq \ldots \leq v_{q-1}$, with $q \geq \lceil F/2 \rceil$ (but for the root, for which $1 \leq q \leq F$). Each of the q children of the node is the root of a subtree storing all the values in the range $[v_i,v_{i+1}]$, $i = 1,\ldots,q-2$. The first child of the node stores all the values lower than v_1, while the last child of the node stores all the values greater than v_{q-1}. Leaf nodes store, together with key values, the corresponding data items. Figure 5(a) illustrates an example of an abstract shuffle index with fan-out $F = 3$.

At the logical level, the shuffle index is a collection of nodes, each associated with a unique randomly assigned logical identifier. Hence, logical identifiers do not reflect the natural order relationship among the values in nodes content. Logical identifiers are used to represent pointers to children in the internal nodes of the $B+$-tree structure. Consider the abstract structure in Figure 5(a). Figure 5(b) illustrates an example of its logical representation where, for the sake of readability, logical node identifiers are reported on top of each node. The first digit of logical identifiers correspond to the level of the node in the tree.

At the physical level, the logical identifier of each node translates into the physical address where the corresponding block is stored. The block representing a logical node is obtained by encrypting the logical node content, concatenated with a random nonce, to destroy plaintext distinguishability. Consider the logical shuffle index in Figure 5(b). Figure 5(c) illustrates an example of its physical representation, which corresponds to the view of the provider over the shuffle index.

Read Access. For each access operation aimed at searching a value v of the candidate key over which the index has been defined, the shuffle index combines the following three protection techniques for providing access and pattern confidentiality.

- *Cover searches.* The search for the target value is complemented with *num_cover* additional *fake* searches, not recognizable as such by the storage server. Cover searches are chosen in such a way to visit *num_cover* disjoint paths, that is, paths including a disjoint set of nodes, apart from the root. Intuitively, for each level of the shuffle index, the client downloads the node along the path to the target, and *num_cover* additional nodes along the paths to the covers. Therefore, from the point of view of the storing server, any of

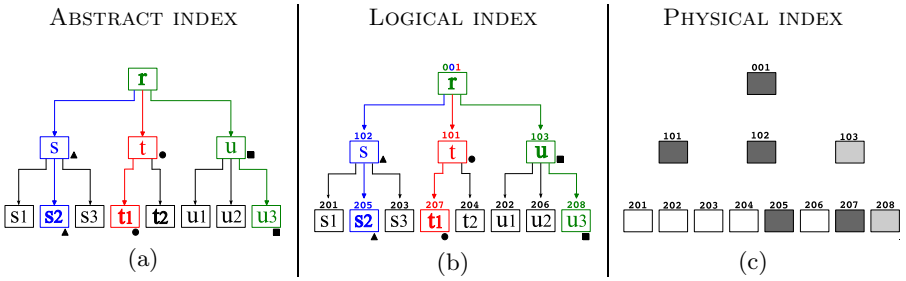


Fig. 5. An example of abstract (a), logical (b), and physical (c) shuffle index
Legend: ■ target, ● node in cache, ▲ cover; blocks read and written: dark gray filling, blocks written: light gray filling

the *num_cover* + 1 downloaded nodes at each level could be the one along the path to the target.

- *Cached searches.* To prevent the storing server from identifying repeated accesses by observing that subsequent searches download the same (or a common subset of) physical blocks, the shuffle index uses a client-side cache structure. The cache is a layered structure, with a layer for each level in the shuffle index, storing the nodes along the (target) paths to the *num_cache* most recent accesses to the shuffle index. If the target of an access is in cache, an additional cover is used to guarantee that each access operation downloads exactly the same number of nodes (i.e., *num_cover* + 1) at each level of the shuffle index, apart from the root.
- *Shuffling.* Shuffling consists in changing the allocation of nodes to blocks at each access. Every block downloaded from the storage server is then decrypted, associated with a different physical address among the accessed ones, re-encrypted using a different random nonce, and written back at the server. Clearly, the parents of shuffled nodes are updated accordingly, to maintain the correctness of the underlying abstract B+-tree structure. Shuffling breaks the (otherwise static) node-block association. Hence, different searches for the same key value will imply accesses to different blocks and, vice versa, accesses reading/writing for a same physical block are not necessarily due to searches for the same key value (i.e., repeated searches).

To retrieve the data item with candidate key equal to v, the client interacts with the server to visit the shuffle index. Starting from the root level, for each level in the shuffle index, the client: downloads the nodes along the paths to the target and cover searches; decrypts their content; updates the cache structure for the visited level; shuffles accessed nodes; updates the parents of shuffled nodes; re-encrypts and re-writes back at the server the nodes read during the previous iteration. Consider a search for value $u3$ in the shuffle index in Figure 5, and assume that the cache stores the path to $t1$ and that value $s2$ is chosen as cover. The client first accesses the root node, which is stored in the first level of the local cache, and identifies the blocks at level 1 along the path to the target (block 103),

to the cover (block 102), and in cache (block 101). It then downloads blocks 102 and 103, decrypts them, and inserts node s in the second level of the cache. The client then shuffles nodes 101, 102, and 103 (e.g., it assigns s to 101, t to 103, and u to 102), updates the root node accordingly, re-encrypts its content and stores it at the server side. The client operates in a similar manner at the second level of the tree: it downloads the blocks along the path to the target (208) and to the cover (205), decrypts their content and updates the cache inserting node $u3$. The client then shuffles blocks 205, 207, and 208 (e.g., it assigns $s2$ to 208, $t1$ to 205, and $u3$ to 207), updates and re-encrypts nodes s, t, and u accordingly, and re-writes them back at the server. Finally, the client re-encrypts the accessed leaf nodes and sends the corresponding blocks to the server for storage. Figure 5(c) illustrates the cloud provider's view over the access in terms of blocks read and written (dark gray) and only written (light gray). Note that the server cannot determine which, among the accessed leaves, is the target of the search operation, nor reconstruct shuffling operations.

The shuffle index exhibits an $O(\lceil \log N \rceil)$ non-amortized access overhead and a number of communications rounds equal to the height of the $B+$-tree, with $O(1)$ and $O(N)$ storage overhead at the client and at the server, respectively.

Write Access. Similarly to ORAM-based structures, also the shuffle index implies a re-write, for each read access, of any accessed blocks. Hence, an update to the data content that does not modify the value of the key attribute can be easily accommodated during any read access operation. On the contrary, an update of the key value (as well as the insertion or removal of data items) deserve a special treatment if the client wants to keep the nature of the access confidential. While the deletion of a data item can be easily managed by marking it as invalid, the insertion of a new data item and of the corresponding key value, or its update may imply a change in the underlying data structure. Indeed, if the leaf node where the data item should be inserted is full, the accommodation of the insert operation requires a split of the node itself. To prevent the storing server from distinguishing read from write accesses, the solution in [11] proposes to probabilistically split nodes at every access, be it associated with a read or a write operation. Hence, during (read and write) access operations the client chooses whether to split each visited node, with a probability that grows with the number of key values in the node. This approach guarantees that split operations can happen during both read and write accesses, thus limiting the ability of the storing server to distinguish between read and write accesses.

3.2 A Dynamic Tree-Based Data Structure

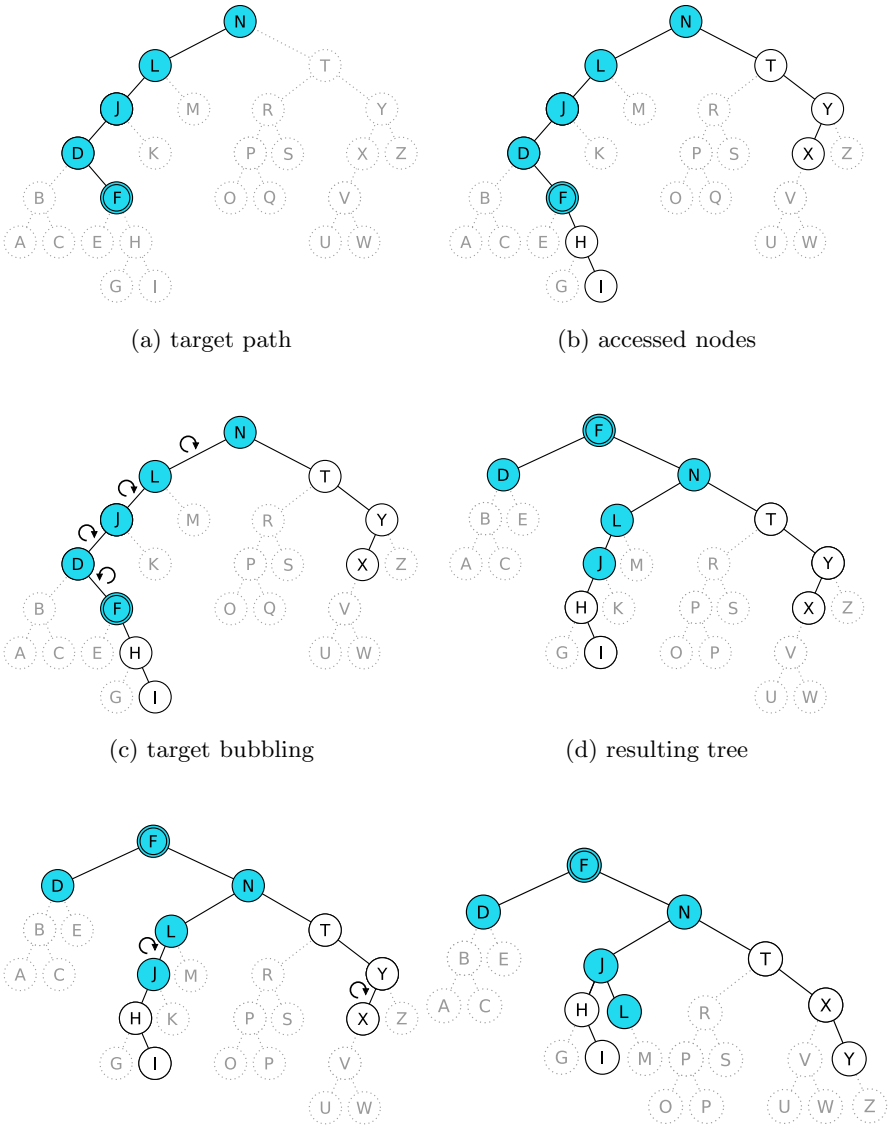
The technique for protecting access and pattern confidentiality presented in [7] aims at enhancing the shuffle index approach along two directions: *i)* it does not require the client to commit storage resources for accessing data; and *ii)* it supports accesses by multiple clients.

Structure. To the aim of supporting efficient accesses, outsourced data are organized in a binary search tree with maximum height $h = \lfloor 2 \log(N) \rfloor$, with

N the number of nodes in the tree. The nodes in the tree are buckets, each storing a set of Z data items. The mapping function, associating each data item with the bucket storing it, is a non-invertible and non-order preserving function, defined in such a way to guarantee a balanced distribution of the data items among the buckets. Since the mapping function is not invertible, exposure of the bucket index does not expose sensitive values. Also, since the mapping function is not order preserving, the binary search tree efficiently supports searches over the outsourced data collection without revealing the relative order among data. The buckets composing the binary search tree are encrypted at the client side before storage at the server.

Read Access. Access operations combine a traditional visit of the BST with the following four protection techniques aimed to protect access confidentiality.

- *Uniform accesses.* All the accesses download from the provider the same number of blocks, independently from the level where the target of the search operation is located. The number of accessed blocks is fixed to $h + 2$. If the path to the target node is shorter than $h + 2$, the client downloads a set of *filler nodes*, that is, of nodes that are not along the path to the target. To guarantee that filler nodes are not recognizable as such, they are randomly chosen among the children of already accessed nodes, and nodes (be them along the path to the target or fillers) are dowloaded level by level. This guarantees that any of the $h + 2$ accessed nodes could be the target of the access. Consider, as an example, a search for value F in the binary search tree in Figure 6 with $N = 26$ and $h + 2 = 10$. Since the path to the target node includes only 5 nodes (light blue background in Figure 6(a), light gray in b/w printout), the search is complemented with 5 filler nodes (white with solid fence in Figure 6(b)). Note that any of the 10 downloaded nodes could be the target of the access since nodes along the path to the target are indistinguishable from filler nodes.
- *Target bubbling.* After each access, the target node is moved up (close to the root) in the tree by properly rotating the nodes along its path. This technique protects against repeated accesses. Indeed, if two subsequent searches look for the same target, the second access will find the target high in the tree and will therefore choose a high number of filler nodes. Hence, the two searches will visit two different sets of nodes, reducing the effectiveness of intersection attacks (i.e., of attacks that exploit the common downloaded blocks in subsequent accesses to infer the target of the searches). Target bubbling has also the advantage of changing the topology of the binary tree structure, further enhancing protection. With reference to the example in Figure 6, the nodes along the path to F are rotated as illustrated in Figure 6(c), obtaining the binary tree in Figure 6(d), where F is the root. A search for F over this tree could visit any subtree including 10 nodes rooted at F, thus considerably enhancing protection guarantees.
- *Speculative rotations.* Each access operation, because of target bubbling, can increase or decrease the heigh of the tree by one. To guarantee that the height of the tree remains within the limit of $h = \lfloor 2\log(N) \rfloor$, speculative



(a) target path (b) accessed nodes

(c) target bubbling (d) resulting tree

(e) speculative rotations (f) resulting tree

Fig. 6. An example of search for value F in a dynamic tree-based data structure
The visit of the path to F (a) is complemented with five filler nodes (b)
The nodes along the path to F are rotated (c), moving the target to the root (d)
Two additional speculative rotations (e) are performed to reduce the height of the tree (f)

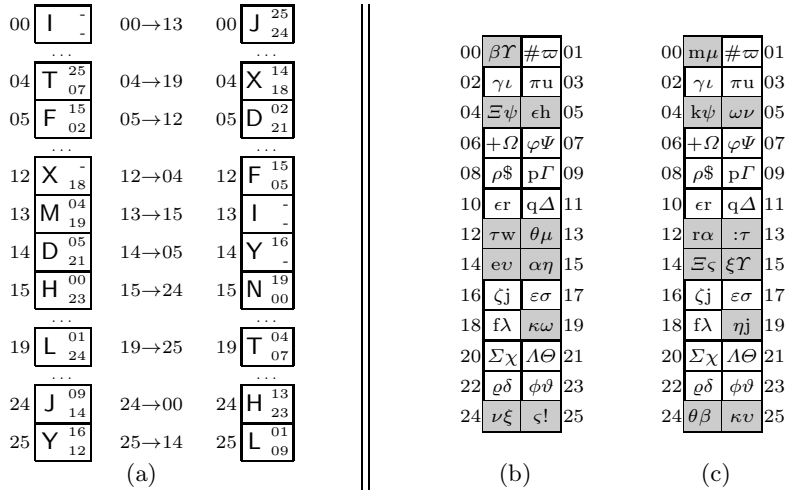


Fig. 7. An example of physical re-allocation (a) and of view of the server before (b) and after (c) the access in Figure 6

rotations possibly rotate accessed nodes, when it could be useful for reducing the height of the tree. Clearly, speculative rotations do not operate on the target node (or its ancestors) because this would possibly nullify (or mitigate the advantages of) target bubbling. Even if speculative rotations do not represent a protection technique per se, they provide benefits as they change the tree topology (and hence paths reaching nodes). With reference to the example in Figure 6, the rotations in Figure 6(e) could reduce the height of the tree. The tree resulting after the application of speculative rotations is illustrated in Figure 6(f) and has a completely different topology than the tree in Figure 6(a) on which the access operated.

– *Physical re-allocation.* At each access, the allocation of all the accessed nodes to physical blocks is changed. Re-allocation implies the need to decrypt and re-encrypt all the accessed nodes, concatenated with a different random salt to make the re-allocation untraceable by a possible observer. Also, it requires to update the pointers to children in the parents of re-allocated nodes. Note that, since all the accessed nodes are in a parent-child relationship, this does not require to download additional nodes. By changing the node-block correspondence at every access, physical re-allocation prevents the provider from determining whether two accesses visited the same node (sub-path) by observing accesses to physical blocks, and hence it prevents accumulating information on the topology of the tree. Indeed, accesses aimed at the same node will visit different blocks (and vice versa). Figure 7(a) illustrates an example of physical re-allocation of the nodes/blocks accessed by the search Figure 6, illustrating the nodes content before and after re-allocation. Fig-

ure 7(b) illustrates the view of the provider over the blocks composing the binary search tree, and its observations of accessed blocks (in gray).

The combined adoption of the protection techniques illustrated above, which imply both physical re-allocation and logical restructuring of the binary search tree, guarantees access confidentiality. Indeed, it makes skewed profiles of access to the plaintext data statistically indistinguishable from uniform access profiles [7]. The approach illustrated in this section provides access and pattern confidentiality at the cost of retrieving $\lceil \log N \rceil$ blocks, and has limited client side storage overhead, O(1), due to the storage of the address of root node only.

4 Conclusion

In this chapter, we have illustrated different solutions for protecting access and pattern confidentiality. The approaches illustrated have been classified in two main classes: ORAM-based techniques, and dynamically allocated data structures. For each of these classes, we have described some representative approaches, discussing the structure for data storage and the working of access operations.

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

References

1. Bacis, E., De Capitani di Vimercati, S., Foresti, S., Paraboschi, S., Rosa, M., Samarati, P.: Distributed shuffle index in the cloud: Implementation and evaluation. In: Proc. of the 4th IEEE International Conference on Cyber Security and Cloud Computing (IEEE CSCloud 2017). New York, USA (June 2017)
2. Bindschaedler, V., Naveed, M., Pan, X., Wang, X., Huang, Y.: Practicing oblivious access on cloud storage: the gap, the fallacy, and the new way forward. In: Proc. of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS 2015). Denver, CO, USA (Oct 2015)
3. Chen, C., Cichocki, A., McIntosh, A., Panagos, E.: Privacy-protecting index for outsourced databases. In: Proc. of the Workshops of the 29th IEEE International Conference on Data Engineering (ICDE 2013). Brisbane, Australia (Apr 2013)
4. De Capitani di Vimercati, S., Foresti, S., Paraboschi, S., Pelosi, G., Samarati, P.: Supporting concurrency and multiple indexes in private access to outsourced data. Journal of Computer Security 21(3), 425–461 (2013)
5. De Capitani di Vimercati, S., Foresti, S., Samarati, P.: Managing and accessing data in the cloud: Privacy risks and approaches. In: Proc. of the 7th International Conference on Risks and Security of Internet and Systems (CRiSIS 2012). Cork, Ireland (Oct 2012)
6. De Capitani di Vimercati, S., S.Foresti, Paraboschi, S., Pelosi, G., Samarati, P.: Enforcing authorizations while protecting access confidentiality. Journal of Computer Security 26(2), 143–175 (Jan 2018)

7. De Capitani di Vimercati, S., Foresti, S., Moretti, R., Paraboschi, S., Pelosi, G., Samarati, P.: A dynamic tree-based data structure for access privacy in the cloud. In: Proc. of the 8th IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2016). Luxembourg (Dec 2016)
8. De Capitani di Vimercati, S., Foresti, S., Paraboschi, S., Pelosi, G., Samarati, P.: Efficient and private access to outsourced data. In: Proc. of the 31st International Conference on Distributed Computing Systems (ICDCS 2011). Minneapolis, MN, USA (Jun 2011)
9. De Capitani di Vimercati, S., Foresti, S., Paraboschi, S., Pelosi, G., Samarati, P.: Supporting concurrency in private data outsourcing. In: Proc. of the 16th European Symposium on Research in Computer Security (ESORICS 2011). Leuven, Belgium (Sep 2011)
10. De Capitani di Vimercati, S., Foresti, S., Paraboschi, S., Pelosi, G., Samarati, P.: Distributed shuffling for preserving access confidentiality. In: Proc. of the 18th European Symposium on Research in Computer Security (ESORICS 2013). Egham, UK (Sep 2013)
11. De Capitani di Vimercati, S., Foresti, S., Paraboschi, S., Pelosi, G., Samarati, P.: Shuffle index: Efficient and private access to outsourced data. ACM Transactions on Storage 11(4), 19:1–19:55 (Oct 2015)
12. De Capitani di Vimercati, S., Foresti, S., Paraboschi, S., Pelosi, G., Samarati, P.: Three-server swapping for access confidentiality. IEEE Transactions on Cloud Computing (Jun 2015 (pre-print))
13. Goldreich, O.: Towards a theory of software protection and simulation by Oblivious RAMs. In: Proc. of the 19th Annual ACM Symposium on Theory of Computing (STOC 1987). New York, NY, USA (May 1987)
14. Goldreich, O., Ostrovsky, R.: Software protection and simulation on Oblivious RAMs. Journal of the ACM 43(3), 431–473 (May 1996)
15. Islam, M.S., Kuzu, M., Kantarcioglu, M.: Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In: Proc. of the 19th Annual Network and Distributed System Security Symposium (NDSS 2012). San Diego, California, USA (Feb 2012)
16. Kellaris, G., Kollios, G., Nissim, K., O'Neill, A.: Generic attacks on secure outsourced databases. In: Proc. of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS 2016). Vienna, Austria (Oct 2016)
17. Lin, P., Candan, K.S.: Hiding traversal of tree structured data from untrusted data stores. In: Proc. of the 2nd International Workshop on Security In Information Systems (WOSIS 2004). Porto, Portugal (Apr 2004)
18. Naveed, M., Kamara, S., Wright, C.V.: Inference attacks on property-preserving encrypted databases. In: Proc. of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS 2015). Denver, CO, USA (Oct 2015)
19. Ostrovsky, R.: Efficient computation on Oblivious RAMs. In: Proc. of the 22nd Annual ACM Symposium on Theory of Computing (STOC 1990). Baltimore, MD, USA (May 1990)
20. Ostrovsky, R., Skeith, W.E.: A survey of single-database private information retrieval: Techniques and applications. In: Proc. of the 10th International Conference on Practice and Theory in Public-Key Cryptography (PKC 2007). Beijing, China (Apr 2007)
21. Ren, L., Fletcher, C.W., Kwon, A., Stefanov, E., Shi, E., van Dijk, M., Devadas, S.: Constants count: Practical improvements to Oblivious RAM. In: Proc. of the 24th USENIX Security Symposium (USENIX 2015). Washington, DC, USA (Aug 2015)

22. Stefanov, E., van Dijk, M., Shi, E., Fletcher, C.W., Ren, L., Yu, X., Devadas, S.: Path ORAM: an extremely simple Oblivious RAM protocol. In: Proc. of the 20th ACM SIGSAC Conference on Computer and Communications Security (CCS 2013). Berlin, Germany (Nov 2013)
23. Tang, J., Cui, Y., Li, Q., Ren, K., Liu, J., Buyya, R.: Ensuring security and privacy preservation for cloud data services. ACM Computing Surveys 49(1), 13:1–13:39 (Jun 2016)