# S-ORAM: A Segmentation-based Oblivious RAM

Jinsheng Zhang
Department of Computer Science
Iowa State University
Ames, IA, USA
alexzjs@iastate.edu

Wensheng Zhang
Department of Computer Science
Iowa State University
Ames, Iowa, USA
wzhang@iastate.edu

Daji Qiao
Department of Electrical and Computer Engineering
Iowa State University
Ames, Iowa, USA
daji@iastate.edu

## ABSTRACT

As outsourcing data to remote storage servers gets popular, protecting user's pattern in accessing these data has become a big concern. ORAM constructions are promising solutions to this issue, but their application in practice has been impeded by the high communication and storage overheads incurred. Towards addressing this challenge, this paper proposes a segmentation-based ORAM (S-ORAM). It adopts two segment-based techniques, namely, *piece-wise shuffling* and *segment-based query*, to improve the performance of shuffling and query by factoring block size into design. Extensive security analysis proves that S-ORAM is a highly secure solution with a negligible failure probability of $O(N^{-\log N})$. In terms of communication and storage overheads, S-ORAM outperforms the Balanced ORAM (B-ORAM) and the Path ORAM (P-ORAM), which are the state-of-the-art hash and index based ORAMs respectively, in both practical and theoretical evaluations. Particularly under practical settings, the communication overhead of S-ORAM is 12 to 23 times less than B-ORAM when they have the same constant-size user-side storage, and S-ORAM consumes $80\%$ less server-side storage and around $60\%$ to $72\%$ less bandwidth than P-ORAM when they have the similar logarithmic-size user-side storage.

## Categories and Subject Descriptors

D.0 [**Software**]: General; E.3 [**Data**]: Data Encryption

## Keywords

Data outsourcing; Access pattern; Privacy; Oblivious RAM

## 1. INTRODUCTION

Along with the increasing popularity of outsourcing data services to remote storage servers, arise also security and privacy concerns. Although encrypting data content has been a common practice for data protection, it does not fully eliminate the concerns, because users' data access pattern is not preserved and researchers have found that a wide range of private information could be revealed by observing the data access pattern [12].


*ASIA CCS'14*, June 4–6, 2014, Kyoto, Japan.

http://dx.doi.org/10.1145/2590296.2590323.

To address this issue, more and more efficient constructions have been developed to implement oblivious RAM (ORAM) [6], which was originally proposed for software protection but also is a provable solution to data access pattern preservation. Among these efforts, hash based ORAMs [6,8–11,13,15,20] utilize hash functions (including ordinary hash functions, cuckoo hash functions, bloom filters, etc.) to distribute data blocks to storage locations when data is stored or shuffled and to look up intended data blocks when data is queried. In comparison, index based ORAMs [16–19] maintain index structures to record the mapping between data blocks and locations and facilitate data lookup at the query time.

Though a large variety of techniques has been proposed and adopted, most existing ORAM constructions are still not applicable in practice because of the high communication and/or storage overheads incurred. Particularly, hash based ORAMs require a large extra storage space at the server side to deal with hash collisions; hence, access pattern privacy usually has to be preserved via heavy data retrievals and complicated data shuffling. Index based ORAMs rely on index structures to avoid the above problems. However, they fail to provide an efficient solution with which the index structures can be stored in a space-efficient manner and meanwhile can be searched and updated in a time-efficient manner. This limitation has also impeded their applications in practice.

We propose a novel ORAM scheme, called *segmentation-based oblivious RAM (S-ORAM)*, aiming to bring theoretical ORAM constructions one step closer to practical applications. Our proposal is motivated by the observation that a large-scale storage system (e.g., a cloud storage system such as Amazon S3 [2]) usually stores data in blocks and such a block typically has a large size [18], but most existing ORAM constructions treat data blocks as atomic units for query and shuffling, and do not factor block size into their designs. S-ORAM is designed to make better use of the large block size by introducing two *segment-based* techniques, namely, *piece-wise shuffling* and *segment-based query*, to improve the efficiency in data shuffling and query. With piece-wise shuffling, data can be perturbed across a larger range of blocks in a limited user-side storage; this way, the shuffling efficiency can be improved, and the improvement gets more significant as the block size increases. With segment-based query, S-ORAM organizes the data storage at the server side as a hierarchy of single-segment and multi-segment layers, and an encrypted index block is introduced to each segment. With these two techniques at the core, together with a few supplementary algorithms for distributing blocks to segments, S-ORAM can accomplish efficient query with only $O(\log N)$ communication overhead and a constant user-side storage, while existing ORAM constructions have to use a larger user-side storage to achieve the same level of communication efficiency in query.

Extensive security analysis has been conducted to verify the se-

curity of the proposed S-ORAM. Particularly, S-ORAM has been shown to be a provably highly secure solution that has a negligible failure probability of $O(N^{-\log N})$, which is no higher than that of existing ORAM constructions.

In terms of communication and storage overheads, S-ORAM outperforms the Balanced ORAM (B-ORAM) [13] and the Path ORAM (P-ORAM) [19], which are the best known theoretical hash-based and practical index-based ORAMs under small local storage assumption, respectively. Particularly, under practical settings [18] where the number of data blocks $N$ ranges from $2^{20}$ to $2^{36}$ and the block size is 32 KB to 256 KB, (i) the communication overhead of S-ORAM is 12 to 23 times less than B-ORAM when they have the same constant-size user-side storage; (ii) S-ORAM consumes 80% less server-side storage and around 60% to 72% less bandwidth than P-ORAM when they have the similar logarithmic-size user-side storage.

The rest of the paper is organized as follows. Section 2 briefly reviews existing ORAM constructions. In Section 3, the basic system model and threat model are described, and a formal security definition is provided. Our proposed S-ORAM is described in detail in Section 4. The subsequent Section 5 provides the security and overhead analysis as well as the comparisons between S-ORAM and two representative existing ORAM constructions. Finally, Section 6 concludes the paper.

# 2. RELATED WORK

In the past decades, there are numerous ORAM schemes proposed to hide user's pattern of access to remote data storage. We roughly classify them into two categories based on the data lookup technique used. From each category, one representative ORAM with the best performance among its peers is chosen to be compared with S-ORAM in Section 5.3.

- Hash based ORAMs: A number of ORAMs [6, 8–11, 13, 15, 20] belong to this category. With hash functions used for data lookup, these ORAMs require facilities such as buckets and stashes to deal with hash collisions. To the best of our knowledege, the Balanced ORAM (B-ORAM) [13] proposed by Kushilevitz et. al. achieves the best asymptotical communication efficiency among hash based ORAMs.

- Index based ORAMs: For ORAMs [16–19] belonging to this category, an index structure is used for data lookup. Therefore, it requires that the user-side storage stores the index, which is feasible only if the number of data blocks is not quite large. When the user-side storage cannot afford to store the index, it can outsource the index to the server in a way similar to storing real data blocks at the cost of increased communication overhead. The Path ORAM (P-ORAM) [19] proposed by Stefanov et. al. is a representative scheme in this category.

## 2.1 B-ORAM

In B-ORAM, the server-side storage is organized as a hybrid hierarchy with a total of $\frac{\log N}{\log \log N}$ layers, where each layer consists of $\log N$ equal-size sublayers. For the top $O(\log \log N)$ layers, the bucket-hash structure [6] proposed by Goldreich and Ostrovsky is deployed and the remaining layers are cuckoo-hash structures with a shared stash [8]. Since each layer is extended to multiple sublayers, the shuffling frequency is reduced while the query overhead is increased; a balance is struck between the query and shuffling overheads. The randomized shellsort [7] is selected as the underlying oblivious sorting algorithm for the shuffling process. In theory, the amortized communication overhead of B-ORAM is $O\left(\frac{\log^2 N}{\log \log N}\right)$ blocks per query. In practice, however, the overhead is on the magnitude of $\log^3 N$ due to a large constant ignored in the above big-O notation; particularly, querying one data block may require the user to access at least 1000 data blocks, which may not be acceptable in many practical applications.

## 2.2 P-ORAM

In P-ORAM, the server-side storage is organized as a binary tree in which each node contains a constant-size bucket for storing data blocks. Initially, data are randomly stored at leaf nodes, and an index structure is maintained to record the mapping between the IDs of data blocks and the IDs of the leaf nodes storing the blocks. Based on the index, a data query process retrieves all blocks on the path that contains the query target block and then moves the target block to the root node. In addition, a background eviction process is performed after each query process, to gradually evict blocks from the root node to nodes of lower-height so as to avoid or reduce node overflowing. The index can also be outsourced to the server and stored in a similar binary tree. Besides, to keep bucket size constant at each node, a user-side storage whose size is a logarithmic function of the number of data blocks is needed to form a stash. P-ORAM achieves a communication overhead of $O\left(\frac{\log^2 N}{\log(Z/\log N)}\right) \cdot \omega(1)$ blocks per query, where $Z$ is data block size and $\omega(1)$ is a security parameter. Though the communication overhead is considered to be acceptable in practice [19], the overhead of server-side storage, which is about $32N$ blocks, may pose as a big cost to the user.

Note that, other ORAM constructions such as the partition based ORAM [18], PrivateFS [21], and the single-round ORAM [22], either are based on different user-side storage assumptions than ours or focus on aspects other than bandwidth and storage efficiency, which is the main focus of our work. Due to these prominent differences, we do not compare them with our proposed S-ORAM.

# 3. PROBLEM STATEMENT

## 3.1 System Model

Similar to existing ORAM constructions [6, 8–11, 13, 15, 20], we consider a system composed of a user and a remote storage server. The user exports a large amount of data to store at the server, and wishes to hide from the server the pattern of his/her accesses to the data. Data are assumed to be stored and accessed in the unit of blocks, and the typical size of a block ranges from 32 KB to 256 KB [18]. Let $N$ denote the total number of data blocks exported by the user. For simplicity, we assume $\log N$ is an even number.

Each data request from the user, which the user wishes to keep private, can be one of the following types:

- read a data block $D$ of unique ID $i$ from the storage, denoted as a 3-tuple $(read, i, D)$; or

- write/modify a data block $D$ of unique ID $i$ to the storage, denoted as a 3-tuple $(write, i, D)$.

To accomplish a data request, the user may need to access the remote storage multiple times. Each access to the remote storage, which is observable by the server, can be one of the following types:

- retrieve (read) a data block $D$ from a location $l$ at the remote storage, denoted as a 3-tuple $(read, l, D)$; or

- upload (write) a data block $D$ to a location $l$ at the remote storage, denoted as a 3-tuple $(write, l, D)$.

## 3.2 Threat Model

We assume the user is trusted but the remote server is not. Particularly, the server is assumed to be honest but curious; that is, it behaves correctly in storing data and serving users' data accesses, but it may attempt to figure out the user's access pattern. The network connection between the user and the server is assumed to be secure; in practice, this can be achieved using well-known techniques such as SSL [4].

We inherit the standard security definition of ORAMs [6, 18, 19] to define the security of our proposed ORAM. Intuitively, an ORAM system is considered secure if the server learns nothing about the user's data access pattern. More precisely, it is defined as follows:

**Definition** Let $\vec{x} = \langle\ (op_1, i_1, D_1),\ (op_2, i_2, D_2),\ \cdots\ \rangle$ denote a private sequence of the user's intended data requests, where each $op$ is either a $read$ or $write$ operation. Let $A(\vec{x}) = \langle\ (op'_1, l_1, D'_1),\ (op'_2, l_2, D'_2),\ \cdots\ \rangle$ denote the sequence of the user's accesses to the remote storage (observed by the server), in order to accomplish the user's intended data requests. An ORAM system is said to be secure if (i) for any two equal-length private sequences $\vec{x}$ and $\vec{y}$ of the intended data requests, their corresponding observable access sequences $A(\vec{x})$ and $A(\vec{y})$ are computationally indistinguishable; and (ii) the probability that the ORAM system fails to operate is negligibly small, i.e., $O(N^{-\log N})$.

# 4. SCHEME

## 4.1 Overview

The design of S-ORAM is motivated by the observation that a large-scale storage system usually stores data in blocks and such a block typically has a large size. To the best of our knowledge, most existing ORAM constructions treat data blocks as atomic units for query and shuffling, and do not factor block size into their designs. The recently proposed index-based ORAM constructions [5, 16–19] have used large-size blocks to store indices to improve index search efficiency; still, more opportunities wait to be explored to fully utilize this feature.

S-ORAM is designed to make better use of the large block size to improve the efficiency in data shuffling and query, which are two critical operations in an ORAM system. Specifically, we propose the following two *segment-based* techniques:

- *Piece-wise Shuffling*. In S-ORAM, each data block is segmented into smaller *pieces*, and in a shuffling process, data is shuffled in the unit of pieces rather than blocks. As we know, data shuffling has to be performed at the user-side storage in order to achieve obliviousness. With the same size of user-side storage, shuffling data in pieces rather than blocks enables data perturbation across a larger range of blocks. This way, the shuffling efficiency can be improved, and the improvement gets more significant as the block size increases.

- *Segment-based Query*. In order to improve query efficiency, S-ORAM organizes the data storage at the server side as a hierarchy of single-segment and multi-segment layers. In each segment, an encrypted index block (with the same size as a data block) is introduced to maintain the mapping between data block IDs and their locations within the segment. This way, when a user needs to access a block in a segment, he/she only needs to access two blocks - the index block and the intended block. By adopting this technique together with supplementary algorithms for distributing blocks to segments, S-ORAM can accomplish efficient query with only $O(\log N)$ communication overhead and a constant user-side storage, while existing ORAM constructions have to use a larger user-side storage to achieve the same level of communication efficiency in query.

The following sections elaborate the details of the proposed S-ORAM construction, emphasizing on the above two techniques.

## 4.2 Storage Organization and Initialization

### 4.2.1 Data Block Format

Similar to existing ORAMs, S-ORAM stores data in blocks, and a data block is the basic unit for read/write operations by the user. A plain-text data block can be split into *pieces* and each piece is $z = \log N$ bits long, where $N$ is the total number of data blocks. The first piece contains the ID of the data block, say $i$, which is also denoted as $d_{i,1}$. The remaining pieces store the content of the data block, denoted as $d_{i,2}$, $d_{i,3}$, $\cdots$, $d_{i,P-1}$. Before being exported to the remote storage server, the plain-text data block is encrypted piece by piece with a secret key $k$, as shown in Figure 1:

$$\begin{aligned} &c_{i,0} = E_k(r_i), \text{ where } r_i \text{ is a random number;} \\ &c_{i,1} = E_k(r_i \oplus d_{i,1}); \\ &c_{i,2} = E_k(c_{i,1} \oplus d_{i,2}); \\ &\cdots, \\ &c_{i,P-1} = E_k(c_{i,P-2} \oplus d_{i,P-1}). \end{aligned} \tag{1}$$

Thus, the encrypted data block (denoted as $D_i$ and hereafter called data block for brevity) has the following format:

$$D_i = (c_{i,0}, c_{i,1}, c_{i,2}, \cdots, c_{i,P-1}). \tag{2}$$

It contains $P$ pieces and has $Z = z \cdot P$ bits.

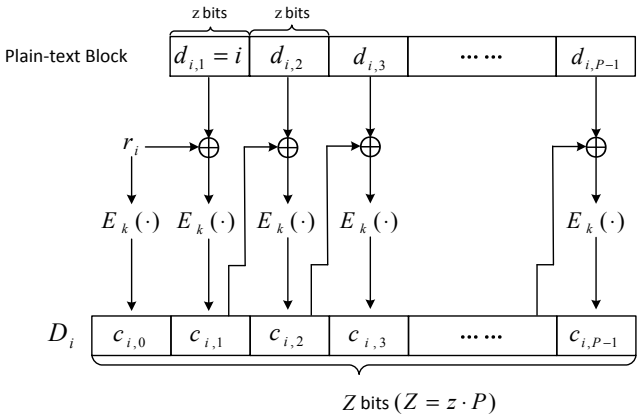


**Figure 1: Format of a data block in S-ORAM.**

### 4.2.2 Server-side Storage

S-ORAM stores data at the remote server in a pyramid-like structure as shown in Figure 2. The top layer, called *layer 1*, is an array containing at most four data blocks. The rest of the layers are divided into two groups as follows.

*T1 (Tier 1) Layers: Single-Segment Layers.* T1-layers refer to those between (inclusive) layer 2 and layer $L_1 = \lfloor 2 \log\log N \rfloor$. As illustrated in Figure 3, each T1-layer consists of a single segment, which includes an encrypted index block $I_l$ and $2^{l+1}$ data blocks. Among the data blocks, at most half of them are real data blocks as formatted in Figure 1, while the rest are dummy blocks each

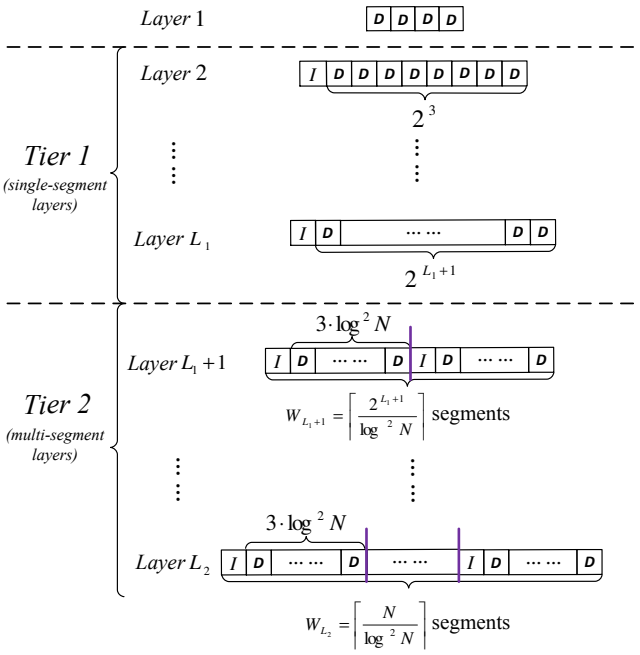


**Figure 2: Organization of the server-side storage.**

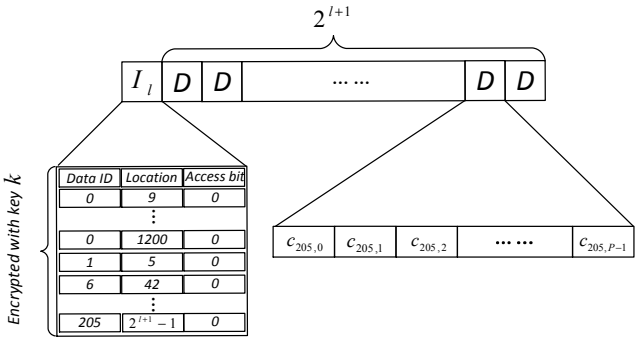


**Figure 3: Structure of a T1-layer.**

with ID 0 and randomly-stuffed content. The index block has $2^{l+1}$ entries; each entry corresponds to a data block in the segment which consists of three fields: *ID of the data block*, *location of the data block in the segment*, and *access bit* indicating whether the block has been accessed since it was placed to the location.

*T2 (Tier 2) Layers: Multi-Segment Layers.* T2-layers refer to those between (inclusive) layer $L_1 + 1$ and layer $L_2$, where $L_2 = \log N$. Each T2-layer consists of $W_l = \lceil \frac{2^l}{\log^2 N} \rceil$ segments, and each T2-layer segment has the same format as a T1-layer segment except that a T2-layer segment contains $3\log^2 N$ data blocks.

Note that, in the above storage structure, a segment (regardless whether at a T1-layer or T2-layer) contains at most $3\log^2 N$ data blocks. Therefore, the index block of a segment has at most $3\log^2 N$ entries. As each entry contains three fields: *ID of the data block* (needing $\log N$ bits), *location of the data block in the segment* (needing $\log(3\log^2 N)$ bits), and *access bit*, an index block needs at most $3\log^2 N[\log N + \log(3\log^2 N) + 1]$ bits. In practice, with $N \leq 2^{36}$ which is considered large enough to accommodate most practical applications, the size of an index block is less than 32 KB, which can fit into a typical data block assumed in the existing studies of practical ORAM schemes [18].

### 4.2.3 User-side Storage

The user organizes its local storage into two parts: *cache (temporary storage)* and *permanent storage*. Cache is used to buffer and process (including encrypt and decrypt) data blocks downloaded from the server. We assume that the size of the cache is $\alpha Z$ bits where $\alpha$ is a constant. In the S-ORAM design presented in this section, we set $\alpha = 2$. This design can be conveniently adapted to other configurations of cache size, as will be discussed in Section 5.3.

Permanent storage stores the user's secret information, including (i) a query counter keeping track of the number of queries that have been issued, (ii) a secret key $k$, and (iii) a one-way hash function $H_l(\cdot)$ for each T2-layer $l$, which maps a data block to one of the segments belonging to the layer. Note that, the size of permanent storage is much smaller than that of the cache, since only several hundreds of bits are needed to store the query counter, secret key, and hash functions.

### 4.2.4 Storage Initialization

The user initializes the S-ORAM system as follows:

- It randomly selects a secret key $k$ and a one-way hash function $H_{L_2}(\cdot)$ of layer $L_2$, i.e., the bottom layer.
- $N$ plain-text data blocks are encrypted into blocks $D_i$ where $i = 1, \cdots, N$ with the secret key $k$ in the format illustrated by Figure 1. In addition, $2N$ dummy blocks are randomly generated and encrypted also with key $k$.
- $N$ real data blocks and $2N$ dummy blocks are uploaded to layer $L_2$ of the server storage in a delicate manner to ensure that (i) each real data block $D_i$ of unique ID $i$ is distributed to segment $H_{L_2}(i)$ at layer $L_2$, (ii) each segment is assigned with exactly $3\log^2 N$ data blocks, and (iii) data blocks distributed to the same segment are randomly placed within the segment. Note that, a process like *data shuffling* elaborated in Section 4.4.4 can be adopted to distribute and place the data blocks to satisfy the above properties.

Besides, the user upload a dummy block $D$ to the server and let the server know it is a dummy block.

## 4.3 Data Query

As formally described in Algorithm 1, the process for querying a data block $D_t$ of ID $t$ consists of the following four phases.

In Phase I, the user retrieves and decrypts all data blocks stored at layer 1, attempting to find $D_t$ in the layer.

In Phase II, each non-empty T1-layer $l$ is accessed sequentially. Specifically, the index block $I_l$ of the layer is first retrieved and decrypted, and then one of the following two operations is performed:

- If $D_t$ has not been found at any layer prior to layer $l$ and $I_l$ indicates that $D_t$ is at layer $l$, record the location where $D_t$ resides, set the access bit of the location to 1, and re-encrypt and upload $I_l$ to save cache space. Then, retrieve $D_t$. Meanwhile, the server makes a copy of user uploaded dummy block $D$ to the location where $D_t$ was retrieved.
- Otherwise, the location of a dummy block $D_{t'}$ whose access bit in $I_l$ is 0 (i.e., it has not been accessed since last time it was distributed to its current location) is randomly picked and recorded. After the block's access bit is set to 1 in $I_l$, $I_l$ is re-encrypted and uploaded. Then, $D_{t'}$ is retrieved and discarded. The server also makes a copy of dummy block $D$ to fill in this location.

**Algorithm 1** Query data block $D_t$ of ID $t$.

```
found ← false
/* Phase I: access layer 1 */
Retrieve & decrypt blocks in layer 1
if D_t is found in layer 1 then found ← true
/* Phase II: access T1-layers */
for each non-empty layer l ∈ {2, ··· , L_1} do
    Retrieve & decrypt I_l – index block of the layer
    if (found = false ∧ t ∈ I_l) then
        Set the access bit of D_t to 1 in I_l
        Re-encrypt & upload I_l
        Retrieve & decrypt D_t
        found ← true
    else
        Randomly pick a dummy D_t' with access bit 0
        Set the access bit of D_t' to 1 in I_l
        Re-encrypt & upload I_l
        Retrieve & discard D_t'
    end if
end for
/* Phase III: access T2-layers */
for each non-empty layer l ∈ {L_1 + 1, ··· , L_2} do
    if (found = false) then
        s ← H_l(t)
    else
        s is randomly picked from {0, ··· , W_l − 1}
    end if
    Retrieve & decrypt I_l^s – index block of segment s
    if (found = false ∧ t ∈ I_l^s) then
        Set the access bit of D_t to 1 in I_l^s
        Re-encrypt & upload I_l^s
        Retrieve & decrypt D_t
        found ← true
    else
        Randomly find a dummy D_t' with access bit 0
        Set the access bit of D_t' to 1 in I_l^s
        Re-encrypt & upload I_l^s
        Retrieve & discard D_t'
    end if
end for
/* Phase IV: wrap up */
if (D_t is found in layer 1) then
    Encrypt an extra dummy D in local storage
else
    Re-encrypt D_t in local storage
end if
Upload all blocks in local storage back to layer 1
```

In Phase III, each non-empty T2-layer $l$ is accessed sequentially as follows.

- If $D_t$ has not been found at any layer prior to layer $l$, segment $s = H_l(t)$ of layer $l$ is picked to access. The index block $I_l^s$ of the segment is first retrieved and decrypted to check whether $D_t$ is at this segment. If so, the access bit of $D_t$ is set to 1 in $I_l^s$ before $I_l^s$ is encrypted and uploaded; then, $D_t$ is retrieved, server fill up $D_t$'s original location with a copy of dummy block $D$. Else, the user randomly selects a dummy block $D_{t'}$ in this segment whose access bit in $I_l^s$ is 0; after the access bit of $D_{t'}$ is set to 1, $I_l^s$ is re-encrypted and uploaded; then, $D_{t'}$ is retrieved and discarded, while a copy of dummy block $D$ is filled in $D_{t'}$'s original location.
- If $D_t$ has already been found at a layer prior to layer $l$, a segment is randomly selected from layer $l$ and the user randomly selects a dummy block $D_{t'}$ in this segment whose access bit in $I_l^s$ is 0. After the access bit of $D_{t'}$ is set to 1, $I_l^s$ is re-encrypted and uploaded. Then, $D_{t'}$ is retrieved, discarded, and a copy of dummy block $D$ is filled in $D_{t'}$'s original location.

Finally in Phase IV, the user wraps up the query process to ensure that $D_t$ is at layer 1, i.e., the top layer. To achieve this, the user first checks whether $D_t$ has been found at layer 1. If so, add a dummy block $D$ to local storage, re-encrypt all blocks in local storage (including $D_t$ and all blocks fetched from layer 1), and upload them back to layer 1; otherwise, the user directly re-encrypts all blocks in local storage and uploads them back to layer 1.

## 4.4 Data Shuffling

A critical step in S-ORAM is *data shuffling* which is used to perturb data block locations. It may occur at all layers of the storage hierarchy. Specifically, data shuffling at layer $l$ ($l = 2, \cdots, L_2 - 1$) is triggered when the total number of queries that have been processed is an odd multiple of $2^l$ (i.e., a multiple of $2^l$ but not a multiple of $2^{l+1}$). At this moment, layer $l$ is empty because: (i) it was empty immediately after data shuffling for some layer $l'$, where $l' > l$, has completed; (ii) since then, only $2^l$ queries have been processed, and during this course no data block has been added to this layer. During data shuffling at layer $l$, all data blocks in layers $\{1, \cdots, l-1\}$ are re-distributed randomly to layer $l$, and dummy blocks may be introduced to make layer $l$ full. Data shuffling at layer $L_2$, i.e., the bottom layer, however, is triggered when the total number of processed queries is any multiple of $2^{L_2}$; it re-distributes all real data blocks and selected dummy blocks in the entire hierarchy to fully occupy the bottom layer.

### 4.4.1 Preliminary: A Segment-Shuffling Algorithm

Compared to existing ORAM schemes, S-ORAM utilizes the user cache space more efficiently to speed up data shuffling. Specifically, the user cache is divided into four parts:

- $\pi$, which is a buffer to store a *permutation* of up to $2m^2$ inputs and thus needs $2m^2 \log(2m^2)$ bits, where $m$ is a system parameter.
- $B_0$, $B_1$, and $B_2$, which are three buffers and each may temporarily store up to $2m^2$ data pieces.

Recall that the size of a data piece is $z$ bits and the size of user cache is $\alpha Z$. Therefore, the following relation shall hold between $m$, $z$, $\alpha$, and $Z$:

$$2m^2 \cdot [\log(2m^2) + 3z] \leq \alpha Z. \tag{3}$$

Data shuffling in S-ORAM is based on a *segment-shuffling algorithm* (as shown in Algorithm 2). It is able to shuffle $n$ ($\leq 3\log^2 N$) data blocks with a communication cost of $O(n)$ data blocks, by setting the system parameter $m$ to $\sqrt{1.5}\log N$, under the following practical assumptions: (1) $N \leq 2^{36}$ which is considered large enough to accommodate most practical applications [18]; (2) the size of $Z$ is between 32 KB and 256 KB which is typically assumed in practical ORAM schemes [18]; and (3) $\alpha = 2$ meaning that a small local cache of two data blocks is assumed. It is easy to verify that, under these assumptions, Equation (3) holds. Moreover, as $n \leq 3\log^2 N = 2m^2$, $\pi$ is large enough to store a permutation of the IDs of $n$ data blocks, and $B_0$, $B_1$, and $B_2$ are large enough to store $n$ data pieces, which are required in the algorithm.

The segment-shuffling algorithm has two phases. Phase I processes the first two data pieces of all $n$ blocks as follows. After the first two pieces of all $n$ blocks are retrieved, IDs of the blocks are obtained and permuted according to a newly picked permutation function, and then re-encrypted using the key and newly-picked random numbers. After that, the new random numbers are uploaded after being encrypted, which is followed by the uploading of the shuffled and re-encrypted block IDs.

In Phase II, the remaining pieces of all $n$ blocks are retrieved, shuffled according to the new permutation function (newly picked in Phase I), re-encrypted, and then uploaded back to the server. This phase runs iteratively and the $(v+1)$-st pieces are retrieved and processed at the $v$-th ($v = 1, \cdots, P-2$) iteration. Particularly, when the $(v+1)$-st pieces are retrieved, two encrypted versions of the $v$-th pieces are present in the user cache. Using the key and the older version of the $v$-th pieces, the plain-text embedded in the $(v+1)$-st pieces are obtained; then, the pieces are permuted, and re-encrypted using the same key and the newer version of the $v$-th pieces, before being uploaded back to the server. At the end of the iteration, two encrypted versions of the $(v+1)$-st pieces are left in the user cache, which will be used in the processing of the $(v+2)$-nd pieces in the next iteration.

#### 4.4.2 *Shuffling a T1-layer $l$ ($2 \le l \le L_1$)*

When a T1-layer $l$ is to be shuffled, all the blocks belonging to the layers above shall be shuffled and distributed to layer $l$, which has $4 + 2^{2+1} + \cdots + 2^l = 2^{l+1} - 4$ blocks in total. The server first makes 4 copies of dummy block $D$ such that the total number of blocks to be shuffled is $2^{l+1}$. Then, the segment-shuffling algorithm is invoked to shuffle these blocks to layer $l$.

**Algorithm 2** Segment-Shuffling of Blocks $(D_{i_1}, \cdots, D_{i_n})$.

*/* Phase I: shuffling first two pieces of all blocks */*
Retrieve $(c_{i_1,0}, \cdots, c_{i_n,0})$ to $B_0$
Decrypt $B_0$ to $(r_{i_1,0}, \cdots, r_{i_n,0})$ using $k$
Retrieve $(c_{i_1,1}, \cdots, c_{i_n,1})$ to $B_1$
Decrypt $B_1$ to $(i_1, \cdots, i_n)$ using $k$ and $B_0$
Store $(i_1, \cdots, i_n)$ in $B_2$
Pick & store a random permutation in $\pi$
Permute $B_2$ to $(i'_1, \cdots, i'_n)$ according to $\pi$
Generate, re-encrypt & upload entries of a new index block based on $B_2$ and $\pi$
**for** each $i'_j$ in $B_2$ **do**
    Randomly picks $r'_{i'_j}$
    Encrypt $r'_{i'_j}$ to $c'_{i'_j,0}$ using $k$, and upload it
    Encrypt $i'_j$ to $c'_{i'_j,1}$ using $k$ and $c'_{i'_j,0}$
**end for**
Upload $B_2$ to designated locations
*/* Phase II: shuffling remaining pieces of all blocks */*
**for** each $v \in \{2, \cdots, P-1\}$ **do**
    Retrieve $(c_{i_1,v}, \cdots, c_{i_n,v})$ to $B_0$
    **for** each $j \in \{1, \cdots, n\}$ **do**
        Decrypt $c_{i_j,v}$ to $d_{i_j,v}$ using $k$ and $c_{i_j,v-1}$ in $B_1$
        Replace $c_{i_j,v-1}$ in $B_1$ by $c_{i_j,v}$ from $B_0$
        Replace $c_{i_j,v}$ by $d_{i_j,v}$ in $B_0$
    **end for**
    Permute $B_0$ to $(d_{i'_1,v}, \cdots, d_{i'_n,v})$ according to $\pi$
    Encrypt $(d_{i'_1,v}, \cdots, d_{i'_n,v})$ in $B_0$ using $k$ and $B_2$
    Replace $B_2$ by $B_0$
    Upload $B_2$ to designated locations
**end for**

#### 4.4.3 *Shuffling a T2-layer $l$ ($L_1 < l < L_2$)*

Similar to a T1-layer, when a T2-layer $l$ (excluding the bottom layer $L_2$) is to be shuffled, all the blocks belonging to the layers above shall be shuffled and distributed to layer $l$. The total number of these blocks is $w = 4 + 2^{2+1} + \cdots + 2^{L_1+1} + 3 \cdot 2^{L_1+1} + \cdots + 3 \cdot 2^{l-1}$ which is less than $3 \cdot 2^l$. Note that, among these blocks, the number of real data blocks is at most $2^l$ as data shuffling is triggered every $2^l$ queries.

Before shuffling, the user updates the hash function $H_l(\cdot)$ used for layer $l$. Then, it uploads a dummy block to the server, and requests the server to make $4 \cdot 2^l - w$ copies of the dummy block to be temporarily stored at layer $l$. This way, the total number of data blocks to be shuffled becomes $4 \cdot 2^l$, among which there are at most $2^l$ real data blocks.

Data shuffling at layer $l$ consists of the following three rounds of scanning and two rounds of oblivious sorting.

**Round I: Scanning**. Blocks are retrieved, labeled, re-encrypted, and then uploaded. Labeling obeys the following rules: (i) Each block is labeled with a tuple of two tags; (ii) Each real data block of ID $i$ has $H_l(i)$ as its first-tag and its second-tag is 0; (iii) Dummy blocks are labeled in such a way that exactly $3\log^2 N$ dummy blocks have $j$ as their first-tag for each $j \in \{1, \cdots, \lceil \frac{2^l}{\log^2 N} \rceil\}$ while all other dummy blocks have $\infty$ as their first-tag. All dummy blocks have $\infty$ as the second-tag.

**Round II: Oblivious Sorting**. All the labeled blocks are sorted obliviously (using the *oblivious data sorting* scheme presented in Section 4.4.5 and the Appendix) in the non-descending order based on the tag-tuple. Particularly, a block with a smaller first-tag should precede ones with larger first-tags; blocks with the same first-tag are sorted in the non-descending order based on the second-tag. This way, real data blocks are sorted to precede dummy blocks.

**Round III: Scanning**. The sorted sequence of blocks is scanned and divided into segments each containing $3\log^2 N$ blocks. A counter is used to facilitate the process. Specifically, the following rule is applied when a block is scanned:

- If the block is the very first one or it has a different first-tag from its immediate predecessor, it becomes the first one of a new segment, and the counter is reset to 1.
- Otherwise: If the counter is less than $3\log^2 N$, the counter is incremented by 1. If the counter reaches $3\log^2 N$, the block is considered redundant and hence its first-tag is relabeled as $\infty$, which means this block is a redundant dummy blocks.

**Round IV: Oblivious Sorting**. This round sorts all the redundant blocks (i.e., those with $\infty$ as the first-tag) to the end of the sequence. Similar to Round II, this is achieved by obliviously sorting the blocks in the non-decreasing order based on the tag-tuple. Then, the redundant blocks are removed.

**Round V: Scanning**. This round is to rebuild an index block for each segment. For each segment formed in the previous round, the segment-shuffling algorithm is applied to distribute the $3\log^2 N$ data blocks back to the server.

#### 4.4.4 *Shuffling the Bottom Layer $L_2$*

Every time when the number of queries is a multiple of $2^{L_2} = N$, layer $L_2$ needs to be shuffled, which means the entire storage shall be shuffled and all blocks from every layer shall participate in data shuffling. Hence, the total number of blocks to be shuffled is $w' = 4 + 2^{2+1} + \cdots + 2^{L_1+1} + 3 \cdot 2^{L_1+1} + \cdots + 3 \cdot 2^{L_2-1} + 3 \cdot 2^{L_2} < 6N$.

Similar to the shuffling of other T2-layers, there are also three rounds of scanning and two rounds of oblivious sorting to accom-

plish layer $L_2$ shuffling. To be more specific, Round I scanning and Round II oblivious sorting are performed on $w' < 6N$ blocks instead of $4 \cdot 2^l$ blocks in T2-layer shuffling. After Round II oblivious sorting, only the first $4N$ blocks participate in Rounds III, IV, and V; therefore, they are identical to the ones in T2-layer shuffling.

#### 4.4.5 Oblivious Data Sorting

Existing oblivious sorting techniques for ORAMs with constant local storage either incurs high asymptotical overhead (for example, Batcher's sorting network [3] incurs $O(n \log^2 n)$ communication overhead) or large hidden constant behind the big-O notations (e.g., AKS sorting network [1] incurs $c \cdot n \log n$ communication overhead with $c \geq 10^3$ and randomized shellsort [7] incurs $> 24 \cdot n \log n$ overhead), which significantly impede their practical efficiency. Hence, a more practically efficient sorting method is needed.

In S-ORAM, we develop an m-way oblivious sorting scheme based on the m-way sorting algorithm in [14]. It sorts data in pieces rather than blocks, which exploits the user cache space more efficiently and thus achieves a better performance than the aforementioned algorithms, particularly when the block size is relatively large (which is common in practice [18]). Modifications have also been made to the original m-way sorting algorithm to ensure the obliviousness of data sorting. Details of the proposed m-way oblivious sorting scheme are omitted due to space limitation. Please refer to the Appendix for a complete description.

## 5. ANALYSIS

### 5.1 Security Analysis

To prove the security of S-ORAM, we describe three lemmas before presenting the main theorem.

LEMMA 1. *When shuffling a T2-layer $l$, the probability that more than $1.5 \log^2 N$ real data blocks are distributed to any given segment is $O(N^{-\log N})$.*

PROOF. When shuffling a T2-layer $l$ as in Section 4.4.3, up to $2^l$ real data blocks are mapped (by a hash function) to $\lceil \frac{2^l}{\log^2 N} \rceil$ segments uniformly at random. In the following proof, we first assume the number of real data blocks is $2^l$ and compute the probability that there exists a segment with at least $1.5 \log^2 N$ real blocks.

Let us consider a particular segment, and define $X_1, \cdots, X_{2^l}$ as random variables such that

$$X_i = \begin{cases} 1 & \text{the } i^{\text{th}} \text{real block mapped to the segment,} \\ 0 & \text{otherwise.} \end{cases} \quad (4)$$

Note that, $X_1, \cdots, X_{2^l}$ are independent of each other, and hence for each $X_i$, $\mathbf{Pr}[X_i = 1] = \frac{1}{2^l/\log^2 N} = \frac{\log^2 N}{2^l}$. Let $X = \sum_{i=1}^{2^l} X_i$. The expectation of $X$ is

$$E[X] = E\left[\sum_{i=1}^{2^l} X_i\right] = \sum_{i=1}^{2^l} E[X_i] = 2^l \cdot \frac{\log^2 N}{2^l} = \log^2 N. \quad (5)$$

According to the multiplicative form of Chernoff bound, for any $j \geq E[X] = \log^2 N$, it holds that

$$\mathbf{Pr}[\text{at least } j \text{ real data blocks in this particular segment}] = \mathbf{Pr}[X \geq j] < \left(\frac{e^{\delta-1}}{\delta^\delta}\right)^{\log^2 N}, \quad (6)$$

where $\delta = \frac{j}{\log^2 N}$. By applying the union bound, we can obtain

$$\mathbf{Pr}[\exists \text{ a segment with at least } j \text{ real data blocks}] < \frac{2^l}{\log^2 N} \cdot \left(\frac{e^{\delta-1}}{\delta^\delta}\right)^{\log^2 N}. \quad (7)$$

Further considering that $2^l \leq N$, it follows that

$$\mathbf{Pr}[\exists \text{ a segment with at least } 1.5 \log^2 N \text{ real data blocks}] < \frac{N}{\log^2 N} \cdot \left(\frac{e^{0.5}}{1.5^{1.5}}\right)^{\log^2 N} = O(N^{-\log N}). \quad (8)$$

When the number of real blocks is less than $2^l$, obviously, the above probability is also $O(N^{-\log N})$. Therefore, the lemma is proved. □

LEMMA 2. *(Failure probability of S-ORAM). The probability that the S-ORAM construction fails is $O(N^{-\log N})$. Particularly, a data query or shuffling process will never fail on any T1-layer; a data query or shuffling process on a T2-layer may fail with probability $O(N^{-\log N})$.*

PROOF. The S-ORAM construction fails if a query or shuffling process fails.

A data query process fails only if: (Q1) the process fails to find the target data block; or (Q2) the process fails to find a non-accessed dummy block on a layer when it needs to retrieve one according to the query algorithm. As the storage server is assumed to be honest, case (Q1) will not occur. Case (Q2) will not occur when the query process is accessing a T1-layer, due to the following reasons: Each layer $l$ contains $2^{l+1}$ blocks, among which the number of dummy blocks is at least $2^l$; since the data blocks in the layer are shuffled once every $2^l$ queries, there must exist at least one non-accessed dummy block for each of the $2^l$ queries.

A data shuffling process for layer $l$ fails only if: (S1) layer overflow occurs, i.e., the process tries to store more data blocks to the layer than its capacity; or (S2) segment overflow occurs when layer $l$ is a T2-layer, i.e., the process tries to store more than $3 \log^2 N$ real data blocks to a segment. As discussed in Sections 4.4.2, case (S1) will not occur when shuffling a T1-layer $l$ because the total number of blocks to be shuffled is $2^{l+1}$, which is the capacity of the layer. According to Section 4.4.3, case (S1) will not occur when shuffling a T2-layer $l$, because Round IV of the shuffling algorithm marks and removes redundant blocks to make the total number of blocks less than the capacity of the layer.

Hence, we only need to study the probability for cases (Q2) and (S2) to occur on a T2-layer.

Case (Q2) occurring on a T2-layer $l$ means that a query process fails to find a non-accessed dummy block on a segment of the layer. This can only happen in one of the following two scenarios: (i) more than $1.5 \log^2 N$ real data blocks are distributed to this segment, or (ii) more than $1.5 \log^2 N$ dummy data blocks are accessed from this segment since last time the blocks were shuffled. According to Lemma 1, scenario (i) occurs with probability $O(N^{-\log N})$. As the selections of dummy blocks during the query processes are also randomly distributed among all segments of the layer, which is the same as the distribution of real data blocks to the segments during the shuffling process, the probability for scenario (ii) to occur is also $O(N^{-\log N})$. Hence, the probability for case (Q2) to occur is $O(N^{-\log N})$.

When case (S2) occurs on a T2-layer, there must be at least one segment of the layer distributed with more than $3 \log^2 N$ blocks. The probability that this case occurs is smaller than the probability that at least one segment of the layer is distributed with at least

$1.5\log^2 N$ blocks, which is $O(N^{-\log N})$. Hence, the probability for case (S2) to occur is also $O(N^{-\log N})$.

To summarize, the probability that the S-ORAM construction fails is $O(N^{-\log N})$. □

LEMMA 3. *(Random and non-repeated location access in S-ORAM). In S-ORAM, a query process accesses locations from each non-empty layer $l$ ($l > 1$) in a random and non-repeated manner. Here, the non-repeatedness means that, a data block is accessed for at most once between two consecutive shuffling processes that involve the block.*

PROOF. When layer $l$ is a T1-layer, there are two cases. *Case 1.1.* If the query target data block $D_t$ has not been found at any layer prior to layer $l$, and layer $l$ contains $D_t$, $D_t$ is accessed. Due to the randomness of the hash function $H_l(\cdot)$ used to distribute data blocks to locations, the location of $D_t$ is randomly distributed among all the locations of layer $l$. Hence, the access is random. Also, $D_t$ must not have been accessed since last time it was involved in data shuffling; otherwise, the block must have been a query target of an earlier query and then moved to layer 1 already. Hence, the access is also non-repeated. *Case 1.2.* Otherwise, a non-access dummy block is randomly selected to access, which makes the access to be random and non-repeated.

When layer $l$ is a T2-layer, there are following cases. *Case 2.1.* If the query target $D_t$ has not been found at any layer prior to layer $l$, a segment $s = H_l(t)$ of layer $l$ is picked to access. Due to the randomness of the hash function $H_l(\cdot)$, the selection of $s$ is random. Then:

- If $D_t$ is in segment $s$, the block is accessed. As the shuffling process randomly permutes blocks within the same segment, the access of $D_t$ within segment $s$ is random. The access is also non-repeated due to the same reasoning as in *Case 1.1.*
- If $D_t$ is not in segment $s$, a non-accessed dummy block is randomly picked to access in the segment. Hence, the access is random and non-repeated.

*Case 2.2.* If the query target $D_t$ has already been found above layer $l$, segment $s$ is randomly selected and a non-accessed dummy block is randomly picked and accessed in the selected segment. Hence, the access is random and non-repeated. □

THEOREM 1. *S-ORAM is secure under the security definition in Section 3.2.*

PROOF. Given any two equal-length sequence $\vec{x}$ and $\vec{y}$ of data requests, their corresponding observable access sequences $A(\vec{x})$ and $A(\vec{y})$ are computationally indistinguishable, because of the following reasons:

- Firstly, according to the query algorithm, sequences $A(\vec{x})$ and $A(\vec{y})$ should have the same format; that is, they contain the same number of accesses, and each pair of corresponding accesses have the same format.
- Secondly, all blocks in the storage of S-ORAM are randomized encrypted and each block is re-encrypted after each access. Hence, the two sequences could not be distinguished based on the appearance of blocks.
- Thirdly, according to the query algorithm, the $j$-th accesses ($j = 1, \cdots, |A(\vec{x})|$) of the $A(\vec{x})$ and $A(\vec{y})$ are from the same non-empty layer of the storage; and according to Lemma 3, the locations accessed from the layer are random and non-repeated in both sequences.

Also, according to Lemma 2, the S-ORAM construction fails with probability $O(N^{-\log N})$, which is considered negligible and no larger than the failure probability of existing ORAMs [6, 8–11, 13, 15–20]. □

## 5.2 Overhead Analysis

We analyze the overhead of S-ORAM including bandwidth consumption (i.e., communication overhead), user-side storage overhead, and server-side storage overhead.

The server-side storage in S-ORAM is no more than $6N \cdot Z$ bits at any time. Note that a storage of at most $6N \cdot Z$ bits is needed only when shuffling layer $L_2$, i.e., the bottom layer; for all other layers, a storage of at most $3N \cdot Z$ bits is needed. The user-side storage is constant; specifically, it is $2 \cdot Z$ bits.

The bandwidth consumption consists of two parts: query overhead $Q(N)$ and shuffling overhead $S(N)$, which are analyzed next.

The query overhead includes the retrieval and uploading of up to four data blocks for layer 1 and one data block (i.e., the index block) for each non-empty layer. Hence, the maximum communication cost $Q(N)$ is the retrieval and uploading of $1.5\log N + 2$ blocks per query.

When shuffling a T1-layer $l$ of $2^{l+1}$ data blocks, each data block is processed once in the user cache. Hence, the communication cost is the retrieval and uploading of $2^{l+1}$ blocks.

When shuffling a T2-layer $l$ of $n = 4 \cdot 2^l$ data blocks or the bottom layer $L_2$ of $n < 6N$ data blocks, the shuffling process includes three rounds of scanning and two rounds of oblivious sorting. The scanning rounds can be integrated into the oblivious sorting rounds. Specifically, Round I (scanning round) can be performed side-by-side with the segment-sorting (line 2 of Algorithm 3) of Round II (oblivious sorting round). Round III (scanning round) can be performed concurrently with the last step of merging (line 19 of Algorithm 5) in Round II. Similarly, Round V (the third scanning round) can also be performed concurrently with the last step of merging in Round IV (oblivious sorting round). This way, the shuffling cost becomes the cost for two rounds of oblivious sorting.

Next, we compute the cost of m-way obliviously sorting $n$ data blocks. With Algorithm 3, $n$ blocks are divided into $\frac{n}{2m^2}$ subsets of equal size. These subsets are sorted at the user cache and then recursively merged into a large sorted set by Algorithm 5. During each merging phase, every $m$ smaller sorted subsets are merged into one larger sorted subset. Thus, there is a total of $\log_m \frac{n}{2} - 1$ merging phases needed to form the final sorted set. Let $G(m, s)$ denote the number of times that each block is retrieved and then uploaded during a merging phase, where $m$ smaller sorted subsets are merged into one larger sorted subset and each smaller subset contains $s$ data blocks.

We have the following recursive relation:

$$G(m, s) = G\left(m, \frac{s}{m}\right) + 2. \tag{9}$$

This is because, during the merging phase, each block should (i) perform another phase of merging in which smaller subsets each containing $s/m$ blocks are merged into subsets of $s$ blocks (line 10 in Algorithm 5), incurring $G(m, \frac{s}{m})$ times of retrieval and uploading for each block, and then (ii) perform steps 13-20 in Algorithm 5, incurring 2 times of retrieval and uploading of each block. Hence, each data block should be retrieved and uploaded for

$$T(n) = \sum_{i=1}^{\log_m \frac{n}{2} - 1} G(m, 2m^{i+1}) = \left(\log_m \frac{n}{2} - 1\right)^2 \tag{10}$$

times during the entire shuffling process.

As shuffling is performed periodically at layers, the amortized shuffling overhead consists of the following:

- Each T1-layer $l$ ($2 \leq l \leq L_1$) is shuffled once every time when an odd multiple of $2^l$ queries have been made, and each of the $2^l$ data blocks at T1-layer $l$ is scanned once for every shuffling. Hence, the amortized overhead is $S_l(N) = \frac{2^{l+1}}{2^{l+1}} = 1$ block scanning per query.
- Each T2-layer $l$ ($L_1 < l < L_2$), except the bottom layer $L_2$, is shuffled also once every time when an odd multiple of $2^l$ queries have been made, and two rounds of oblivious sorting are performed on $4 \cdot 2^l$ data blocks. Hence, the amortized overhead is $S_l(N) = \frac{2 \cdot 4 \cdot 2^l \cdot T(4 \cdot 2^l)}{2^{l+1}} = 4 \cdot T(4 \cdot 2^l)$ block scannings per query.
- The bottom layer $L_2$ is shuffled every time when a multiple of $N$ queries have been made, and two rounds of oblivious sorting are performed. The first oblivious sorting is performed on $w < 6N$ blocks and second one is performed on $4N$. Hence, the amortized overhead is at most $S_{L_2}(N) = \frac{6N \cdot T(6N)}{N} + \frac{4N \cdot T(4N)}{N} = 6 \cdot T(6N) + 4 \cdot T(4N)$ block scannings per query.

Therefore, amortized shuffling overhead $S(N)$ is:

$$S(N) = \sum_{l=2}^{L_1} S_l(N) + \sum_{l=L_1+1}^{L_2-1} S_l(N) + S_{L_2}(N) = O\left(\frac{\log^3 N}{\log^2 m}\right).$$

To summarize, the bandwidth consumption for S-ORAM is

$$Q(N) + S(N) = O\left(\frac{\log^3 N}{\log^2 m}\right). \quad (11)$$

## 5.3 Overhead Comparison

We now compare the performance of S-ORAM with that of B-ORAM and P-ORAM from both theoretical and practical aspects. The theoretical results of bandwidth, user-side storage and server-side storage overheads are denoted as $T_b$, $T_c$, and $T_s$, and the practical results as $P_b$, $P_c$, and $P_s$, respectively. The practical settings used here are as follows: the number of data blocks $N$ ranges from $2^{20}$ to $2^{36}$ and the block size ranges from 32 KB to 256 KB, which are similar to the practical settings adopted in [18]. In the comparisons, system parameter $\alpha$ in S-ORAM may be set to a value other than 2. If $\alpha \neq 2$, the scheme presented in Section 4 can be modified to accommodate this by simply setting parameter $m$ to the largest integer satisfying Equation (3).

| | S-ORAM | B-ORAM |
|---|---|---|
| $T_b$ | $O(\frac{\log^3 N}{\log^2(Z/\log N)} \cdot Z)$ | $O(\frac{\log^2 N}{\log \log N} \cdot Z)$ |
| $T_c$ | $O(Z)$ | $O(Z)$ |
| $T_s$ | $O(N \cdot Z)$ | $O(N \cdot Z)$ |
| $P_b$ | $c \log^2 N \cdot Z (0.599 \leq c \leq 0.978)$ | $> \frac{60 \log^2 N}{\log \log N} \cdot Z$ |
| $P_c$ | 512 KB | 512 KB |
| $P_s$ | $\leq 6N \cdot Z$ | $\geq 8N \cdot Z$ |

**Table 1: Performance Comparison: S-ORAM vs. B-ORAM**

### 5.3.1 S-ORAM vs. B-ORAM

In order to compare S-ORAM with B-ORAM, the user cache size is set to 512 KB in both constructions.

| | S-ORAM | P-ORAM |
|---|---|---|
| $T_b$ | $O(\frac{\log^3 N}{\log^2(Z/\log N)} \cdot Z)$ | $O(\frac{\log^2 N}{\log(Z/\log N)} \cdot Z) \cdot \omega(1)$ |
| $T_c$ | $O(Z)$ | $O(\log N \cdot Z) \cdot \omega(1)$ |
| $T_s$ | $O(N \cdot Z)$ | $O(N \cdot Z)$ |

**Table 2: Theoretical Performances: S-ORAM vs. P-ORAM**

As shown in Table 1, the bandwidth consumption of S-ORAM is 12 to 23 times less than that of B-ORAM under practical settings, while the server-side storage overhead of S-ORAM is about 75% of that of B-ORAM. The improvement in bandwidth efficiency is attributed to two factors: (i) the query overhead of S-ORAM is only $2 \log N$ blocks while the overhead of B-ORAM is $\frac{2 \log^2 N}{\log \log N}$; and (ii) the shuffling algorithm of S-ORAM is more efficient than that of B-ORAM. In addition, the failure probability S-ORAM is $O(N^{-\log N})$, which is asymptotically lower than that of B-ORAM which is $O(N^{-\log \log N})$ [13].

### 5.3.2 S-ORAM vs. P-ORAM

To fairly compare the performance of S-ORAM and P-ORAM, their user-side storage sizes are both set to around $\log^2 N$ blocks and their failure probabilities are set to the same level, which are both $O(N^{-\log N})$. For this purpose, the security parameter $\omega(1)$ of P-ORAM has to be set to $\frac{\log^2 N}{\log(Z/\log N)}$, and the user-side storage size of P-ORAM is set to $\frac{\log^3 N}{\log(Z/\log N)} \cdot Z$ bits; the user-side storage size of S-ORAM is expanded to $\log^2 N \cdot Z$ bits. Note that, $\frac{\log^3 N}{\log(Z/\log N)} \cdot Z \geq \log^2 N \cdot Z$ as long as $Z \leq N$ (which is usually true in practice).

Table 2 shows the theoretical performances of both S-ORAM and P-ORAM and Table 3 is the practical performance comparison of these two ORAMs.

In Table 3, it can be seen that S-ORAM outperforms P-ORAM in both bandwidth efficiency and server-side storage efficiency. It requires 80% less server-side storage and consumes around 60% to 72% less bandwidth than P-ORAM.

# 6. CONCLUSION AND FUTURE WORK

In this paper, we propose a segmentation-based Oblivious RAM (S-ORAM). S-ORAM adopts two segment-based techniques, i.e., piece-wise shuffling and segment-based query, to improve the performance of shuffling and query by factoring block size into design. Extensive security analysis proves that S-ORAM is a highly secure solution with a negligible failure probability of $O(N^{-\log N})$. In terms of communication and storage overheads, S-ORAM outperforms the Balanced ORAM (B-ORAM) and the Path ORAM (P-ORAM), which are two state-of-the-art hash and index based ORAMs respectively, in both practical and theoretical evaluations.

In the future, we plan to study the de-amortization of S-ORAM, and completely implement the S-ORAM design.

## Acknowledgments

This work is supported partly by the ONR under grant N00014-09-1-0748 and by the NSF under grant EECS-1128312.

# 7. REFERENCES

[1] M. Ajtai, J. Komlos, and E. Szemeredi. An $O(n \log n)$ sorting network. In *Proc. STOC*, 1983.

[2] Amazon. http://aws.amazon.com/s3/. In *Amazon S3*, 2006.

| | $N = 2^{20}$ | | $N = 2^{36}$ | |
|---|---|---|---|---|
| | S-ORAM | P-ORAM | S-ORAM | P-ORAM |
| $P_b(Z = 32$ KB) | $0.394 \log^2 N \cdot Z$ | $1.170 \log^2 N \cdot Z$ | $0.456 \log^2 N \cdot Z$ | $1.247 \log^2 N \cdot Z$ |
| $P_b(Z = 64$ KB) | $0.334 \log^2 N \cdot Z$ | $1.090 \log^2 N \cdot Z$ | $0.456 \log^2 N \cdot Z$ | $1.157 \log^2 N \cdot Z$ |
| $P_b(Z = 128$ KB) | $0.334 \log^2 N \cdot Z$ | $1.021 \log^2 N \cdot Z$ | $0.392 \log^2 N \cdot Z$ | $1.079 \log^2 N \cdot Z$ |
| $P_b(Z = 256$ KB) | $0.259 \log^2 N \cdot Z$ | $0.959 \log^2 N \cdot Z$ | $0.392 \log^2 N \cdot Z$ | $1.011 \log^2 N \cdot Z$ |
| $P_c$ | $\log^2 N \cdot Z$ | $\frac{\log^3 N}{\log(Z/\log N)} \cdot Z$ | $\log^2 N \cdot Z$ | $\frac{\log^3 N}{\log(Z/\log N)} \cdot Z$ |
| $P_s$ | $< 6N \cdot Z$ | $32N \cdot Z$ | $< 6N \cdot Z$ | $32N \cdot Z$ |

**Table 3: Practical Performances: S-ORAM vs. P-ORAM**


[3] K. E.Batcher. Sorting networks and their applications. In *Proc. AFIPS*, 1968.

[4] A. O. Freier, P. Karlton, and P. C. Kocher. The secure sockets layer (SSL) protocol version 3.0. In *RFC 6101*, 2011.

[5] 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.

[6] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAM. *Journal of the ACM*, 43(3), May 1996.

[7] M. T. Goodrich. Randomized shellsort: a simple oblivious sorting algorithm. In *Proc. SODA*, 2010.

[8] M. T. Goodrich and M. Mitzenmacher. Mapreduce parallel cuckoo hashing and oblivious RAM simulations. In *Proc. CoRR*, 2010.

[9] M. T. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *Proc. ICALP*, 2011.

[10] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Oblivious RAM simulation with efficient worst-case access overhead. In *Proc. CCSW*, 2011.

[11] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *Proc. SODA*, 2012.

[12] M. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: ramification, attack and mitigation. In *Proc. NDSS*, 2012.

[13] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *Proc. SODA*, 2012.

[14] D.-L. Lee and K. E. Batcher. A multiway merge sorting network. *IEEE Transactions on Parallel and Distributed Systems*, 6(2), February 1995.

[15] B. Pinkas and T. Reinman. Oblivious RAM revisited. In *Proc. CRYPTO*, 2010.

[16] 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.

[17] E. Stefanov and E. Shi. ObliviStore: high performance oblivious cloud storage. In *Proc. S&P*, 2013.

[18] E. Stefanov, E. Shi, and D. Song. Towards practical oblivious RAM. In *Proc. NDSS*, 2011.

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

[20] P. Williams and R. Sion. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *Proc. CCS*, 2008.

[21] P. Williams, R. Sion, and A. Tomescu. PrivateFS: a parallel oblivious file system. In *Proc. CCS*, 2012.

[22] P. Williams, R. Sion, and A. Tomescu. Single round access privacy on outsourced storage. In *Proc. CCS*, 2012.

# Appendix

We present the details of the proposed m-way oblivious sorting algorithm in this Appendix.

As shown in Algorithm 3, to sort a set $\mathcal{D}$ of $n$ blocks, the m-way oblivious sorting algorithm works recursively as follows: if $n \leq 2m^2$, a *segment-sorting algorithm* similar to the segment-shuffling algorithm is applied to sort the $n$ blocks at the communication cost of $O(n)$ blocks; otherwise, the $n$ blocks are split into $m$ subsets each of $\frac{n}{m}$ blocks, the m-way oblivious sorting algorithm is applied to sort each of the subsets, and finally a *merging algorithm* is used to merge the sorted subsets into a sorted set of $n$ blocks.

Next, we describe the segment-sorting algorithm (Algorithm 4) and the merging algorithm (Algorithm 5) The segment-sorting algorithm is based on the segment-shuffling algorithm (Algorithm 2) with the following revisions: (1) The segment-sorting algorithm sorts blocks that are labeled with tags. The format of a labeled block is slightly different from the one shown in Figure 1; particularly, the encrypted tag is inserted as an extra piece before the encrypted block ID. (2) While the segment-shuffling algorithm can randomly pick a permutation function to shuffle pieces and blocks, the segment-sorting algorithm must permute pieces and blocks according to the non-decreasing order of tags. (3) The segment-sorting algorithm does not need to re-construct index blocks.

Finally, Algorithm 5 formally presents the merging algorithm.

**Algorithm 3** m-way Oblivious Sorting ($\mathcal{D}$: a set of data blocks)

**if** ($|\mathcal{D}| \leq 2m^2$) **then**
    Apply Algorithm 4 to sort $\mathcal{D}$
**else**
    Split $\mathcal{D}$ into $m$ equal-size subsets of blocks $\mathcal{D}_0, \cdots, \mathcal{D}_{m-1}$
    **for** each $i$ ($0 \leq i \leq m-1$) **do**
        Apply Algorithm 3 to sort $\mathcal{D}_i$
    **end for**
    Apply Algorithm 5 to merge $\mathcal{D}_0, \cdots, \mathcal{D}_{m-1}$
**end if**

**Algorithm 4** Segment-Sorting of Blocks $(D_{i_1}, \cdots, D_{i_n})$.

1-5: the same as in Algorithm 2
6: Construct a permutation function that sorts $B_2$ in the non-decreasing order
7: the same as in Algorithm 2
8: blank
9-14: the same as in Algorithm 2
15: **for** each $v \in \{2, \cdots, P\}$ **do**
16-26: the same as in Algorithm 2

**Algorithm 5** Merging Sorted-subsets of Blocks $(\mathcal{D}_0, \cdots, \mathcal{D}_{m-1})$

*/* Regroup blocks */*
$s = |\mathcal{D}_0|$
**for** each $i$ ($0 \leq i \leq m-1$) **do**
    **for** each $j$ ($0 \leq j \leq m-1$) **do**
        Add $\mathcal{D}_i[j], \mathcal{D}_i[m+j] \cdots, \mathcal{D}_i[s-m+j]$ to $\mathcal{D}'_j$
    **end for**
**end for**
*/* Recursively merge regrouped blocks */*
**for** each $j$ ($0 \leq j \leq m-1$) **do**
    **if** $|\mathcal{D}'_j| \leq 2m^2$ **then**
        Apply Algorithm 4 to sort $\mathcal{D}'_j$
    **else**
        Apply Algorithm 5 to merge sort $\mathcal{D}'_j$
    **end if**
**end for**
*/* Merge sorted blocks */*
**for** each $i$ ($0 \leq i \leq \frac{s}{m} - 2$) **do**
    **for** each $j$ ($0 \leq j \leq m-1$) **do**
        Add $\mathcal{D}'_j[im], \mathcal{D}'_j[im+1], \cdots, \mathcal{D}'_j[im+2m-1]$ to $\mathcal{D}''_i$
    **end for**
**end for**
**for** each $i$ ($0 \leq i \leq \frac{s}{m} - 1$) **do**
    Apply Algorithm 4 to sort $\mathcal{D}''_i$
**end for**