Efficient and Accountable Oblivious Cloud Storage with Three Servers

Qiumao Ma, Wensheng Zhang
Iowa State University, Ames, IA, USA
{qmma,wzhang}@iastate.edu

***Abstract*—As the adoption of cloud storage service has been pervasive, more and more attentions have been paid to the related security and privacy risks, among which, data access pattern privacy is an important aspect. Lots of solutions have been proposed, but most are infeasible due to high overheads in communication and storage. In this paper, we propose a new solution to address the limitations by leveraging the moderate storage capacity in the increasingly popular cloud storage gateways and the existence of multiple competing and independent cloud storage servers. Extensive analysis and evaluation have shown that, our proposed system can simultaneously attain the features of provable protection of data access pattern, low data query delay, low server storage overhead, low communication costs, and accountability.**

I. Introduction

For the attractive features of self-provisioning, pay-as-you-go, economic efficiency and high availability, cloud storage model has been hailed by organizational and individual clients. In the past years, the model has kept evolving, and the hybrid variant based on cloud storage gateway has increasing popularity [1]. By the hybrid model, a client runs an on-premise gateway with moderate storage resources. For scalability and cost-efficiency, the gateway outsources majority data to one or multiple off-premise cloud storage servers; meanwhile, it stores the meta-data and the frequently or recently-accessed data, to manage data as well as reduce the frequency and latency in accessing data directly from the remote server.

In spite of the pervasive usage of cloud storage, the clients have also raised various privacy concerns, among which the data access pattern privacy is gaining more awareness. Data encryption has been common for data privacy protection, but it cannot protect data access patterns for cloud storage clients. A curious owner or employee of a cloud storage service, or an intruder invading the storage server, can observe a client's data access pattern. Based on the observed pattern and the client's activities that could be obtained through some side channels, the attacker could develop a model relating them. Later on, the attacker may use the model and newly observed access patterns to infer or predict the client's activities. Therefore, exposed data access pattern can potentially reveal some private information about cloud storage clients. Especially, military, homeland security and public safety agencies should protect these private information from the enemies; businesses should protect the information from their rivals and competitors.

In the past decades, the problem of protecting data access pattern has attracted a lot of interest from the researchers, but the advancements are mainly with the theoretical aspect. It is still an un-attained goal to build an oblivious cloud storage system that can protect the data access pattern but meanwhile can deliver a performance similar to an existing non-oblivious system. Specifically, it is desirable to have a storage system with the following features: (i) provable security in access pattern protection; (ii) low data query delay experienced by the client; (iii) low communication costs incurred; and (iv) low storage costs incurred.

Related Works. The oblivious RAM (ORAM) model [2], originally proposed for software protection, is a well-known provable approach to protect the client's access pattern. Following this model, a large variety of schemes [2]–[13] have been developed, with the goal to implement the ORAM model more efficiently and practically. We here briefly review a few that are most related or newest schemes.

Communication efficiency has been one major optimization objective in the research. As one of the most communication-efficient ORAM schemes, Partition ORAM [3] organizes the server storage as $\sqrt{N}$ partitions (N is the number of exported data blocks), and each partition works as an ORAM module. The client storage is utilized to contain a location map for blocks, a buffer for storing and shuffling data blocks of an ORAM partition, and $\sqrt{N}$ stash slots. Based on this storage arrangement, together with optimizations in query and shuffling algorithm, the scheme incurs a communication cost of about $1.25 \log N$ blocks per query. Burst ORAM [5] improves upon Partition ORAM [3] by introducing a new XOR technique to reduce the online bandwidth cost to a constant, and priority scheduling algorithms to deal with request bursts. Ring ORAM [8] further improves the communication efficiency by combining the best qualities from the Partition ORAM [3] and Path ORAM [4], an efficient tree-based ORAM. CURIOUS [14] presents a partition-based ORAM framework and each partition is a small ORAM and can be organized as Path ORAM [4]. It doubles the overall communication cost of Partition ORAM, but reduces the response time. Although the above efforts have advanced the communication efficiency, especially in the regard of client-server communication latency, the client-server communication cost is still higher than a non-oblivious storage system by a factor of $O(\log N)$. Besides, the server storage costs also remain high; to export N data blocks, the server needs to store $3N$ or even more data blocks, most of which are dummy.

To address the problem with client-server communication, Hoang et al. [11] propose S^3ORAM based on the utilization of multiple (at least three) non-colluding servers. This

978-1-5386-7117-7/19/$31.00 ©2019 IEEE

scheme incurs $O(1)$ bandwidth consumption for client-server communication, which is similar to the cost incurred by a non-oblivious storage system; it still requires $O(\log N)$ bandwidth consumption for communication between the servers, but the inter-server communication occurs behind the scene and does not consume client-server bandwidth. However, there are several limitations with this scheme. First, the scheme requires multiple servers each storing a copy of the outsourced data blocks, which significantly increases the server storage cost to $12N$ blocks for every N real data blocks exported. Second, the scheme assumes the servers to be semi-honest, which may not be realistic in practice; a dishonest server could deviate from the designated protocol, and if not detected immediately, could lead to big overhead to recover the system.

Our Contributions. To address the afore-discussed limitations of the existing works, we propose a new oblivious cloud storage system, which is also an implementation of the ORAM model. Similar to S^{3}ORAM, our proposed system also recruits three non-colluding cloud servers to act as the oblivious storage. But, through leveraging the moderate storage space at the client and optimizing the storage arrangement at the servers, we significantly reduce the server storage overhead by storing only around $0.3N$ extra dummy blocks compared to $11N$ dummy blocks required by S^{3}ORAM. Moreover, we design and employ several lightweight accountability mechanisms for the servers, such that each server can detect the misbehavior of other servers that interact with it. Compared to the state of the art, our proposed system can simultaneously attain the following features:

- provable protection of clients' access pattern privacy;
- low server storage overhead, which is around $0.3N$ blocks for every N real data blocks exported;
- low data query delay, only slightly longer than a communication round trip time between the client and server;
- accountability with multiple servers, which removes the less-realistic semi-honest assumption in a multi-server oblivious storage system;
- lower communication costs than S^{3}ORAM, the most related state-of-the-art scheme.

The above features are achieved by our novel designs of storage arrangement, data query algorithm and eviction algorithm. Note that, our system has made full use of the available moderate level of client-side storage, but the required storage capacity is still only as small as around 0.1% of the cloud server's storage capacity.

Organization. In the rest of the paper, we first define the system model and security in Section II, which is followed by the detailed design in terms of storage organization, query algorithm, eviction algorithm and accountability enhancements in Section III-VI respectively. Section VII presents the security analysis. Performance evaluation and comparisons are reported in Section VIII. Finally, section IV concludes the paper.

II. Preliminaries

We consider a distributed system that consists of a client and three cloud servers. The client has an on-premise cloud storage gateway with a moderate storage capacity, though much smaller than the capacity of the cloud storage servers. The cloud servers are assumed to be non-colluding. However, different from the often-taken assumption that the servers are semi-honest, we assume the servers could be malicious and some accountability mechanisms will be deployed to each server to detect the misbehavior of other servers. The client is assumed to be honest; note that, this assumption could be removed by, for example, requiring the client to electronically sign each message and data block that it sends.

Assume the client outsources N data blocks each with the same size of z bits to the cloud storage server, and then needs to access the outsourced data every now and then.

Each *data access* intended by the client, which should be kept private, is of two types: read a data block D of unique ID i from the storage, denoted as $(read, i, D)$; write a data block D of unique ID i to the storage, denoted as $(write, i, D)$. To hide a private data access, the client and servers need to access multiple locations of the server-side storage and exchange some messages with each other. Each *location access* or *message exchange*, which can be observed by the servers, is one of the following types: retrieve (i.e., read) a data block D from location l at the storage, denoted as $(read, l, D)$; upload (i.e., write) a data block D to location l at the storage, denoted as $(write, l, D)$; send a message from one party to another (note: a party could be the client or a server), denoted as $(send, s, d)$ where s and d are the source and destinations.

Extending the security definition of ORAM in prior works [2]–[4], we define the security of our proposed oblivious storage system as follows.

Definition Let λ be a security parameter, and $\vec{x} = \langle (op_1, i_1, D_1), (op_2, i_2, D_2), \cdots \rangle$ denote a private sequence of the client's data accesses, where each op is either a $read$ or $write$. Let $A(\vec{x}) = \langle (op'_1, p_{1,1}, p_{1,2}), (op'_2, p_{2,1}, p_{2,2}), \cdots \rangle$ denote the sequence of the location accesses or message exchanges (observed by the server) in order to accomplish the data access sequence $\vec{x}$. An oblivious storage system is secure if (i) for any two equal-length private sequences $\vec{x}$ and $\vec{y}$ of data accesses, their corresponding location access and message exchange sequences $A(\vec{x})$ and $A(\vec{y})$ are computationally indistinguishable; and (ii) the system fails to operate with a probability of $O(2^{-\lambda})$.

III. System Architecture and Initialization

As shown in Fig. 1, our proposed system is composed of one client and three servers, denoted as $\mathcal{S}_0$, $\mathcal{S}_1$ and $\mathcal{S}_2$. Only one server (i.e., $\mathcal{S}_0$) needs to permanently store the data blocks exported by the client. The other two servers only store some meta-data and temporarily buffer some data blocks, to facilitate data query and eviction processes as well as to maintain system accountability, which are detailed in Sections IV, V and VI. The client also stores meta-data and a small subset of data blocks. In the following, we elaborate the storage organization at server $\mathcal{S}_0$ and the client.

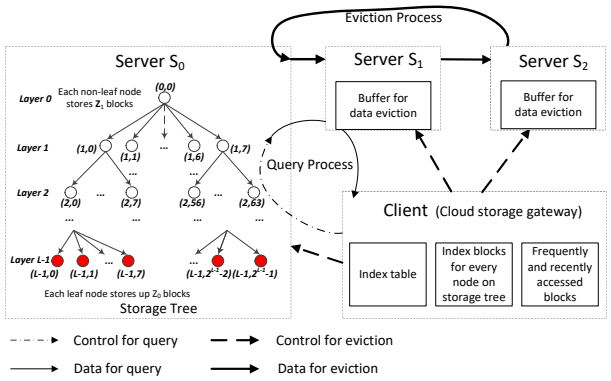


Figure 1. Proposed System Architecture.

A. Storage Organization at Server $\mathcal{S}_0$

Let positive integers m (a power of 2) and q, and positives fractions α and β be system parameters. Let

$$\xi(m,q) = \max(\frac{(m-1)\cdot q}{2}, 2q). \quad (1)$$

The data blocks are stored into an m-ary *storage tree*, in which each non-leaf node can have up to m child nodes. When constructing the storage tree, we make the tree to be balanced and the number of data blocks at each leaf node to vary between $(1+\beta)\cdot\xi(m,q)$ and $2(1+\beta)\cdot\xi(m,q)$, for certain security purposes explained later in Section VII.

Specifically, the tree is constructed as follows:

- Let $L' = \lfloor \log_m \frac{N}{\xi(m,q)} \rfloor$ and $Z' = \frac{N}{m^{L'}}$. Obviously, $Z' \geq \xi(m,q)$.
- If $Z' \leq 2\xi(m,q)$, the storage tree is organized as a complete m-ary tree with height $L = L'+1$ where the capacity of each leaf node is $Z_0 = \lceil (1+\beta)\cdot Z' \rceil$ blocks.
- Otherwise (i.e., $Z' > 2\xi(m,q)$), the storage tree is organized as a tree of height $L = L'+2$, and the root has $\lfloor \frac{Z'}{\xi(m,q)} \rfloor$ child nodes while each child node is a root of a complete m-ary tree with $L'+1$ layers and $Z_0 = \lceil (1+\beta)\cdot \frac{Z'}{\lfloor \frac{Z'}{\xi(m,q)} \rfloor} \rceil$ blocks at each leaf node.
- Each non-leaf node has a capacity of $Z_1 = \lceil (1+\alpha)\cdot \xi(m,q) \rceil$ blocks.

Each node $\mathcal{N}_{l,i}$ is identified by a unique tuple (l,i), where $l \in \{0,\cdots,L-1\}$ is the ID of the layer that the node resides (note: the root node is at layer 0 while the leaf nodes are at layer $L-1$), and $i \geq 0$ is the ID of the node on layer l that equals to the offset of the node on layer l (from 0 at the leftmost towards right). Note that Fig. 1 shows a storage tree when $m = 8$.

B. Storage Organization at the Client

The client maintains an *index table for all of the N real data blocks* and an *index block for each node* on the storage tree. The index table has N entries and each entry $i \in \{0,\cdots,N-1\}$ has the following fields:

- *path ID* of block i, i.e., the ID of the leaf node on the path that block i is assigned to;
- *secret key* k_i which, as detailed in Section III-C, randomly selected by the client to *encrypt* the block based on XOR operation;
- three *message authentication codes (MACs)* of the block, of which the computation and usage are explained in detail in Section VI.

Note that, following most of the tree-based ORAM constructions [3], [4], [6], [7], [9], our proposed scheme also enforces the policy that, a block is assigned to a path and the block must be stored on the path.

For each node on the tree, the index block has one entry $(id,\ ah)$ for each block it stores, where id is the ID of the block, no matter whether the block is real or dummy, and $ah \in \{0,1,2\}$ indicates the access history of the block since the system initialization or the most recent data eviction process involving the node, whichever is more recent: (i) $ah = 0$ if the block has not been accessed; (ii) $ah = 1$ if the block has been accessed as a query target; and (iii) $ah = 2$ if the block has been accessed but never as a query target.

In addition, the client maintains a local buffer that stores the most recently accessed data blocks. The capacity of the buffer is at least q blocks.

C. System Initialization

The client picks a pseudo random number generator $PRG_0(k)$, which takes a secret seed k of λ bits and outputs a pseudo-random sequence of 3λ bits. The client also picks and shares with the servers another pseudo random number generator function, denoted as $PRG_1(k)$, which takes a secret seed k and outputs a pseudo-random sequence of bytes with the same length as a data block.

Before each real block (denoted as $\vec{D}_i$ which is a sequence of bits) of ID i is exported to server $\mathcal{S}_0$, the client encrypts the block as follows.

1) It randomly picks a secret seed k_i, and computes $PRG_0(k_i)$ whose output is denoted as $k_{i,0}|k_{i,1}|k_{i,2}$ where each $k_{i,j}$ has λ bits and $|$ represents concatenation.
2) It computes $PRG_1(k_{i,0})$, $PRG_1(k_{i,1})$ and $PRG_1(k_{i,2})$ to generate three pseudo-random sequences of bytes, denoted as $\vec{R}_{i,0}$, $\vec{R}_{i,1}$ and $\vec{R}_{i,2}$, each of the same length as a data block.
3) It performs bit-wise XOR operations on *each group of four bits with the same offset* of the four bit-sequences $\vec{D}_i$, $\vec{R}_{i,0}$, $\vec{R}_{i,1}$ and $\vec{R}_{i,2}$, to *encrypt* $\vec{D}_i$ to

$$\vec{D}'_i = \vec{D}_i \oplus \vec{R}_{i,0} \oplus \vec{R}_{i,1} \oplus \vec{R}_{i,2}. \quad (2)$$

IV. Data Query Algorithm

Assume the client wishes to query data block $\vec{D}_t$, where t denotes the block ID, and the block is not in its local buffer. It looks up its index table to find path p_t that contains $\vec{D}_t$, and looks up the index blocks of the path to locate the node containing $\vec{D}_t$. Then, it launches a query process in two phases: selecting some data blocks to access from $\mathcal{S}_0$, based on the index table and index blocks that it stores, in order to hide the query target; interacting with the servers to retrieve query target $\vec{D}_t$.

A. Phase I: Selecting Data Blocks to Access

For each node $\mathcal{N}'_i$ on path p_t, where $i \in \{0, \cdots, L-1\}$ represents the layer ID of the node, let $\Delta_{i,0}$, $\Delta_{i,1}$ and $\Delta_{i,2}$ denote the block sets with ah being 0, 1 and 2, and $\delta_{i,0}$, $\delta_{i,1}$ and $\delta_{i,2}$ denote the sizes of these sets, respectively. The client selects data blocks from each node $\mathcal{N}'_i$ to download, according to the rules presented in Algorithm 1, with the dual goals of *hiding data access pattern* and *communication efficiency*.

Algorithm 1 Rules for Selecting Blocks from $\mathcal{N}'_i$ to Access (Output: Δ - a set of blocks selected to access)

1: $\Delta \leftarrow \emptyset$
2: **if** $\mathcal{N}'_i$ contains query target $\vec{D}_t$ **then**
3: add $\vec{D}_t$ to Δ
4: $\forall \vec{D} \in \Delta_{i,1}$, add $\vec{D}$ to Δ with probability $\frac{1}{\delta_{i,0}}$
5: **if** $\vec{D}_t$ belongs to $\Delta_{i,0}$ **then**
6: $\forall \vec{D} \in \Delta_{i,2}$, add $\vec{D}$ to Δ with probability $\frac{\delta_{i,2}}{\delta_{i,0}^2}$
7: **else** //i.e., $\vec{D}_t$ belongs to $\Delta_{i,2}$
8: randomly picks one $\vec{D}$ from $\Delta_{i,0}$; adds it to Δ
9: **else**
10: randomly picks one $\vec{D}$ from $\Delta_{i,0}$; adds it to Δ
11: $\forall \vec{D} \in \Delta_{i,1} \cup \Delta_{i,2}$, add $\vec{D}$ to Δ with probability $\frac{1}{\delta_{i,0}}$

First, the algorithm hides data access pattern by making each block in $\mathcal{N}'_i$ to be accessed with the same probability independent of where the query target resides, as stated in the following Lemma 1 with proof.

Lemma 1: During a query process with query path p_t, each block $\vec{D}$ in node $\mathcal{N}'_i$ on p_t is selected to access with the same probability of $\frac{1}{\delta_{i,0}}$, which is obviously independent of the location of the query target.

Proof: When the query target does not belong to $\mathcal{N}'_i$, every block $\vec{D}$ is accessed with probability $\frac{1}{\delta_{i,0}}$ based on lines 10-11 of Algorithm 1. Otherwise, each block $\vec{D}$ must be in $\Delta_{i,0}$, $\Delta_{i,1}$ or $\Delta_{i,2}$. So we consider the three cases respectively.

- *Case I:* $\vec{D} \in \Delta_{i,0}$. Further there are two subcases: $\Delta_{i,0}$ contains the query target or not.
  - *Subcase I-a:* $\Delta_{i,0}$ *contains query target* $\vec{D}_t$. Only $\vec{D}_t$ is accessed from $\Delta_{i,0}$. Further due to the random distribution of blocks in $\Delta_{i,0}$, every $\vec{D}$ has the same probability of $\frac{1}{\delta_{i,0}}$ to be accessed as query target.
  - *Subcase I-b:* $\Delta_{i,0}$ *contains query target* $\vec{D}_t$. Based on line 8 of the algorithm, every $\vec{D}$ has the probability of $\frac{1}{\delta_{i,0}}$ to be accessed.
- *Case II:* $\vec{D} \in \Delta_{i,1}$. Every $\vec{D}$ has the probability of $\frac{1}{\delta_{i,0}}$ to be accessed, based on line 4 of the algorithm.
- *Case III:* $\vec{D} \in \Delta_{i,2}$. Further there are two subcases:
  - *Subcase III-a:* $\Delta_{i,0}$ *contains query target* $\vec{D}_t$, which occurs with probability $\frac{\delta_{i,0}}{\delta_{i,0}+\delta_{i,2}}$ in case III. In this subcase, every $\vec{D}$ is accessed with probability $\frac{\delta_{i,2}}{\delta_{i,0}^2}$.
  - *Subcase III-b:* $\Delta_{i,0}$ *contains query target* $\vec{D}_t$, which occurs with probability $\frac{\delta_{i,2}}{\delta_{i,0}+\delta_{i,2}}$ in case III. In this subcase, $\vec{D}$ is accessed as the query target with probability $\frac{1}{\delta_{i,2}}$.

  To summarize, $\vec{D}$ is accessed with the following probability in Case III:

$$\frac{\delta_{i,0}}{\delta_{i,0}+\delta_{i,2}} \cdot \frac{\delta_{i,2}}{\delta_{i,0}^2} + \frac{\delta_{i,2}}{\delta_{i,0}+\delta_{i,2}} \cdot \frac{1}{\delta_{i,2}} = \frac{1}{\delta_{i,0}}.$$

Hence, the Lemma is proved. ■

Second, in terms of communication efficiency, the query algorithm requires only $1+\frac{\delta_{i,1}+\delta_{i,2}}{\delta_{i,0}}$ blocks accessed from each node $\mathcal{N}'_i$. Further, as we study later in Section VII, $\frac{\delta_{i,1}+\delta_{i,2}}{\delta_{i,0}} < 1$ with an overwhelming probability of $1-2^{-\lambda}$. That is, no more than 2 blocks are accessed from each node $\mathcal{N}'_i$ on the query path with a probability at least $1-2^{-\lambda}$.

B. Phase II: Retrieving Query Target

The client sends a request to $\mathcal{S}_0$, which contains:

- list $\vec{B} = \langle b_1, \cdots, b_x \rangle$ of x block indices where, in each $b_i = (n_i, o_i)$, n_i is the ID of a block on query path and o_i is the offset of a block selected to access in Phase I;
- random permutation vector $\vec{V} = \langle v_1, \cdots, v_x \rangle$ of integers $\{1, \cdots, x\}$, which directs $\mathcal{S}_0$ to put every block b_i to offset v_i after the permutation.

It also sends a request to $\mathcal{S}_1$, which only contains one number in $\{1, \cdots, x\}$.

In response to the client's request, $\mathcal{S}_0$ makes a copy of the blocks indicated by $\vec{B}$, permutes the blocks as directed by $\vec{V}$, and then forwards the resulting block sequence to $\mathcal{S}_1$.

Upon receiving the sequence, $\mathcal{S}_1$ retains only the query target block, whose offset on the sequence is the index contained in the client's request, and immediately returns the block to the client.

Having received the query target, the client updates its local meta-data to make the copy of the query target left on the storage tree as a dummy block. Then, it can start reading or writing to the query target locally.

V. Data Eviction Algorithm

After every q queries, the client has retained at its buffer q blocks that are the targets of the most recent q queries. We call these blocks the current *evicting blocks*. The client randomly re-assigns a path for each evicting block, sends all these blocks in an ordered list to server $\mathcal{S}_1$, and then launches a *data eviction process* to evict them into the storage tree at server $\mathcal{S}_0$. Note that, as in existing ORAM constructions such as [3], the eviction process can be carried out concurrently with data query processes through some de-amortization mechanism. Due to page limit, we skip the de-amortization detail and focus on the main idea.

Every eviction process involves only one root-to-leaf path, which we call *eviction path*, on the storage tree at server $\mathcal{S}_0$. The eviction path is selected in the reverse-lexicographic order, as illustrated by Fig. 2.

An eviction process runs iteratively, one iteration for each node on the eviction path from the root to the leaf. We

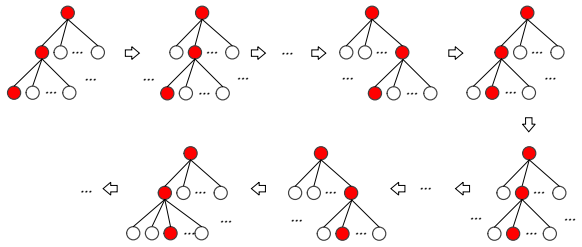


Figure 2. **Reverse-lexicographic Order**: Every eviction process involves one root-to-leaf *eviction path* selected in the *reverse-lexicographic order*.

introduce variable $\mathcal{N}'_e$ to denote the node currently involved in the eviction. Hence, $\mathcal{N}'_e$ is initialized to $\mathcal{N}_{0,0}$ (i.e., the root node). Also, when an eviction iteration begins, $\mathcal{S}_0$ has an ordered list (denoted as $\vec{\mathcal{L}}_0$) containing Z_0 or Z_1 blocks stored at node $\mathcal{N}'_e$, depending on whether $\mathcal{N}'_e$ is leaf or not; $\mathcal{S}_1$ has an ordered list (denoted as $\vec{\mathcal{L}}_1$) of q blocks; $\mathcal{S}_2$ has no data blocks. Then, the iteration, which involves the client and all the servers, runs as follows.

1) For each block $\vec{D}_i \in \vec{\mathcal{L}}_0 \cup \vec{\mathcal{L}}_1$, where i represents the ID of the block, the client randomly picks a new key k'_i and then generates a new set of keys $k'_{i,0}$, $k'_{i,1}$ and $k'_{i,2}$ where $PRG_0(k'_i) = k'_{i,0}|k'_{i,1}|k'_{i,2}$. Also, from the current version of key k_i recorded in the index table, the client derives the current set of keys $k_{i,0}$, $k_{i,1}$ and $k_{i,2}$ where $PRG_0(k_i) = k_{i,0}|k_{i,1}|k_{i,2}$.
2) The client randomly constructs a permutation vector π_0 for $|\vec{\mathcal{L}}_0|$ elements (i.e., a random permutation of numbers $0, \cdots, |\vec{\mathcal{L}}_0| - 1$) where $|\vec{\mathcal{L}}_0|$ denotes the length of $\vec{\mathcal{L}}_0$, and sends the vector to $\mathcal{S}_0$.
3) Upon receiving π_0, $\mathcal{S}_0$ permutes $\vec{\mathcal{L}}_0$ to $\vec{\mathcal{L}}'_0 = \pi_0(\vec{\mathcal{L}}_0)$, and sends $\vec{\mathcal{L}}'_0$ to $\mathcal{S}_1$.
4) Letting $\vec{\mathcal{L}}'_0|\vec{\mathcal{L}}_1 = \langle \vec{D}_{i_0}, \cdots, \vec{D}_{i_{x-1}} \rangle$ where $x = |\vec{\mathcal{L}}'_0| + |\vec{\mathcal{L}}_1|$, the client randomly constructs a permutation vector π_1 for x elements and the following ordered list (denoted as $\vec{\mathcal{R}}_1$):
$$\vec{\mathcal{R}}_1 = \langle (k_{i_0,0}, k'_{i_0,1}), \cdots, (k_{i_{x-1},0}, k'_{i_{x-1},1}) \rangle. \quad (3)$$
Then, the client sends π_1 and $\vec{\mathcal{R}}_1$ to $\mathcal{S}_1$.
5) Upon receiving $\vec{\mathcal{L}}'_0$ from $\mathcal{S}_0$ as well as π_1 and $\vec{\mathcal{R}}_1$ from the client, $\mathcal{S}_1$ first constructs $\vec{\mathcal{L}}'_1 = \vec{\mathcal{L}}'_0|\vec{\mathcal{L}}_1$, which we also denote as $\langle \vec{D}_{i_0}, \cdots, \vec{D}_{i_{x-1}} \rangle$. Next, it *re-encrypts* each block $\vec{D}_{i_j}$ (where $j = 0, \cdots, x-1$), based on key pair $(k_{i_j,0}, k'_{i_j,1})$ in $\vec{\mathcal{R}}_1$, through the following steps:
   - It computes pseudo-random blocks $\vec{R}_{i_j,0} = PRG_1(k_{i_j,0})$ and $\vec{R}'_{i_j,1} = PRG_1(k'_{i_j,1})$.
   - It updates $\vec{D}_{i_j}$ to $\vec{D}'_{i_j} = \vec{D}_{i_j} \oplus \vec{R}_{i_j,0} \oplus \vec{R}'_{i_j,1}$, where $\oplus$ is the bit-wise XOR between two blocks (i.e., bit sequences).

   Then, list $\langle \vec{D}'_{i_0}, \cdots, \vec{D}'_{i_{x-1}} \rangle$ is permuted according to π_1, and the resulting list (denoted as $\vec{\mathcal{L}}_2$) is sent to server $\mathcal{S}_2$.
6) Letting $\langle i'_0, \cdots, i'_{x-1} \rangle$ be the ordered list of IDs of the blocks in $\vec{\mathcal{L}}_2$, the client sends to $\mathcal{S}_2$ the following list of key pairs
$$\vec{\mathcal{R}}_2 = \langle (k_{i'_0,1}, k'_{i'_0,2}), \cdots, (k_{i'_{x-1},1}, k'_{i'_{x-1},2}) \rangle. \quad (4)$$
The client also constructs a permutation π_2 for x elements, and sends π_2 to $\mathcal{S}_2$.
7) Upon receiving π_2 and $\vec{\mathcal{R}}_2$ from the client, as well as $\vec{\mathcal{L}}_2 = \langle \vec{D}_{i'_0}, \cdots, \vec{D}_{i'_{x-1}} \rangle$ from $\mathcal{S}_1$, server $\mathcal{S}_2$ first re-encrypts each block in $\vec{\mathcal{L}}_2$ based on the key pairs in $\vec{\mathcal{R}}_2$, and then permutes the re-encrypted list according to π_2, as server $\mathcal{S}_1$ does. The resulting list (denoted as $\vec{\mathcal{L}}'_2$ is sent to server $\mathcal{S}_0$.
8) Letting $\langle i''_0, \cdots, i''_{x-1} \rangle$ be the ordered list of IDs of the blocks in $\vec{\mathcal{L}}'_2$, the client sends to $\mathcal{S}_0$ the following list of key pairs
$$\vec{\mathcal{R}}_0 = \langle (k_{i''_0,2}, k'_{i''_0,0}), \cdots, (k_{i''_{x-1},2}, k'_{i''_{x-1},0}) \rangle. \quad (5)$$
Besides, the client further constructs and sends to $\mathcal{S}_0$ an ordered list $\mathcal{I}$ with q elements, which is a sub-stream of $\langle 0, \cdots, x-1 \rangle$. The construction should meet the following *requirements*:
   - *Case I: $\mathcal{N}'_e$ is a non-leaf node*. For each $j \in \{0, \cdots, x-1\}$, if $\vec{D}_{i''_j}$ is a real block and it cannot be evicted to the next evicting node (i.e., the path that $\vec{D}_{i''_j}$ is assigned to does not pass the next evicting node), then j must not be in $\mathcal{I}$.
   - *Case II: $\mathcal{N}'_e$ is a leaf node*. $\mathcal{I}$ should contain only the IDs for dummy blocks.
9) Upon receiving $\mathcal{L}'_2$ from $\mathcal{S}_2$ as well as $\vec{\mathcal{R}}_0$ and $\mathcal{I}$ from the client, server $\mathcal{S}_0$ re-encrypts each block in $\mathcal{L}'_2$ based on the key pairs in $\vec{\mathcal{R}}_0$, as $\mathcal{S}_1$ and $\mathcal{S}_2$ do. Then, from the resulting list of blocks, $\mathcal{S}_0$ removes the list of blocks with offsets specified in $\mathcal{I}$; these removed blocks are sent to server $\mathcal{S}_1$ and become the new version of $\vec{\mathcal{L}}_1$ if $\mathcal{N}'_e$ is a non-leaf node, or discarded if $\mathcal{N}'_e$ is a leaf node. All blocks written back to $\mathcal{N}'_e$ are now with $ah = 0$.

Fig. 3 illustrates how the client and the servers cooperate during the eviction process, in a high level.

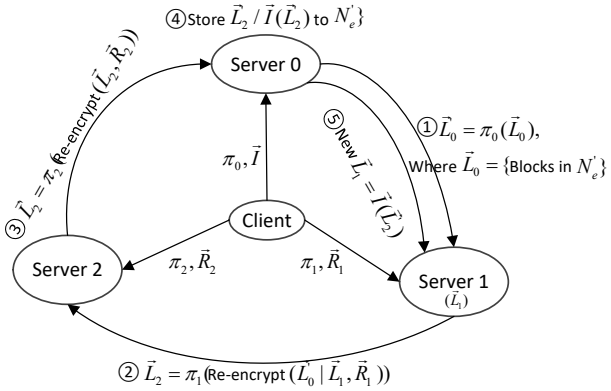


Figure 3. A High-level Illustration of Eviction Process.

VI. Accountability Enhancements

In this section, we propose several accountability enhancements to the above data query and eviction algorithms, so that if a server maliciously changes a block, another server is able to detect. The enhancements affect the storage organization, system initialization, data query algorithm and data eviction algorithm, in the following ways.

A. Enhancements to Storage and System Initialization

When the system is initialized, for each server $\mathcal{S}_i$ where $i \in \{0,1,2\}$, the client randomly constructs λ blocks each with z bits, denoted as $\vec{A}_{i,j} = \langle a_{i,j,0}, \cdots, a_{i,j,z-1}\rangle$ for $j \in \{0,\cdots,\lambda-1\}$, where each $a_{i,j,y} \in \{0,1\}$ for $y \in \{0,\cdots,z-1\}$. Then, the client sends each $\vec{A}_{i,j}$ to server $\mathcal{S}_i$, and the block should be kept secret only between server $\mathcal{S}_i$ and the client.

For each exported data block $\vec{D}$, letting $\langle d_0,\cdots,d_{z-1}\rangle$ denote its plain text, the client computes 3 message authentication codes (MACs) as follows.

- First, the client computes the following 3λ message authentication bits (MABs) for $\vec{D}$:

$$MAB_{i,j}(\vec{D}) = \oplus_{y\in\{0,\cdots,z-1\}} d_y \cdot a_{i,j,y}, \tag{6}$$

  where $i \in \{0,1,2\}$ and $j \in \{0,\cdots,\lambda-1\}$.
- Based on the MABs, the client computes the following 3 MACs for $\vec{D}$:

$$MAC_i(\vec{D}) = MAB_{i,0}|\cdots|MAB_{i,\lambda-1}, \tag{7}$$

  where $i \in \{0,1,2\}$ and $|$ denotes concatenation.

Finally, the client stores $MAC_0(\vec{D})$, $MAC_1(\vec{D})$ and $MAC_2(\vec{D})$ to the entry of $\vec{D}$ in the index table.

B. Enhancement to Data Query Algorithm

In the data query algorithm, we introduce an accountability enhancement to allow $\mathcal{S}_1$ to check if $\mathcal{S}_0$ has sent to it a correct sequence $\vec{\mathcal{L}}$. The detail is as follows.

During the query process, the client completely knows which blocks should be in $\vec{\mathcal{L}}$. Let $\vec{\mathcal{I}} = \langle i_0, i_1, \cdots\rangle$ denote the IDs of the blocks in the sequence. For each block with ID $i_x \in \vec{\mathcal{I}}$, the client computes an MAC of the block that can be checked by the $\mathcal{S}_1$ as follows:

- From the index table, it retrieves $MAC_1(\vec{D}_{i_x})$ (i.e., the MAC computed based on the block's plain text and the secret block $\vec{A}_1$ known by $\mathcal{S}_1$) and the current version of encryption key k_{i_x} for the block.
- It computes the two pseudo-random blocks that have been used to encrypt the block, i.e., $\vec{R}_{i_x,0} = PRG_1(k_{i_x,0})$, $\vec{R}_{i_x,1} = PRG_1(k_{i_x,1})$, and $\vec{R}_{i_x,2} = PRG_1(k_{i_x,2})$. Note that, the x-th block received by $\mathcal{S}_1$ should be equal to $\vec{D}_{i_x} \oplus \vec{R}_{i_x,0} \oplus \vec{R}_{i_x,1} \oplus \vec{R}_{i_x,2}$ if it is correct.
- It computes $MAC'_1(\vec{D}_{i_x})$ as

$$MAC_1(\vec{D}_{i_x}) \oplus MAC_1(\vec{R}_{i_x,0}) \oplus MAC_1(\vec{R}_{i_x,1}) \oplus MAC_1(\vec{R}_{i_x,2}), \tag{8}$$

  which should be equal to

$$MAC_1(\vec{D}_{i_x} \oplus \vec{R}_{i_x,0} \oplus \vec{R}_{i_x,1} \oplus \vec{R}_{i_x,2}) \tag{9}$$

  according to the definition of $MAC_1(\cdot)$.

Then, $MAC'_1(\vec{D}_{i_x})$ is sent to $\mathcal{S}_1$ for checking.

Upon receiving $\vec{\mathcal{L}}$ from $\mathcal{S}_0$ and the ordered list of MACs from the client, $\mathcal{S}_1$ applies $MAC_1(\cdot)$ to compute the MAC for each block in $\vec{\mathcal{L}}$, and compares the resulting MAC with the MAC sent from the client. If a mismatch is found, $\mathcal{S}_0$ will be identified to have modified some block.

C. Enhancements to Data Eviction Algorithm

The accountability enhancements to data eviction algorithm are similar to that applied for data query algorithm. That is, whenever a server $\mathcal{S}_i$ ($i \in \{0,1,2\}$) receives a list of blocks from another server, $\mathcal{S}_i$ needs to: (1) receive from the client an MAC_i for each block on the list; (2) re-computes the MAC_i for each block on the list; (3) find out if the above values match. We skip the detail due to the page limit.

VII. Security Analysis

According to the definition of security in Section II, we first study the security of the proposed system in terms of obliviousness and failure probability. Then, we study the accountability of the system.

A. Obliviousness Analysis

In this subsection, we show the obliviousness of the query and eviction processes; i.e., these processes are random and independent of the client's data access pattern. First of all, it is obvious that the interactions between servers and the client follow the same pattern, independent of the client's access pattern. Hence, we focus to analyze the obliviousness of the processes inside server $\mathcal{S}_0$.

1) Obliviousness in Query Path Selection: When the system is initialized, the path assigned to each block is selected randomly and independently of each other. After a block has been queried, its path is re-assigned randomly and independently of the client's data access pattern. Due to the randomness in path assignment, the query path for each query process, which is determined by the path assigned to the query target block, is random and independent of the client's access pattern.

2) Obliviousness in Block Access from Query Path: According to the data query algorithm, the following block access pattern has been enforced: from each node on the query path, the client must select one block that has not been accessed; meanwhile, every block that has already been accessed has the same probability to be accessed again according to Lemma 1.

3) Obliviousness in eviction process: The eviction process is random and independent of the client's data access pattern, due to the following reasons: (i) Each eviction process involves only one root-to-leaf path (called eviction path), and the order in which the paths are selected for as eviction paths is fixed and independent of data access pattern. (ii)During each eviction process, the processing for each node on the selected eviction path follows a fixed pattern which is independent of data access pattern. Specifically, all the data blocks on the node are re-encrypted and re-permuted by all the servers; then, the same number of blocks are stored back to the node.

B. Failure Analysis

In this subsection, we study the probabilities for a query process and an eviction process to fail.

1) Failure Probability for A Query Process: According to Algorithm 1, a query process fails only when the probability $\frac{\delta_{i,2}}{\delta_{i,0}^2}$ used in selecting a block (in Line 6) becomes greater than 1. Also, as discussed in Section IV, we aim to make $\frac{\delta_{i,1}+\delta_{i,2}}{\delta_{i,0}} \leq 1$ (which obviously makes $\frac{\delta_{i,2}}{\delta_{i,0}^2} \leq 1$) such that on average no more than 2 blocks are accessed from each layer of the storage tree during each query process. Hence, we here study the probability for $\frac{\delta_{i,1}+\delta_{i,2}}{\delta_{i,0}} > 1$, which is no less than the probability for a query process to fail. Our result is stated in the following Lemma 2.

Lemma 2: As long as $q \geq 25\lambda$ and $\alpha \geq 0.25 \ \wedge \ \beta \geq 0.25$ when $m = 2, 4$, $Pr[\frac{\delta_{i,1}+\delta_{i,2}}{\delta_{i,0}} \leq 1] > 1 - 2^{-\lambda}$, i.e., any query process fails with a probability less than $2^{-\lambda}$.

Proof: Consider an arbitrary node $\mathcal{N}_i$ on a m-ary storage tree, and let random variable X denote the times that $\mathcal{N}_i$ has been selected to be on a query path during two consecutive evictions involving the node. Obviously, $X \geq \delta_{i,1} + \delta_{i,2}$.

When $m = 2, 4$, according to the storage organization, the size of each node $\mathcal{N}_i'$ on the storage tree is at least $2q \cdot \min(1+\alpha, 1+\beta) \geq 2.5q$; i.e., $\delta_{i,0} + \delta_{i,1} + \delta_{i,2} \geq 2.5q$. Since an eviction process is launched every q queries, the mean of X is q. Further according to the multiplicative Chernoff bound,

$$Pr[X \leq 1.25q] > 1 - (\frac{e^{0.25}}{1.25^{1.25}})^q > 1 - 2^{-2\lambda}. \quad (10)$$

Hence,

$$Pr[\delta_{i,1} + \delta_{i,2} < \delta_{i,0}] \geq Pr[\delta_{i,1} + \delta_{i,2} \leq 1.25q] > 1 - 2^{-\lambda}. \quad (11)$$

When $m \geq 8$, the size of each node on the storage tree is at least $\frac{m-1}{2} \cdot q \geq 3.5q$; i.e., $\delta_{i,0} + \delta_{i,1} + \delta_{i,2} \geq 2.5q$. Due to Equation (10),

$$Pr[\delta_{i,1} + \delta_{i,2} < \delta_{i,0}] > Pr[\delta_{i,1} + \delta_{i,2} \leq 1.25q] > 1 - 2^{-\lambda}. \quad (12)$$

■

2) Failure Probability for An Eviction Process: An eviction process fails iff the following scenarios occur in Step 8) of the eviction algorithm. (i) *Failure Scenario I:* The current evicting node (i.e., $\mathcal{N}_e'$) is a non-leaf node, and so q out of the x blocks in $\mathcal{L}_2'$ need to be picked to send from $\mathcal{S}_0$ to $\mathcal{S}_1$. According to Case-I of the requirement, the q blocks should not contain any real block that cannot be evicted to the next evicting node, but failure will occur if there are more than $x - q$ real blocks that cannot be evicted to the next evicting node. (ii) *Failure Scenario II:* The current evicting node $\mathcal{N}_e'$ is a leaf node, and so q dummy blocks out of the x blocks in $\mathcal{L}_2'$ need to be discarded. Failure will occur if there are less than q dummy blocks (i.e., more than $x - q$ real blocks) in $\mathcal{L}_2'$.

The results of our analysis are summarized as the following Lemmas 3 and 4. The proofs, which have to be skipped due to space limit, can be developed based on the analysis of the eviction process and the application of the multiplicative Chernoff bound.

Lemma 3: With $q \geq 25\lambda$ and the following combinations of system parameters, i.e., $(m = 2, \alpha \geq 0.25)$, $(m = 4, \alpha \geq 0.25)$, $(m = 8, \alpha \geq 0.34)$ and $(m = 16, \alpha \geq 0.34)$, the Failure Scenario I occurs with a probability of $O(2^{-\lambda})$.

Lemma 4: With $q \geq 25\lambda$ and the following combinations of system parameters, i.e., $(m = 2, \beta \geq 0.25)$, $(m = 4, \beta \geq 0.25)$, $(m = 8, \beta \geq 0.13)$ and $(m = 16, \beta \geq 0.09)$, the Failure Scenario II occurs with a probability of $O(2^{-\lambda})$.

Based on the above analysis on obliviousness and failure probablities, we get the following theorem.

Theorem 1: The proposed system is secure under the security definition in Section II with $q \geq 25\lambda$ and the following combinations of system parameters: $(m = 2, \alpha \geq 0.25, \beta \geq 0.25)$, $(m = 4, \alpha \geq 0.25, \beta \geq 0.25)$, $(m = 8, \alpha \geq 0.34, \beta \geq 0.13)$ and $(m = 16, \alpha \geq 0.34, \beta \geq 0.09)$.

C. Accountability Analysis

The accountability of the proposed system relies on the security of the proposed MAC mechanism, which is formally stated and proved in the following.

Lemma 5: For $\forall j \in \{0, 1, 2\}$ and distinct blocks $\vec{D}$ and $\vec{D}'$,

$$Pr[MAB_{j,u}(\vec{D}) = MAB_{j,u}(\vec{D}')] = \frac{1}{2}, \ u = 0, \cdots, \lambda - 1. \quad (13)$$

Proof: (By induction). Let $\vec{D}$ and $\vec{D}'$ differ by n bits on indices $v_0, \cdots, v_{n-1}$; let $j \in \{0, 1, 2\}$, $u \in \{0, \cdots, \lambda - 1\}$ and $v \in \{0, \cdots, z - 1\}$; let $\vec{A}_{j,u}[v]$ denote the v-th bit on $\vec{A}_{j,u}$ (recall that $\vec{A}_{j,u}$ is a secret block shared only between the client and server $\mathcal{S}_j$).

When $n = 1$, $MAB_{j,u}(\vec{D}) = MAB_{j,u}(\vec{D}')$ iff $\vec{A}_{j,u}[v_0] = 0$. Because $\vec{A}_{j,u}$ is randomly picked from $\{0, 1\}^z$, $Pr[\vec{A}_{j,u}[v_0] = 0] = \frac{1}{2}$. Hence, Equation (13) holds.

Assuming Equation (13) holds when $\vec{D}$ and $\vec{D}'$ differ by $n \leq t$, we next prove the equation holds when $n = t + 1$. Without loss of generality, assume $\vec{D}[v_t] = 0$ and $\vec{D}'[v_t] = 1$. Let $\vec{I}_0$ be the z-bit block with 0 on every bit, $\vec{I}_1$ be the z-bit block with 1 on bit v_t but 0 on all other bits, and $\vec{D}'' = \vec{D}' \oplus \vec{I}_1$ (i.e., $\vec{D}' = \vec{D}'' \oplus \vec{I}_1$). Hence, $\vec{D}$ and $\vec{D}''$ differ in t bits $v_0, \cdots, v_{t-1}$. According to the induction assumption, $Pr[MAB_{j,u}(\vec{D}) = MAB_{j,u}(\vec{D}'')] = \frac{1}{2}$. Also note that, $Pr[MAB_{j,u}(\vec{I}_0) = MAB_{j,u}(\vec{I}_1)] = \frac{1}{2}$ and $\vec{D} = \vec{D} \oplus \vec{I}_0, \ \ \vec{D}' = \vec{D}'' \oplus \vec{I}_1$. Therefore, we have

$$\begin{aligned}
& Pr[MAB_{j,u}(\vec{D}) = MAB_{j,u}(\vec{D}')] && (14)\\
= \ & Pr[MAB_{j,u}(\vec{D} \oplus \vec{I}_0) = MAB_{j,u}(\vec{D}'' \oplus \vec{I}_1)] && (15)\\
= \ & Pr[MAB_{j,u}(\vec{D}) = MAB_{j,u}(\vec{D}'')] \times \\
& Pr[MAB_{j,u}(\vec{I}_0) = MAB_{j,u}(\vec{I}_1)] + \\
& Pr[MAB_{j,u}(\vec{D}) \neq MAB_{j,u}(\vec{D}'')] \times \\
& Pr[MAB_{j,u}(\vec{I}_0) \neq MAB_{j,u}(\vec{I}_1)] && (16)\\
= \ & \frac{1}{2} \times \frac{1}{2} + \frac{1}{2} \times \frac{1}{2} = \frac{1}{2}. && (17)
\end{aligned}$$

■

Theorem 2: If server $\mathcal{S}_i$ sends data block $\vec{D}'$ instead of $\vec{D}$ to server $\mathcal{S}_j$, where $i \neq j$ and $\vec{D} \neq \vec{D}'$, then:

$$Pr[MAC_j(\vec{D}') = MAC_j(\vec{D})] = 2^{-\lambda}; \quad (18)$$

i.e., the misbehavior of $\mathcal{S}_i$ is detected with a probability of $1 - 2^{-\lambda}$.

Proof: According to the MAC definition in Section VI and Lemma 5,

$$Pr[MAC_j(\vec{D}') = MAC_j(\vec{D})] \quad (19)$$

$$= \prod_{u=0}^{\lambda-1} Pr[MAB_{j,u}(\vec{D}') = MAB_{j,u}(\vec{D})] = 2^{-\lambda}. \quad (20)$$

■

VIII. Performance Evaluation and Comparisons

We have implemented the proposed system, and conducted performance comparisons with S^{3}ORAM [11], which is the newest and most-efficient ORAM construction that employs multiple non-colluding servers.

A. System Settings

We rent four AWS EC2 instances to run our implemented servers and client. As the communication latency between these instances are smaller than those between client and server and between the servers owned by different cloud owners, we conducted experiments to measure the communication latencies between AWS EC2 and Microsoft Compute Engine instances and add the measured average round trip delay 29 ms to the communication between our servers; we also measured the communication latencies between these cloud servers and a rented client located at the center of North America Continent, and add the measured average round trip delay 177.5 ms to the communication between our servers and client.

We set security parameter $\lambda = 40$, which makes the failure probability of each query and eviction process to be lower than 2^{-40}. According to Theorem 1, we set $q = 1024$ which is greater than 25λ; with different m, we adopt the following combinations of system parameter by default: $(m = 2, \alpha = \beta = 0.25)$, $(m = 4, \alpha = \beta = 0.25)$, $(m = 8, \alpha = 0.34, \beta = 0.13)$ and $(m = 16, \alpha = 0.34, \beta = 0.09)$.

In each evaluation, we vary N (i.e., the number of real data blocks to export) between 2^{20} to 2^{26} and vary B (i.e., the size of each data block in bytes) between $16K$ to $1M$.

We measure the following metrics: (1) client-server communication cost, which is measured as the average number of blocks sent between the client and the servers to serve each data query request; (2) inter-server communication cost, which is measured as the average number of blocks sent among the servers per data query; (3) query delay, which is measured as the average time elapse from a query request is sent from the client till the requested data block arrives at the client; (4) server storage overhead, which is measured as the amount of storage consumed at the server other than that for storing N exported data blocks; and (5) client storage cost, which is measured as the amount of storage consumed at the client.

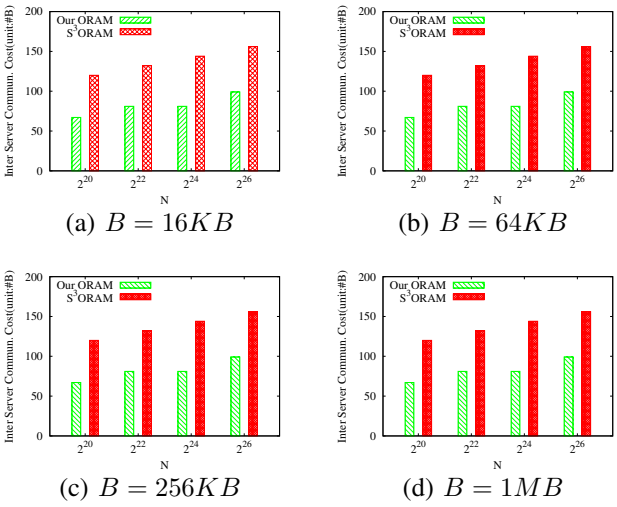
To optimize the system parameter selection, we also measure the system costs with varying m and results are shown in Table I. Note that, the table only shows the results when $N = 2^{20}$, as the trend is similar with different N. As we can see from the table, when m increases, the client-server communication cost does not change; the inter-server communication cost decreases and then increases; the server storage overhead decreases. Hence, in the following experiments, we set $m = 8$ to make our system to have low communication and storage overheads.

Table I
System Costs with Varying m

m	Client-Server Comm. Cost	Inter-Server Comm. Cost	Server Storage Overhead
2	1.3B	96B	1.5 N
4	1.3B	58B	1.1 N
8	1.3B	67B	0.3N
16	1.3B	90B	0.17N

B. Comparison with S^{3}ORAM

Comparisons are conducted over various metrics.



(a) $B = 16KB$ (b) $B = 64KB$

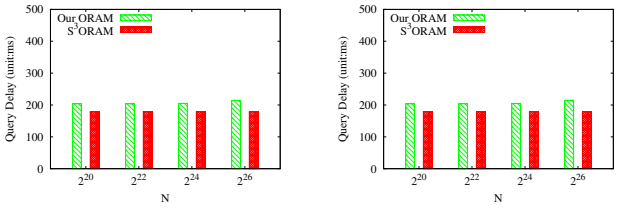
(c) $B = 256KB$ (d) $B = 1MB$

Figure 4. Inter-Server Communication Cost.

1) Communication Costs and Query Delay: As shown in Fig. 4, our ORAM system incurs smaller inter-server communication cost, which is about 60-80% of that of S^{3}ORAM. Both schemes require a constant number of blocks to be transferred between the client and server for each query. Specifically, in our system, the communication cost ranges from 1 to 1.3 data blocks per query, which includes 1 target block downloaded from $\mathcal{S}_1$ and some control messages for query and eviction. S^{3}ORAM needs to download 3 data blocks as well as a small size of meta-data.

In terms of query delay, as shown in Fig. 5, our ORAM has similar but a slightly higher query delay. This is due to the fact that, the request data block needs to travel through the path of $\mathcal{S}_0 \rightarrow \mathcal{S}_1 \rightarrow$ client; the detour between the servers incurs some extra delay, but it is very small compared to the delay between client and server.

2) Storage Overheads: Both schemes require 3 non-colluding servers. For S^{3}ORAM, all servers have the same

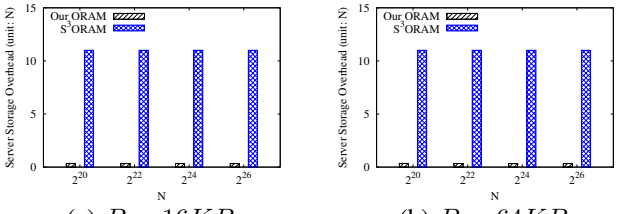


(a) $B = 16KB$ (b) $B = 64KB$

(c) $B = 256KB$ (d) $B = 1MB$

Figure 5. Query Delay

structure, different in that each server stores a different secret-shared version of blocks. Our system stores data blocks on one server, i.e., $\mathcal{S}_0$, while the other two servers only need to allocate small storage to facilitate query and eviction. Fig. 6 shows the server-side storage overheads. Specifically, the server-side storage overhead of S^{3}ORAM is $11N$ data blocks, while the overhead of our system is $(\beta + \frac{1+\alpha}{7})N + \frac{(1+\alpha)s}{2}$, which is no more than $0.3N$ blocks.



(a) $B = 16KB$ (b) $B = 64KB$

(c) $B = 256KB$ (d) $B = 1MB$

Figure 6. Server Storage Overhead

As the cost of the increased server storage efficiency, our system requires a larger client-side storage space, which is around 0.1% of the server-side storage cost.

Also node that, both schemes require some computation at the server side. S^{3}ORAM requires its servers to execute addition and multiplication of Shamir Secret Sharing operations, while our ORAM requires server to run random number generator to produce pseudo random sequences and then perform XOR operations to decrypt or re-encrypt data blocks. Our ORAM also requires the server to conduct authentication, which is also XOR operations. Our evaluations show that, the delay caused by the computations is nearly negligible compared to the communication delay.

3) Summary: Compared to S^{3}ORAM, our ORAM achieves a same level of efficiency in client-server communication, a higher level of efficiency in server-server communication, and a significantly higher level of server-side storage efficiency, at the price of increased client-side storage requirement, which is only around 0.1% of the server-side storage capacity and thus should be affordable for a client who runs an on-premise facility such as a cloud storage gateway.

IX. Conclusion and Future Work

This paper proposes an oblivious cloud storage system to address the limitations of existing research efforts. Extensive analysis and evaluation have shown that, the system can simultaneously attain the features of provable protection of data access pattern, low data query delay, low server storage overhead; low communication costs, and accountability. In the future, we plan to improve the performance of the system by further reducing the communication costs, especially the inter-server communication costs.

Acknowledgement

This work is partly supported by NSF grant CNS-1844591.

References

[1] Research and Markets, "Cloud storage market - forecasts from 2017 to 2022," *https://www.researchandmarkets.com/research/lf8wbx/cloud_storage*, 2017.

[2] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious RAMs," *Journal of the ACM*, vol. 43, no. 3, 1996.

[3] E. Stefanov, E. Shi, and D. Song, "Towards practical oblivious RAM," in *Proc. NDSS*, 2011.

[4] E. Stefanov, M. V. Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: an extremely simple oblivious RAM protocol," in *Proc. CCS*, 2013.

[5] J. Dautrich and E. Stefanov, "Burst ORAM: Minimizing ORAM Response Times for Bursty Access Patterns," in *Proc. 23rd USENIX Security Symposium*, 2014.

[6] C. Gentry, K. Goldman, S. Halevi, C. Julta, M. Raykova, and D. Wichs, "Optimizing ORAM and using it efficiently for secure computation," in *Proc. PETS*, 2013.

[7] E. Stefanov and E. Shi, "Multi-cloud oblivious storage," in *Proc. CCS*, 2013.

[8] L. Ren, C. W. Fletchery, A. Kwony, E. Stefanov, E. Shi, M. van Dijkz, and S. Devadasy, "Ring ORAM: Closing the Gap Between Small and Large Client Storage Oblivious RAM," in *Proc. IACR Cryptology ePrint Archive 2014:997*, 2014.

[9] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li, "Oblivious RAM with $O((\log N)^3)$ worst-case cost," in *Proc. ASIACRYPT*, 2011.

[10] S. Devadas, M. van Dijk, C. W. Fletcher, L. Ren, E. Shi, and D. Wichs, "Onion oram: A constant bandwidth blowup oblivious ram," in *Proc. TCC*, 2016.

[11] T. Hoang, C. Ozkaptan, A. Yavuz, J. Guajardo, and T. Nguyen, "S3oram: A computation-efficient and constant client bandwidth blowup oram with shamir secret sharing," in *Proc. ACM CCS*, 2017.

[12] X. Wang, T.-H. H. Chan, and E. Shi, "Circuit oram: On tightness of the goldreich-ostrovsky lower bound," in *Proc. ACM CCS*, 2015.

[13] X. Wang, S. Gordon, J. Katz, "Simple and Efficient Two-Server ORAM," in *Proc. ASIACRYPT*, 2018

[14] V. Bindschaedler, M. Naveed, X. Pan, X. Wang, and Y. Huang, "Practicing Oblivious Access on Cloud Storage: the Gap, the Fallacy, and the New Way Forward," in *Proc. CCS*, 2015.