Supporting Concurrency in Private Data Outsourcing

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 Milano, Italy
pelosi@elet.polimi.it

Abstract. With outsourcing emerging as a successful paradigm for delegating data and service management to third parties, the problem of guaranteeing proper privacy protection against the external server is becoming more and more important. Recent promising solutions for ensuring privacy in such scenarios rely on the use of encryption and on the dynamic allocation of encrypted data to memory blocks for destroying the otherwise static relationship between data and blocks in which they are stored. However, dynamic data allocation implies the need to re-write blocks at every read access, thus requesting exclusive locks that can affect concurrency.

In this paper we present an approach that provides support for concurrent accesses to dynamically allocated encrypted data. Our solution relies on the use of multiple differential versions of the data index that are periodically reconciled and applied to the main data structure. We show how the use of such differential versions guarantees privacy while effectively supporting concurrent accesses thus considerably increasing the performance of the system.

1 Introduction

The evolution of information and communication technology is leading to information system architectures that rely more and more on the outsourcing to other parties of IT functions that were typically managed within an organization. A major motivation for such trend, is economical: with outsourcing an organization can simplify its structure and benefit from the large scale economies of ad-hoc IT services, with low costs and high availability. However, a significant obstacle to a greater adoption of outsourcing is today represented by possible concerns over improper exposure of confidential or sensitive information. As a matter of fact, while the external service provider can be relied upon for guaranteeing security of data and services managed, it is of utmost importance to protect possible sensitive information from the eyes of the service provider itself.

V. Atluri and C. Diaz (Eds.): ESORICS 2011, LNCS 6879, pp. 648–664, 2011.

The research and development communities have devoted significant attention to the problem of protecting data confidentiality in outsourcing scenarios, producing several solutions addressing different aspects of the problem. All proposals apply encryption to make data not intelligible to the server, providing support for query execution either by associating additional indexes with the encrypted data [1,3,8,14,15,20,21] or extending tree-based indexing structures typically adopted in DBMSs [8,17]. Tree-based approaches, unlike additional indexes, are not vulnerable to privacy breaches exploiting the possible correlation between frequencies of the index values and of the actual data behind them [3]. However, even tree-based data structures remain vulnerable to attacks based on the observation of sequences of accesses and on the analysis of the frequency distribution of access requests (i.e., by observing that certain physical blocks are often accessed). Such vulnerability can be counteracted by adopting approaches that change the location of the encrypted data at every access, so to break the otherwise static relationship between data and their physical location [10,17,22]. Dynamically allocated data structures represent the best defense against frequency attacks by the server. Among them, the shuffle index [10] extends the classical $B+$-tree structure used in databases with encryption, cover searches (to cover the actual target search with additional fake searches to "hide" it in a set and provide uncertainty over the block actually aimed by the access), and shuffling to enforce dynamic allocation. Although the shuffle index enjoys limited overhead with respect to the protection guarantees it offers [10], like other dynamically allocated data structures, it can potentially affect performance in scenarios where accesses need to operate concurrently. In fact, reallocating data at the server side requires write (hence exclusive) locks on the blocks involved in an access even in the execution of read-only operations.

In this paper, we extend the shuffle index to support a scenario where the data owner – who outsources data to the external server – wants to be able to execute several concurrent read-only transactions that need to access the remote data. Our solution to provide concurrent accesses to the shuffle index (Sect. 2) stored at the external server consists in having transactions operating on dynamically created portions of the index, which we call *delta versions* (Sect. 3). Delta versions are maintained in the server main memory, are managed – and shuffled at each access – independently one from the other (Sect. 4), and are periodically reconciled and applied to the main data structure on disk (Sect. 5). The use of periodically reconciled and merged delta versions offers protection against frequency attacks similar or better than the use of a single main index (Sect. 6) while producing an up to fourfold increase in system throughput (Sect. 7), thus offering a convincing argument for its adoption.

2 Preliminary Concepts

Before introducing our approach, we illustrate the shuffle index with which outsourced data are organized [10]. We assume data to be indexed over a candidate key and organized as an abstract *unchained $B+$-tree*, with actual data stored in

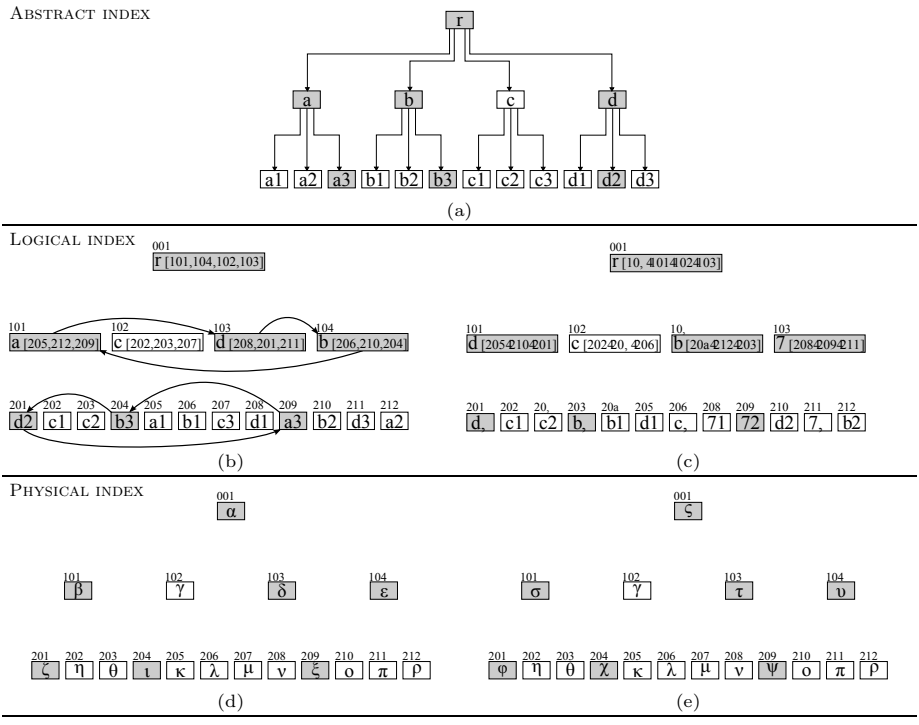

Fig. 1. An example of abstract (a), logical (b)-(c), and physical (d)-(e) index before (b)-(d) and after (c)-(e) the execution of a search operation

the leaves of the tree in association with their index values. The fact that the tree is unchained means that there are no links connecting the leaves. The fan-out F of the tree regulates the number of index values stored in the nodes. Each node stores a list $v[1,\ldots,q]$ of q values, with $\lceil \frac{F}{2} \rceil - 1 \leq q \leq F-1$ (the lower-bound does not apply to the root) ordered from the smallest to the greatest, and has $q+1$ children. The i-th children of a node is the root of the subtree containing the values val with $v[i-1] \leq val < v[i]$, $i = 2,\ldots,q$; the first child is the root of the subtree with all values $val < v[1]$, while the last child is the root of the subtree with all values $val \geq v[q]$. Figure 1(a) illustrates a graphical representation of our abstract data structure. For simplicity, in our examples we refer to nodes with a label (not explicitly reporting values in them). At the logical level, nodes are allocated to logical addresses that work as logical *identifiers*.

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 storing server. In the following, we assume that the physical address corresponds to the logical identifier of the node stored in it. 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 at the logical level of the abstract data structure in Fig. 1(a). In the figure, nodes appear ordered

(left to right) according to their identifiers, which are reported on the top of each node. Pointers to children are represented by reporting in the node the ordered list of the identifiers of its children. For simplicity and easy reference, in our example, the first digit of the node identifier denotes the level of the node in the tree. For external outsourcing, node's content is prefixed with a random salt and then encrypted in CBC mode with a symmetric encryption function producing an encrypted block. Figure 1(d) illustrates the physical representation of the logic data structure in Fig. 1(b) (Greek letters represent the result of encryption). Since the block content is encrypted, the server does not have any information on the content or on the parent-child relationship between nodes stored in blocks. Retrieval of the leaf block containing the tuple corresponding to an index value requires an iterative process. Starting from the root of the tree and ending at a leaf, the read block is decrypted retrieving the address of the child block to be read at the next step. To avoid leaking to the server *i)* the fact that some blocks represent a path in the tree and *ii)* different accesses aim at the same content, the shuffle index extends the search operation by:

- performing, in addition to the target search, other fake *cover searches*, guaranteeing indistinguishability of target and cover searches and operating on disjoint paths of the tree (retrieving at every level of the tree $num\_cover+1$ blocks at the same time);
- maintaining a set of blocks in a local *cache*;
- mixing (*shuffling*) the content of all retrieved blocks as well as those maintained in cache and rewriting them accordingly on the server.

Intuitively, cover searches introduce uncertainty over the leaf block actually belonging to the target search and do not allow the server to establish the parent-child relationship between blocks (since multiple blocks are retrieved at every level). The cache is used to make searches repeated within a short time interval not recognizable as being the same search (if the nodes in the target path are already in cache, an additional cover search will be executed instead). Shuffling moves content among blocks, thus breaking the correspondence between nodes (contents) and blocks (addresses). Note that shuffling requires to re-encrypt the involved nodes with a different random salt, so to produce a different encrypted text, and changing the pointers to them in their parents (which will have to point to the new blocks at which nodes have been allocated). Changing the allocation of nodes to blocks provides confidentiality: *i)* subsequent searches looking for the same content would aim at different blocks, and *ii)* subsequent searches hitting the same block would involve a different content.

As an example, consider a search for value $b3$ over the abstract index in Fig. 1(a) that adopts $a3$ as cover, and assume that the local cache contains the path to $d2$ (i.e., (001,103,201)). The nodes involved in the search operation are denoted in gray in the figure. Figure 1(b) illustrates the logical representation of the abstract index before the execution of the search operation and how accessed blocks are shuffled, level by level, to obtain the structure in Fig. 1(c). Note that although the server knows which blocks have been accessed (gray blocks in

Figs. 1(d)-(e)) it cannot detect which of those is the actual search target and how the content of blocks has been shuffled, since blocks are encrypted using a different salt at each encryption.

3 Main Index and Delta Versions

Before introducing the concept of delta version, we need to formalize the different components of the shuffle index data structure and of the shuffling (which were only procedurally managed in the original proposal). Data can be seen at the abstract, logical, and physical levels, which we formally capture as follows.

- *Abstract* ($\mathcal{T}^a$): set $\{n_1^a, \ldots, n_m^a\}$ of abstract nodes forming an unchained B+-tree. Each internal node in $\mathcal{T}^a$ is a pair $n^a = \langle values, children \rangle$ with *values* a list of index values and *children* a list of $q+1$ child nodes. Leaf nodes have *tuples*, representing the tuples with index value in *values*, instead of *children*.
- *Logical* ($\mathcal{T}$): triple $(\mathcal{T}^a, \mathcal{ID}, \phi)$, where $\mathcal{T}^a$ is an abstract data structure, $\mathcal{ID}$ is a set of logical identifiers, and $\phi : \mathcal{T}^a \rightarrow \mathcal{ID}$ is a bijective function associating each abstract node n^a in $\mathcal{T}^a$ with a logical identifier *id* in $\mathcal{ID}$. Triple $(\mathcal{T}^a, \mathcal{ID}, \phi)$ determines how the abstract nodes in $\mathcal{T}^a$ are allocated to logical identifiers in $\mathcal{ID}$. Each internal node $n^a = \langle values, children \rangle \in \mathcal{T}^a$ is then represented by a (logical) node of the form $\langle id, v, p \rangle$, where $id = \phi(n^a)$, $v = values$, and $p[j] = \phi(children[j])$, $j = 1, \ldots, q+1$. Leaf nodes are represented with logical nodes of the form $\langle id, v, t \rangle$ that include tuples t instead of pointers to children.
- *Physical* ($\mathcal{T}^e$): set of (disk) blocks storing $\mathcal{T}$. Each logical node $\langle id, v, p \rangle \in \mathcal{T}$ (leaf $\langle id, v, t \rangle \in \mathcal{T}$, resp.) is stored in a block that can be seen as a pair of the form $\langle id, b \rangle$, where $b = E_k(salt||id||v||p)$ ($b = E_k(salt||id||v||t)$, resp.) with E a symmetric encryption function, k the encryption key, and *salt* a value chosen at random during each encryption.

In the following, we use the term node to refer to an abstract content and block to refer to a specific memory slot in the logical/physical structure. When either term can be used, we will use node/block interchangeably.

Shuffling executed at every access randomly exchanges the content among blocks. A shuffling of logical index $\mathcal{T} = (\mathcal{T}^a, \mathcal{ID}, \phi)$ is equivalent to reallocating nodes to potentially different blocks (the corresponding abstract index remains unaltered), as formally defined in the following.

Definition 1 (Shuffling). *Let $\mathcal{T} = (\mathcal{T}^a, \mathcal{ID}, \phi)$ be a logical index and $\pi : \mathcal{ID} \rightarrow \mathcal{ID}$ be a random permutation of $\mathcal{ID}$. The* shuffling *of $\mathcal{T}$ with respect to π is a logical index $\mathcal{T}' = (\mathcal{T}^a, \mathcal{ID}, \phi')$, where $\forall n^a \in \mathcal{T}^a$, $\phi'(n^a) = \pi(\phi(n^a))$.*

Note that a change in the allocation of nodes to blocks implies that the pointers to children must be updated to reflect their new allocation, thus preserving the correct parent-child relationship. In the following, for convenience we assume

shuffling to operate within the boundary of the tree level (i.e., permutations are always performed among nodes of the same level of the tree).

A delta version is essentially a – potentially shuffled – portion of the main index, as captured by the following definition.

Definition 2 (Delta version). *Let* $\mathcal{T} = (\mathcal{T}^a, \mathcal{ID}, \phi)$ *be a logical index. A* delta version $\Delta_i = (\Delta_i^a, \mathcal{ID}_i, \phi_i)$ *of* $\mathcal{T}$ *is a shuffling of* $(\Delta_i^a, \mathcal{ID}_i, \phi)$*, where* $\Delta_i^a \subseteq \mathcal{T}^a$ *such that* $\forall n^a \in \Delta_i^a$*, the parent of* n^a *belongs to* Δ_i^a*;* $\mathcal{ID}_i = \bigcup \phi(n^a)$*,* $n^a \in \Delta_i^a$*; and* $\phi_i : \mathcal{T}^a \rightarrow \mathcal{ID}$ *such that* $\phi_i(n^a) = \phi(n^a)$ *if* $n^a \notin \Delta_i^a$*.*

Figure 2(c) illustrates an example of delta version of the logical index in Fig. 2(a). Note that, since a delta version is composed of nodes forming paths that are traversed when executing search operations, the parent of each node in the delta version also belongs to the delta version. As a consequence, every delta version always includes the root of $\mathcal{T}^a$.

Merging a delta version with a main index implies enforcing on the main index the allocation of nodes to blocks prescribed by the delta version, as captured by the following definition.

Definition 3 (Merge). *Let* $\mathcal{T} = (\mathcal{T}^a, \mathcal{ID}, \phi)$ *be a logical index and* $\Delta_i = (\Delta_i^a, \mathcal{ID}_i, \phi_i)$ *be a delta version of* $\mathcal{T}$*. The* merge *of* $\mathcal{T}$ *and* Δ_i*, denoted* $\mathcal{T} \oplus \Delta_i$*, is logical index* $\mathcal{T}' = (\mathcal{T}^a, \mathcal{ID}, \phi_i)$*.*

In terms of actual enforcement, $\mathcal{T}'$ can be simply obtained by flushing the blocks of the delta version to the main index (overwriting the corresponding blocks on disk), while leaving the other blocks unaltered. Such an operation – which can be performed without any need to download the involved blocks or performing computation by the client – produces an index that correctly represents the original data structure and includes the shuffling operated in the delta version.

4 Operating on Delta Versions

The basic idea of our approach is that transactions operate on delta versions (dynamically created and maintained in main memory at the server) rather than on the main shuffle index.

Access Execution. Every access operation is executed over a delta version. If the operation needs to read a block that does not belong to the delta version, such a block is taken from the main index and included in the delta version. Access execution works essentially like in the original shuffle index proposal requesting at every level at least *num_cover*+1 blocks. Apart from the need to include new blocks in the delta version, the only notable difference with respect to the original shuffle index proposal is that we depart from the local cache originally maintained for hiding the fact that subsequent searches were aiming at the same node. The reason for departing from the cache is that its maintenance would impose a strong synchronization overhead among the different transactions operating at the client side. To prevent the server from recognizing

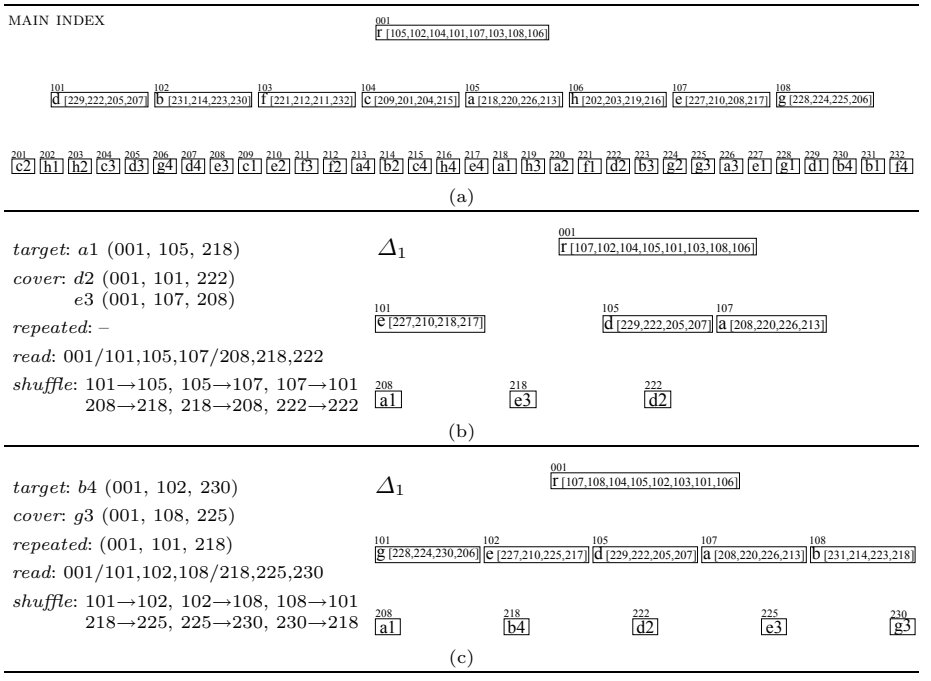

Fig. 2. An example of main index (a) and of execution of two subsequent searches (b)-(c) over it using delta version Δ_1

that two subsequent accesses aim at the same block, we take a dual approach and adopt *repeated searches*. Intuitively, while the cache ensured consequent searches never accessed the same block (if a value just retrieved was needed, a fake value was searched instead, so to ensure no intersection between the two searches and the same number of blocks is accessed at each level), repeated searches always ensure intersection between subsequent searches (regardless of whether the two searches are looking or not for the same value). For enforcing repeated searches, we store, in conjunction with each delta version, a layered structure that keeps track of the identifiers of the blocks accessed during the last search. Execution of an access on a delta version will also request at least one block per level among those appearing in the last search. Each search then accesses *num_cover*+2 blocks at every level of the index, since, besides the blocks of the target and cover searches, an additional block is necessary for the repeated search (the additional blocks are two if the target or cover searches correspond to a repeated search). At the beginning, when the delta version is empty, there is no search to repeat and an additional cover is requested instead. To illustrate, consider the index in Fig. 2(a) and a request for value $a1$ that adopts one cover and operates on empty delta version Δ_1. In this case, two covers (e.g., $d2$ and $e3$) are needed. The blocks on the paths to $a1$, $d2$, and $e3$ are all read from the main index, shuffled, and written back in Δ_1 as illustrated Fig. 2(b). Suppose now to execute another search for value $b4$ over Δ_1, with cover $g3$, and one repeated

access (e.g., 001, 101, 218). Since the nodes along the paths to $b4$ and $g3$ (except the root) do not belong to Δ_1 they are read from the main index, and after shuffling their content with all accessed blocks, are copied in the delta version. Figure 2(c) illustrates Δ_1 after the execution of the second search operation.

Delta Version Assignment. To avoid imposing synchronization constraints at the client side, we assume the allocation of delta versions to each transaction to be determined by the server. However, we need to provide a means at the client side to control the proper behavior of the server in the allocation of the versions. It is important to ensure that the server does not discard the shuffling requested, creates a new delta version at each access and having then transactions always operating on the main index (and therefore on a static data structure), or selectively allocates versions to monitor specific activities. Therefore, we assume that the client sets the number of delta versions (i.e., amount of concurrent operations). At the client side, we maintain a table VERSION($\underline{\Delta id}$, *ts*, *status*), reporting for each delta version Δid the time *ts* of last access and whether its *status* is busy or free. We assume synchronization before execution of each search operation, requesting the transaction at the client side to update the entry for the version allocated to it setting *ts* to the current time and *status* to busy. We instead account for a lazy process for the transactions in setting that the version allocated to them has been released (*status* free). Hence, while a version appearing free in the table is certainly free, a version appearing busy could actually have been released (but the transaction be late in reporting the status change). We request the server to manage delta version allocation according to the MRU policy, that is, an access should be enforced on the most recently used version. The client can then check that the server has performed proper allocation by checking that the delta version allocated to the request has *ts* greater than the greatest *ts* associated with a free version in the table (the greater than condition is to accommodate for possible delays at the client side to set version status free). We also assume the root of every delta version to be timestamped at each access. This allows checking that the root is actually the result of the access executed at the time *ts* recorded in the table for the delta version and, therefore (since the root points to the other blocks in the tree) the freshness of the whole version.

5 Reconciling Delta Versions and Main Index

A delta version grows at every access by including new requested blocks that were not previously contained in the delta version. In the long run, a delta version could potentially grow to include all the blocks of the main index saturating the server main memory. Hence, we periodically synchronize the main index with the delta versions, reporting shuffling operations on the main index and resetting the delta versions. Note that we cannot simply destroy the delta versions without changing the main index. In fact, although all operations are read-only (i.e., the abstract data structure remains unaltered), the principle of the shuffle index is that the allocation of nodes to blocks is dynamic. It is therefore important to apply the shuffle

performed on the delta versions to the main index, so to enjoy the protection of shuffling for subsequent accesses.

If there were a single delta version, applying the performed shuffling on the main index would be simple. Indeed, it would be sufficient to simply flush to the main index on disk the blocks included in the delta version. The situation is however complicated by the existence of several delta versions, which can have operated independently on the same nodes/blocks. In this case, a reconciliation is needed to ensure correctness of the index and, in particular, to ensure no content is lost and pointers to child blocks are properly set. We first note that, while it is important that shuffling is enforced in the main index, the specific way in which nodes are shuffled (i.e., which node goes to which block) does not have any impact, provided it represents a random permutation. As long as allocation is dynamic, any rearrangement would do. Hence, a straightforward approach to enforce shuffling on the main index would be to download all the blocks contained in the delta versions at the client side, retrieve (by decrypting) the corresponding nodes, allocate them to blocks, and re-uploading them at the server by rewriting the involved blocks on the main index. Such a naive approach, requiring to download all the blocks and to re-encrypt all the nodes, is clearly too expensive and not needed. Our approach aims at minimizing the blocks to be downloaded and re-uploaded by limiting these blocks to the ones strictly needed to guarantee correctness or to avoid leakage on the node allocation, while flushing as many blocks as possible directly to disk.

To determine which blocks need to be downloaded and re-encrypted, we have to identify the blocks for which the presence of multiple delta versions represents a problem. In principle, it is sufficient for two delta versions to have a block (and hence the corresponding node) in common to require checking all the blocks in them, since the node (which should be reported in only one block to the main index) may have been re-allocated to any of the blocks within each delta version. In practice, however, only the block where the node was originally allocated in the main index and the new block where it has been allocated in each of the delta versions need to be strictly involved in some re-encryption, since the delta versions have conflicting node/block allocation.

We then start by characterizing conflicting node/block allocation among a set of delta versions as follows.

Definition 4 (Conflicting allocations). *Let $\mathcal{T} = (\mathcal{T}^a, \mathcal{ID}, \phi)$ be a logical index and $\{\Delta_1, \ldots, \Delta_n\}$ be a set of delta versions of $\mathcal{T}$. The* conflicting allocations *of Δ_i with respect to $\{\Delta_1, \ldots, \Delta_n\} \setminus \{\Delta_i\}$ is a set $\mathcal{C}_i$ of pairs $\langle n_i^a, id_i \rangle$, where $n_i^a \in \Delta_i^a$, $id_i = \phi_i(n_i^a)$, and $\exists n_j^a \in \Delta_j^a$, $\Delta_j \in \{\Delta_1, \ldots, \Delta_n\}$ and $i \neq j$, such that either:* 1) $n_i^a = n_j^a$ *(same node); or* 2) $\phi_i(n_i^a) = \phi_j(n_j^a)$ *(same block).*

It is easy to see that, with respect to nodes, the nodes that are in conflict for a given delta version Δ_i are all those nodes that are also present in another version (i.e., belong to $\Delta_i^a \cap \Delta_j^a$, for some j) or are contained in a block which is also present in another version (i.e., are allocated to a block in $\mathcal{ID}_i \cap \mathcal{ID}_j$, for some j). Analogously, with respect to blocks, the blocks that are in conflict for a given delta version Δ_i are all those blocks that are also present in another

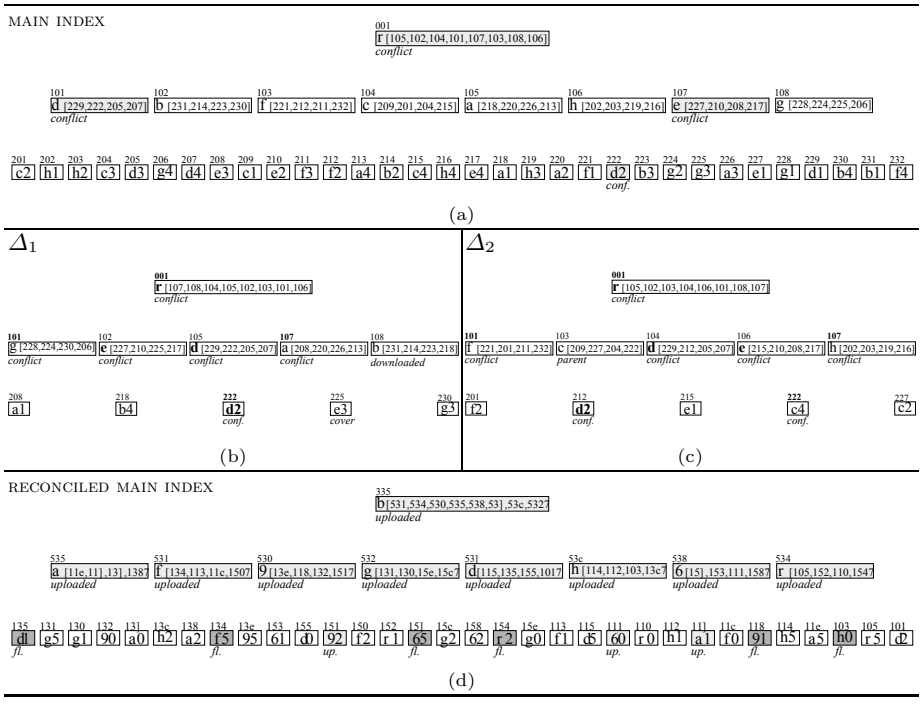

Fig. 3. An example of main index (a), two delta versions Δ_1 (b) and Δ_2 (c), and the result of their reconciliation (d)

version (i.e., belong to $\mathcal{ID}_i \cap \mathcal{ID}_j$, for some j) or that contain a node that is also present in another version (i.e., belong to $\Delta_i^a \cap \Delta_j^a$, for some j). For completeness, Definition 4 captures both components representing conflicts, in terms of pairs $\langle node, block \rangle$ since the conflict requires to revisit the allocation of the *node* contained in *block*. To illustrate, consider the two delta versions Δ_1 and Δ_2 in Figs. 3(b)-(c). The nodes/blocks representing a conflicting allocation in each version are marked with the word *conflict* below the block.

All blocks involved in a conflict for some delta version are blocks that cannot be simply written to disk as the resulting index would not be correct (some nodes would be lost and others would appear replicated). To ensure consistency of the content, it is important to reconcile the delta versions so that there is agreement – with respect to common nodes or common blocks – on which node is allocated to which block. We capture this by formalizing the definition of reconciled delta version, resulting from a reconciliation of different delta versions, as follows.

Definition 5 (Reconciled delta version). *Let $\mathcal{T} = (\mathcal{T}^a, \mathcal{ID}, \phi)$ be a logical index, $\{\Delta_1, \ldots, \Delta_n\}$ be a set of delta versions of $\mathcal{T}$, and $\mathcal{C}_i$ be the conflicting allocations of Δ_i with respect to $\{\Delta_1, \ldots, \Delta_n\} \setminus \{\Delta_i\}$, $i = 1, \ldots, n$. A* reconciled delta version *of $\{\Delta_1, \ldots, \Delta_n\}$ is a delta version $\Delta_r = (\Delta_r^a, \mathcal{ID}_r, \phi_r)$ where $\Delta_r^a =$*

$\Delta_1^a \cup \ldots \cup \Delta_n^a$, $\mathcal{ID}_r = \mathcal{ID}_1 \cup \ldots \cup \mathcal{ID}_n$, *and* $\phi_r(n^a) = \phi_i(n^a)$ *if* $n^a \in \Delta_i^a$ *and* $\langle n^a, \phi_i(n^a)\rangle \notin \mathcal{C}_i$.

The reconciled delta version can then be enforced on the shuffle index as in the case of a single delta version, by merging $\mathcal{T}$ and Δ_r producing logical index $\mathcal{T}_r = \mathcal{T} \oplus \Delta_r$ that represents the same abstract index represented by $\mathcal{T}$.

For producing the reconciled version, in addition to blocks in conflict also the blocks containing a pointer to a block in conflict (e.g., block 103 in Δ_2 in Fig. 3(c)) need to be re-written, as the pointer should be changed to refer to the new block where the child node (e.g., $c4$) has been allocated.

While the blocks in conflict and their parents are the only ones that should be downloaded by the client and re-uploaded (after shuffling the nodes in conflict) to produce a correct reconciled version (all other blocks in the delta versions could simply be flushed to disk directly by the server), we may need to download (and either include in the shuffling or simply re-write) other blocks. The reason is to ensure that the server cannot infer node/block allocation by observing that only few blocks have been involved in a reconciliation. As an example, for Δ_1 in Fig. 3(b), the only leaf block to download and re-upload would be conflicting block 222, therefore the server can infer that it stores the value accessed (as target or cover) by two searches performed with different delta versions. To avoid leakages like this, and providing the same uncertainty over the block allocation enjoyed by the original shuffle index proposal in the access execution, we require each version, for each level of the index, to: *i)* perform shuffling of either 0 or at least *num_cover*+1 blocks and *ii)* flush directly either 0 or not less that *num_cover*+1 blocks. If for a given level there are less than *num_cover*+1 blocks to flush, additional cover blocks are also downloaded and re-uploaded after re-encrypting them with a new salt (to make them not recognizable). Like parents, these latter nodes are not involved in the shuffling to avoid propagating the need for changes to higher levels of the index. For instance, with reference to Δ_1 in Fig. 3(b): *i)* 225 is added as cover to perform shuffling among at least two nodes at leaf level, and *ii)* 108 is also downloaded since it would have been the only one flushed at level one. Figure 3(d) illustrates the merging of the index in Fig. 3(a) after reconciliation of delta versions Δ_1 and Δ_2 in Figs. 3(b)-(c). The gray blocks are those that have been written on disk because flushed from main memory or re-uploaded by the client.

6 Security Analysis

We analyze the protection offered by our proposal for the new aspects introduced with respect to the serial version operating only with the main index. Like in the original proposal, we focus the analysis on leaves of the index (nodes at a higher level are subject to a greater number of accesses, due to the multiple paths that pass through them, and are then involved in a larger number of shuffling operations, which increase their protection). Since our search operations execute essentially like in the original proposal (with repeated searches instead of cache), our solution enjoys the protection guarantees given by covers like in [10]. The only

potential exposure in our solution is when two different delta versions require access to a block in the main index for the first time. Since the main index changes only upon reconciliation, the server can infer that the two requests actually refer to the same node. However, since every access execution entails reading at least *num_cover*+1 blocks (in addition to the repeated search) at every level, and covers are chosen guaranteeing indistinguishability (with respect to access profiles) between target and covers, the server cannot determine whether the transactions operating on the two different delta versions are actually aiming at the same target, or either or both of them are accessing the block as a cover. The probability that the two transactions aimed at the same target is then $\frac{1}{(num\_cover+1)^2}$; when m delta versions request access to the same block from the main index, the probability that all the transactions aimed at the same target is $\frac{1}{(num\_cover+1)^m}$.

The crucial property we are interested in evaluating is the protection against the inferences the server may make on the data content by exploiting information on the frequency of accesses to the blocks. Applying classical concepts of information theory, we can model the information available to the server on the association between a node n_i^a and block id_j storing it as probability $\mathcal{P}(n_i^a, id_j)$. A value equal to 1 for this probability means that the server will be able to correctly identify a node, whereas a value equal to $\frac{1}{|\mathcal{T}^a|}$ will correspond to the absence of any knowledge. If the block is replicated in delta versions, each instance will be associated with the analogous probability. Let ID' be the set of blocks involved in an access in a version (excluding the repeated search). For all $n_i^a \in \mathcal{T}^a$, and all $id_j \in ID'$, $\mathcal{P}(n_i^a, id_j)$ after the shuffling becomes $\sum_{id_j \in ID'} \frac{\mathcal{P}(n_i^a, id_j)}{num\_cover+1}$, because the shuffling can associate each node with any of the blocks involved in the access with equal probability, thus flattening the probability distribution. After the reconciliation, all the blocks that have been accessed by a single version will be transferred to the main index, where they will be associated with the probabilities computed in the version. Blocks accessed by multiple versions will be shuffled together, with a further averaging of probabilities among the blocks. As a consequence, $\mathcal{P}(n_i^a, id_j)$ for each node n_i^a after each access and each reconciliation will progressively move toward value $\frac{1}{|\mathcal{T}^a|}$.

It is natural to study the evolution of these probabilities using the concept of entropy, which allows us to identify at an aggregate level the knowledge of the server and its degradation due to shuffling and merging. In particular, we are interested in the impact of delta versions over the entropy, which we evaluated – as common in the study of codes and channels when analytical models become unmanageable – experimentally. We then designed a set of experiments with an initial configuration corresponding to a worst case assumption where the server has a precise knowledge about the node-block correspondence, and then the entropy is equal to zero, and evaluated how the entropy increases with access execution (for the serial index) and with access execution and merging after reconciliation (for our proposal). The experiments have considered a variety of configurations, with different numbers of nodes, number of versions, *num_cover*, and access profiles. Access profiles have been simulated by synthetically generating a sequence of

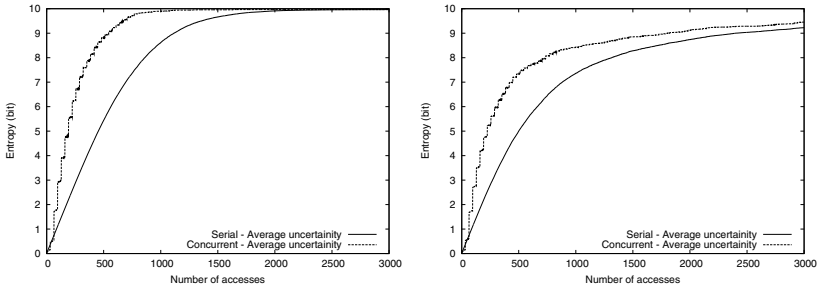

Fig. 4. Evolution of the entropy for values of γ equal to 0.5 (a) and 0.25 (b)

accesses that follow a self-similar probability distribution with skewness γ in the range $[0.25, 0.5]$ (given a domain of cardinality d, a self-similar distribution with skewness γ provides a probability equal to $1 - \gamma$ of choosing one of the first γd domain values). We then applied the same sequence of accesses to the serial and concurrent shuffle index and evaluated the growth of the entropy. Figure 4 illustrates the experimental results using 4 covers, 4 versions, 1000 nodes, skewness γ equal to 0.5 and 0.25, and varying the number of accesses. Experiments with different configurations presented a similar behavior.

As visible from the figure, before the first reconciliation, the entropy is slightly lower in the concurrent scenario with respect to the serial index. The reason is that each delta version serves a smaller number of accesses than the index in the serial version (assuming uniform distribution of load among versions, each transaction has one fourth of the accesses operating on the main index). However, already at the first reconciliation, the entropy for the concurrent scenario becomes higher than that of the serial scenario, and keeps maintaining higher. While an even higher entropy might sound not intuitive and an unexpected advantage (more protection with better performances), the explanation for such a behavior is simply that reconciliation and merging enjoy shuffling over a larger number of nodes all at one time. In fact, reconciliation makes the concurrent shuffle index stronger because this phase applies a shuffle over all the nodes in the conflict set. The size of this set depends on the number of conflicts and our model forces it to be for each delta version at least as large as the number of covers used for every shuffle. The size of the conflict set will often be greater than the number of covers, and the growth of entropy produced by a shuffle increases more than linearly with the number of blocks involved in the shuffle (i.e., the execution of two shuffles over two sets of m distinct elements produces lower entropy than a single shuffle over the set of $2m$ elements). The cost of such better protection can be reconducted to the cost of the reconciliation, which is below 10% of the access cost in the configuration that maximizes the server throughput (Sect. 7).

7 Performance Analysis

We implemented the search and reconciliation algorithms with Java programs. To assess the system performance, we used a data set of 1TB stored in the leaves of a shuffle index with 4 levels, built on a numerical candidate key of fixed-length, with fan-out 512, and representing 2^{32} (over 4 billion) different index values. The hardware used in the experiments included a server machine with 2 Intel Xeon Quad 2.0GHz L3-4MB, 12GB RAM, four 1TB disks, 7200RPM, 32MB cache, and Linux Ubuntu 9.04 x86_64 with the ext4 file system, and a client machine with an Intel Core 2 Duo CPU T5500 at 1.66GHz, 2GB DRAM, and Linux Ubuntu 9.04 x86. The client and the server operate in a local area network (100Mbps Ethernet, with average RTT of 0.48ms). The results reported in this section have been obtained as the average over 50 runs and, for each run, the number of accesses is 5000 and the number of covers adopted at each access is 4. The inverse of the average disk time needed to perform a single search is 52tps and represents the upper bound for the maximum throughput of the system.

To emulate the workload of an outsourcing service, we designed a generator scheme, modeling the number of access requests per second as a random variable following a Poisson distribution with mean arrival rate λ (the time when an access request arrives is independent from the time of arrival of previous requests). In our experiments, we considered λ=16tps and λ=32tps, which correspond to 30% and to 60%, respectively, of the physical maximum throughput (52tps). These are sensible workloads for a service hosted on a single machine and a robust test for the deployment of the proposed solution in a real world scenario. In fact, a workload of 60% of the maximum disk service rate is known to be optimal with respect to the upper bound of the physical maximum throughput [16].

To evaluate the performance gain obtained with the support of concurrent searches and the overhead due to reconciliation, we compare the server throughput in three different scenarios: *i)* serial shuffle index [10]; *ii)* concurrent shuffle index where delta versions are never reconciled; and *iii)* concurrent shuffle index where delta versions are periodically reconciled. In the experiments, delta versions are reconciled every 128 and every 256 access requests, for the configuration with λ=16tps and λ=32tps, respectively. A higher reconciliation frequency increases overhead because it more often requires write locks on the disk blocks to be re-written. On the other hand, a lower frequency requires less often such locks but over a considerably larger number of blocks (conflicts among versions grow more than linearly with respect to the number of searches). Experiments (which we do not present here for space reasons) show that the chosen threshold values balance the two aspects offering the maximum server throughput for the employed operating setup. Figure 5 reports the server throughput, varying the maximum number of delta versions between 1 and 128 with access request arrival rate equal to λ=16tps and λ=32tps, respectively. Although the performance overhead of concurrent applications highly depends on the random disk access patterns required to execute read and write accesses to blocks, Fig. 5 demonstrates how the adoption of our concurrency support offers a threefold (fourfold, respectively) increase of the server throughput compared to the serial shuffle index when

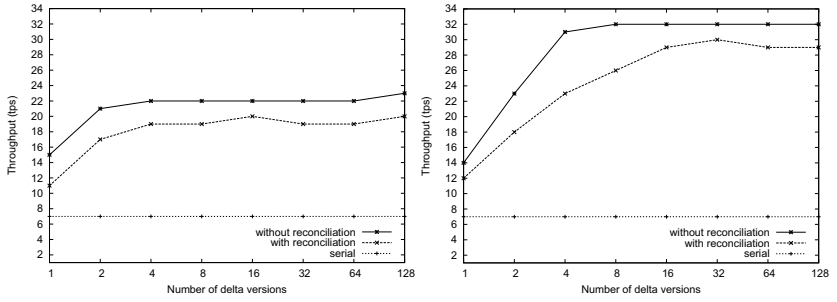

Fig. 5. Server throughput varying the number of delta versions between 1 and 128, with access request arrival rate equal to λ=16tps (a) and λ=32tps (b)

λ=16tps (λ=32tps, respectively). Note that the server throughput is higher than or equal to the mean arrival rate λ of client requests, meaning that the time necessary to the server to process an access request is lower than the time between two consecutive accesses. Figure 5 also highlights the limited cost due to reconciliation, which has a maximum of 25% and is 6% in the configuration that maximizes the server throughput.

8 Related Work

Previous work is related to the definition of indexing structures for the execution of queries on encrypted outsourced data (e.g., [1,8,14,15,20,21]). The proposals in [8,21] specifically adopt the $B+$-tree and the B-tree data structures to define an index able to efficiently support search operations on the key attribute. Although these solutions efficiently support accesses to the outsourced data, they are static and do not offer protection against the attacks based on the frequency of the accesses. Another line of work related to our is represented by Private Information Retrieval (PIR) [4,18]. These proposals typically protect the confidentiality of users' queries while data confidentiality is not considered an issue.

The proposals in [10,17,22] aim to protect data confidentiality and the accesses realized by the client over the data. The solution in [17] is based on the definition of a B-tree index and of a technique for accessing the content of a node in the tree that prevents the server from inferring which node has been accessed. However, the server can observe repeated accesses to the same physical block, which correspond to repeated searches for the same values, and apply a frequency attack to infer information about the values stored by each node in the B-tree. The proposal in [22] adopts the pyramid-shaped database layout of Oblivious RAM [13] and an enhanced reordering technique between adjacent levels of the data structure to protect both data confidentiality and the secrecy of users' queries. The performance of a search operation is however highly affected by the reordering of lower levels of the database, since this reordering can take

hours and needs to be periodically performed. This appears a strong obstacle to the real deployment of such a solution. The architecture proposed in [22] also requires a secure coprocessor trusted by the client on the server. The first proposal combining shuffling, cover searches, and cache to offer an extensive protection of confidentiality with a limited overhead in response times is illustrated in [10], where data are organized according to a novel data structure whose management does not rely on a trusted component at the server side. However, such proposal as well as the proposals in [17,22] do not support concurrency, with consequent performance limits in many real life scenarios.

9 Conclusions

Dynamically allocated data structures have recently emerged as a promising solution to provide privacy protection of data whose storage and management are delegated to external servers. However, even solutions guaranteeing limited performance overheads could be affected in scenarios where several accesses need to operate concurrently, therefore impacting their application. In this paper, we have addressed this problem and presented a proposal for accommodating concurrent executions over a shuffle index whose working (based on multiple searches and dynamic data allocation) would otherwise require several exclusive locks which, while causing only a limited overhead in serial environments, could considerably affect concurrent accesses. Our proposal, based on operating on multiple differential versions of the index, enjoys a privacy protection against frequency attacks comparable to or better than the serial solution while offering up to fourfold throughput, thus providing a convincing argument for its adoption.

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

References

1. Agrawal, R., Kierman, J., Srikant, R., Xu, Y.: Order preserving encryption for numeric data. In: Proc. of ACM SIGMOD 2004, Paris, France (June 2004)
2. Atallah, M., Frikken, K.: Securely outsourcing linear algebra computations. In: Proc. of ASIACCS 2010, Beijing, China (April 2010)
3. Ceselli, A., Damiani, E., De Capitani di Vimercati, S., Jajodia, S., Paraboschi, S., Samarati, P.: Modeling and assessing inference exposure in encrypted databases. ACM TISSEC 8(1), 119–152 (2005)
4. Chor, B., Kushilevitz, E., Goldreich, O., Sudan, M.: Private information retrieval. JACM 45(6), 965–981 (1998)
5. Cimato, S., Gamassi, M., Piuri, V., Sassi, R., Scotti, F.: Privacy-aware biometrics: Design and implementation of a multimodal verification system. In: Proc. of ACSAC 2008, Anaheim, CA, USA (December 2008)

6. Ciriani, V., De Capitani di Vimercati, S., Foresti, S., Jajodia, S., Paraboschi, S., Samarati, P.: Fragmentation design for efficient query execution over sensitive distributed databases. In: Proc. of ICDCS 2009, Montreal, Canada (June 2009)
7. Ciriani, V., De Capitani di Vimercati, S., Foresti, S., Jajodia, S., Paraboschi, S., Samarati, P.: Combining fragmentation and encryption to protect privacy in data storage. ACM TISSEC 13(3), 22:1–22:33 (2010)
8. Damiani, E., De Capitani Vimercati, S., Jajodia, S., Paraboschi, S., Samarati, P.: Balancing confidentiality and efficiency in untrusted relational DBMSs. In: Proc. of CCS 2003, Washington, DC, USA (October 2003)
9. De Capitani di Vimercati, S., Foresti, S., Jajodia, S., Paraboschi, S., Samarati, P.: Encryption policies for regulating access to outsourced data. ACM TODS 35(2), 12:1–12:46 (2010)
10. De Capitani di Vimercati, S., Foresti, S., Paraboschi, S., Pelosi, G., Samarati, P.: Efficient and private access to outsourced data. In: Proc. of ICDCS 2011, Minneapolis, MN, USA (June 2011)
11. Gamassi, M., Lazzaroni, M., Misino, M., Piuri, V., Sana, D., Scotti, F.: Accuracy and performance of biometric systems. In: Proc. of IMTC 2004, Como, Italy (May 2004)
12. Gamassi, M., Piuri, V., Sana, D., Scotti, F.: Robust fingerprint detection for access control. In: Proc. of RoboCare Workshop 2005, Rome, Italy (May 2005)
13. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious RAMs. JACM 43(3), 431–473 (1996)
14. Hacigümüs, H., Iyer, B., Mehrotra, S.: Providing database as a service. In: Proc. of ICDE 2002, San Jose, CA, USA (February 2002)
15. Hacigümüs, H., Iyer, B., Mehrotra, S., Li, C.: Executing SQL over encrypted data in the database-service-provider model. In: Proc. of SIGMOD 2002, Madison, WI, USA (June 2002)
16. Lazowska, E., Zahorjan, J., Graham, G., Sevcik, K.: Quantitative system performance: Computer system analysis using queueing network models. Prentice-Hall, Inc., Upper Saddle River (1984)
17. Lin, P., Candan, K.: Hiding traversal of tree structured data from untrusted data stores. In: Proc. of WOSIS 2004, Porto, Portugal (April 2004)
18. Olumofin, F., Goldberg, I.: Privacy-preserving queries over relational databases. In: Atallah, M.J., Hopper, N.J. (eds.) PETS 2010. LNCS, vol. 6205, pp. 75–92. Springer, Heidelberg (2010)
19. Sadeghi, A., Schneider, T., Winandy, M.: Token-based cloud computing. In: Acquisti, A., Smith, S.W., Sadeghi, A.-R. (eds.) TRUST 2010. LNCS, vol. 6101, pp. 417–429. Springer, Heidelberg (2010)
20. Shmueli, E., Waisenberg, R., Elovici, Y., Gudes, E.: Designing secure indexes for encrypted databases. In: Proc. of IFIP DBSec 2005, Storrs, CT, USA (August 2005)
21. Wang, H., Lakshmanan, L.: Efficient secure query evaluation over encrypted XML databases. In: Proc. of VLDB 2006, Seoul, Korea (September 2006)
22. Williams, P., Sion, R., Carbunar, B.: Building castles out of mud: Practical access pattern privacy and correctness on untrusted storage. In: Proc of CCS 2008, Alexandria, VA, USA (October 2008)