Efficient and Private Access to Outsourced Data

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

***Abstract*—As the use of external storage and data processing services for storing and managing sensitive data becomes more and more common, there is an increasing need for novel techniques that support not only data confidentiality, but also confidentiality of the accesses that users make on such data. In this paper, we propose a technique for guaranteeing content, access, and pattern confidentiality in the data outsourcing scenario. The proposed technique introduces a *shuffle index* structure, which adapts traditional $B+$-trees. We show that our solution exhibits a limited performance cost, thus resulting effectively usable in practice.**

***Keywords*-shuffle index, private access, content confidentiality, access confidentiality, pattern confidentiality**

I. Introduction

The research and industrial communities have been recently showing considerable interest in the outsourcing of data and computation. The motivations for this trend come from the economics of system administration, which present large scale economies, and by the evolution of ICT, which offers universal network connectivity that makes it convenient for users owning multiple devices to store personal data into an external server. A major obstacle toward the large adoption of outsourcing, otherwise particularly attractive to individuals and to small/medium organizations, is the perception of insecurity and potential loss of control on sensitive data and the exposure to privacy breaches. Guaranteeing privacy in a context where data are externally outsourced entails protecting the confidentiality of the data as well as of the accesses to them. In particular, it requires to maintain confidentiality on: the data being outsourced (*content confidentiality*), the fact that an access aims at a specific data (*access confidentiality*), the fact that two accesses aim at the same data (*pattern confidentiality*).

Several solutions have been proposed in the past few years, both in the theoretical and in the system communities, for protecting the confidentiality of the outsourced data [1]. Typically, such solutions (e.g., [2], [3]) consider a honest-but-curious server (i.e., a server trusted to provide the required storage and management service but not authorized to read the actual data content) and resort to encryption to protect the outsourced data. Since the server is not allowed to decrypt the data for access execution, these solutions provide different techniques for elaborating queries on encrypted data. Furthermore, they aim at content confidentiality but do not address the problem of access and pattern confidentiality.

Access and pattern confidentiality have been traditionally addressed within a different line of work by *Private Information Retrieval* (PIR) proposals (e.g., [4]–[6]), which provide protocols for querying a database that prevent the storage server from inferring which data are being accessed. PIR approaches typically work on a different problem setting. As a matter of fact, in most proposals, the external database being accessed is in plaintext (i.e., content confidentiality is not an issue). Regardless of whether the external database is plaintext or encrypted, PIR solutions have high computational complexity and are therefore not applicable to real systems. It has been proved [6] that the execution of information-theoretic PIR protocols require more resources than those required for a complete transfer of the database from the server to the client.

In this paper, we aim at providing a novel efficient approach addressing the different aspects of the privacy problem. We consider a reference scenario where a *data owner* outsources data to an external honest-but-curious server, and accesses her data by submitting requests to a *client* that directly interacts with the server. Our goal is to enable the owner to efficiently access the outsourced data while guaranteeing content, access, and pattern confidentiality from any observer, including the server itself.

We propose a novel data structure, called *shuffle index*, with which the data to be outsourced are organized (Section II). Our shuffle index assumes an unchained $B+$-tree organization of data and applies node-level encryption to hide actual data from the external storage. In the working of the system, the client can hide the actual request within cover (fake) requests, cache nodes, and shuffle the content among blocks stored at the server. In this way, no observer, including the server itself, can reconstruct the association between blocks read and actual accessed data (Section III). Our solution combines cover, caching, and shuffling techniques in an effective way (Section IV) to provide confidentiality (Sections V) while maintaining a limited performance overhead (Section VI). Our approach is the only solution known to us that delivers content, access, and pattern confidentiality at the same time, offering a performance profile adequate for real applications.

II. SHUFFLE INDEX DATA STRUCTURE

For outsourcing, we assume data to be indexed over a candidate key K defined for the data collection and organized as an *unchained $B+$-tree*, with data stored in the leaves in association with their index values, and where there are no links from a leaf to the next, representing a chain. Accesses to the data (searches) are based on the value of the index. The reason for not representing the links between the leaves is that following such links, when accessing data, would leak to the server (to which the content of the nodes is not known) *i)* the fact that the query being executed is a range query, and *ii)* the order relationship among index values in different nodes.[1] Our data structure is therefore characterized by a fan out *F*, meaning that each node (except the root) has $q \geq \lceil F/2 \rceil$ children and stores $q-1$ values $v_1, \ldots, v_{q-1}$, ordered from the smallest to the greatest. The i-th child of any internal node in the unchained $B+$-tree is the root of a subtree containing the values v with: $v < v_1$; $v_{i-1} \leq v < v_i$, $i = 2, \ldots, q-2$; $v \geq v_{q-1}$. Figure 1(a) illustrates a graphical representation of our data structure. Pointers between nodes of the abstract data structure correspond, at the logical level, to *node identifiers*, which can then be easily translated at the physical level into physical addresses. At the logical level, our data structure can be seen as a set of nodes, where each node is a pair $\langle id, n \rangle$, with *id* the node identifier and n the node content. Note that the possible order between identifiers does not necessarily correspond to the order in which nodes appear in the value-ordered abstract representation. Figure 1(b) illustrates a possible representation of the data structure in Figure 1(a), where nodes appear ordered (left to right) according to their identifiers, which are reported on the top of each node. For simplicity and easy reference, in our example, the first digit of the node identifier denotes the level of the node in the tree. The reason why we distinguish between node identifier and node content is that, as we will see later on, our approach is based on shuffling content among nodes. In other words, a given content may be associated with different identifiers at different times. In the following, when clear from the context, we will use the term *node* to refer to either the content of a node or to the content together with the identifier.

As typical in emerging outsourcing solutions, we use *encryption* to preserve *content* confidentiality. We assume encryption to be applied at the node level (i.e., each node is individually encrypted). To destroy plaintext distinguishability, the encryption function adopts a random salt. Also, the encrypted node is concatenated with the result of a MAC function applied to the encrypted node and its identifier. In this way, the client can assess the authenticity of the node returned by the server. Note that, since nodes contain pointers to children, the ability to establish authenticity of a node (starting from the root) implies the ability to establish authenticity, and therefore integrity, of the whole data structure.

[1]Range queries are supported with only the additional cost of accessing the next leaf, starting the access from the root. With a collection of shuffle indexes, the overhead due to restarting the access from the root, rather than going directly to the next leaf, causes an increase of only a few percentage points in the overall access times.

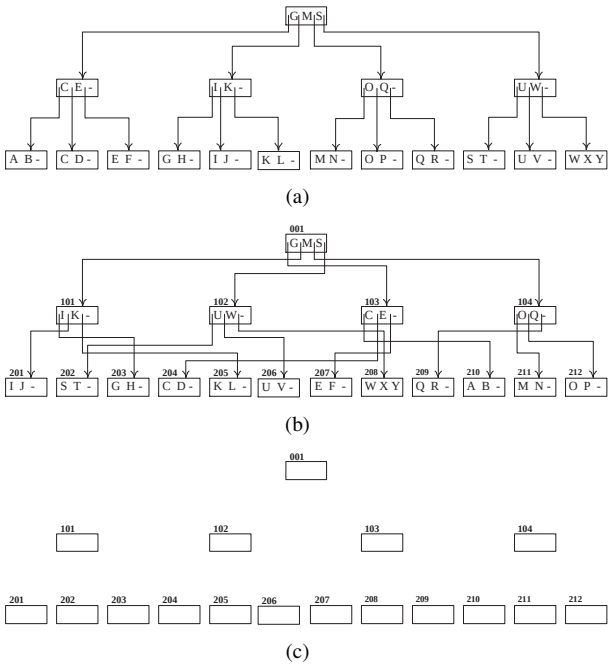


Figure 1. An example of abstract (a) and logical (b) representation of a data structure to be outsourced, and of the corresponding view of the server (c)

In the realization of physical accesses, for efficiency reasons, the size of the node to be stored (i.e., its encrypted version together with the result of the MAC function) should be a multiple of the size of the disk block. For simplicity, we assume the size of each encrypted node to be equal to the size of one disk block of the server, and the identifier of the block to be the same as the identifier of the node. We refer to an encrypted node as a *block*. Blocks are formally defined as follows.

Definition 2.1 (Block): Let $\langle id, n \rangle$ be a node of an unchained $B+$-tree. The encrypted version of $\langle id, n \rangle$, called *block*, is a pair $\langle id, b \rangle$, with $b = \mathcal{C} || \mathcal{T}$, $\mathcal{C} = E_k(salt||n)$, $\mathcal{T} = MAC_k(id||\mathcal{C})$, with E a symmetric encryption function, k the encryption key, *salt* a value chosen at random during each encryption, and *MAC* a strongly un-forgeable keyed cryptographic hash function.

We refer to the encrypted version of the logical data structure outsourced to the server and on which accesses are executed as *shuffle index*. The reason for the term *shuffle* is due to the way the structure is dynamically modified at each access, shuffling content among blocks (see Section III). Our shuffle index is defined as follows.

Definition 2.2 (Shuffle index): Let $\{\langle id_0, n_0 \rangle, \ldots, \langle id_m, n_m \rangle\}$ be a set of nodes of an unchained $B+$-tree.

The *shuffle index* is the set $\{\langle id_0, b_0\rangle, \ldots, \langle id_m, b_m\rangle\}$ of corresponding blocks (Definition 2.1).
According to the definition of shuffle index, the server just sees a collection of blocks, each with a given identifier but whose content is encrypted. Access to the data requires an iterative process between the client and the server [7]. The client performs an iteration for each level of the shuffle index starting from the root. At each iteration it determines the node to be read (i.e., the block to be retrieved from the server) at the next level. The process ends when a leaf block is retrieved, which is the block that contains the index value searched (or where it would have appeared, if the index value does not belong to the database).

III. Protection Techniques

We first describe the different aspects of confidentiality we want to guarantee against non authorized observers. We then illustrate our protection techniques complementing encryption for ensuring confidentiality.

A. Problem Statement

Our goal is to protect the confidentiality of the outsourced data against any possible observer. Since, among all possible observers, the server is the party that has the highest potential for observations (all accesses are executed by it), without loss of generality in the following we assume the server as our observer.

The server receives from the data owner a set of blocks to store and receives requests to access such blocks with the iterative process described in Section II. The server has therefore knowledge of the number m of blocks (nodes) and their identifiers, and the height h of the shuffle index (because the iterative process requires the retrieval of a block for each level of the shuffle index). Also, by observing a long enough history of accesses, the server can easily establish the level associated with each block. Note instead that the topology of the shuffle index (i.e., the pointers between parent and children) is not known to the server. Figure 1(c) illustrates the view of the server on the shuffle index in Figure 1(b).

Before defining the confidentiality we want to guarantee, we note that the server can only monitor accesses at the granularity of a block (node). The basic protection granted by encryption already ensures uncertainty on the actual index value (and therefore on the specific data) requested by an access, since any of the index values stored in the returned node could potentially be the target. Such a basic protection cannot be considered sufficient, also because index values stored in the same node will all be close within a given range. Given this observation, in the following, we consider confidentiality breaches at the granularity of nodes.

In the working of the system, every access request translates into an *observation* o_i of the server corresponding to a sequence of blocks $\{b_{i1}, \ldots, b_{ih}\}$ accessed. At any point in time, given a sequence of observations $o_1, \ldots, o_z$ corresponding to all the accesses performed, the server should not be able to infer: *i)* the data stored in the shuffle index (*content confidentiality*); *ii)* the data to which access requests are aimed, that is, $\forall i = 1, \ldots, z$, the server should not infer that o_i aims at a specific node (*access confidentiality*); and *iii)* that o_i aims at accessing the same node as o_j, $\forall i, j = 1, \ldots, z, i \neq j$ (*pattern confidentiality*).
Intuitively content confidentiality refers to the data stored in the leaves of the unchained $B+$-tree, access confidentiality to the data targeted by a request, and pattern confidentiality to the relationship between the data targeted by different requests. It is easy to see that encryption provides content confidentiality of data at rest and access confidentiality of individual requests. It is however not sufficient for providing pattern confidentiality of a set of observations. To illustrate, suppose that a shuffle index never changes. By observing that two accesses retrieve the same blocks, an observer could easily determine that the accesses refer to the same node, thus breaching pattern confidentiality. An observer can then exploit the possible information on the frequencies with which different values can be accessed and a set of observations to reconstruct the correspondence between plaintext values and blocks and infer (or restrict her uncertainty on) the specific node to which a specific access refers, thus breaching access confidentiality.

Since the information that the server can exploit in the working of the system is the comparison between the frequencies with which blocks are accessed and the frequencies of accesses to different values, the key aspect for guaranteeing all forms of confidentiality above is to destroy such a correspondence. Our approach to protect confidentiality is based on the combination of three basic strategies: *1) cover searches*, *2) cached searches*, and *3) shuffling*.

B. Cover Searches

As noted above, the execution of an access over the shuffle index can trivially leak information on the fact that two accesses aim, or do not aim, at the same node. Also, combined with the possible knowledge of the server on frequencies of accesses to node contents, it can help the server to establish the correspondence between node contents and blocks where they are stored (frequently accessed data will correspond to frequently accessed blocks) [2]. For instance, consider the logical representation of a shuffle index in Figure 1(b), and two consecutive requests for index value 'F' translating into accesses to blocks $\{$(001); (103); (207)$\}$ and $\{$(001); (103); (207)$\}$, respectively. By observing these sequences of accessed blocks, the server can infer that the two requests refer to the same data (i.e., the content of block 207). Our first protection technique aims at introducing confusion on the target of an access request by hiding it within a group of other requests that work as covers.

Cover searches are fake searches that the client executes in conjunction with the actual *target* search of the index value it aims to access. The number of cover searches is a protection parameter of our approach.

Since, as noted in Section III-A, the granularity of protection is the block (node), cover searches must provide block diversity, that is, must translate into accesses to different blocks at each level of the shuffle index, but the root. As a matter of fact, covers translating to the same block would not provide any additional protection than that offered by encryption. For instance, 'E' cannot be chosen as a cover for 'F' as both would translate into accesses to block 207, thus disclosing that the access requests refer to the content of block 207. Given a shuffle index built over a candidate key with domain $\mathcal{D}$ and a value $v \in \mathcal{D}$, *path*(v) denotes the set of blocks in the unique path of the shuffle index that starts at the root and ends in the leaf block where v is possibly stored, if v is in the database. Cover searches are formally defined as follows.

Definition 3.1 (Cover searches): Let $\{\langle id_0, b_0\rangle, \ldots, \langle id_m, b_m\rangle\}$ be a set of blocks forming a shuffle index built over a candidate key with domain $\mathcal{D}$, and let v_0 be a value in $\mathcal{D}$. A set $\{v_1, \ldots, v_n\}$ of values in $\mathcal{D}$ is a set of *cover searches* for v_0 if $\forall v_i, v_j \in \{v_0, v_1, \ldots, v_n\} : v_i \neq v_j \implies$ *path*(v_i)$\cap$*path*(v_j)=$\langle id_0, b_0\rangle$, that is, contains only the root of the shuffle index.

Basically, assuming *num_cover* searches are adopted in the execution of an access, instead of asking the server to retrieve, for each level in the shuffle index, the block in the path from the root to the target, the client asks the server to retrieve *num_cover*$+1$ blocks: one corresponds to the block on the path to the target, and each of the others corresponds to the block on the path to one cover.

Intuitively, cover searches hide the actual search within a set of searches, since any of the *num_cover* $+ 1$ leaf blocks have the same probability of containing the actual target. This requires cover searches to be indistinguishable from actual searches. We guarantee this cover/target indistinguishability property by ensuring that the frequency distribution with which values in the candidate key domain $\mathcal{D}$ are used as cover searches is the same as the frequency distribution with which values are searched upon client's request (see Section IV). For instance, consider again the two searches above for index value 'F' (block 207), and assume the first uses cover 'I' while the second one uses cover 'M'. The sequences of accesses to blocks observed by the server would now be {(001); (101,103); (201,207)} and {(001); (103,104); (207,211)}, respectively. While without cover the server was able to detect that the two requests aimed at the same block (node), with one cover the server can assess this only with probability $0.5 \cdot 0.5 = 0.25$.

The fact that searches are all executed in parallel (i.e., all the *num_cover* $+ 1$ blocks at each level of the shuffle index are retrieved before proceeding at the next level), confuses the parent-child relationship of the different blocks. In fact, at each level any of the *num_cover* $+ 1$ parents could be associated with any of the *num_cover*$+1$ children, producing therefore $(num\_cover + 1)^h$ potential paths. For instance, with reference to the example above, 201 could be child of either 101 or 103. Of course, parent-child information (like actual targets) can be disclosed by intersection attacks, observing the same set of blocks in different accesses (103 and 207 in the example above). Intersection attacks are counteracted by caching and shuffling, as explained in the remainder of this section.

C. Cached Searches

Our second protection technique aims at counteracting *intersection* attacks in the short term and consists in maintaining at the trusted client side a local copy, called *cache*, of nodes in the path to the target. Being client side, we maintain the cache in plaintext (i.e., the cache stores plaintext nodes and not their encrypted version).

Definition 3.2 (Cache): Let $\{\langle id_0, n_0\rangle, \ldots, \langle id_m, n_m\rangle\}$ be a set of nodes forming an unchained $B+$-tree of height *h*. A cache $\mathcal{C}$ of size *num_cache* for the unchained $B+$-tree is a layered structure of $h + 1$ sets $Cache_0, \ldots, Cache_h$, where:

- $Cache_0$ contains the root node $\langle id_0, n_0\rangle$;
- $Cache_l$, $l = 1, \ldots, h$, contains *num_cache* nodes belonging to the l-th level of the unchained $B+$-tree;
- $\forall n \in Cache_l$, $l = 1, \ldots, h$, the parent of n in the unchained $B+$-tree belongs to $Cache_{l-1}$ (path continuity property).

Path continuity guarantees that the parent of any node in the cache belongs to the cache. As a consequence, the path connecting the root of the unchained $B+$-tree to every node in the cache completely belongs to the cache itself. We assume the cache to be properly initialized by the data owner at the time of outsourcing, by locally storing nodes in *num_cache* disjoint paths (i.e., with only the root in common) of the unchained $B+$-tree.

In the working of the system, the cache will be updated and will keep track only of actual (and not of cover) searches, since it is intended to work as an actual cache. We assume the cache at each level to be managed according to the LRU policy, that is, when a new node is added to $Cache_l$, the node least recently used is pushed out from $Cache_l$. The application of the LRU policy guarantees the satisfaction of the path continuity property (Section IV).

The cache helps in counteracting short term intersection attacks since it avoids the client to search for a repeated target of two close access requests. For instance, with reference to the two consecutive requests for index value 'F' in Section III-B, the second request would find 'F' in cache. Since the number of blocks requested to the server has always to be the same (i.e., *num_cover*$+1$), the client would generate, for the second request, two cover searches (e.g.,

'M' and 'W'). Consequently, the observations of the server on the two requests would be {(001); (101,103); (201,207)} and {(001); (102,104); (208,211)}, respectively. The server would not be able to determine whether the two requests aim at the same target. The reader may wonder why we perform *num_cover* $+1$ fake cover searches when the target node is already in cache. First, if the observer knows that an access was to be executed, not performing it would leak information on the fact that the target node is in the cache. Second, the protection given by the cache does not work only as an independent technique, but plays a role together with the other protection techniques.

D. Shuffling

Caching does not prevent intersection attacks on observations that go beyond the size of the cache. As an example, suppose that no cache is used (i.e., *num_cache*=0), and with reference to Figure 1(b) consider three consecutive requests all for index value 'F', using one cover search for each request (e.g., 'I', 'M', and 'W', respectively). These access requests will translate into the following sequences of accesses to blocks {(001); (101,103); (201,207)}, {(001); (103,104); (207,211)}, and {(001); (102,103); (207,208)}, respectively. Assuming the indistinguishability of targets and covers, by the observation of these sequences of accesses the server can infer with probability $0.5{\cdot}0.5{\cdot}0.5 = 0.125$ that the three access requests refer to the same data (i.e., the content of block 207). Also, accesses leak to the server the parent-child relationship between blocks. While the information on the parent-child relationship by itself might seem to not compromise confidentiality, it can easily open the door to privacy breaches and should then remain confidential. Given a long enough history of observations, the server will be able to reconstruct the topology of the shuffle index and therefore gain knowledge on the similarity between values stored in the blocks.

Our third protection technique starts from the observation that inferences such as the one mentioned above are possible to the server by exploiting the one-to-one correspondence between a block and the node stored in it: accesses to the same block trivially correspond to accesses to the same node. *Node shuffling* breaks this one-to-one correspondence by exchanging the content among nodes (and therefore blocks). Since a block depends on the content of the corresponding node and on the node identifier (Definition 2.1), shuffling clearly requires the re-computation of the blocks associated with shuffled nodes and then requires node decryption and re-encryption. Note how the re-encryption of a node, applied to the node content concatenated with a possibly different node identifier and a different random salt, produces a different encrypted text (block). This aspect is particularly important since encrypted text corresponding to a given node automatically changes at each access, making it impossible to track the shuffling executed and to determine if the node content stored in a block has been changed or has remained the same. Node shuffling is formally defined as follows.

Definition 3.3 (Shuffling): Let $\mathcal{N}$=$\{\langle id_1, n_1\rangle, \ldots, \langle id_m, n_m\rangle\}$ be a set of nodes at the same level of an unchained B+-tree and π be a permutation of $id_1, \ldots, id_m$. The *node shuffling* of $\mathcal{N}$ with respect to π is the set $\{\langle id_1, n'_1\rangle, \ldots, \langle id_m, n'_m\rangle\}$ of nodes, where $id_i = \pi(id_j)$ and $n'_i = n_j$, with $i, j = 1, \ldots, m$.

Intuitively, our approach exploits shuffling by exchanging the contents of all blocks read in the execution of an access and the nodes in cache (so that their contents are shuffled), and rewriting all of them back on the server. In this way, the correspondence existing between block identifiers and the content of the nodes they store is destroyed. For instance, assume that shuffling is used and that the server observes the following sequence of accesses to blocks {(001); (101,103); (201,207)}; {(001); (103,104); (207,211)}; and {(001); (102,103); (207,208)}. The server can only note that the three sequences have a leaf block in common (i.e., 207). The three requests aim at accessing the same node only if: the second and third requests are for the content of block 207 (the probability is $0.5 \cdot 0.5 = 0.25$); the data target of the first request coincides with the content of block 207 after the first shuffling operation (the probability is 0.5); and the content of block 207 is not moved by the second shuffling operation (the probability is 0.5). As a consequence, 0.0625 is the probability that the three requests aim at the same node.

Note that shuffling among nodes at a given level requires to update the parents of the nodes so that the pointers in them properly reflect the shuffling. For instance, consider Figure 1(b) and assume nodes (103,104) are shuffled so that π(103)=104 and π(104)=103, (i.e., their contents are swapped). As a consequence, root node $[_{103}\mathrm{G}_{101}\mathrm{M}_{104}\mathrm{S}_{102}]$ must be updated to be $[_{104}\mathrm{G}_{101}\mathrm{M}_{103}\mathrm{S}_{102}]$.

IV. Access Execution and Shuffle Index Management

We illustrate how the protection techniques described in Section III (cover, cache, and shuffling) are applied in a joint way in the execution of an access and how the shuffle index is managed. Figure 2 illustrates the algorithm, executed client-side, enforcing the search process and the shuffle index updates.

Given a request for searching *target_value* in shuffle index $\mathcal{S}$, the algorithm first determines *num_cover* $+ 1$ values, *cover_value*[1], ..., *cover_value*[*num_cover* $+ 1$] to be used as cover searches (Definition 3.1) for *target_value* (lines 16-20). Note that the number of cover searches is *num_cover* $+ 1$, because for each level of the shuffle index, *num_cover*$+1$ blocks have to be downloaded from the server and therefore, if the block in the path to the target value belongs to the cache, an additional cover search becomes

necessary. For each level $l = 1, \ldots, h$, the algorithm then executes the following process. The algorithm first determines the identifiers (i.e., *ToRead_ids*) of the blocks at level l in the path to the target value (i.e, *target_id*, lines 23-24) and to the cover searches (i.e., *cover_id*[1], . . . ,*cover_id*[*num_cover* + 1], lines 32-35). If the node in the path to the target value does not belong to $Cache_l$ (i.e., a cache miss occurs), one of the values initially chosen as a cover is discarded and only *num_cover* out of the *num_cover* + 1 cover searches are performed (lines 26-31). It sends to the server a request for the blocks with identifiers in *ToRead_ids* and decrypts their content, obtaining a set *Read* of nodes (line 37). The nodes in *Read* and $Cache_l$ are then shuffled according to a random permutation π (Definition 3.3) (lines 39-40). As a consequence, the pointers stored in the nodes that are parents of the nodes in *Read* and $Cache_l$, which belong either to $Cache_{l-1}$ or to *Non_Cached_P*, are updated according to permutation π, encrypted, and sent back to the server for storage (lines 42-44). To reflect the effects of the shuffling on all the variables of interest, *target_id* and *cover_id*[i], $i = 1, \ldots,$ *num_cover* + 1, are updated according to π (lines 45-46). The algorithm finally updates $Cache_l$ by possibly inserting, if a cache miss occurred, the most recently accessed node in the path to the target value (lines 48-54). When the visit of the shuffle index terminates, the node identified by *target_id*, which is the leaf node where *target_value* is stored (if present in the database), is returned (lines 58-59).

We note that the choice of cover searches (lines 16-20) has to satisfy the target/cover *indistinguishability property*. Intuitively, indistinguishability is guaranteed if cover searches and target searches follow the same frequency distribution. However, the frequency distribution with which the target values are accessed may not be known in advance. If this is the case, the client can build a simple statistical model [8] that: *i)* estimates the probability density function bound to the occurrence of target values; and *ii)* chooses cover values by sampling from the estimated distribution. Our implementation of the shuffle index concretely implements indistinguishability. To empirically demonstrate this property, we considered recurrences within 100 accesses of the same physical blocks, and analyzed, for every recurrence, if this was due to a target or a cover access. The average value of the absolute difference in probability between targets and covers was equal to 0.0001.

Example 4.1: Figure 3 illustrates an example of algorithm execution. The columns of the table represent: the level of the shuffle index (l); the content of the cache ($Cache_l$ in *Retrieved nodes*) and the nodes read from the server (*Read* in *Retrieved nodes*); the permutation (π); the nodes in the cache and read after the shuffling (*Shuffled nodes*); the nodes written on the server (*Written nodes*). In column *Retrieved nodes*, a $*$ denotes the node along the path to *target_value*.

Consider the index in Figure 1(b) (reported for convenience at the top of Figure 4) and assume *num_cover*=1, *num_cache*=2, and *target_value*='F'. Initially, *target_id* is set to 001, the identifier of the root n_0 in $Cache_0$, and two

```
1: /* S : shuffle index on a cand. key with domain D, height h, fan out F */
2: /* Cache_l, l=0,...,h : cache */
3: /* num_cache : number of nodes in Cache_l, l=1,...,h */
4: /* num_cover : number of cover searches */
5: INPUT    target_value : value to be searched in the shuffle index
6: OUTPUT   n : leaf node that contains target_value
7: MAIN
8: /* Initialize variables*/
9: Non_Cached := Non_Cached_P := ∅
10: let n_0 be the unique node in Cache_0
11: target_id := n_0.id
12: cache_hit := TRUE /* the root always belongs to Cache_0 */
13: num_cover := num_cover + 1
14: for i:=1...num_cover do cover_id[i] := target_id
15: /* Choose cover searches*/
16: for i:=1...num_cover do
17:   randomly choose cover_value[j] in D s.t. ∀j=1,...,i−1,
18:   ChildToFollow(n_0,cover_value[i]) ≠ ChildToFollow(n_0,cover_value[j])
19:   ChildToFollow(n_0,cover_value[i]) ∉ {n.id|n∈Cache_1} and
20:   ChildToFollow(n_0,cover_value[i]) ≠ ChildToFollow(n_0,target_value)
21: /* Search, shuffle, and update cache and index structure */
22: for l:=1...h do
23:   let n∈Cache_{l−1} such that n.id=target_id
24:   target_id := ChildToFollow(n,target_value)
25:   /* identify the blocks to read from the server */
26:   if target_id∉{n.id|n∈Cache_l} then
27:     ToRead_ids := {target_id}
28:     if cache_hit then
29:       cache_hit := FALSE
30:       num_cover := num_cover − 1
31:   else ToRead_ids := ∅
32:   for i:=1...num_cover do
33:     let n∈Cache_{l−1}∪Non_Cached_P such that n.id=cover_id[i]
34:     cover_id[i] := ChildToFollow(n,cover_value[i])
35:     ToRead_ids := ToRead_ids ∪ {cover_id[i]}
36:   /* read blocks */
37:   Read := Decrypt(ReadBlocks(ToRead_ids))
38:   /* shuffle nodes */
39:   let π be a permutation of ToRead_ids∪{n.id|n∈Cache_l}
40:   for each n∈Read∪Cache_l do n.id := π(n.id)
41:   /* determine effects on parents and store nodes at level l−1 */
42:   for each n∈Cache_{l−1}∪Non_Cached_P do
43:     for i:=0..F do n.pointers[i] := π(n.pointers[i])
44:     WriteBlock(n.id, Encrypt(n))
45:   target_id := π(target_id)
46:   for i:=1...num_cover do cover_id[i] := π(cover_id[i])
47:   /* update cache at level l */
48:   Non_Cached := Read
49:   if cache_hit then refresh the timestamp of n∈Cache_l s.t. n.id=target_id
50:   else let deleted be the least recently used node in Cache_l
51:     let n∈Read s.t. n.id=target_id
52:     insert n into Cache_l
53:     Non_Cached := Non_Cached ∪ {deleted} \ {n}
54:   Non_Cached_P := Non_Cached
55: /* Write nodes at level h */
56: for each n∈Cache_h∪Non_Cached_P do WriteBlock(n.id, Encrypt(n))
57: /* Return the target leaf node */
58: let n∈Cache_h such that n.id=target_id
59: return(n)
60: CHILDTOFOLLOW(n, v)
61: i := 0
62: if v≥n.values[1] then
63:   while i+1<Length(n.values) AND v>n.values[i+1] do i := i+1
64: return(n.pointers[i])
```

Figure 2. Shuffle index access algorithm

l	**Retrieved nodes**		π	**Shuffled nodes**		**Written nodes**	
	Cache$_l$	*Read*		*Cache$_l$*	*Non_Cached*	*Cache$_{l-1}$*	*Non_Cached_P*
0	001 [$_{103}$G$_{101}$M$_{104}$S$_{102}$] *						
1	101 [$_{203}$I$_{201}$K$_{205}$- -] 103 [$_{210}$C$_{204}$E$_{207}$- -] *		101→102 103→101	102 [$_{203}$I$_{201}$K$_{205}$- -] 101 [$_{210}$C$_{204}$E$_{207}$- -]		001 [$_{101}$G$_{102}$M$_{103}$S$_{104}$]	
		102 [$_{202}$U$_{206}$W$_{208}$- -]	102→104		104 [$_{202}$U$_{206}$W$_{208}$- -]		
		104 [$_{211}$O$_{212}$Q$_{209}$- -]	104→103		103 [$_{211}$O$_{212}$Q$_{209}$- -]		
2	203 [GH-] 210 [AB-]		203→207 210→203	207 [GH-]	203 [AB-]	102 [$_{207}$I$_{201}$K$_{205}$- -] 101 [$_{203}$C$_{204}$E$_{202}$- -]	
		207 [EF-] *	207→202	202 [EF-]			104 [$_{210}$U$_{206}$W$_{208}$- -]
		202 [ST-]	202→210		210 [ST-]		103 [$_{211}$O$_{212}$Q$_{209}$- -]
						207 [GH-] 202 [EF-]	203 [AB-] 210 [ST-]

Figure 3. An example of access to the shuffle index in Figure 4 with *target_value*='F', *cover_value*[1]='S', *cover_value*[2]='M'

values, for example, 'S' and 'M', are randomly chosen as covers for 'F'.

For $l = 1$, **ChildToFollow**(001,F) returns 103, which is in *Cache*$_1$. Therefore, the two nodes in the paths to the cover searches (i.e., 104 and 102, respectively) are read from the server. The nodes in *Cache*$_1$ and in *Read* are shuffled following the permutation in Figure 3 (i.e., 101's content moves to block 102; 103's to 101; 102's to 104; and 104's to 103). *Cache*$_1$ is updated by refreshing the timestamp of node 101 (i.e., *target_id*). Then, *Non_Cached* contains the nodes in the paths to the cover searches (i.e., 104 and 103). Finally, the pointers in n_0 are updated according to π, and node 001 is encrypted and stored at the server.

For $l = 2$, **ChildToFollow**(101,F) returns 207, which does not belong to *Cache*$_2$, and hence the second cover is dropped. Nodes 202 and 207 are read. The nodes in *Cache*$_2$ and in *Read* are shuffled following the permutation in Figure 3. Node 202 (i.e., *target_id*) is inserted into *Cache*$_2$ and node 203 (which we suppose the least recently used) is pushed out and inserted into *Non_Cached*, along with node 210. The pointers in the nodes in *Cache*$_1$ and *Non_Cached_P* are updated according to π, encrypted, and sent to the server.

Finally, nodes in *Cache*$_2$ and in *Non_Cached_P* are encrypted and sent to the server. Node 202 (i.e., *target_id*) is returned. Figure 4 shows the evolution of the shuffle index.

The algorithm guarantees the following properties:

- it returns the unique node where the target value is (or should be) stored;
- it maintains at the server a shuffle index representing the original unchained $B+$-tree;
- it maintains the correctness of the cache according to Definition 3.2;
- it operates in $O((1 + \textit{num\_cover} + \textit{num\_cache}) \log_F(m))$ time, where m is the number of blocks in the shuffle index and F is the fan out.

The theorems proving the correctness and complexity of the algorithm are omitted for space constraints.

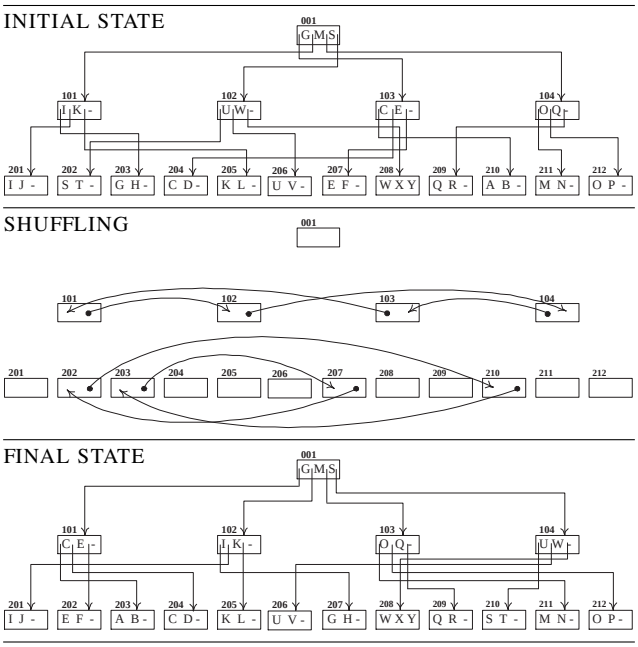


Figure 4. Evolution of the shuffle index for Example 4.1

V. Protection Analysis

In this section, we provide the intuition of how shuffling degrades the information on the correlation between nodes and blocks and how the shuffle index supports access and pattern confidentiality.

For simplicity and without loss of generality, our analysis will consider only the leaf blocks accessed at each request, since leaves are more exposed than internal nodes, which, representing only summary information on the descendants, are more protected.

Degradation due to shuffling. The continuous shuffling, which occurs at every access, is able to degrade any information the server may possess on the correspondence between nodes and blocks, reaching, after a sufficient number of accesses, a complete loss of information. This result shows an interesting feature of the shuffle index behavior and the absence of long term accumulation of information.

Access confidentiality. Access confidentiality is characterized as the protection against the server ability to associate a specific access request with a specific node/data. Static encrypted indexing structures do not exhibit access confidentiality, because the server may exploit information on the frequency of accesses (e.g., the server may know that people last names are used as key and "Smith" is the most frequently accessed value) and may thus identify the content associated with a specific node.

The shuffle index offers a natural protection against this attack. Even disregarding the caching and considering only the contribution offered by covers, every time an access is performed any information on the specific access has to be divided among all the *num_cover* + 1 nodes involved in the access request. After the nodes are shuffled, the information on the correspondence between nodes and blocks is further destroyed. In general, we observe here a reenforcing mechanism: access confidentiality is typically at risk when there are values that are characterized by high access frequency, but the higher the access frequency, the greater the destruction of information realized by shuffling.

Pattern confidentiality. Pattern confidentiality is characterized as the protection against the server ability to recognize that two separate accesses refer to the same node. We first consider a generic scenario, for which we quantify the minimum level of protection offered by the shuffle index. We then extend the analysis to the consideration of patterns separated by a number of steps smaller than the size of the cache. We can observe that the degradation of information that derives from shuffling guarantees that accesses separated by a significant number of steps will not be recognizable.

Protection by covers and shuffling. To simplify the analysis, here we suppose that the cache is not used. The server observes two consecutive requests that translate into accesses to the following two sets of leaf blocks: $\{b_{i_1}, \ldots, b_{i_{num\_cover+1}}\}$ and $\{b_{j_1}, \ldots, b_{j_{num\_cover+1}}\}$, respectively (non-consecutive requests are characterized by better protection). Two cases may occur: *i)* the two sets do not have any block in common, or *ii)* there is (at least) one block that appears in their intersection. In the first case, there is no repeated access and therefore no pattern to protect. In the second case, there is the possibility that the access to the same node has been repeated. By the cover/target indistinguishability, the probability that the intersection identifies a repeated access is $1/(num\_cover + 1)^2$. We observe that the consideration of patterns presenting a greater number of accesses (i.e., the identification of z accesses to the same node) will be characterized by a probability decreasing at a geometric rate (i.e., a sequence of z accesses presenting a non-empty intersection will be due with probability $1/(num\_cover+1)^z$ to the execution of z accesses to the same node). The server then cannot use the information on the accessed blocks to recognize accesses to the same nodes.

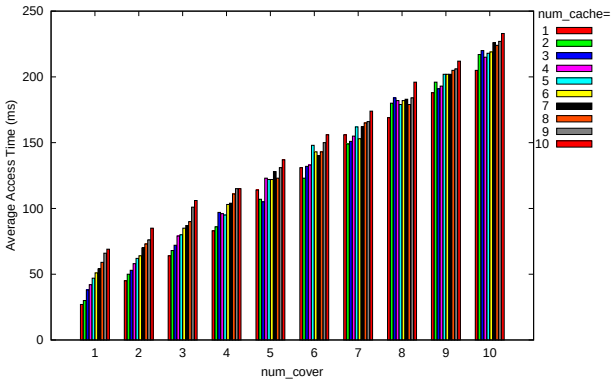
Protection by caching. Considering a worst case scenario where the server knows which are the nodes in the cache before the i-th access (with $i \leq num\_cache$), pattern confidentiality is violated when the server can identify if the i-th target access refers to the same node n_1 as the first access or not. This situation cannot happen since the server is not able to distinguish cover accesses from target accesses. Therefore, the shuffle index fully protects pattern confidentiality when the distance between the observations is within the size of the cache.

VI. Performance Analysis

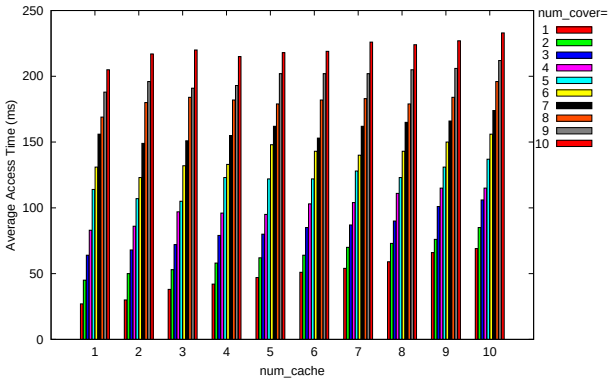
We implemented our algorithm with a Java program. To assess its performance, we used a data set of 1 TB stored in the leaves of a shuffle index with 4 levels, built on a numerical candidate key K of fixed-length, with fan out 512, and representing 2^{32} (over 4 billion) different index values. The size of the nodes of the shuffle index was 8 KB. The hardware used in the experiments included a server machine with 2 Intel Xeon Quad 2.0 GHz L3–4 MB, 12 GB RAM, four 1 TB disks, 7200 RPM, 32 MB cache, and Linux operating system with the ext4 file system. The client machine was running an Intel Core 2 Duo CPU at 2.4 GHz, with 4 GB RAM. The index was stored on all the 4 drives of the server. The performance analysis started after the system had processed a significant number of accesses, to be in a steady state. To evaluate the performance of the shuffle index we took into consideration the cost of: CPU, disk, and network.

CPU. The computational load required for the management of the shuffle index is quite limited. The algorithm uses only symmetric encryption and a MAC; the execution times we measured on an 8 KB block for both cryptographic functions are under 100 μs, a negligible fraction of the time required by network and disk accesses. The performance of the shuffle index is then driven by disk and network performance.

Disk. We analyzed the performance of the shuffle index when client and server operate in a local area network (we used a 100 Mbps Ethernet network). In this configuration, disk performance becomes the limiting factor. These experiments then permit to identify the maximum rate of queries that a server can support. Figure 5(a) reports observed times in milliseconds. The values are grouped by the same value of *num_cover* and for the same value of *num_cache*, both varying from 1 to 10. As expected, the access time grows linearly with the number of cover searches, since every additional cover requires the traversing of an additional path in the shuffle index. Although an increase in *num_cache* causes a growth in the number of blocks written for each level of the shuffle index, the number of cached nodes has a smaller impact on the access time. This is justified by the fact that the disk operations caused by the increase in *num_cache*



(a)



(b)

Figure 5. Access time in a LAN as a function of the number of covers (a) and of the size of the cache (b)

greatly benefit from buffering and cache mechanisms at the operating system and disk controller level. We claim that, as it is typical for database index structures, the bottleneck in the performance of the shuffle index in a LAN is the number and profile (e.g., random, sustained/repeated) of read and write operations on the hard disks. The access times show that the system is able, when retrieving randomly chosen 8 KB blocks over a 1 TB collection, to manage up to 40 requests per second. The best performance is obtained when using a single cover and a single cache; increasing the number of covers there is an impact on performance, but in every tested configuration the access time was below 250 ms. We also note that no solution providing support for access and pattern confidentiality offers comparable performance. The approach nearest in performance to our technique is the one in [9], discussed in the related work, which presents average access times significantly greater than the shuffle index (considering an analysis of the number of read/write accesses, supported by the experiments in [9], we estimate that the best shuffle index configuration offers a 10x-20x advantage). Also, in [9] when a specific reordering phase is triggered, access times for a single request can be in the order of hours.

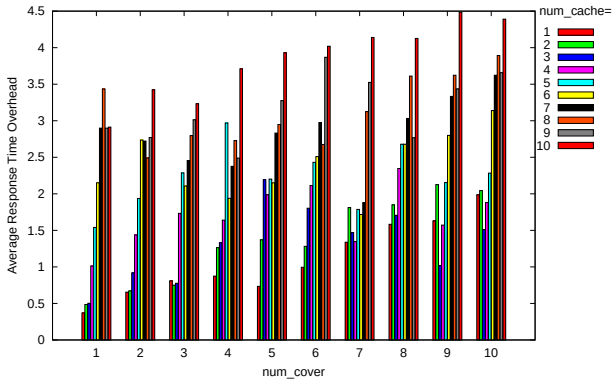


Figure 6. Overhead in a WAN compared to the use of a plain encrypted index as a function of the number of covers

Network. We analyzed the performance of the shuffle index when client and server operate in a wide area network. The server was in a University lab and the client was located in the same city, accessing the server using a 10 Mbps domestic Internet connection. This scenario, where a client uses a remote untrusted party for the private access to data, is the most interesting and natural for the shuffle index. In this configuration, network performance becomes the limiting factor. Rather than focusing on absolute numbers that strongly depend on network configuration parameters that are not under control, we were especially interested in comparing the performance of the shuffle index and the performance offered by a plain encrypted index (this is essentially the tree structure that was proposed in [7]). A plain encrypted index has the same static structure of the shuffle index, but it does not use covers, caching, and shuffling to provide access and pattern confidentiality. The plain encrypted index still requires the client to visit the nodes in the tree level-by-level. Figure 6 reports the overhead compared to the use of the plain encrypted index. The reported measures were obtained by averaging over 100 experiments for each data point. We built a statistical model to analyze the results of experiments. From the model, we derive that each increase in the number of covers or cache searches adds respectively 30% and 10% of the plain encrypted access time. The difference between the impact of covers and caches is due to the different disk costs discussed above. Again, even in a WAN configuration, our solution enjoys considerably better performance with respect to approaches providing comparable protection [9]. Also, we note that configurations with *num_cover*=1 and *num_cache* between 1 and 2 already provide a strong degree of access and pattern confidentiality with a performance overhead factor below 50% (the measured values were 170 ms for the plain encrypted index and less than 240 ms for the shuffle index). Hence, we believe our approach to be particularly appealing to many application scenarios, providing adequate access and pattern confidentiality at an affordable overhead.

VII. Related Work

Previous work is related to proposals aimed at defining indexing structures for the execution of queries on encrypted outsourced data (e.g., [2], [3], [7], [10], [11]). These proposals however focus on content confidentiality and do not guarantee access and pattern confidentiality.

Other related work is represented by classical studies on Private Information Retrieval (PIR) (e.g., [4], [5]). In these works a database is typically modeled as an N-bit string and a user is interested in retrieving the i-th bit of the collection without allowing the server to know/infer which is the bit the user is interested in. PIR protocols however suffer from high computation and communication costs [6] and typically do not address content confidentiality.

The problem presented in this paper has some affinity with the proposals in [9], [12]. In [12] the authors describe a B-tree based indexing technique that allows a user to access the content of a node in the B-tree, while guaranteeing that the server cannot infer which node has been accessed. This proposal however does not guarantee pattern confidentiality, since repeated accesses to the same node in the B-tree can leak information about its content. In [9] the authors introduce a model aimed at preserving both access and pattern confidentiality. The main similarity with our solution is that also this work is based on a data structure managed by the data owner for organizing and securely querying the remotely stored encrypted data. Their data structure is however clearly different from ours. In [9] the authors exploit the pyramid-shaped database layout of Oblivious RAM [13] and an enhanced reordering technique between adjacent levels of the data structure. Response time of any access request submitted during the reordering of lower levels of the database reaches the order of hours. This appears a strong obstacle to the real deployment of this solution; also, the amortized cost per query is $O(\log m \log \log m)$, with $O(\sqrt{m})$ temporary client storage, $O(m)$ server storage overhead (where m is the size of the data set), and the architecture requires a secure coprocessor trusted by the client on the server. Our shuffle index permits to reduce the (non amortized) computational cost for query evaluation to $O(\log m)$, maintains a constant computational and communicational overhead, and does not rely on any trust assumption on server components.

VIII. Conclusions

We presented an indexing technique for data outsourcing that proves to be efficient while ensuring content, access, and pattern confidentiality. To our knowledge this is the first work providing such a guarantee of protection while enjoying actual applicability. The shuffle index presents additional advantages. First, the underlying structure is that of $B+$-trees, which are commonly used in relational DBMSs to support the efficient execution of queries. This similarity can facilitate the integration between shuffle indexes and traditional query processing. A second advantage is the possibility for the use of multiple indexes, defined on distinct search keys, over the same collection of data.

Acknowledgment

This work was supported in part by the EC within the 7FP, under grant agreements 216483 (PrimeLife) and 257129 (PoSecCo), and by the Italian Ministry of Research within the PRIN 2008 project "PEPPER" (2008SY2PH4).

References

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

[2] A. Ceselli, E. Damiani, S. De Capitani di Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati, "Modeling and assessing inference exposure in encrypted databases," *ACM TISSEC*, vol. 8, no. 1, pp. 119–152, 2005.

[3] 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, USA, June 2002.

[4] B. Chor and N. Gilboa, "Computationally private information retrieval (extended abstract)," in *Proc. of STOC*, El Paso, USA, May 1997.

[5] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan, "Private information retrieval," *JACM*, vol. 45, no. 6, pp. 965–981, 1998.

[6] R. Sion and B. Carbunar, "On the computational practicality of private information retrieval," in *Proc. NDSS*, San Diego, USA, February/March 2007.

[7] E. Damiani, S. De Capitani Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati, "Balancing confidentiality and efficiency in untrusted relational DBMSs," in *Proc. of CCS*, Washington, USA, October 2003.

[8] B. W. Silverman, *Density Estimation for Statistics and Data Analysis*. Chapman & Hall Monographs on Statistics & Applied Probability, 1986, 1st edition.

[9] 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, USA, October 2008.

[10] C. Wang, N. Cao, J. Li, K. Ren, and W. Lou, "Secure ranked keyword search over encrypted cloud data," in *Proc. of ICDCS*, Genoa, Italy, June 2010.

[11] H. Wang and L. V. Lakshmanan, "Efficient secure query evaluation over encrypted XML databases," in *Proc. of VLDB*, Seoul, Korea, September 2006.

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

[13] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious RAMs," *JACM*, vol. 43, no. 3, pp. 431–473, 1996.