A Light-Weight Solution to Preservation of Access Pattern Privacy in Un-trusted Clouds

Ka Yang, Jinsheng Zhang, Wensheng Zhang, and Daji Qiao

Iowa State University, Ames, Iowa 50010, USA
{yangka,alexzjs,wzhang,daji}@iastate.edu

Abstract. Cloud computing is a new computing paradigm that is gaining increased popularity. More and more sensitive user data are stored in the cloud. The privacy of users' access pattern to the data should be protected to prevent un-trusted cloud servers from inferring users' private information or launching stealthy attacks. Meanwhile, the privacy protection schemes should be efficient as cloud users often use thin client devices to access the cloud. In this paper, we propose a lightweight scheme to protect the privacy of data access pattern. Comparing with existing state-of-the-art solutions, our scheme incurs less communication and computational overhead, requires significantly less storage space at the cloud user, while consuming similar storage space at the cloud server. Rigorous proofs and extensive evaluations have been conducted to demonstrate that the proposed scheme can hide the data access pattern effectively in the long run after a reasonable number of accesses have been made.

1 Introduction

Cloud computing [1, 12] enables enterprise and individual users to enjoy flexible, on-demand and high-quality services such as huge-volume data storage and processing, without the need to invest on expensive infrastructure, platform or maintenance. As more and more sensitive user data (e.g., financial records, health information, etc.) have been centralized into the cloud, cloud computing is facing great privacy and security challenges that may impede its fast growth and increased adoption if not well addressed. Rising to the challenges, researchers have proposed many schemes [7, 18, 23] to protect confidentiality and integrity of cloud data. Unfortunately, limited research has been conducted on the protection of users' privacy during their access to the cloud, such as the access frequency to each data item and the linkage between accesses of data items. Leakage of such access pattern information may enable potential privacy attacks such as focused attacks against selected data items. Cloud server may also infer a cloud user's activity pattern or private interest by tracking the user's access to a particular data item.

To strictly protect the privacy of data access pattern, the intention of every data access operation should be hidden so that observers of the operations cannot gain any meaningful information. Conforming to this strict requirement of access pattern privacy, Chor *et al.* [4], Ostrovsky *et al.* [14] and Itkis [10] introduced the notions of the private information retrieval (PIR) in an information theoretical setting and the computational PIR by restricting the database to perform only polynomial-time computations. Fully implementing the PIR notion is, however, expensive. As shown by Sion *et al.* [15], deployment of any single-server PIR protocol is not necessarily more efficient than a simple transfer of the entire database. Another approach to the strict preservation of data access pattern privacy is based on the notion of oblivious RAM (ORAM) [9]. In a latest ORAM implementation [20], about $\log n$ data items of the database should be scrambled every time after a single data item has been requested, where n is the total number of data items in the database. Let τ denote the size of a data item in bits. This ORAM scheme incurs a communication and computational complexity of $O(\log n \cdot \log\log n \cdot \tau)$ and requires $O(\sqrt{n} \cdot \tau)$ temporary user storage. The cost of this scheme is still rather expensive especially when the data are accessed frequently.

Although strict protection of data access pattern privacy is attractive, less strict protection, such as protecting the privacy of long-term access pattern, is also very useful in practice. For example, a malicious cloud server may use the statistical data access pattern of a user to infer the user's private information or conduct stealthy attacks. Moreover, being lightweight is also highly desired by users in cloud computing, as many of them often access the cloud with thin client devices such as smartphones. *Based on these considerations, we propose a lightweight scheme to preserve the privacy of long-term data access pattern in this paper.* The outline of the proposed scheme is as follows. Every time when a data item is needed by a user, (i) the user retrieves the desired data item together with additional dummy data items to hide the actual retrieval target; and (ii) the retrieved data items are re-encrypted and re-positioned before being stored back to the server to perturb the connections between data items and their storage locations at the server. The scheme records the storage locations of data items in index files, which are stored in a pyramid-like hierarchical structure at the cloud server to reduce communication, computational and storage overheads. Similar to data items, the access pattern to index files is also protected with additional dummies and re-positioning of the files after access. A set of delicately designed rules are used in the selection of dummy data items and index files as well as the repositioning of the files, which ensures that the connections between data items and their storage locations are reshuffled gradually, become more and more difficult to trace as the number of accesses increases, and eventually become fully un-trackable. Rigorous proofs and extensive evaluations have been conducted to demonstrate that the proposed scheme can hide the data access pattern in the long run, and the number of accesses required to preserve the access pattern privacy is reasonable in many situations.

The rest of the paper is organized as follows. Section 2 describes the system models. The proposed scheme is elaborated in Section 3, and Section 4 analyzes its security and overhead performances. Section 5 reports the evaluation results and Section 6 discusses the related work. Finally, Section 7 concludes the paper.

2 Models and Assumptions

2.1 System Model

We consider a basic cloud system with *a cloud server* and *a single cloud user*. The cloud user stores its sensitive data on the cloud server, which in turn provides an online interface for the cloud user to access the outsourced data. Later on, when the need for a data item arises, the cloud user requests it from the cloud server, updates the data item after usage, and then uploads the updated data item back to the server. Similar to [20, 9], we assume that all the data items stored at the cloud server have the same size so the server cannot identify a data item from its size. In practice, this can be achieved conveniently by appending padding bits to short data items or dividing large data items into smaller ones.

2.2 Security Model

We assume that a cloud server is curious about the user's private information and may launch malicious attacks. Specifically, it may be interested in obtaining the user's data access pattern over the long term, which primarily includes the following information: *which data items that have been requested by the user and the number of times that a particular data item has been requested by the user.*

If the access pattern information is obtained, the cloud server may be able to launch various attacks. For example, the cloud server may attempt to infer the user's activity pattern or private interest via tracking the user's access to some particular data items. The cloud server may also launch focused attacks towards user's data that are accessed with very high frequency, or stealthily delete data that are never accessed to save its storage and maintenance costs without being noticed by the user.

As for the cloud user, we assume that it has a primitive encryption function that generates different cipher-texts over different input, and the cloud server does not have non-negligible advantage over the cloud user at determining whether a pair of encrypted items of the same length represent the same data item. We assume that data confidentiality and integrity are protected using existing techniques and the communication channel between the cloud user and the cloud server is secured using mechanisms such as SSL/IPSec. We do not consider denial of service attacks or timing attacks as they can be addressed independently from this work.

2.3 Design Goal

Our main design goal is to develop a lightweight solution to prevent the cloud server from knowing the cloud user's long-term access pattern to the data stored at the cloud server, while allowing the user to access the outsourced data with low communication and computational overhead. Specifically, we preserve the access pattern privacy by breaking the connections between the data items and their storage locations gradually.

3 The Proposed Scheme

3.1 System Setup

Before describing our proposed scheme in detail, we first explain the system setup.

Hierarchical Storage Structure at the Cloud Server. We study a system where a cloud user stores n distinct data items (denoted by d_i, $i = 1, \cdots, n$) at a cloud server. All data items are encrypted with the user's secret key before uploading. In addition to data items, the cloud server stores a hierarchy of index files with the following features:

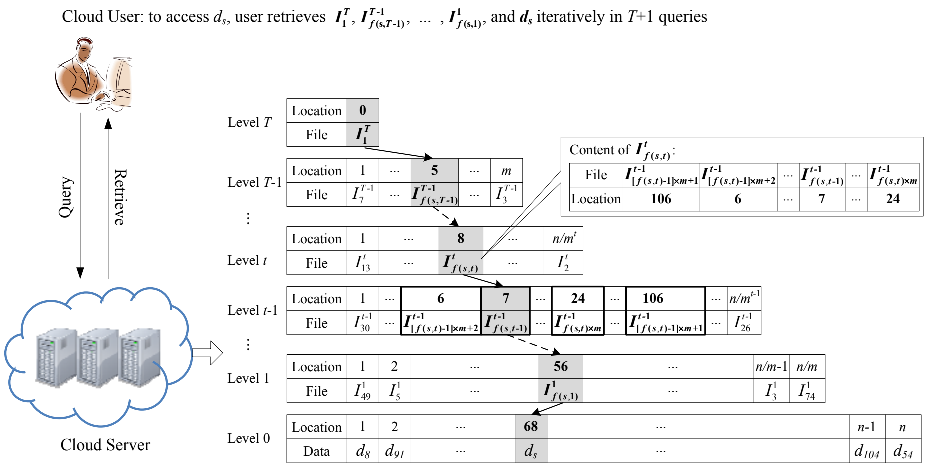

Cloud User: to access d_s, user retrieves $I_1^T, I_{f(s,T-1)}^{T-1}, \cdots, I_{f(s,1)}^1$, and d_s iteratively in $T+1$ queries
Query
Retrieve
Cloud Server
Level T
Level T-1
Level t
Level t-1
Level 1
Level 0
Location
File
Data
Content of $I_{f(s,t)}^t$:
106
6
7
24
68
56
8
5
0

Fig. 1. System setup. Data items and index files form a pyramid-like hierarchical storage structure at the cloud server. Each index file records the storage locations of m index files at its next lower level. For example, the content of $I^t_{f(s,t)}$ is shown in the callout box, and the m level-$(t-1)$ index files associated with $I^t_{f(s,t)}$ are shown as bold boxes in the figure. Here, $f(s,t) = \lceil \frac{s}{m^t} \rceil$. To obatin data item d_s, the cloud user performs a sequence of queries iteratively in a top-down manner, to obtain T index files (marked as gray boxes), one at each level of the hierarchy.

- As shown in Fig. 1, there is a total of $T = \lceil \log_m n \rceil \geqslant 1$ levels of index files, where $m > 1$ is a design parameter. In Section 4.2, we analyze the relation between m and the communication, computational and storage overheads incurred by our solution. To simplify the presentation, we assume that $\log_m n$ is an integer in the rest of the paper.
- At level t ($t = 1, \cdots, T$), there are $\frac{n}{m^t}$ index files (denoted by I^t_j, $j = 1, \cdots, \frac{n}{m^t}$). So the total number of index files in the hierarchy is $\sum\limits_{t=1}^{T} \frac{n}{m^t} = \frac{n-1}{m-1}$.
- Each index file records the storage locations of m index files at its next lower level. Specifically, I^t_j at level t contains the storage location information of the following index files at level $(t-1)$: $I^{t-1}_{(j-1)m+1}, I^{t-1}_{(j-1)m+2}, \cdots, I^{t-1}_{jm}$, as illustrated in the callout box in Fig. 1.
- There is only a single index file at the top level (i.e., level T): I^T_1.
- Data items form the bottom level (i.e., level 0) of the hierarchy.
- We assume that the files at different levels of the hierarchy are stored at non-overlapping storage spaces.

Note that, as shown in Fig. 1, there is no fixed order-correspondence between an index file (or a data item) and its storage location. This is due to the design nature of our proposed scheme, whose key idea is to randomize the storage locations of index files and data items after each access. Details of the scheme will be discussed in Section 3.2.

Iterative Query Process by the Cloud User. With such a pyramid-like hierarchical storage structure, we have the following observation about the relation between a data item and its index files: *the storage location of the data item d_s is recorded in the level-1 index file $I^1_{f(s,1)}$, whose storage location information is in turn recorded in the level-2 index file $I^2_{f(s,2)}$, so on and so forth, till the top-level index file I^T_1; here, $f(s,t)$ is defined as* $f(s,t) = \lceil \frac{s}{m^t} \rceil$. This relation is illustrated in Fig. 1 as a linked chain of gray boxes from top level T to bottom level 0.

Based on the above observation, we know that the user can obtain the desired data item d_s by performing a sequence of queries to obtain these T index files in the chain: I^T_1, $I^{T-1}_{f(s,T-1)}$, $\cdots$, $I^1_{f(s,1)}$, in a top-down manner through the hierarchy; once $I^1_{f(s,1)}$ is obtained, the user gets to know the storage location of d_s and can then issue the final query to obtain the data item. After the access, the data items and index files are updated, re-encrypted and uploaded back to the server.

We assume that the user requests the data items in rounds. To simplify the presentation, we assume that the user requests a single data item in each round. The proposed scheme may be extended to support requests of multiple data items in each round without much difficulty. In the following section, we explain our proposed scheme in detail. Table 1 lists the notations to be used in the rest of the paper.

Table 1. Notations Used in the Paper

Notation	Description
n	the total number of data items
$\mathcal{D}$	the set of all data item IDs
m	the number of storage locations recorded in an index file
I^t_j	the j-th index file at level t of the hierarchy
$\xi(j,t)$	the set of IDs of files whose storage locations are recorded in the level-t index file of ID j
$\mathcal{L}^t$	the set of storage locations of level-t files
$f(i,t)$	the ID of the index file that corresponds to data item d_i at level t
$\mathcal{Q}^t_{\text{pre}}(t \geqslant 1)$	the set of IDs and locations of level-t index files accessed in the previous round
$\mathcal{Q}^t_{\text{cur}}(t \geqslant 1)$	the set of IDs and locations of level-t index files to be accessed in the current round
$\mathcal{Q}^0_{\text{pre}}$	the set of IDs and locations of data items accessed in the previous round
$\mathcal{Q}^0_{\text{cur}}$	the set of IDs and locations of data items to be accessed in the current round

3.2 Scheme Description

Scheme Overview. Our proposed scheme is executed every time when the cloud user needs to request a data item. The key ideas of the scheme include: (i) extra dummy data items and index files (called *dummies* for short) are requested to hide the actual files of the user's interest; (ii) multiple dummies are selected so that the user's request at each round has the same format, which is a necessity to hide the access pattern [9] and (iii) the retrieved files are re-encrypted and re-positioned before being stored back to the server so as to break the connections between files and their storage locations at the server. Generally, these rules ensure that the connections between files and their storage locations are reshuffled gradually, become more and more difficult to trace as the number of accesses increases, and eventually become fully un-trackable. Detailed explanations and analysis will be presented in the following sections.

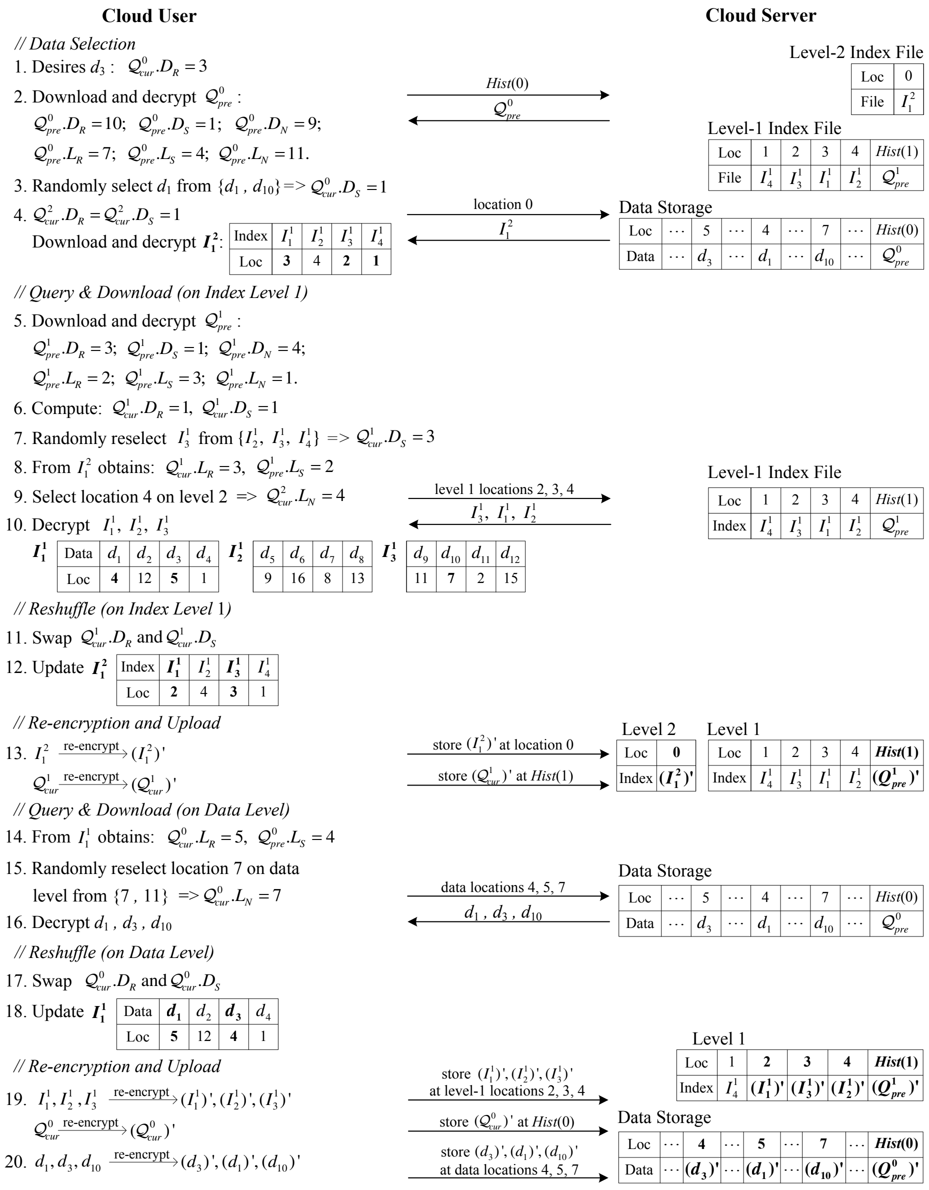
- *Assumption:* The following assumption is made on the initial condition when our scheme starts: *for any* $t = 1, \cdots, T-1$, *the mappings between level-*t *and level-*$(t-1)$ *files are unknown to the cloud server.* In other words, for any particular data item, the server has no knowledge about the corresponding index files; similarly, for any particular index file, the server has no knowledge about the corresponding index files at the upper layers.
- *Data Structures Recording Access History:* Our scheme makes use of past file access history when selecting dummies. To facilitate such mechanism, the historical information about the previous round of file access at layer t is recorded in a data structure denoted as $\mathcal{Q}^t_{\text{pre}}$, which consists of six fields: D_R, D_S and D_S recording the file IDs, and L_R, L_S and L_S recording their storage locations, respectively. The data structures are stored in cipher-text in a designated storage space at the server, and we denote the storage location of $\mathcal{Q}^t_{\text{pre}}$ as $\mathit{Hist}[t]$.
- *Structure of the Algorithm:* The pseudo-code of our scheme is presented in Algorithm 1 in Appendix 1. The scheme starts by selecting dummy data items. Then, it works iteratively to select, download, process and upload the index files, from the top level to the bottom level of the index hierarchy. In each iteration, it performs similar operations including *Selection & Downloading*, *Random Reshuffling*, and *Re-encryption & Uploading* of index files. Finally, the desired data item and the selected dummy data items are downloaded, randomly reshuffled, re-encrypted and uploaded. Detailed explanations of the operations are presented next, with a simple example given in Fig. 2.

Selection of Dummy Data Items. When the cloud user intents to retrieve a data item (denote its ID by $\mathcal{Q}^0_{\text{cur}}.D_R$), it also requests the following dummy data items to conceal its intention:

- the first dummy (whose ID is denoted as $\mathcal{Q}^0_{\text{cur}}.D_S$): the dummy that may swap its storage location with $\mathcal{Q}^0_{\text{cur}}.D_R$ after access with a probability of $1/2$;
- the second dummy (whose ID is denoted as $\mathcal{Q}^0_{\text{cur}}.D_N$): the dummy that will not swap its storage location with others.

$\mathcal{Q}^0_{\text{cur}}.D_S$ and $\mathcal{Q}^0_{\text{cur}}.D_N$ are selected to make sure that the user's request at each round has the same format: *the user always requests three data locations, out of which two and only two of them are from the ones accessed in the previous round.* Note that requiring user's request at each round to have the same format is necessary to hide the true access pattern [9]. Specifically, it hides the information about whether user's requests at two rounds are intended for the same data item. Also note that the second dummy is needed in order to guarantee that each access can keep the same format. Detailed explanations are presented in Appendix 2. To maintain the same format in each access, the data structure $\mathcal{Q}^0_{\text{pre}}$ is downloaded from the server, which records the information about the data items (namely, the data IDs and their corresponding locations) accessed in the previous round. Then, the dummies for the current round are selected according to the following rules:

- For the first dummy (i.e., $\mathcal{Q}^0_{\text{cur}}.D_S$): (i) If the intended data item is the same as the intended data item or the first dummy in the previous round, then the first dummy



Cloud User

// Data Selection

1. Desires d_3 : $Q^0_{cur}.D_R = 3$
2. Download and decrypt Q^0_{pre} :
   $Q^0_{pre}.D_R = 10$; $Q^0_{pre}.D_S = 1$; $Q^0_{pre}.D_N = 9$;
   $Q^0_{pre}.L_R = 7$; $Q^0_{pre}.L_S = 4$; $Q^0_{pre}.L_N = 11$.
3. Randomly select d_1 from $\{d_1, d_{10}\}$ => $Q^0_{cur}.D_S = 1$
4. $Q^2_{cur}.D_R = Q^2_{cur}.D_S = 1$
   Download and decrypt $\boldsymbol{I^2_1}$:

Index	I^1_1	I^1_2	I^1_3	I^1_4
Loc	**3**	4	**2**	**1**

// Query & Download (on Index Level 1)

5. Download and decrypt Q^1_{pre} :
   $Q^1_{pre}.D_R = 3$; $Q^1_{pre}.D_S = 1$; $Q^1_{pre}.D_N = 4$;
   $Q^1_{pre}.L_R = 2$; $Q^1_{pre}.L_S = 3$; $Q^1_{pre}.L_N = 1$.
6. Compute: $Q^1_{cur}.D_R = 1$, $Q^1_{cur}.D_S = 1$
7. Randomly reselect I^1_3 from $\{I^1_2, I^1_3, I^1_4\}$ => $Q^1_{cur}.D_S = 3$
8. From I^2_1 obtains: $Q^1_{cur}.L_R = 3$, $Q^1_{pre}.L_S = 2$
9. Select location 4 on level 2 => $Q^2_{cur}.L_N = 4$
10. Decrypt I^1_1, I^1_2, I^1_3

$\boldsymbol{I^1_1}$

Data	d_1	d_2	d_3	d_4
Loc	**4**	12	**5**	1

$\boldsymbol{I^1_2}$

Data	d_5	d_6	d_7	d_8
Loc	9	16	8	13

$\boldsymbol{I^1_3}$

Data	d_9	d_{10}	d_{11}	d_{12}
Loc	11	**7**	2	15

// Reshuffle (on Index Level 1)

11. Swap $Q^1_{cur}.D_R$ and $Q^1_{cur}.D_S$
12. Update $\boldsymbol{I^2_1}$

Index	$\boldsymbol{I^1_1}$	I^1_2	$\boldsymbol{I^1_3}$	I^1_4
Loc	**2**	4	**3**	1

// Re-encryption and Upload

13. $I^2_1 \xrightarrow{\text{re-encrypt}} (I^2_1)'$
    $Q^1_{cur} \xrightarrow{\text{re-encrypt}} (Q^1_{cur})'$

// Query & Download (on Data Level)

14. From I^1_1 obtains: $Q^0_{cur}.L_R = 5$, $Q^0_{pre}.L_S = 4$
15. Randomly reselect location 7 on data level from $\{7, 11\}$ => $Q^0_{cur}.L_N = 7$
16. Decrypt d_1, d_3, d_{10}

// Reshuffle (on Data Level)

17. Swap $Q^0_{cur}.D_R$ and $Q^0_{cur}.D_S$
18. Update $\boldsymbol{I^1_1}$

Data	$\boldsymbol{d_1}$	d_2	$\boldsymbol{d_3}$	d_4
Loc	**5**	12	**4**	1

// Re-encryption and Upload

19. $I^1_1, I^1_2, I^1_3 \xrightarrow{\text{re-encrypt}} (I^1_1)', (I^1_2)', (I^1_3)'$
    $Q^0_{cur} \xrightarrow{\text{re-encrypt}} (Q^0_{cur})'$
20. $d_1, d_3, d_{10} \xrightarrow{\text{re-encrypt}} (d_3)', (d_1)', (d_{10})'$

$Hist(0)$ →
← Q^0_{pre}
location 0 →
← I^2_1
level 1 locations 2, 3, 4 →
← I^1_3, I^1_1, I^1_2
store $(I^2_1)'$ at location 0 →
store $(Q^1_{cur})'$ at $Hist(1)$ →
data locations 4, 5, 7 →
← d_1, d_3, d_{10}
store $(I^1_1)', (I^1_2)', (I^1_3)'$ at level-1 locations 2, 3, 4 →
store $(Q^0_{cur})'$ at $Hist(0)$ →
store $(d_3)', (d_1)', (d_{10})'$ at data locations 4, 5, 7 →

Cloud Server

Level-2 Index File

Loc	0
File	I^2_1

Level-1 Index File

Loc	1	2	3	4	$Hist(1)$
File	I^1_4	I^1_3	I^1_1	I^1_2	Q^1_{pre}

Data Storage

Loc	⋯	5	⋯	4	⋯	7	⋯	$Hist(0)$
Data	⋯	d_3	⋯	d_1	⋯	d_{10}	⋯	Q^0_{pre}

Level-1 Index File

Loc	1	2	3	4	$Hist(1)$
Index	I^1_4	I^1_3	I^1_1	I^1_2	Q^1_{pre}

Level 2

Loc	**0**
Index	$\boldsymbol{(I^2_1)'}$

Level 1

Loc	1	2	3	4	$\boldsymbol{Hist(1)}$
Index	I^1_4	I^1_3	I^1_1	I^1_2	$\boldsymbol{(Q^1_{pre})'}$

Data Storage

Loc	⋯	5	⋯	4	⋯	7	⋯	$Hist(0)$
Data	⋯	d_3	⋯	d_1	⋯	d_{10}	⋯	Q^0_{pre}

Level 1

Loc	1	**2**	**3**	**4**	$\boldsymbol{Hist(1)}$
Index	I^1_4	$\boldsymbol{(I^1_1)'}$	$\boldsymbol{(I^1_3)'}$	$\boldsymbol{(I^1_2)'}$	$\boldsymbol{(Q^1_{pre})'}$

Data Storage

Loc	⋯	**4**	⋯	**5**	⋯	**7**	⋯	$\boldsymbol{Hist(0)}$
Data	⋯	$\boldsymbol{(d_3)'}$	⋯	$\boldsymbol{(d_1)'}$	⋯	$\boldsymbol{(d_{10})'}$	⋯	$\boldsymbol{(Q^0_{pre})'}$

Fig. 2. An example of the access procedure of a cloud user. There is a total of $n = 16$ data items and $T = 2$ levels of index files stored at the cloud server. We use d'_i to represent that data item d_i appears differently after re-encryption. In this example, data items d_1, d_9, d_{10} were accessed in the previous round. It shows how the user operates when it is interested in obtaining data item d_3 in the current round.

will be selected uniformly at random from the set of all data items excluding the intended data item of the current round. (ii) Otherwise, the first dummy will be randomly selected from the intended data item or the first dummy in the previous round with equal probability. (Refer to lines 3 to 7 in Step 1 of Algorithm 1.)

- For the second dummy (i.e., $\mathcal{Q}^0_{\text{cur}}.D_N$), its selection depends on the selection results of the first dummy: (i) If both the intended data item and the first dummy have appeared in the previous round, the second dummy will be selected uniformly at random from the set of all data storage locations excluding the locations accessed in the previous round. (ii) Otherwise, the second dummy will be selected uniformly at random from the locations accessed in the previous round excluding locations of the already-selected files. (Refer to lines 12 to 20 in Step 2 of Algorithm 1 when $t = 0$.)

In the example given in Fig. 2, in the previous round, data $\#10$ was intended by the user and data $\#1$ was selected as the first dummy. Since data $\#3$ is needed in the current round (i.e., case (ii) in the first dummy selection rules), the user randomly selects the first dummy, which is data $\#1$ in this example, from data $\#10$ and data $\#1$ (as shown by step 3). As the selected data items did not both appear in the previous round (i.e., case (ii) in the second dummy selection rules), the second dummy's location, which is 7 in this example (as shown by step 15), is selected from data $\#10$ and data $\#9$'s locations (i.e., data locations $\#7$ and $\#11$).

Selection, Downloading, Processing and Uploading of Index Files. First, the single top-level index file is downloaded and decrypted, and its ID is recorded in $\mathcal{Q}^T_{\text{cur}}.D_R$, $\mathcal{Q}^T_{\text{cur}}.D_S$, and $\mathcal{Q}^T_{\text{cur}}.D_N$, i.e., $\mathcal{Q}^T_{\text{cur}}.D_R = \mathcal{Q}^T_{\text{cur}}.D_S = \mathcal{Q}^T_{\text{cur}}.D_N = 1$ (as shown by step 4 in the example of Fig. 2). Then, three index files for each level t, where $(T-1) \geqslant t \geqslant 1$, are selected, downloaded, processed and uploaded, in an iterative and top-down manner. Without loss of generality, the following describes the operations for iteration t.

Selection & Downloading of Level-t Index Files. The files that contain the level-t indices of the intended data item ($\mathcal{Q}^0_{\text{cur}}.D_R$) and the first dummy ($\mathcal{Q}^0_{\text{cur}}.D_S$) are first selected to access. The IDs of these files are denoted as $\mathcal{Q}^t_{\text{cur}}.D_R$ and $\mathcal{Q}^t_{\text{cur}}.D_S$ respectively. Note that, these file IDs can be found out by using the afore-defined $f(\cdot,\cdot)$ function, i.e., $\mathcal{Q}^t_{\text{cur}}.D_R = f(\mathcal{Q}^0_{\text{cur}}.D_R, t)$ and $\mathcal{Q}^t_{\text{cur}}.D_S = f(\mathcal{Q}^0_{\text{cur}}.D_S, t)$. Then, similar to the selection of dummy data items, additional dummy index files are selected to make sure that, in each round, three level-t index files are accessed and exactly two of them appeared in the previous round. The following rules are applied in the selection:

- For the first dummy index file (i.e., $\mathcal{Q}^t_{\text{cur}}.D_S$): If the intended data item and the first dummy share the same level-t index file, the first dummy index file is re-selected uniformly at random from the index files whose storage locations are stored in files $\mathcal{Q}^{t+1}_{\text{cur}}.D_R$ or $\mathcal{Q}^{t+1}_{\text{cur}}.D_S$, i.e., the level-$(t+1)$ intended index file and the first dummy index file downloaded in the previous iteration of this algorithm. (Refer to lines 8 to 10 in Step 2 of Algorithm 1.)
- For the second dummy index file (i.e., $\mathcal{Q}^t_{\text{cur}}.D_N$): (i) If the intended index file and the first dummy index file have both appeared in the previous round, the second

dummy index file will be selected uniformly at random from all level-t index file locations excluding the locations that appeared in the previous round. (ii) Otherwise, the second dummy index file will be selected uniformly at random from the locations that appeared in the previous round excluding locations of the already-selected files. (Refer to lines 12 to 20 in Step 2 of Algorithm 1 when $t \neq 0$.)

After the level-t index files have been selected, the locations of files $\mathcal{Q}^t_{\text{cur}}.D_R$ and $\mathcal{Q}^t_{\text{cur}}.D_S$ can be found by searching their indices in the downloaded level-$(t+1)$ index files, i.e., files $\mathcal{Q}^{t+1}_{\text{cur}}.D_R$ and $\mathcal{Q}^{t+1}_{\text{cur}}.D_S$. Then the locations of the three level-t index files are provided to the server and the files can be downloaded. Note that, the locations are presented to the server in an arbitrary order, so that the server cannot distinguish between desired index files and dummies. The downloaded files are then decrypted with the user's key.

In the example given in Fig. 2, since the intended data item and the first dummy share the same level-1 index file I^1_1, the user randomly selects a new first dummy index file, which is I^1_3 in this example, from level-1 index files $\{I^1_2, I^1_3, I^1_4\}$ (as shown by steps 6 and 7). Then the user looks up I^2_1 to find out the storage locations $\mathcal{Q}^1_{\text{cur}}.L_R$ and $\mathcal{Q}^1_{\text{cur}}.L_S$ (as shown by step 8). Since both I^1_1 and I^1_3 were accessed in the previous round, the user selects the second dummy index file with location $\#4$ (as shown by step 9). Hence, the user retrieves the files from level-1 storage locations $\#2$, $\#3$ and $\#4$.

Random Reshuffling of Selected Level-t Index Files. The intended index file ($\mathcal{Q}^t_{\text{cur}}.D_R$) and the first dummy index file ($\mathcal{Q}^t_{\text{cur}}.D_S$) may swap their storage locations with a probability of $1/2$. If the swap happens, the index information of these files should be updated in their index files $\mathcal{Q}^{t+1}_{\text{cur}}.D_R$ and $\mathcal{Q}^{t+1}_{\text{cur}}.D_S$, respectively. In the example given in Fig. 2, since files $\mathcal{Q}^1_{\text{cur}}.D_R$ and $\mathcal{Q}^1_{\text{cur}}.D_S$ are swapped, the user updates I^2_1 accordingly (as shown by steps 11 and 12).

Re-encryption & Uploading of Index Files. Now, we have completed the processing of level-$(t+1)$ index files $\mathcal{Q}^{t+1}_{\text{cur}}.D_R$, $\mathcal{Q}^{t+1}_{\text{cur}}.D_S$ and $\mathcal{Q}^{t+1}_{\text{cur}}.D_N$. To hide content and/or location changes made to them, these files should be re-encrypted before being uploaded back to the server. In our scheme, re-encryption is performed by applying the Cipher Block Chaining (CBC) encryption techniques [13] on the file content, where the first block of the file is a non-reappearing nonce. The user's key is used in the re-encryption. This way, the same secret key can be reused for encrypting all files, which simplifies the key management at the cloud user. Such re-encryption process ensures that a computationally bounded adversary does not have non-negligible advantage at determining whether a pair of encrypted data items (before and after re-encryption, respectively) carry the same data content.

After re-encryption, files $\mathcal{Q}^{t+1}_{\text{cur}}.D_R$, $\mathcal{Q}^{t+1}_{\text{cur}}.D_S$ and $\mathcal{Q}^{t+1}_{\text{cur}}.D_N$ are uploaded to their locations, respectively, but in an arbitrary order to make it difficult for the cloud server to track these files. At the end of iteration t, data structure $\mathcal{Q}^t_{\text{pre}}$ should be replaced by $\mathcal{Q}^t_{\text{cur}}$, then re-encrypted and uploaded to location *Hist*$[t]$. This way, next time when $\mathcal{Q}^t_{\text{pre}}$ is downloaded, it will reflect the mostly recent access history.

In the example given in Fig. 2, I^2_1 and $\mathcal{Q}^1_{\text{cur}}$ are re-encrypted and uploaded to the server at the storage locations $\#0$ and *Hist*$[1]$, respectively (as shown by step 13).

Downloading, Processing and Uploading of Data Items. After the above steps, the level-1 index files have been downloaded and decrypted. Based on the index information in these files, the desired data item and two additional dummy data items can be downloaded from the cloud server and decrypted with the user's key. Upon the user's access to the desired data item has been completed, the intended data item and the first dummy may swap their storage locations with a probability of $1/2$, and if the swap happens, changes will be made to the level-1 index files $\mathcal{Q}^1_{\text{cur}}.D_R$ and/or $\mathcal{Q}^1_{\text{cur}}.D_S$, respectively. Finally, the three level-1 index files and the three data items are re-encrypted and uploaded to the cloud server. Also, data structure $\mathcal{Q}^0_{\text{pre}}$ is updated to $\mathcal{Q}^0_{\text{cur}}$, re-encrypted and uploaded to the server. The re-encryption and uploading operations are performed in the similar manner as described above.

In the example given in Fig. 2, the user looks up I^1_1 to find the storage locations $\mathcal{Q}^0_{\text{cur}}.L_R = 5$ and $\mathcal{Q}^0_{\text{cur}}.L_S = 4$. As afore-explained, the user selects the second dummy's storage location $\mathcal{Q}^0_{\text{cur}}.L_R = 7$ (as shown by steps 14 and 15). Since data items $\mathcal{Q}^0_{\text{cur}}.D_R$ and $\mathcal{Q}^0_{\text{cur}}.D_S$ are swapped, the content of I^1_1 is updated (as shown by steps 17 and 18). Finally, the re-encrypted level-1 index files, $\mathcal{Q}^0_{\text{cur}}$ and data items are uploaded to the server respectively.

4 Security and Overhead Analysis

In this section, we first show that the proposed scheme can preserve the privacy of user data access pattern in the long run. That is, after a sufficiently large number of accesses, the frequency with which each data item has been accessed cannot be figured out by the cloud server. Then we discuss the practical implications of this security property through analyzing how our scheme can deal with some typical attacks that are based on the knowledge of data access pattern. Finally we analyze the overhead of the proposed scheme.

4.1 Security Analysis

We first show that the access pattern of index file locations, which can be observed by the cloud server, does not reveal extra information about the data access pattern. In the proposed scheme, index files are used to facilitate user query and data access. The content of an index file is protected by being re-encrypted after each access, based on the user's secret key and a random non-repeating nonce. Hence, it is impossible for the server to gain information about the data access pattern from the content of index files. The following theorem states that observing the access pattern of index file storage locations does not reveal more information about data access pattern than observing only the access pattern of data storage locations.

Theorem 1. The cloud server cannot gain any advantage in inferring user's data access pattern through observing the access pattern of index file storage locations.

Proof. Refer to [21].

As the observed access pattern of index file locations does not help in inferring data access pattern, we next study what can be inferred from observing only the access pattern of data storage locations. The following theorem formally states the property that, if the cloud server can only observe the access pattern of data storage locations, the data access pattern, namely, the data item requested by a cloud user and the frequency with which each data item has been accessed by a cloud user, can be preserved in the long run.

Theorem 2. If a cloud user has accessed the data items, despite the user access sequence, for a sufficiently large number of times, each storage location at the cloud server is accessed uniformly at random.

Proof. Refer to Appendix 3 for a sketch of the proof and [21] for the detailed proof.

Note that the proof of Theorem 2 also implies that, after a sufficiently large number of accesses, the server does not have non-negligible advantage at determining whether a specific data storage location corresponds to a particular data item.

Discussion. To further understand the practical implications of the above security property, we now discuss a few typical attacks that are based on the knowledge of data access pattern, and analyze how our scheme can deal with the attacks.

Security Against Tracking Data Items. Suppose the cloud server has identified a particular user data item via other means, e.g., physical spying. It may want to keep track of this data item thereafter. Using our proposed scheme, due to the property described in Theorem 2, after a sufficiently large number of accesses, the server does not have non-negligible advantage at determining which location the target data item is at. For example, after the first round that the target item has been accessed, from the server's perspective, the target item may be stored at any of the three accessed locations with an equal probability of $1/3$. Then if any of these three locations is accessed in the next round, the probability will be divided further among the newly accessed locations. Therefore, by solely observing the storage locations accessed by the user, the server could lose track of the target data item quickly.

Security Against Focused Attacks on Selected Data Items. Some of the cloud user's data items may be requested with very high frequency. These files are often important to the user. If a malicious cloud server knows which data items are frequently accessed, it may launch intensive attacks on the data, attempting to find out the content or contextual information of the data. Note this, such attacks are sometimes feasible in practice, for example, when the adopted data encryption algorithm or the key chosen by the user is not sophisticated enough, or some side information about the data can be obtained in other means. Using our proposed scheme, due to the property described in Theorem 2, all data storage locations will be equally accessed in the long run. Hence, the server cannot identify which data items are frequently requested by the user. Similarly, some of the cloud user's data items may be requested with very low frequency, e.g., backup data. A malicious cloud server may want to stealthily delete these rarely-accessed user data items to save storage and maintenance cost for itself without being noticed by the user. Such attack can also be stopped as our proposed scheme prevents the server from identifying rarely requested data items.

4.2 Overhead Analysis

Communication and Computational Overhead. With our proposed scheme, to access a single data item, the cloud user needs to obtain the following information from the cloud server:

- Three index files at each level of the storage hierarchy; each index file records the storage locations of m index files at its next lower level and it takes $\log n$ bits to represent a storage location.
- One access history file at each level of the storage hierarchy; each access history file records the IDs and storage locations of three index files (at this level) that were accessed in the previous round; hence, it contains six fields and each field is $\log n$-bit long.
- The desired data item and two additional dummy data items; let τ denote the size of each data item in bits.

Recall that there is a total of $\log_m n$ levels in our proposed hierarchical storage structure. Therefore, the overall communication and computational overhead for accessing a single data item can be calculated as:

$$\text{OH}_{\text{c\&c}} = m \log n \cdot 3 \log_m n + 6 \log n \cdot \log_m n + 3\tau. \tag{1}$$

It is easy to verify that:

$$\begin{cases} \min \text{OH}_{\text{c\&c}} = \text{OH}_{\text{c\&c}}|_{m=4} = 9(\log n)^2 + 3\tau; \\ \max \text{OH}_{\text{c\&c}} = \text{OH}_{\text{c\&c}}|_{m=n} = (3n+6)\log n + 3\tau. \end{cases} \tag{2}$$

Storage Overhead. As explained in Section 3.1, the total number of index files in our proposed scheme is $\frac{n-1}{m-1}$. Each index file records the storage locations of m index files at its next lower level and it takes $\log n$ bits to represent a storage location. Therefore, the overall storage overhead at the cloud server can be calculated as:

$$\text{OH}_{\text{s\_server}} = m \log n \cdot \frac{n-1}{m-1} + n\tau. \tag{3}$$

It is easy to verify that:

$$\begin{cases} \min \text{OH}_{\text{s\_server}} = \text{OH}_{\text{s\_server}}|_{m=n} = n \log n + n\tau; \\ \max \text{OH}_{\text{s\_server}} = \text{OH}_{\text{s\_server}}|_{m=2} = 2(n-1)\log n + n\tau. \end{cases} \tag{4}$$

At the user side, to operate our proposed scheme, the cloud user needs to store one access history file, three index files, and three more index files or data items at any given time. Therefore, the required storage at the user side is:

$$\text{OH}_{\text{s\_user}} = 6 \log n + 3m \log n + \max\{3m \log n, 3\tau\}. \tag{5}$$

Overhead Comparison. Based on the above overhead analysis, we set $m = 4$ in our scheme. In Table 2, we compare our scheme with one of the state-of-the-art access pattern preservation schemes for single-cloud-server systems [20].

Table 2. Overhead Comparison

	Comm./Comp.	Storage (server side)	Storage (user side)
Our Scheme ($m = 4$)	$O((\log n)^2 + \tau)$	$O(n \max\{\log n, \tau\})$	$O(\max\{\log n, \tau\})$
Scheme in [20]	$O(\log n \cdot \log\log n \cdot \tau)$	$O(n \cdot \tau)$	$O(\sqrt{n} \cdot \tau)$

It is interesting to see that, as long as the size of a data item (τ, in bits) is larger than $\log n$ where n is the total number of data items, which usually holds true in practical cloud storage applications, our scheme is more efficient. Specifically, our scheme (i) consumes similar storage space at the cloud server; (ii) usually incurs significantly less communication and computational overhead; and (iii) requires significantly less storage space at the cloud user, which facilitates the employment of our proposed scheme on thin user devices such as mobile phones. Note that the better efficiency performance of our scheme is achieved under a less stringent privacy requirement than [20]; instead of requiring strict privacy protection to the data access pattern, our scheme aims to protect the privacy of the data access pattern in the long run.

5 Performance Evaluation

5.1 Evaluation Setup

To evaluate the performance of the proposed scheme, we have collected two user access traces from two popular cloud service providers: *Youtube* [22] and *Baidu* [2]. As shown in Figs. 3(i) and (ii), both the Youtube user and the Baidu user have 256 files stored at the server. Different files have been accessed with different frequencies over time. Moreover, we have created an additional user who always requests the same file from the server, called the *SFA* (Single File Access) user, as shown in Fig. 3(iii). We use the SFA user to emulate an extreme access pattern. The total number of files stored at the server for the SFA user is also 256.

5.2 Preservation of Access Frequency Privacy

To study how well our proposed scheme preserves a cloud user's access frequency privacy, we propose to use *entropy* to measure the distribution of the user's access frequencies to different files. Specifically, let C_i denote the number of accesses to the file stored at storage location i. Then, the access frequency to location i is $F_i = \frac{C_i}{\sum_i C_i}$, and the entropy of access frequency is $H_F = -\sum_i F_i \log(F_i)$. For example, H_F of the Youtube and Baidu traces is around 7.6 and 6.5, respectively, which can be calculated by counting the number of accesses to each file in Figs. 3(i) and (ii). Clearly, for a given set of files stored at the server, the maximum entropy is achieved when all file locations

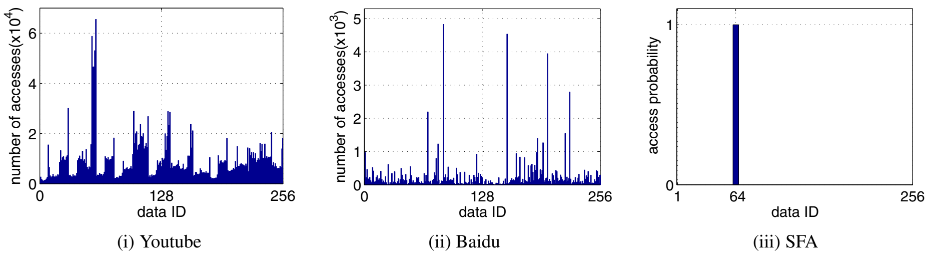


Fig. 3. Data access traces and distribution used in the performance evaluation

have been accessed with an equal probability. This means that, the maximum entropy for accessing 256 files is $H_F^{\max}(256) = -256 \times \frac{1}{256} \log(\frac{1}{256}) = 8$.

We evaluate how the entropy of access frequency changes as the number of access rounds increases. Fig. 4 plots the results (averaged over 100 simulation runs) for different access scenarios. It can be seen clearly from the figures that, with our scheme, the entropy of access frequency improves over the original trace, and converges gradually to the maximum entropy in all simulated scenarios. This confirms our analytical study in Section 4 and Theorem 2 that the access frequency distribution converges towards the uniform distribution in the long run.

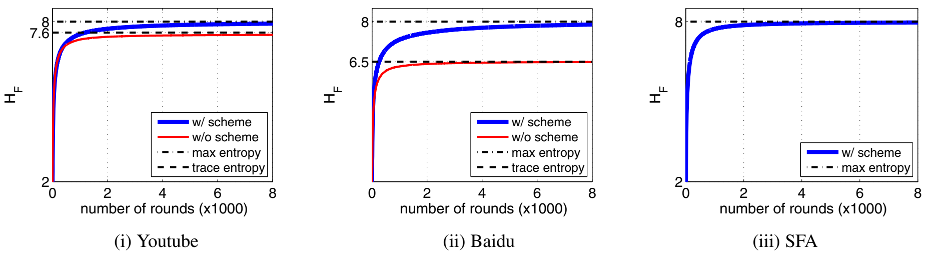


Fig. 4. The entropy of access frequency vs. the number of access rounds for a particular simulation run under different access scenarios. In (iii), because the SFA user always requests the same data item at each round, the entropy of access frequency without using our proposed scheme is always zero, which is not shown in the figure.

5.3 Preservation of Access Order Privacy

In this section, we demonstrate the effectiveness of the proposed scheme in preserving the access order privacy. We do so by evaluating the correlation between the output access sequences (i.e., the sequence of the requested data items' storage locations) for the same input access sequence (i.e., the sequence of actual data items requested by the user). Specifically, in each simulation run, we simulate the access procedure using the same input access sequence twice and calculate the *correlation coefficient* (denoted as Φ) between the two output sequences. A smaller Φ indicates that the two output sequences are less correlated, and thus the access order privacy is better preserved. Note that, using our scheme, the server observes accesses to three storage locations at each round. Therefore, it won't be able to get the exact sequence of the requested data items' storage locations, which also helps to preserve the access order privacy.

Figs. 5(i) plot the Φ values (averaged over 100 simulation runs) as the number of access rounds increases for different access scenarios. We can see that Φ decreases as the number of access rounds increases, thus the correlation between the output sequences becomes looser. Notice that Φ never reaches zero (i.e., perfect access order privacy) in the simulation, which is due to the randomness and finite length of the output sequence. As a result, Φ remains at small values (e.g., < 0.1) after a number of accesses.

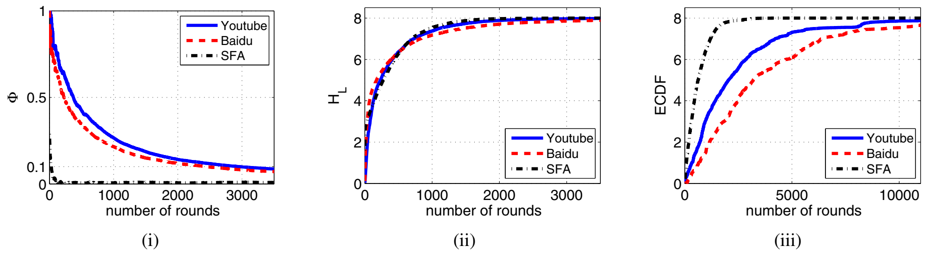


Fig. 5. (i) Average correlation coefficient (Φ) between output sequences for the same input sequences with our proposed scheme. (ii) Average entropy of location distribution vs. the number of access rounds for the most frequently requested data item. (iii) Average entropy of location distribution vs. the number of access rounds for the least frequently requested data item.

5.4 Preservation of Data Item's Location Privacy

As discussed in Section 4.1, when the user employs our proposed scheme, the cloud server loses track of a certain data item gradually over time. In other words, from the server's perspective, the uncertainty of a data item's storage storage location increases gradually over time. Similar to the evaluation of access frequency privacy, we also use *entropy* to measure the uncertainty of a particular data item's storage location from the server's perspective. It is defined as $H_L = -\sum_i p_i \log(p_i)$, where p_i is the probability that the data item is at storage location i from the server's perspective. We evaluate how the entropy of the data item's location distribution grows as the number of access rounds increases. For each access scenario, we collect the statistics of the most accessed data item and the least accessed data item, and results (averaged over 100 simulation runs) are plotted in Figs. 5(ii) and (iii), respectively. From the figures, we can see that a data item's location distribution entropy reaches the maximum regardless of their real access frequency. Note that, without our proposed scheme, a data item's location distribution entropy is zero because its location is fixed and known to the server.

6 Related Work

Although many schemes [19,18,23] have been proposed to protect data confidentiality and data integrity for the cloud computing paradigm, little effort has been made to protect users' access pattern privacy. Private Information Retrieval (PIR) [5, 15, 11], Oblivious RAM [9,20] and Steganographic File Systems (SFS) [24,16,6]are the works most related to our solution.

Private Information Retrieval: PIR schemes aim to allow clients to retrieve information from a database while maintaining the privacy of the queries to the database. Fully implementing the PIR notion is, however, expensive. As shown by Sion *et al.* [15], deployment of any single-server PIR protocol is not necessarily more efficient than a simple transfer of the entire database due to computational costs. On the other hand, PIR schemes typically do not address data confidentiality, which makes PIR schemes unsuitable to be applied in the un-trusted cloud environments.

Oblivious RAM: In order to prevent the users' access pattern from being revealed, *Oblivious RAM (ORAM)* [20, 8] has been proposed. In a latest version of ORAM, Williams *et al.* [20] proposed to user encrypted Bloom Filter [3] to reshuffle and scramble data in the database. In Section 4.2, we have shown that our scheme is much more efficient in terms communication, computational and storage overheads in practical cloud storage applications under a less stringent privacy requirement.

Steganographic File Systems: Research efforts on steganographic file systems [24, 16, 6] are also related to our proposed design. The major differences lie in that, the research on SFS targets at protecting the information about existence and/or locations of sensitive files through hiding both short-term and long-term access patterns, while our proposal mainly targets at protecting long-term access pattern at low cost.

Recently, there is a concurrent effort [17] that addresses a similar problem as the one in our work. Their solution and ours share similar high-level ideas such as usage of dummies, hierarchical storage structure and file reshuffling. However, there are several key differences between the two solutions. For example, our solution yields provable security and overhead performances and does not require user-side LRU cache or an empirical statistical access model.

7 Conclusions and Future Work

In this paper, we present a lightweight solution to the preservation of a cloud users' data access pattern privacy in un-trusted clouds. Rigorous proofs have been provided to show that the proposed scheme can provide full protection to data access pattern privacy in the long run. Extensive evaluations have also been conducted to show that the scheme can protect the data access pattern privacy effectively after a reasonable number of accesses have been made. In the future work, we plan to enhance the scheme such that it can support private and efficient data updates, including data changes, data insertions and data deletions.

Acknowledgments. We would like to thank the shepherd and the reviewers for their valuable suggestions for strengthening the paper. The research reported in this paper was supported in part by the Information Infrastructure Institute (iCube) of Iowa State University, the Security and Software Engineering Research Center (S^{2}ERC), the National Science Foundation under Grants CNS 0831874 and CNS 0716744, and the office of Naval Research under Grant N000140910748.

References

1. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R.H., Konwinski, A., Lee, G., Patterson, D.A., Rabkin, A., Stoica, I., Zaharia, M.: Above the clouds: A berkeley view of cloud computing. Tech. Rep. UCB-EECS (2009)
2. Baidu, `http://passport.baidu.com/?business&aid=6&un=chenfoxlord#7`
3. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM 13 (1970)
4. Chor, B., Gilboa, N.: Computationally private information retrieval. In: Proc. STOC 1997 (1997)
5. Chor, B., Goldreich, O., Kushilevitz, E., Sudan, M.: Private information retrieval. In: Proc. FOCS 1998 (1998)
6. Diaz, C., Troncoso, C., Preneel, B.: A framework for the analysis of mix-based steganographic file systems. In: Jajodia, S., Lopez, J. (eds.) ESORICS 2008. LNCS, vol. 5283, pp. 428–445. Springer, Heidelberg (2008)
7. Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., Boneh, D.: Terra: A virtual machine-based platform for trusted computing. In: Proc. SOSP 2003 (2003)
8. Goldreich, O.: Towards a theory of software protection and simulation by oblivious rams. In: Proc. STOC 1987 (1987)
9. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious ram. In: JACM 1996 (1996)
10. Itkis, G.: Personal communication, via oded goldreich (1996)
11. Kushilevitz, E., Ostrovsky, R.: Replication is not needed: Single database, computationally-private information retrieval. In: Proc. IEEE Symposium on Foundations of Computer Science (1997)
12. Mell, P., Grance, T.: Draft: Nist working definition of cloud computing (2010)
13. Menezes, A.J., van Oorschot, P.C., Vanstone, S.A.: Handbook of Applied Cryptography. CRC Press, Boca Raton (1996)
14. Ostrovsky, R., Shoup, V.: Private information storage. In: Proc. STOC 1997 (1997)
15. Sion, R., Carbunar, B.: On the computational practicality of private information retrieval. In: Proc. NDSS 2007 (2007)
16. Troncoso, C., Diaz, C., Dunkelman, O., Preneel, B.: Traffic analysis attacks on a continuously-observable steganographic file system. In: Furon, T., Cayre, F., Doërr, G., Bas, P. (eds.) IH 2007. LNCS, vol. 4567, pp. 220–236. Springer, Heidelberg (2008)
17. De Capitani di Vimercati, S., Foresti, S., Paraboschi, S., Pelosi, G., Samarati, P.: Efficient and private access to outsourced data. In: Proc. ICDCS 2011 (2011)
18. Wang, C., Wang, Q., Ren, K., Lou, W.: Ensuring data storage security in cloud computing. In: Proc. IWQoS 2009 (2009)
19. Wang, C., Wang, Q., Ren, K., Lou, W.: Secure ranked keyword search over encrypted cloud data. In: Proc. ICDCS 2010 (2010)
20. Williams, P., Sion, R., Carbunar, B.: Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In: Proc. CCS 2008 (2008)
21. Yang, K., Zhang, J., Zhang, W., Qiao, D.: A light-weight solution to preservation of access pattern privacy in un-trusted clouds. Technical Report (2011), `http://www.public.iastate.edu/~yangka/PatternFull.pdf`
22. Youtube, `http://www.youtube.com/user/supercwm`
23. Yu, S., Wang, C., Ren, K., Lou, W.: Achieving secure, scalable, and fine-grained access control in cloud computing. In: Proc. INFOCOM 2010 (2010)
24. Zhou, X., Pang, H., Tan, K.L.: Hiding data accesses in steganographic file system. In: Proc. ICDE 2004 (2004)

Appendix 1

What follows is the algorithm for the proposed access procedure of a cloud user. The details of the algorithm are described and explained in Section 3.2.

Algorithm 1. Proposed Access Procedure of a Cloud User

```
Step 1: Selection of Data Items (of IDs $\mathcal{Q}^0_{\text{cur}}.D_R$ and $\mathcal{Q}^0_{\text{cur}}.D_S$) to Access
1: $\mathcal{Q}^0_{\text{cur}}.D_R \leftarrow$ UserRequest(); $k \leftarrow$ UserKey();    // input user's desired data and secret key
2: Download&Decrypt$_k$($\mathcal{Q}^0_{\text{pre}}$, Hist[0]);
                                        // get access history of data items from location Hist[0] & decrypt it
3: if $\mathcal{Q}^0_{\text{cur}}.D_R \in \{\mathcal{Q}^0_{\text{pre}}.D_R, \mathcal{Q}^0_{\text{pre}}.D_S\}$ then
4:     $\mathcal{Q}^0_{\text{cur}}.D_S \leftarrow$ RandomSelectOne($\mathcal{D} \setminus \{\mathcal{Q}^0_{\text{cur}}.D_R\}$);
5: else
6:     $\mathcal{Q}^0_{\text{cur}}.D_S \leftarrow$ RandomSelectOne($\{\mathcal{Q}^0_{\text{pre}}.D_R, \mathcal{Q}^0_{\text{pre}}.D_S\}$);
7: end if
Step 2: Query for Index Files and Data Items
1: Download&Decrypt$_k$($I^T_1$, 0);        // download top-level index file from location 0 & decrypt it
2: $\mathcal{Q}^T_{\text{cur}}.D_R \leftarrow 1$; $\mathcal{Q}^T_{\text{cur}}.D_S \leftarrow 1$; $\mathcal{Q}^T_{\text{cur}}.D_N \leftarrow 1$;
3: for ($t \leftarrow (T-1)$; $t \geqslant 0$; $t--$) do
4:     // Step 2.1: Selection of Level-t Index Files and $\mathcal{Q}^0_{cur}.L_N$
5:     if $t > 0$ then
6:         Download&Decrypt$_k$($\mathcal{Q}^t_{\text{pre}}$, Hist[t]);        // get access history of level-t index files
7:         $\mathcal{Q}^t_{\text{cur}}.D_R \leftarrow f(\mathcal{Q}^0_{\text{cur}}.D_R, t)$; $\mathcal{Q}^t_{\text{cur}}.D_S \leftarrow f(\mathcal{Q}^0_{\text{cur}}.D_S, t)$;
                                // find out files storing level-t indices of data items $\mathcal{Q}^0_{cur}.D_R$ and $\mathcal{Q}^0_{cur}.D_s$
8:         if $\mathcal{Q}^t_{\text{cur}}.D_R = \mathcal{Q}^t_{\text{cur}}.D_S$ then
9:             $\mathcal{Q}^t_{\text{cur}}.D_S \leftarrow$ RandomSelectOne($\xi(\mathcal{Q}^{t+1}_{\text{cur}}.D_R, t+1) \cup \xi(\mathcal{Q}^{t+1}_{\text{cur}}.D_S, t+1) \setminus \{\mathcal{Q}^t_{\text{cur}}.D_R\}$);
10:        end if
11:    end if
12:    if $\{\mathcal{Q}^t_{\text{cur}}.D_R, \mathcal{Q}^t_{\text{cur}}.D_S\} \subseteq \mathcal{Q}^t_{\text{pre}}$ then
13:        $\mathcal{Q}^t_{\text{cur}}.L_N \leftarrow$ RandomSelectOne($\mathcal{L}^t \setminus \{\mathcal{Q}^t_{\text{pre}}.L_R, \mathcal{Q}^t_{\text{pre}}.L_S, \mathcal{Q}^t_{\text{pre}}.L_N\}$);
14:    else
15:        if $\mathcal{Q}^t_{\text{cur}}.D_R \in \mathcal{Q}^t_{\text{pre}}$ then
16:            $\mathcal{Q}^t_{\text{cur}}.L_N \leftarrow$ RandomSelectOne($\{\mathcal{Q}^t_{\text{pre}}.L_R, \mathcal{Q}^t_{\text{pre}}.L_S, \mathcal{Q}^t_{\text{pre}}.L_N\} \setminus \{\mathcal{Q}^t_{\text{cur}}.L_R\}$);
17:        else
18:            $\mathcal{Q}^t_{\text{cur}}.L_N \leftarrow$ RandomSelectOne($\{\mathcal{Q}^t_{\text{pre}}.L_R, \mathcal{Q}^t_{\text{pre}}.L_S, \mathcal{Q}^t_{\text{pre}}.L_N\} \setminus \{\mathcal{Q}^t_{\text{cur}}.L_S\}$);
19:        end if
20:    end if
21:    Download&Decrypt$_k$($\mathcal{Q}^t_{\text{cur}}.\{D_R, D_S, D_N\}$, $\mathcal{Q}^t_{\text{cur}}.\{L_R, L_S, L_N\}$);
       /* download files of IDs $\mathcal{Q}^t_{cur}.\{D_R, D_S, D_N\}$ from locations specified by $\mathcal{Q}^t_{cur}.\{L_R, L_S, L_N\}$
                                        respectively but in an arbitrary order & decrypt them */
       // Step 2.2: Random Reshuffling
22:    if RandomSelectOne($\{0, 1\}$) $= 1$ then
23:        Swap($\mathcal{Q}^t_{\text{cur}}.D_R$, $\mathcal{Q}^t_{\text{cur}}.D_S$);
24:        Update $\mathcal{Q}^t_{\text{cur}}.\{L_R, L_S\}$ in level-$(t+1)$ index files of IDs $\mathcal{Q}^{t+1}_{\text{cur}}.\{D_R, D_S\}$;
25:    end if
       // Step 2.3: Reencryption/Uploading of Level-$(t+1)$ Files and Level-t Access History
26:    Reencrypt$_k$&Upload($\mathcal{Q}^{t+1}_{\text{cur}}.\{D_R, D_S, D_N\}$, $\mathcal{Q}^{t+1}_{\text{cur}}.\{L_R, L_S, L_N\}$);
       /* reencrypt & upload files of IDs $\mathcal{Q}^{t+1}_{cur}.\{D_R, D_S, D_N\}$ to locations specified by
                                $\mathcal{Q}^{t+1}_{cur}.\{L_R, L_S, L_N\}$ respectively but in an arbitrary order */
27:    Reencrypt$_k$&Upload($\mathcal{Q}^t_{\text{cur}}$, Hist[t]);
28: end for
Step 3: Reencryption and Uploading of Accessed Data Items
1: Reencrypt$_k$&Upload($\mathcal{Q}^0_{\text{cur}}.\{D_R, D_S, D_N\}$, $\mathcal{Q}^0_{\text{cur}}.\{L_R, L_S, L_N\}$);
```

Appendix 2

We now explain why our proposed scheme requires the user to download two dummy data items together with the intended data item in each access.

Suppose the scheme only downloads one dummy data item (whose ID is denoted as $\mathcal{Q}_{\text{cur}}^0.D_R$) together with the intended data item (whose ID is denoted as $\mathcal{Q}_{\text{cur}}^0.D_S$). We let the dummy data item be selected to make sure that the user's request at each round has the same format: *the user always requests two data locations, out of which one and only one of them is from the ones accessed in the previous round.* The rules for selecting the dummy data item are: (i) if the intended data item has been accessed in the previous round, the dummy is selected uniformly at random from the data items that have not been accessed in the previous round; (ii) otherwise, the dummy is selected from the two accessed data items with equal probability.

Similarly, we would like to have the same format at each round of index file access: *at each index level, the user always requests two index file locations, out of which one and only one of them is from the ones accessed in the previous round.* Unfortunately, this may not always be possible with a single dummy index file. An example is given in Fig. 6 to illustrate the problem.

Level-2 Index File

I_1^2

Index	I_1^1	I_2^1	I_3^1	I_4^1
Loc	3	4	2	1

Level-1 Index File

I_1^1

Data	d_1	d_2	d_3	d_4
Loc	4	12	5	1

I_2^1

d_5	d_6	d_7	d_8
9	16	8	13

...

Fig. 6. In this example, there are $n = 16$ data items and $T = 2$ levels of index files stored at the cloud server. The contents of index files I_1^2, I_1^1 and I_2^1 are shown in the figure.

In Fig. 6, suppose in the first round, the user needs data item d_1 and data item d_2 is randomly selected to be the dummy. Since d_1 and d_2 share the same level-1 index file I_1^1, the user needs to randomly select a new dummy index file. Suppose the user selects I_2^1 as the dummy index file. Then in the second round, suppose the user needs data item d_5. According to the selection rules, the user randomly selects a dummy from d_1 and d_2. However, no matter whether d_1 or d_2 is selected as the dummy, the user needs to retrieve I_1^1 and I_2^1 in order to get the storage locations of d_5 and the selected dummy. Note that both I_1^1 and I_2^1 have been accessed in the previous round; this violates the desired access format.

A quick remedy to the problem may be as following: when selecting the dummy, the user randomly selects the dummy from data items that do not share the same index files (except for the top level) with the intended data item. It is easy to see that this selection rule can avoid the afore-described problem. However, such remedial action may leak information about user's access pattern in some situations. For example, in Fig. 6, if the user accesses d_1 consecutively, data locations where d_2, d_3 and d_4 are stored will never be accessed, which may leak information about the data item of user's interest.

There may exist more sophisticated rules that can preserve the user's long-run access pattern using a single dummy, which we are not aware of at the moment. So instead, in this work, we adopt an efficient two-dummy solution to guarantee that user's access at each around has the same format.

Appendix 3

In this section, we sketch the proof for Theorem 2. Please refer to [21] for the detailed proof. In the proposed scheme, at each round of access, the user accesses three data items, where two of them (D_R and D_S) randomly swap their locations after the access and the other D_N does not. Therefore, the selection of D_N does not affect the location distribution of the data items. As a result, in the proof, we only need to consider the behavior of D_R and D_S.

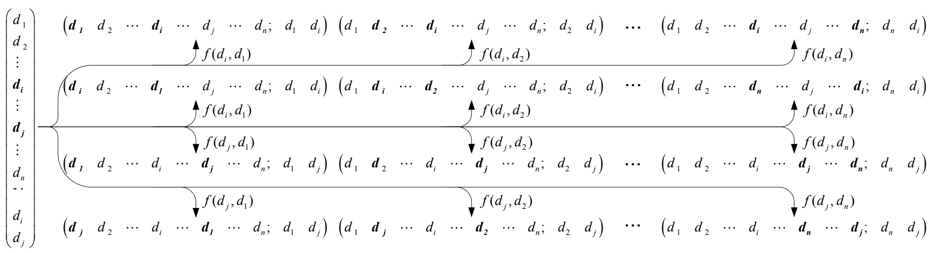


Fig. 7. One-step transition from an arbitrary state $(d_1\ d_2\ \cdots\ d_i\ \cdots\ d_j\ \cdots\ d_n;\ d_i\ d_j)$ to other reachable states in MC-1. $f(\cdot,\cdot)$ is the transition probability function.

Let d_i $(i = 1, \cdots, n)$ denote the data items and P_i denote the probability with which d_i is actually requested by the user in each round of access. We model the data access process with a homogeneous Markov chain denoted as MC-1, as shown in Fig. 7. Each state of MC-1 is $(\sigma;\ d_i\ d_j)$. Here, σ is a permutation of $(d_1, \cdots, d_n)$, which stands for one distribution of the n data items to n storage locations. i and j are two distinct numbers from $\{1,\ \cdots,\ n\}$, and d_i and d_j is D_R and D_S respectively. Hence, there is a total of $n!\binom{n}{2}$ distinct states in MC-1.

In the proof, we show that MC-1 converges to a steady state. In the steady state, all permutations of data items are equally likely to happen. Consequently, every data item is uniformly randomly distributed to all storage locations in the steady state. This implies that each storage location will be accessed uniformly at random in the long run.