MSKT-ORAM: Multi-server K-ary Tree Oblivious RAM without Homomorphic Encryption

Jinsheng Zhang, Qiumao Ma, Wensheng Zhang, Daji Qiao
Iowa State University, Ames, IA, USA 50011
Email:{alexzjs,qmma,wzhang,daji}@iastate.edu

ABSTRACT

This paper proposes MSKT-ORAM, an efficient multiple server ORAM construction, to protect a client's access pattern to outsourced data. MSKT-ORAM organizes each of the server storage as a k-ary tree and adopts XOR based PIR and a novel delayed eviction technique to optimize both the data query and data eviction process. MSKT-ORAM is proved to protect the data access pattern privacy at a failure probability of 2^{-80} when $k \geq 128$. Meanwhile, given constant local storage, when N (i.e., the total number of outsourced data blocks) ranges from 2^{16} to 2^{34} and data block size $B \geq 20$ KB, the communication cost of MSKT-ORAM is only 22 to 46 data blocks. Asymptotical analysis and detailed implementation comparisons are conducted to show that MSKT-ORAM achieves better communication, storage and access delay in practical scenario over the compared state-of-the-art ORAM schemes.

1. INTRODUCTION

1.1 Motivations

With attractive cost-effectiveness, cloud storage services such as Amazon S3 and Dropbox have been popularly utilized by business and individual clients to host their data. Before exporting sensitive data to the cloud storage, clients can encrypt it if they do not trust the storage server. However, data encryption itself is not sufficient for data security, because the secrecy of data can still be exposed if a client's access pattern to the data is revealed [16].

The private information retrieval (PIR) [1–4, 8, 9, 15, 18, 19, 31] and the oblivious RAM (ORAM) [7, 10–14, 17, 25–30, 32–35] are two well-known security-provable approaches to protect a client's access pattern to the data stored in remote storage. While some studies indicate that the PIR schemes may be infeasible for large-scale data sets as they need to process the whole data set in order to hide just one data request [33], the ORAM approaches still appear to be promising as more and more resource-efficient constructions have been proposed. Particularly, communication cost is the most important metric to evaluate the feasibility of an ORAM construction. In the literature, the most communication-efficient ORAM constructions are C-ORAM [24] and CNE-ORAM [23] proposed by Moataz et al. both consuming $O(B)$ bandwidth for each data query, when the total number of exported data blocks is N and each data block is of size $B \geq N^{\epsilon}$ bits for some constant $0 < \epsilon < 1$.

Though C-ORAM and CNE-ORAM have achieved better communication efficiency than prior works, further reducing the requirement of data block size and the query delay is still desirable to make the ORAM construction more feasible to implement in cloud storage systems.

1.2 Assumption Ambiguity of Existing ORAMs

In state-of-the-art ORAMs using tree data structure on server side, there is always an assumption ambiguity on the size of data blocks in the scheme. In tree based ORAM schemes, data block size is usually assumed to be $O(\log^{\alpha} N)$ bits, where $\alpha \geq 2$. However, these ORAM further assume the size of a data block is $O(N^{\epsilon})$ where $0 < \epsilon < 1$, since the number of recursions on the index of the data blocks need to be constant such that the communication cost of these scheme can save a $O(\log N)$ factor.

Let's take one step further on these two assumptions of data block size. In a practical scenario, where $2^{16} \leq N \leq 2^{34}$, given $\alpha = 3$ and $\epsilon = 0.4$, $N^{\epsilon} < \log^{\alpha} N$ always holds. This means the data block size required to make the recursion depth to be $O(1)$ is dominated by the requirement of a normal data block size in these schemes. Theoretically, if the size of the data block is set to be $O(\log^{\alpha} N)$, the recursion depth is $O(\log N / \log \log N)$. Therefore, the above two assumptions are not consistent with each other. In this paper, we assume that $B = O(N^{\epsilon})$ without causing the above ambiguity on data block size.

1.3 Our Results

In this paper, we propose a new ORAM construction, named MSKT-ORAM, to accomplish the following performance goals in practical scenario:

- *Communication efficiency* - Under a practical scenario, the communication cost per data query is about 22 blocks to 46 blocks when $2^{16} \leq N \leq 2^{34}$ and the block size $B \geq N^{\epsilon}$ bits for some constant $0 < \epsilon < 1$. In practice, this is lower or comparable to constant communication ORAM construction C-ORAM and CNE-ORAM. In MSKT-ORAM, there is no server-server communication cost.

- *Low access delay* - Compared to both C-ORAM and CNE-ORAM schemes with constant client-server communication cost, MSKT-ORAM has a low access latency.

- *Small data block size requirement* - Compared to C-ORAM and CNE-ORAM, MSKT-ORAM only requires each data block size $B \geq 20$ KB.

- *Constant storage at the client* - MSKT-ORAM only requires the client storage to store a constant number of data blocks while each server needs to store $O(N \cdot B)$ data blocks.

- *Low failure probability guarantee* - MSKT-ORAM is proved to achieve 2^{-80} failure probability given $k \geq 128$.

1.4 Our Methodology

Similar to P-PIR [22], MSKT-ORAM organizes the cloud storage as a tree with each node acting as fully-functional PIR, where the PIR-read and PIR-write primitives are implemented based on additive homomorphic (AH) operations (i.e., AH encryption, decryption, addition and multiplication). By applying PIR primitives, the communication cost, which is affected by both the height of tree and the size of each tree node in Tree ORAM [26], becomes *determined only by the height of tree*, because only one data block is transferred from/to each accessed tree node. Meanwhile, the PIR primitives can be performed efficiently because they process only a small fraction of the data set stored on the tree.

Different from P-PIR, MSKT-ORAM uses a k-ary tree instead of a binary tree, in order to reduce the height of tree by a factor of $O(\log k)$ and thus *reduce the communication cost also by $O(\log k)$ times*. Note that, Gentry et al. [7] also have used k-ary tree in their designed ORAM construction which we call G-ORAM hereafter. However, MSKT-ORAM and G-ORAM use different eviction algorithms to evict actual data blocks within the tree. This difference has contributed to the different levels of communication efficiency that they can accomplish: when the number of branches k is set to $\log N$, G-ORAM requires to transfer $O(\frac{\log^2 N}{\log\log N})$ data blocks per query with $O(\log^2 N) \cdot \omega(1)$ data blocks in local storage, while MSKT-ORAM only requires to transfer $O(1)$ blocks with constant data blocks in local storage.

MSKT-ORAM adopts an eviction algorithm different from the one used in G-ORAM based on the following observations: The AH operation-based PIR primitives can help to reduce communication cost if, in an eviction process, only a small number of the data blocks need to be obliviously moved out/in from/to a tree node. For example, in P-PIR, at most one out of $O(\log N)$ data blocks is moved out/in from/to each node selected for eviction; the PIR primitives are employed to obliviously download/upload one block from/to each selected node, without transferring all blocks from/to these nodes. However, if the G-ORAM eviction algorithm is applied, the number of blocks moved out/in from/to a node could be as large as $O(\log N)$, which is the total number of blocks in a node; hence, to obliviously perform these movements, all the blocks in the node may have to be downloaded and re-uploaded, even if the PIR primitives are used.

In comparison, we design a novel eviction algorithm in MSKT-ORAM that moves only one data block out/in from/to each selected tree node during an eviction process. Specifically, the algorithm treats the physical k-ary tree as a logical binary tree, where each k-ary tree node is mapped to a logical binary subtree. Over the logical tree, a binary tree-based eviction algorithm, similar to the one used in Tree ORAM [26] and P-PIR [22], is logically simulated but not directly executed. Instead, an idea of *delayed eviction* is employed to defer and aggregate as many as possible the logical eviction operations to ensure that: (i) for each eviction process, only $O(1)$ k-ary tree nodes need to be accessed; (ii) within each of the accessed k-ary tree node, only one data block needs to be downloaded/uploaded.

1.5 Organization of the Paper

In the rest of the paper, Section 2 presents the problem definition. Section 3 and 4 presents our proposed scheme. Section 5 and 6 reports the security and cost analysis. Section 7 makes a detailed comparison between our proposed scheme with existing ORAMs and Section 8 briefly reviews the related work. Finally, Section 9 concludes the paper.

2. PROBLEM DEFINITION

We consider a system as follows. A client exports N equal-size data blocks to two remote storage servers, where the two servers do not collude with each other. Note that, such an architecture is feasible in practice, since the client can select two servers in a way that they will not know the existence of each other. For example, a client can simply select Amazon S3 and Google Drive as two independent storage servers. Each of the two servers has an identical copy of the data storage. The client accesses the exported data every now and then, and wishes to hide the pattern of the accesses from the server.

Each data request from the client, which should be kept private, is one of the following two types: (i) read a data block D of unique ID i from the storage, denoted as a 3-tuple $D = (read, i)$; or (ii) write/modify a data block D of unique ID i to the storage, denoted as a 3-tuple $(write, i, D)$. To simplify the presentation in the security definition, we denote these two types of requests as (op, i, D) where *op* is $read$ or $write$.

To accomplish a private data request, the client needs 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: (i) retrieve (i.e., read) a data block D from a location l at the remote storage, denoted as a 3-tuple $D = (read, l)$; or (ii) upload (i.e., write) a data block D to a location l at the remote storage, denoted as a 3-tuple $(write, l, D)$. Similarly, we denote these two access types as (op, l, D) where *op* is $read$ or $write$.

We assume the client is trusted but the remote server is honest but curious; that is, it stores data and serves the client's requests according to the protocol that we deploy, but it may attempt to figure out the client's access pattern. The network connection between the client and the server is assumed to be secure; in practice, this can be achieved using well-known techniques such as SSL [6].

Following the security definition of ORAMs [10,29,30], we define the security of our proposed ORAM as follows.

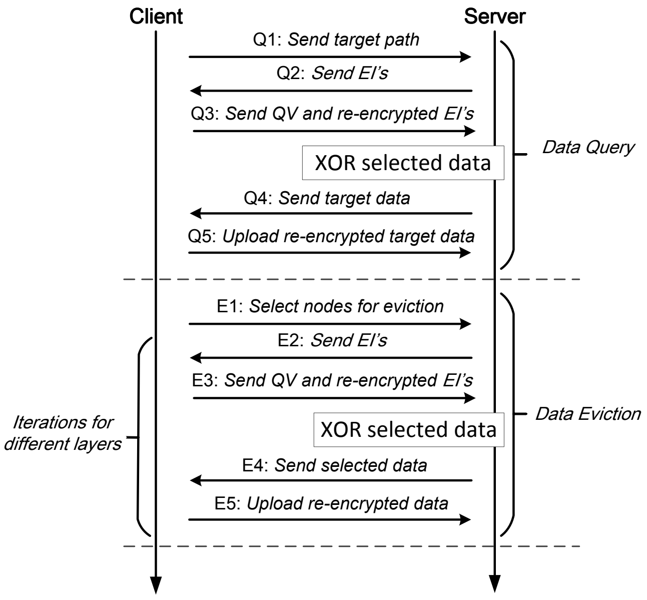
Definition Let $\vec{x} = \langle\ (op_1, i_1, D_1), (op_2, i_2, D_2), \cdots\ \rangle$ denote a private sequence of the client's intended data requests, where each *op* is either a $read$ or $write$ operation. Let $A(\vec{x}) = \langle\ (op'_1, l_1, D'_1), (op'_2, l_2, D'_2), \cdots\ \rangle$ denote the sequence of the client's accesses to the remote storage (observed by the server), in order to accomplish the client's private 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 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 bounded by $2^{-\lambda}$ where λ is a security parameter.

3. FIRST CONSTRUCTION: MSBT-ORAM

Before we present our proposed scheme, we first present a preliminary scheme: MSBT-ORAM (Multi-server Binary Tree ORAM). MSBT-ORAM follows the framework of tree-based ORAMs, especially, P-PIR [22] and aims to improve the access delay of P-PIR [22]. According to the observation that the access delay in P-PIR is mainly caused by the expensive homomorphic operations, MSBT-ORAM replaces the homomorphic encryptions with XOR operations on the server, thus will leverage servers from heavy computation tasks.

The design of MSBT-ORAM includes storage organization, data query, and data eviction. The basic work flow of the MSBT-ORAM is shown in Figure 1.



Client
Server
Q1: Send target path
Q2: Send EI's
Q3: Send QV and re-encrypted EI's
XOR selected data
Q4: Send target data
Q5: Upload re-encrypted target data
Data Query
E1: Select nodes for eviction
E2: Send EI's
E3: Send QV and re-encrypted EI's
XOR selected data
E4: Send selected data
E5: Upload re-encrypted data
Data Eviction
Iterations for different layers

Figure 1: The working flow of MSBT-ORAM. This figure shows how MSBT-ORAM works between the client and the servers.

3.1 Storage Organization

In the design, we require two servers denoted as $\mathcal{S}_1$ and $\mathcal{S}_2$. Since the two servers are almost identical to each other, we only describe how storage is organized in $\mathcal{S}_1$ and call $\mathcal{S}_1$ the "server". Later, we will show the differences between $\mathcal{S}_1$ and $\mathcal{S}_2$.

Assuming N data blocks are exported by the client to the server. The server storage is organized as a binary tree with $L = \log N + 1$ layers, the same as in T-ORAM [26] and P-PIR [22]. Each node can store $3c \log N$ blocks, where c is a system parameter related to the security parameter λ. As the capacity of the storage is larger than the N *real* data blocks, *dummy* blocks are added to fill up the rest of the storage. A real data block is encrypted with symmetric encryption such as AES [5] before it is stored to a node in the binary tree; that is, each data block d_i is stored as $D_i = E(d_i)$ in a node. Each node also contains a symmetrically encrypted index block that records the ID of the data block stored at each position of the node; as the block is encrypted, the index information is not known to the server.

Figure 2 shows an example, where $N = 32$ data blocks are exported and stored in a binary tree-based storage with 6 layers. Starting from the top layer, i.e., layer 0, each node is denoted as $v_{l,i}$, where l is the layer index and i is the node index on the layer.

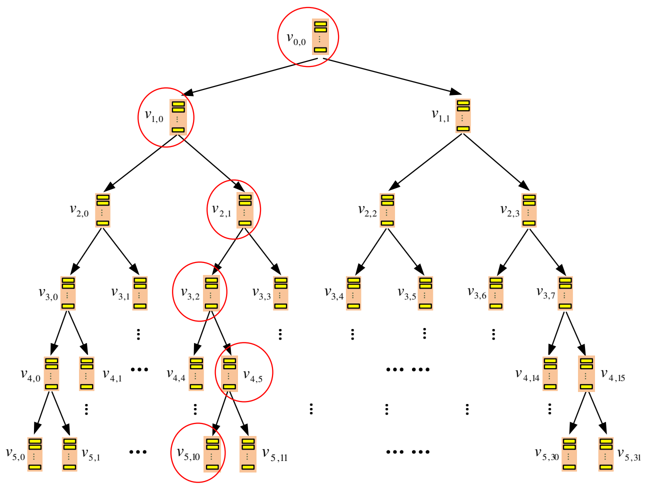


Figure 2: MSBT-ORAM's server-side storage structure. Circled nodes represent the ones accessed by the client during a query process when the target data block is mapped to leaf node $v_{5,10}$.

MSBT-ORAM requires the client to maintain an index table with N entries, where each entry i ($i \in \{0, \cdots, N-1\}$) records the ID of a leaf node on the tree such that data block D_i is stored at some node on the path from the root to this leaf node. As in T-ORAM [26] and P-PIR [22], the index table can be exported to the server as well; hence, the client-side storage is of constant size and only needs to store at most two data blocks and some secret information such as encryption keys.

3.2 Data Query Process

To query a certain data block D_t, the client acts as follows:

- The client checks the index table to find out the leaf node $v_{L-1,f}$ that D_t is mapped to. Hence, a path from the root to $v_{L-1,f}$ is identified. To facilitate presentation, we denote the selected path as follows:

$$\overrightarrow{v} = (v_0, \cdots, v_{L-1}). \tag{1}$$

- For each node v_l ($0 \le l \le L-1$) on the selected path $\overrightarrow{v}$, the client first retrieves the encrypted index block from it, and checks if D_t is in the node. Then, two query bit vectors $\overrightarrow{Q}_1^l$ and $\overrightarrow{Q}_2^l$ are generated by the client, where $|\overrightarrow{Q}_1^l| = |\overrightarrow{Q}_2^l| = c \log N$. Suppose D_t is at a certain position m of the node, the two bit vectors are set as follows:

$$\begin{aligned} \overrightarrow{Q}_1^l &= (r_1, r_2, \cdots, r_m, \cdots, r_{c \log N}), \\ \overrightarrow{Q}_2^l &= (r_1, r_2, \cdots, \overline{r_m}, \cdots, r_{c \log N}), \end{aligned} \tag{2}$$

where $\overline{r_m}$ means the negation of r_m and each r_i denotes a random bit selected by the client from $\{0, 1\}$. (Note that if D_t is not in this node, the two vectors will be the same.) After that, $\overrightarrow{Q}_1^l$ is sent to $\mathcal{S}_1$ and $\overrightarrow{Q}_2^l$ is sent to $\mathcal{S}_2$.

Server $\mathcal{S}_j$ $j \in \{0, 1\}$ computes:

$$\hat{D}_j = \bigoplus_{0 \leq l \leq L-1} \left(\bigoplus D_i^l \right), \forall D_i^l : \overrightarrow{Q_j^l}[i] = 1. \tag{3}$$

After the server $\mathcal{S}_j$ executes the computation, the calculated data block is returned to the client. The client computes $\hat{D}_0 \bigoplus \hat{D}_1$ to get the query target data block. After the target block has been accessed, it will be re-encrypted by the client and uploaded to the root node of each server.

An example is given in Figure 2, where the query target D_t is mapped to leaf node $v_{5,10}$. Hence, each node on the path $v_{0,0} \rightarrow v_{1,0} \rightarrow v_{2,1} \rightarrow v_{3,2} \rightarrow v_{4,5} \rightarrow v_{5,10}$ is involved in data query. Finally, data block D_t is found at node $v_{5,10}$. After being accessed, it is re-encrypted and added to root node $v_{0,0}$.

3.3 Data Eviction Process

To prevent any node on the tree from overflowing, the following data eviction process is conducted by the client in each server after each query.

The basic idea of data eviction is as follows: For each non-bottom layer l, two nodes $v_{l,x}$ and $v_{l,y}$ are randomly selected and the similar operations are executed on each node. In the following, we will elaborate the eviction operation using $v_{l,x}$ as an example. Note that, $v_{l+1,2x}$ and $v_{l+1,2x+1}$ are the children nodes of $v_{l,x}$.

The obliviousness of the data eviction process in the MSBT-ORAM involves the following two aspects:

- For each evicting node (e.g. $v_{l,x}$), the position where the evicted data block reside should be hidden from any server.
- For each receiving node (e.g. $v_{l+1,2x}$), each position that can be used to receive the evicted data block should be selected with an equal probability. In other words, each position of the receiving node has an equal probability to be written during data eviction.

Therefore, to perform the eviction operation, the client first obliviously retrieves an evicting data block D from node $v_{l,x}$. If there is at least one real data block in this node, D is a randomly selected real data block, otherwise, D is a dummy data block. After D has been retrieved by the client, the client re-encrypts D and upload it to the server. The uploading process has the following two cases:

- D is a real data block. Without loss of generality, suppose D needs to be evicted to $v_{l+1,2x}$, a dummy block D' needs to be evicted to $v_{l+1,2x+1}$ to achieve data eviction obliviousness. To reduce the communication cost, D acts as the dummy data block D' when evicting to $v_{l+1,2x+1}$.
- D is a dummy data block. Then, D will be evicted to both $v_{l+1,2x}$ and $v_{l+1,2x+1}$ as a dummy data block.

Next, we explain how data blocks are evicted from parent node to its children. First, the node storage structure is introduced as follows: For each node, the node storage is logically partitioned into 3 equal-size parts, i.e., P_1, P_2 and P_3. Each part can store $c \log N$ data blocks:

- P_1: this part is a storage space that is used to receive the latest evicted data blocks from its parent.
- P_2: this part is used to store real data blocks that still remains in this node after more than $c \log N$ evictions on this node have occurred since these real data blocks were evicted to this node. Note that, as the number of real data blocks stored in any node is at most $c \log N$, this part may contain dummy blocks.
- P_3: the remaining storage space in the node. This part only contains dummy data blocks that could be used to perform real data block receiving from the eviction procedure of its parent or dummy data eviction to its children nodes.

Note that, the servers can only distinguish P_1 and the rest of the storage of any node; P_2 and P_3 are known only to the client. Also, P_1, P_2 and P_3 are only logical partitions and any position of this node will belong to different partitions from time to time. For example, when one position in P_2 or P_3 is used to accept a data block evicted from the parent node to this node, this position is transferred from P_2 or P_3 to P_1; meanwhile, the oldest position in P_1 is automatically transferred from P_1 to P_2 or P_3.

Based on the above logical partitioning, the client applies the following rules to evict D to $v_{l+1,2x}$ (the eviction of D to $v_{l+1,2x+1}$ follows the same rule):

- If D is a real data block, there are two sub-cases:
  - D is intended to be evicted to $v_{l+1,2x}$. Thus, the data block that will be evicted to $v_{l+1,2x+1}$ is a dummy block. In this case, each position $w \in P_3$ will have equal probability to be selected such that D will be evicted to w.
  - D is not intended to be evicted to $v_{l+1,2x}$. Hence, the data block that will be evicted to $v_{l+1,2x+1}$ is the real data block. In this case, one position $w \in P_2$ will be uniformly randomly selected and D will be evicted to w.
- If D is a dummy data block, each position $w \in P_2 \cup P_3$ will have equal probability to be selected such that D will be written to.

After the above operations, w is transferred to P_1. Meanwhile, the oldest position $w' \in P_1$ becomes a member of P_2 or P_3 depending on if it contains a real block or a dummy block.

We prove in Section 5 that, each position in $P_2 \cup P_3$ has the same probability to be selected to access during an eviction process; thus, the eviction process is oblivious and does not reveal the access pattern.

Figure 3 shows an example of the eviction process, where circled nodes are selected to evict data blocks to their child nodes. Let us consider how node $v_{2,2}$ evicts its data block. The index block in the node is first retrieved to check if the node contains any real data block. If there is a real block D_e in $v_{2,2}$ and D_e is mapped to leaf node $v_{5,20}$, D_e will be obliviously evicted to $v_{3,5}$, which is $v_{2,2}$'s child and is on path from $v_{2,2}$ to $v_{5,20}$, while a dummy eviction is performed to another child node $v_{3,4}$. Otherwise, two dummy evictions will be performed to nodes $v_{3,4}$ and $v_{3,5}$.

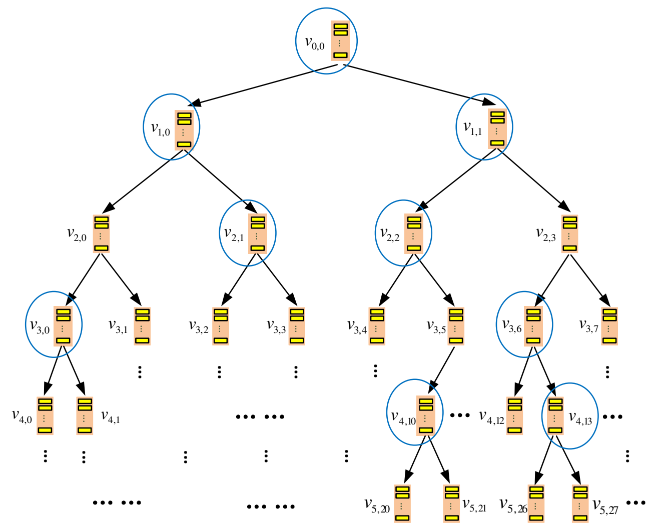


Figure 3: An example of the eviction process in MSBT-ORAM.

4. FINAL CONSTRUCTION: MSKT-ORAM

Our proposed MSKT-ORAM aims to improve the communication efficiency and further reduce the access delay. The idea of MSKT-ORAM is to replace the binary tree in MSBT-ORAM with k-ary tree to reduce the height of the tree. As the tree height is decreased, the communication cost for each query will be reduced therefore the access delay. However, directly replacing the binary tree to k-ary tree requires the data eviction process to change accordingly to ensure the obliviousness (i.e., the overflow probability of each node should be bounded by $2^{-\lambda}$). Hence, we propose new data eviction algorithm.

This section presents the details of the proposed MSKT-ORAM design in terms of storage organization, system initialization, data query process, and data eviction process.

4.1 Storage Organization

At each server, the data storage is physically organized as a k-ary tree of height $H_k = \lceil \frac{\log N+1}{\log k} \rceil$. For simplicity, we set the system parameter k to be a power of two. Each k-node is mapped to a binary subtree with $k-1$ binary nodes (called b-nodes). As shown in Figure 4, each k-node $u_{l,i}$ has the following components:

- *Data Array (DA)*: a data container that stores $3c(k-1)$ data blocks, where $c = 4$ is a system parameter.
- *Encrypted Index Table (EI)*: a table of $3c(k-1)$ entries recording the information for each block stored in the DA. Specifically, each entry is a tuple of format $(ID, lID, bnID)$, which records the following information of each block:
  - ID - ID of the block;
  - lID - ID of the leaf k-node that the block is mapped to;
  - $bnID$ - ID of the b-node (within $u_{l,i}$) that the block logically belongs to.

  In addition, the EI has a ts field which stores a timestamp indicating when this k-node was accessed last time.

For example, k-node $u_{0,0}$ in Figure 4(a) is mapped to the binary subtree with $v_{0,0}$ as root, and $v_{1,0}$ and $v_{1,1}$ as leaves in Figure 4(b). This way, the physical k-ary tree can be treated as a logical binary tree. Note that, all the b-nodes within the same k-node share the storage space (i.e., DA).

4.2 Client-side Storage

At the client side, the following storage structures are maintained:

- *A client-side index table $\mathcal{I}$*: a table of N entries, where each entry i records the ID of the leaf k-node that data block D_i is mapped to (i.e., block D_i is stored at some node on the path from the root to this k-node). In practical implementation of MSKT-ORAM, the table can be exported to the server, just as in T-ORAM [26] and P-PIR [22]; to simplify presentation of the design in this section, however, we assume the table is maintained locally at the client side. Note that, similar to Path ORAM [30] and SCORAM [32], outsourcing the index table of $O(N \log N)$ bits with a uniform block size of $B = N^{\epsilon}$ bits can ensure the metadata recursion to be of $O(1)$ depth $(0 < \epsilon < 1)$.
- *A constant-size temporary buffer*: a buffer used to temporarily store a constant number of blocks downloaded from the server-side storage.
- *A small permanent storage for secrets*: a permanent storage to store the client's secrets such as the keys used for the encryption and decryption of data and index tables.
- $\mathcal{C}$: a counter counting the number of queries that the client has issued to the server.

4.3 System Initialization

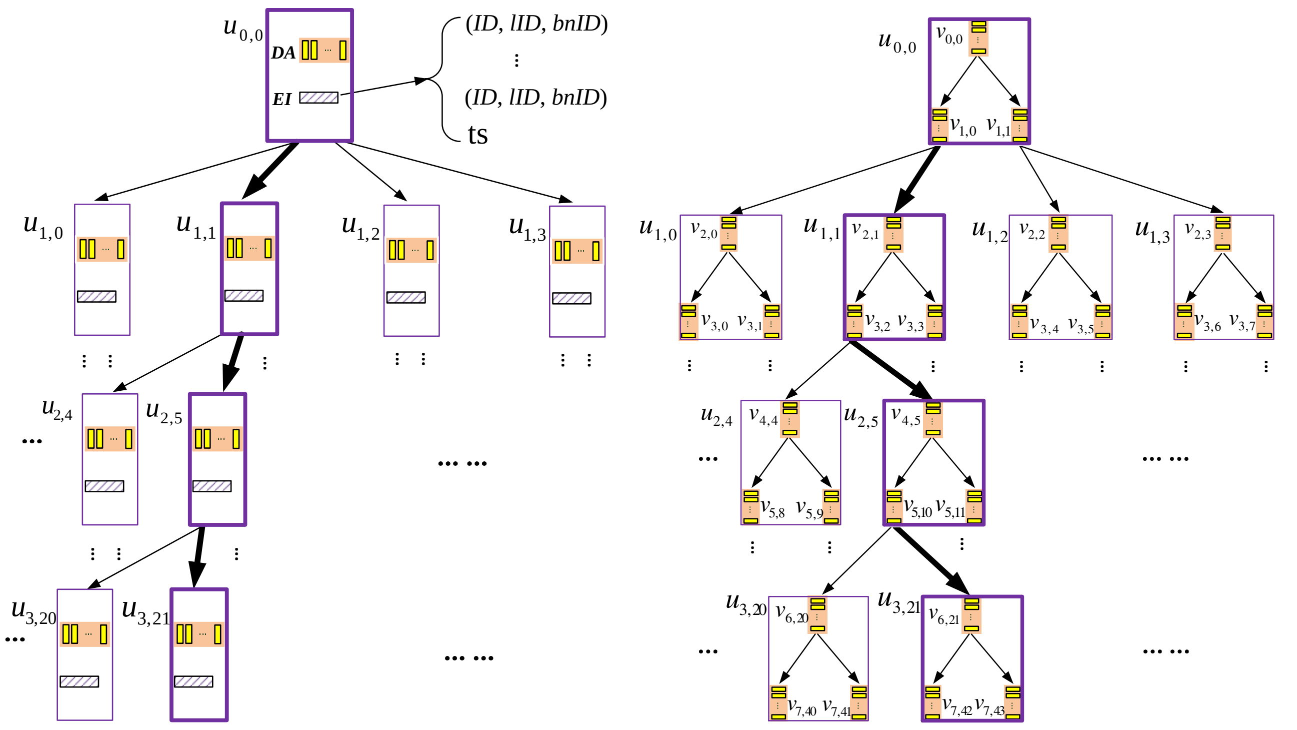
To initialize the system, the client acts as follows. It first encrypts each real data block d_i to D_i, and then randomly assigns it to a leaf k-node on the k-ary tree maintained at each server. The rest of the DA spaces on the tree shall all be filled with dummy blocks.

For each k-node, its EI entries are initialized to record the information of blocks stored in the node. Specifically, the entry for a real data block shall record the block ID to the ID field, the ID of the assigned leaf k-node to the lID field, and the ID of an arbitrary leaf b-node within the k-node to the $bnID$ field. In an entry for a dummy data block, the block ID is marked as "-1" while lID and $bnID$ fields are filled with arbitrary values. The ts field of the EI shall be initialized to 0.

For the client-side storage, the index table $\mathcal{I}$ is initialized to record the mapping from real data blocks to leaf k-nodes, and the keys for data and index table encryption are also recorded to a permanent storage space. Finally, the client initializes its counter $\mathcal{C}$ to 0.

4.4 Data Query

To query a data block D_t with ID t, the client increments the counter $\mathcal{C}$, searches the index table $\mathcal{I}$ to find out the leaf k-node that D_t is mapped to, and then, for each k-node u on the path from the root k-node to this leaf node, the XOR operations similar to those in MSBT-ORAM are performed to retrieve D_t. The only difference is that the query bit vector size of MSKT-ORAM is $3c(k-1)$ bits per vector. As shown in Figure 4(a), to query a data block D_t stored at k-node $u_{3,21}$, the EIs at $u_{0,0}$, $u_{1,1}$, $u_{2,5}$, and $u_{3,21}$ are accessed, as these k-nodes are on the path from the root to the leaf node that D_t is mapped to. A dummy data block is retrieved obliviously from $u_{0,0}$, $u_{1,1}$, and $u_{2,5}$, respectively, while D_t is retrieved obliviously from $u_{3,21}$ and inserted obliviously into $u_{0,0}$.


$u_{0,0}$
DA
EI
(ID, lID, bnID)
(ID, lID, bnID)
ts
$u_{1,0}$
$u_{1,1}$
$u_{1,2}$
$u_{1,3}$
$u_{2,4}$
$u_{2,5}$
$u_{3,20}$
$u_{3,21}$
$v_{0,0}$
$v_{1,0}$ $v_{1,1}$
$v_{2,0}$ $v_{2,1}$ $v_{2,2}$ $v_{2,3}$
$v_{3,0}$ $v_{3,1}$ $v_{3,2}$ $v_{3,3}$ $v_{3,4}$ $v_{3,5}$ $v_{3,6}$ $v_{3,7}$
$v_{4,4}$ $v_{4,5}$
$v_{5,8}$ $v_{5,9}$ $v_{5,10}$ $v_{5,11}$
$v_{6,20}$ $v_{6,21}$
$v_{7,40}$ $v_{7,41}$ $v_{7,42}$ $v_{7,43}$

(a) quaternary tree: physical view of the server storage (b) binary tree: logical view of the server storage

Figure 4: An example MSKT-ORAM scheme with a quaternary-tree storage structure. Bold boxes represent the k-nodes accessed when a client queries a target data block stored at k-node $u_{3,21}$.

4.5 Data Eviction

In each of the servers, to prevent a k-node from overflowing its DA, real data blocks should be gradually evicted from the root k-node towards leaf k-nodes. Similar to T-ORAM and P-PIR, a data eviction process is launched in MSKT-ORAM immediately after each query.

4.5.1 Overview

Data eviction in MSKT-ORAM is conducted over its logical binary tree. In a nutshell, two nodes from each layer of the logical binary tree are randomly selected. In each of the selected b-nodes, if there is a real data block, the block will be evicted to one of its child b-nodes according to the block's path (i.e., the path from the root to the leaf k-node whose ID is stored at lID field of the EI entry corresponding to this block); otherwise, the client performs a dummy eviction. Note that, immediate execution of all of these binary-tree evictions would require the client to access $O(\log N)$ blocks, which incurs the same eviction cost as P-PIR. To reduce the cost, we delay and aggregate certain evictions, and execute them later in a more efficient manner. The idea is developed based on the observation that there are two types of evictions between b-nodes: *intra k-node evictions* and *inter k-node evictions*.

Intra k-node Evictions vs. Inter k-node Evictions An eviction is called an *intra k-node eviction* if the data block is evicted between b-nodes that belong to the same k-node; else it is called an *inter k-node eviction*. For example in Figure 5, the eviction from $v_{2,2}$ to its child nodes is an intra k-node eviction, as $v_{2,2}$ and its child nodes belong to the same k-node $u_{1,2}$. On the other hand, the eviction from $v_{3,2}$ to its child nodes is an inter k-node eviction, as $v_{3,2}$ and its two child nodes belong to different k-nodes.

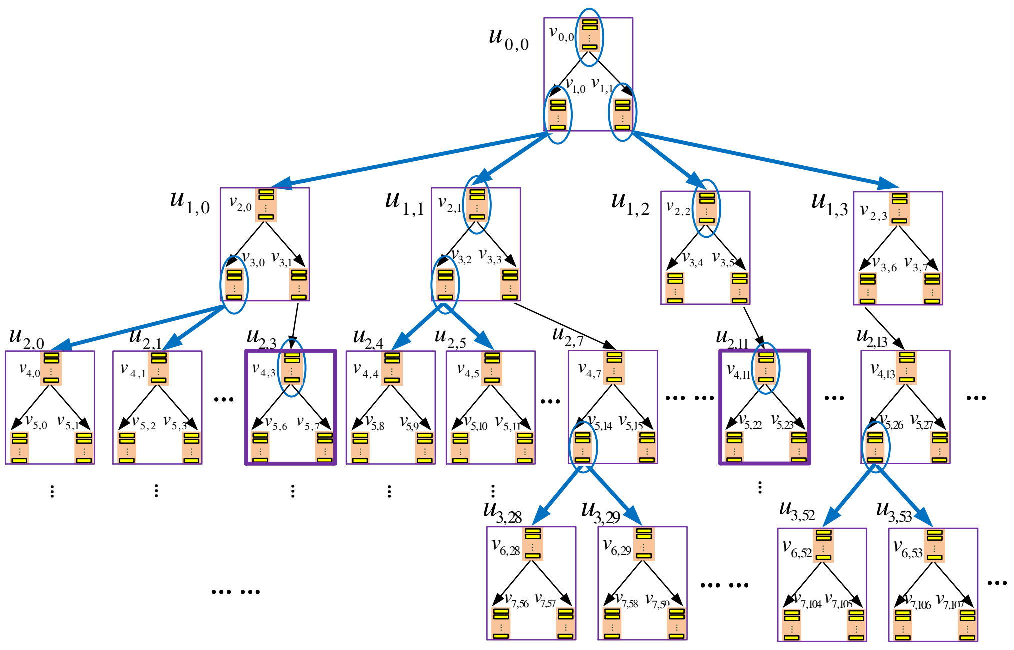
As b-nodes within the same k-node share the same DA space for storing data blocks, an intra k-node eviction only requires an update of the EI to reflect the change of $bnID$ field for the evicted block. Therefore, such an eviction does not incur any communication overhead and thus could be performed more efficiently than inter k-node evictions.

Opportunities to Delay Intra k-node Evictions During a data eviction process, a k-node may not be involved in any inter k-node evictions, i.e., its root b-node is not a child of any evicting b-node meanwhile its own leaf b-nodes do not evict any data blocks. In Figure 5, $u_{2,3}$ and $u_{2,11}$ are two examples of such a k-node. If intra k-node evictions should occur in such a k-node, they can be delayed to perform (i.e., to update the EI) later when the k-node is next accessed during a query process or an inter k-node eviction. This is possible because the EI of the k-node is not accessed until the k-node is next accessed. Moreover, since the client has to download the EI of the k-node anyway during a query process or an inter k-node eviction, updating of the EI to complete delayed intra k-node evictions does not cause any additional communication overhead, thus reducing the eviction cost. For example, as shown in Figure 5, evictions from b-nodes $v_{4,3}$ and $v_{4,11}$ can be delayed. Later on, when $u_{2,3}$ and $u_{2,11}$ are accessed, the delayed evictions shall be executed before any other updates.

More specifically, the eviction process is composed of the following three phases.

4.5.2 Phase I: Selecting k-nodes for Eviction

At the beginning of an eviction process, the client uniform randomly selects two b-nodes from each binary-tree layer l ($l \in \{\log k-$



$u_{0,0}$
$v_{0,0}$
$v_{1,0}$ $v_{1,1}$
$u_{1,0}$ $v_{2,0}$ $v_{3,0}$ $v_{3,1}$
$u_{1,1}$ $v_{2,1}$ $v_{3,2}$ $v_{3,3}$
$u_{1,2}$ $v_{2,2}$ $v_{3,4}$ $v_{3,5}$
$u_{1,3}$ $v_{2,3}$ $v_{3,6}$ $v_{3,7}$
$u_{2,0}$ $v_{4,0}$ $v_{5,0}$ $v_{5,1}$
$u_{2,1}$ $v_{4,1}$ $v_{5,2}$ $v_{5,3}$
$u_{2,3}$ $v_{4,3}$ $v_{5,6}$ $v_{5,7}$
$u_{2,4}$ $v_{4,4}$ $v_{5,8}$ $v_{5,9}$
$u_{2,5}$ $v_{4,5}$ $v_{5,10}$ $v_{5,11}$
$u_{2,7}$ $v_{4,7}$ $v_{5,14}$ $v_{5,15}$
$u_{2,11}$ $v_{4,11}$ $v_{5,22}$ $v_{5,23}$
$u_{2,13}$ $v_{4,13}$ $v_{5,26}$ $v_{5,27}$
$u_{3,28}$ $v_{6,28}$ $v_{7,56}$ $v_{7,57}$
$u_{3,29}$ $v_{6,29}$ $v_{7,58}$ $v_{7,59}$
$u_{3,52}$ $v_{6,52}$ $v_{7,104}$ $v_{7,105}$
$u_{3,53}$ $v_{6,53}$ $v_{7,106}$ $v_{7,107}$

Figure 5: An example data eviction process in MSKT-ORAM with a quaternary-tree storage structure. The b-nodes selected to evict data blocks are circled. The k-nodes scheduled with delayed evictions (i.e., $u_{2,3}$ and $u_{2,11}$) are highlighted with bold boundaries.

$1, 2\log k - 1, \cdots, (\lceil \frac{\log N+1}{\log k} \rceil - 1) \cdot \log k - 1\})$. These layers are the bottom binary subtree layers inside non-bottom k-nodes. Therefore, evicting data blocks from the b-nodes on these layers to their child b-nodes are inter k-node evictions and shall be executed immediately. The k-nodes that contain these selected b-nodes or their child b-nodes shall be processed as specified in Phases II and III. For example, in Figure 5, b-nodes $v_{3,0}$ and $v_{3,2}$, which are on the bottom-layer of k-nodes $u_{1,0}$ and $u_{1,1}$ respectively, are selected for binary tree eviction; hence, k-nodes $u_{1,0}$, $u_{1,1}$, $u_{2,0}$, $u_{2,1}$, $u_{2,4}$ and $u_{2,5}$, which contain either the selected b-nodes or their child b-nodes, shall be processed in the follow-up phases.

Note that, we delay the selection of evicting b-nodes on other layers, as their evictions are intra k-node evictions and can be delayed.

4.5.3 *Phase II: Execution of Delayed Intra k-node Evictions*

For each b-node selected in Phase I, the three k-nodes that contain this selected b-node or its child b-nodes shall execute their delayed intra k-node evictions in this phase. Also, each k-node downloaded during the query process, as elaborated in Section 4.4, shall also execute its delayed intra k-node evictions as follows.

Specifically, for each of these k-nodes, say, $u_{l,i}$, the following operations shall be conducted. First, the client retrieves and decrypts the EI of this k-node to obtain the ts stored there. The difference between the client's current counter $\mathcal{C}$ and the value of the ts, i.e., $\mathcal{C} - ts$, is the number of eviction rounds for which this k-node has delayed its intra k-node evictions.

Then, for each of these delayed eviction rounds, two b-nodes are uniform randomly picked from each layer l' ($l' \in \{l \cdot \log k, l \cdot \log k + 1, \cdots, (l+1) \cdot \log k - 2\}$) of the whole logical binary tree that the k-ary tree is mapped to. For each selected b-node $v_{l',i'}$ belonging to the binary subtree that k-node $u_{l,i}$ is mapped to, eviction from this b-node to its child nodes is executed. Specifically, a real data block d is randomly selected from $v_{l',i'}$ if the b-node has a real data block; then, according to the lID of block d, the block is logically evicted to one of its child b-nodes, say, $v_{l'+1,j'}$, which is done by changing the $bnID$ of block d to the ID of $v_{i'+1,j'}$.

After all the delayed intra k-node evictions have been executed, the ts of k-node $u_{l,i}$ is updated to $\mathcal{C}$.

4.5.4 *Phase III: Execution of Inter k-node Evictions*

For each b-node selected in Phase I, after the k-nodes containing this b-node or its child b-nodes have executed their delayed intra k-node evictions in Phase II, the inter k-node eviction from this b-node to its child b-nodes shall be executed in this phase.

To facilitate data eviction, each k-node partitions its DA space into three logical parts, denoted as P_1, P_2 and P_3, each of which can store $c \cdot (k-1)$ data blocks.

- P_1 stores the $c \cdot (k-1)$ data blocks that have been evicted to this k-node most recently.
- The rest space is evenly divided into two logical parts, P_2 and P_3, and all the real data blocks belong to P_2. As we prove in Section 5, when system parameters are properly configured, each k-node stores at most $c \cdot (k-1)$ real blocks with the probability of $1 - 2^{-\lambda}$; hence, the division fails (and thus the eviction scheme fails) with only a probability of $2^{-\lambda}$.

Note that, the server knows P_1, but does not know the scopes of P_2 and P_3.

Let $v_{l,x}$ denote the selected evicting b-node inside k-node $u_{l',x'}$, and $v_{l+1,y}$ and $v_{l+1,z}$ denote the two child b-nodes of $v_{l,x}$. Also, let $u_{l'+1,y'}$ and $u_{l'+1,z'}$ denote the two k-nodes where b-nodes $v_{l+1,y}$ and $v_{l+1,z}$ reside. The procedure for data eviction from $v_{l,x}$ to $v_{l+1,y}$ and $v_{l+1,z}$ is elaborated as follows.

First, the client needs to obliviously retrieve an evicting data block, denoted as D, from $v_{l,x}$. To accomplish this, the client retrieves the EI of k-node $u_{l',x'}$ which contains $v_{l,x}$, and checks if $v_{l,x}$ contains a real data block. Suppose one real data block D of $v_{l,x}$ is stored at position m of the DA of $u_{l',x'}$, the client constructs two $3c(k-1)$-bit query vectors, denoted as $\overrightarrow{Q}_1$ and $\overrightarrow{Q}_2$. The client sets the two bit vectors as follows:

$$\begin{aligned}\overrightarrow{Q}_1 &= (r_0, r_1, \cdots, r_m, \cdots, r_{3c(k-1)-1}),\\ \overrightarrow{Q}_2 &= (r_0, r_1, \cdots, 1-r_m, \cdots, r_{3c(k-1)-1}),\end{aligned} \tag{4}$$

where each r_i is a bit randomly selected from $\{0,1\}$. Then, the client sends $\overrightarrow{Q}_1$ to $\mathcal{S}_1$ and $\overrightarrow{Q}_2$ to $\mathcal{S}_2$. Each server $\mathcal{S}_j$, where $j \in \{0,1\}$, computes $\hat{D}_j$ as

$$\bigoplus_{\forall D' \in \{D' \ on\ pos\ i\ of\ u_{l',x'} | 0 \le i \le 3c(k-1)-1, Q_j[i]=1\}} D', \tag{5}$$

and sends $\hat{D}_j$ back to the client, who computes $\hat{D}_0 \bigoplus \hat{D}_1$ to get D.

Second, the client applies the following rules to evict a data block D from $v_{l,x}$ to $v_{l+1,2x}$ (the eviction of D to $v_{l+1,2x+1}$ follows the same rule):

- If D is a real data block, there are two sub-cases:
  - D is intended to be evicted to $v_{l+1,2x}$. Thus, the data block that will be evicted to $v_{l+1,2x+1}$ is a dummy block. In this case, each position $w \in P_3$ will have equal probability to be selected such that D will be evicted to w.
  - D is not intended to be evicted to $v_{l+1,2x}$. Hence, the data block that will be evicted to $v_{l+1,2x+1}$ is the real data block. In this case, one position $w \in P_2$ will be uniformly randomly selected and D will be evicted to w.
- If D is a dummy data block, each position $w \in P_2 \cup P_3$ will have equal probability to be selected such that D will be written to.

After the above operations, w is transferred to P_1. Meanwhile, the oldest position $w' \in P_1$ becomes a member of P_2 or P_3 depending on if it contains a real block or a dummy block.

We prove in Section 5 that, each position in $P_2 \cup P_3$ has the same probability to be selected to access during an eviction process; thus, the eviction process is oblivious and does not reveal the access pattern.

5. SECURITY ANALYSIS

In this section, we first show that with proper setting of parameters, MSKT-ORAM construction fails with a probability of $2^{-\lambda}$ ($\lambda \ge \log N + 10$) through proving the DA of each k-node overflows with a probability of $2^{-\lambda}$. Then, we show that both query and eviction processes access k-nodes independently of the client's private data request. Based on the above steps, we finally present the main theorem.

LEMMA 1. *In MSKT-ORAM, if each DA stores $3c(k-1)$ data blocks, where $k \ge 1.36\lambda + 6.44$ and $c = 4$, the probability for the DA of any k-node in the k-ary tree to overflow is $2^{-\lambda}$, where $\lambda > \log N + 10$.*

PROOF. The proof considers non-leaf and leaf k-nodes separately.

Non-leaf k-nodes. The proof for non-leaf k-node proceeds in the following two steps.

In the first step, we consider the binary tree that a k-ary tree in MSKT-ORAM is logically mapped to, and study the number of real data blocks (denoted as a random variable X_v) logically belonging to an arbitrary b-node v on an arbitrary level l of the binary tree.

As the eviction process of MSKT-ORAM completely simulates the eviction process of T-ORAM and P-PIR over the logical binary tree, their results [26] of theoretical study on the number of real data blocks in a binary tree node can still apply. Specifically, X_v can be modeled as a Markov Chain denoted as $\mathcal{Q}(\alpha_l, \beta_l)$. In the Chain, the initial one is $X_v = 0$, The transition from $X_v = i$ to $X_v = i+1$ occurs with probability α_l, and the transition from $X_v = i+1$ to $X_v = i$ occurs with probability β_l, for every non-negative integer i. Here, $\alpha_l = 1/2^l$ and $\beta_l = 2/2^l$ for any level l. Also, for any $l \ge 2$, an unique stationary distribution exists for the Chain; that is,

$$\pi_l(i) = \rho_l^i(1-\rho_l), \tag{6}$$

where

$$\rho_l = \frac{\alpha_l(1-\beta_l)}{\beta_l(1-\alpha_l)} = \frac{2^l - 2}{2(2^l - 1)} \in \left[\frac{1}{3}, \frac{1}{2}\right). \tag{7}$$

In the second step, we consider an arbitrary k-node u on the k-ary tree and study the number of real data blocks stored at the DA of u, which is denoted as a random variable Y_u.

The binary subtree that u is logically mapped to contains $k-1$ b-nodes, which are denoted as $v_1, \cdots, v_{k-1}$ for simplicity. Then $Y_u = \sum_{i=1}^{k-1} X_{v_i}$. Also, as k should be greater than 2 to make MSKT-ORAM nontrivial, any of the b-nodes $v_1, \cdots, v_{k-1}$ should be on a level greater than or equal to 2 on the logical binary tree (Those b-nodes on level 0 and 1 never overflow).

Now, we compute the probability

$$\mathbf{Pr}\,[Y_u = t] = \mathbf{Pr}\,[X_{v_1} + \cdots + X_{v_{k-1}} = t]. \tag{8}$$

Note that, there are $\binom{t+k-2}{k-2}$ different combinations of $X_i = t_i$ $(i = 1, \cdots, k-1)$ such that $t_1 + \cdots + t_{k-1} = t$. Hence, we have:

$$\begin{aligned}\mathbf{Pr}\,[Y_u = t] &\le \binom{t+k-2}{k-2} \prod_{i=1}^{k-1} \left[\left(\frac{1}{2}\right)^{t_i} \left(\frac{2}{3}\right)\right] && (9)\\ &\le \left(\frac{(t+k-2)\cdot e}{k-2}\right)^{k-2} \left(\frac{1}{2}\right)^t \left(\frac{2}{3}\right)^{k-1} && (10)\\ &< \left(\frac{(t+k-2)\cdot e}{k-2}\right)^{k-1} \left(\frac{1}{2}\right)^t \left(\frac{2}{3}\right)^{k-1}\\ &\le \left(\frac{2(t+k-2)\cdot e}{3(k-2)}\right)^{k-1} \left(\frac{1}{2}\right)^t.\end{aligned}$$

Here, Equation (9) is due to $\pi_l(i) = \rho_l^i(1-\rho_l) \leq \rho_l^i \cdot \frac{2}{3} < \left(\frac{1}{2}\right)^i \cdot \frac{2}{3}$, which is due to Equation (6). Inequality (10) is due to $\binom{n}{k} \leq \left(\frac{n \cdot e}{k}\right)^k$ for all $1 \leq k \leq n$. Hence, we have:

$$\begin{aligned}\mathbf{Pr}[Y_u = t] &\leq [\frac{2}{3} \cdot e \cdot (c+1+\frac{c}{k-2}) \cdot (\frac{1}{2})^c]^{k-1} \quad (11)\\ &< (\frac{3}{5})^{k-1} = (\frac{3}{5})^{t/4}. \quad (12)\end{aligned}$$

Here, Inequality (11) is due to $t = c(k-1)$ in the scheme, Inequality (12) is due to $k = 1.36\lambda + 6.44 > \log N$ and $c = 4$.

Then, the following inequalities follows:

$$\begin{aligned}\mathbf{Pr}\,[Y_u \geq t] &= \sum_{i=0}^{\infty} \mathbf{Pr}\,[Y_u = t+i] < \sum_{i=0}^{\infty} [(\frac{3}{5})^{1/4}]^{t+i}\\ &= \frac{(\frac{3}{5})^{t/4}}{1-(\frac{3}{5})^{1/4}} < 9 \cdot 2^{-0.74(k-1)} < 2^{-\lambda}.\end{aligned} \quad (13)$$

Leaf k-nodes. At any time, all the leaf k-nodes contain at most N real blocks and each of the blocks is randomly placed into one of the leaf k-nodes. Thus, we can apply standard balls and bins model to analyze the overflow probability. In this model, N balls (real blocks) are thrown into $2N/k$ bins (i.e., leaf k-nodes) in a uniformly random manner.

We study one arbitrary bin and let $X_1, \cdots, X_N$ be N random variables such that

$$X_i = \begin{cases} 1 & \text{the } i^{\text{th}} \text{ ball is thrown into this bin,} \\ 0 & \text{otherwise.} \end{cases} \quad (14)$$

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

$$E[X] = E\left[\sum_{i=1}^{N} X_i\right] = \sum_{i=1}^{N} E[X_i] = N \cdot \frac{k}{2N} = \frac{k}{2}. \quad (15)$$

According to the Chernoff bound, when $\delta = 2j/k - 1 \geq 2e - 1$, it holds that

$$\begin{aligned}&\mathbf{Pr}\,[\text{at least } j \text{ balls in this bin}]\\ &= \mathbf{Pr}\,[X \geq j] < \left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right)^{k/2}\\ &< \left(\frac{e^\delta}{(2e)^\delta}\right)^{k/2} = 2^{-k\delta/2}.\end{aligned} \quad (16)$$

Hence, letting $j = 4(k-1)$, we have

$$\mathbf{Pr}\,[\text{at least } 4(k-1) \text{ balls in this bin}] < 2^{-1.5k}. \quad (17)$$

Finally, we have the following equation:

$$\begin{aligned}&\mathbf{Pr}\,[\exists \text{ a bin with at least } 4(k-1) \text{ balls}]\\ &< \frac{2N}{k} \cdot 2^{-1.5k} = \frac{2N}{k} \cdot 2^{-1.5(1.36\lambda+6.44)}\\ &= \frac{2N}{k} \cdot 2^{-2.04\lambda+9.66} < 2^{-\lambda}.\end{aligned} \quad (18)$$

The first inequality is due to the union bound. The second inequality is due to the fact that $\lambda \geq \log N + 10$.

According to the above two parts, we have proved that the overflow probability is $2^{-\lambda}$. □

LEMMA 2. *Any query process in MSKT-ORAM accesses k-nodes from each layer of the k-ary tree, uniformly at random.*

LEMMA 3. *Any eviction process in MSKT-ORAM accesses a sequence of k-nodes independently of the client's private data request.*

Please refer to Appendices for proofs of the lemmas.

LEMMA 4. *For any k-node n_i with k-node n_p as its parent, each position in $P_2 \bigcup P_3$ of n_i has the equal probability of $\frac{1}{2c \log N}$ to be selected to access.*

PROOF. During an eviction process, n_p may evict a real or dummy block to one of its child nodes (i.e., n_i or n_{1-i}). In the following, we consider these two cases respectively.

Case I: n_p evicts a real block to a child node. There are two subcases as follows.

Case I-1: the real block is evicted to n_i; this subcase occurs with the probability of 0.5. In this subcase, according to the aforedescribed eviction policy, a position is randomly selected from P_3 to accept the evicted block.

Case I-2: the real block is evicted to n_{1-i}, i.e., a dummy block is evicted to n_i; this occurs with the probability of 0.5. In this subcase, according to the eviction policy, a position is randomly selected from P_2 to access.

Case II: n_p does not evict any real block to its child nodes. That is, both n_i and n_{1-i} are evicted to with dummy blocks. In this case, according to the eviction policy, a position is randomly selected from $P_2 \bigcup P_3$ to access.

Also because P_2 and P_3 have the same size, each position in $P_2 \bigcup P_3$ has the equal probability to be selected to access. □

THEOREM 1. *MSKT-ORAM is secure under Definition 2, if $k \geq 1.36\lambda + 6.44$ and $c = 4$.*

PROOF. Given any two equal-length sequence $\vec{x}$ and $\vec{y}$ of the client's private data requests, their corresponding observable access sequences $A(\vec{x})$ and $A(\vec{y})$ are computationally indistinguishable, because both of the observable sequences are independent of the client's private data request sequences. This is due to the following reasons:

- According to the query and eviction algorithms, sequences $A(\vec{x})$ and $A(\vec{y})$ should have the same format; that is, they contain the same number of observable accesses, and each pair of corresponding accesses have the same access type.
- According to Lemma 2, the sequence of locations (i.e., k-nodes) accessed by each query process are uniformly random and thus independent of the client's private data request.

- According to Lemma 3, the sequence of locations (i.e., k-nodes) accessed by each eviction process after a query process is also independent of the client's private data request.

Moreover, according to Lemma 1, the MSKT-ORAM construction fails with a probability of $2^{-\lambda}$, when $k \geq 1.36\lambda + 6.44$ and $c = 4$. $\square$

6. COST ANALYSIS

We analyze the performance of MSKT-ORAM in terms of communication, server-side and client-side storage costs. To facilitate the analysis, the following notations and assumptions are applied:

- N: the total number of data blocks. In MSKT-ORAM, we assume $2^{16} \leq N \leq 2^{34}$. In practice, suppose the data block size is 1 MB, when $N = 2^{34}$, the total amount of all outsourced data blocks will be 16 PB.
- k: the out-degree of each k-node in MSKT-ORAM. Following the requirement that $k \geq 1.36\lambda + 6.44$, k can be set to $O(N^{\epsilon'})$ where $0 < \epsilon' < \epsilon < 1$. Figure 6 shows the lower bound of k under different security parameter λ. Through out the rest of the paper, we set $k = 128$ such that the overflow probability of MSKT-ORAM is no more than 2^{-80}.
- H_k and H_b: heights of the k-ary and binary trees, respectively. According to the scheme, $H_b = \lceil \log N + 1 \rceil$ and $H_k = \lceil \frac{\log N+1}{\log k} \rceil$. Therefore, H_k is a constant in practice. For example, given $k = 128$, the height of the tree is $H_k \leq 5$ when $2^{16} \leq N \leq 2^{34}$.
- S_{EI}: size of the EI of a k-node. According to the scheme, $S_{EI} = 3c \cdot (k-1) \cdot \{\log N + \log(N/k) + \log[3c \cdot (k-1)]\}$. Given $2^{16} \leq N \leq 2^{34}$ and $\epsilon' = 1/3$, 6.7 KB $\leq S_{EI} \leq$ 13.4 KB.
- B: the size of each data block. In MSKT-ORAM, we assume $\Omega(N^{\epsilon})$ bits ($0 < \epsilon < 1$), such that the number of recursions is $O(1)$. For example, given $2^{16} \leq N \leq 2^{34}$ and $\epsilon = 1/2$, B is only required to be no smaller than 16 KB. Consider the requirement of S_{EI}. In order to absorb the EI in our scheme, each data block size is only required to be greater than or equal to 20 KB.
- S_{QV}: size of the query vector of a k-node. According to the scheme, $S_{QV} = 3c \cdot (k-1) < 24$ bytes.

In order to reduce the communication cost, we suppose only one server maintains the EI for each k-node regardless of how many servers are.

6.1 Communication Cost Per Query

Suppose there are two servers in MSKT-ORAM. During a query process, H_k k-nodes are accessed from the k-ary tree. The client needs to (i) download the EIs of these k-nodes (which is $H_k \cdot S_{EI}$ bits); (ii) send query vectors to two servers(i.e., $2H_k \cdot S_{QV}$ bits); (iii) upload the re-encrypted EIs (i.e., $H_k \cdot S_{EI}$ bits). Then, the client will download two data blocks from the server, one data block from each server, which is $2B$ bits. To wrap up a query process, the target data block is uploaded to the root k-node of each server through multicasting, which incurs B bits communication overhead.

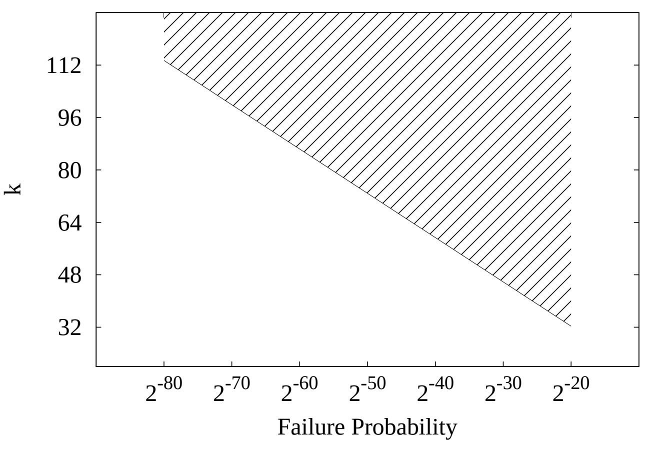


Figure 6: k v.s. failure probability. The shadow area in the figure shows feasible k's for a given security level, i.e., failure probability of MSKT-ORAM, where $2^{16} \leq N \leq 2^{34}$.

Therefore, the communication cost for each query process is:

$$Comm_{Query} = 3B + 2H_k \cdot (S_{EI} + S_{QV}) \text{ bits.} \tag{19}$$

During an eviction process, $2(H_k - 1) - 1$ k-nodes are selected to evict one data block to their children k-node.

For each selected k-node, the client first downloads the evicted block. Therefore, the client needs to do the following: (i) download its EI and its children's EI ($3S_{EI}$ bits); (ii) send query vectors to two servers ($2S_{QV}$ bits); (iii) XOR-download one data block from parent k-node from each server ($2B$ bits) and download one data block from each selected children k-node ($2B$ bits); (iv) upload these EI ($3S_{EI}$ bits) back. Then, two data blocks are uploaded to the two children k-node for all servers through multicasting, which incurs $2B$ bits communication overhead.

Therefore, the communication cost for each eviction process is

$$Comm_{Evict} = [2(H_k - 1) - 1] \cdot (6S_{EI} + 2S_{QV} + 6B) \text{ bits.} \tag{20}$$

Overall, the communication cost per query is:

$$Comm = 6(H_k - 1)(3S_{EI} + S_{QV}) + [12(H_k - 1) - 3]B \text{ bits.} \tag{21}$$

Figure 7 shows the actual communication cost for different k given N and B. From the figure, it is obvious that:

- When N is fixed, the increase of k may lead to a sharp communication cost decrease. The reason is because the decrease of H_k, i.e., the height of the tree decreases by 1.
- When N is fixed and the tree height H_k is not changed, the increase of k will lead to the increase of communication cost since the size of EI is increased as there are more b-nodes for any single k-node.
- When data block size B increases, the EI part and query vector part will be absorbed by data block size. Therefore, the communication cost is $[12(H_k - 1) - 3]B$ bits. For example, in the last figure, when data block size is 1 MB and

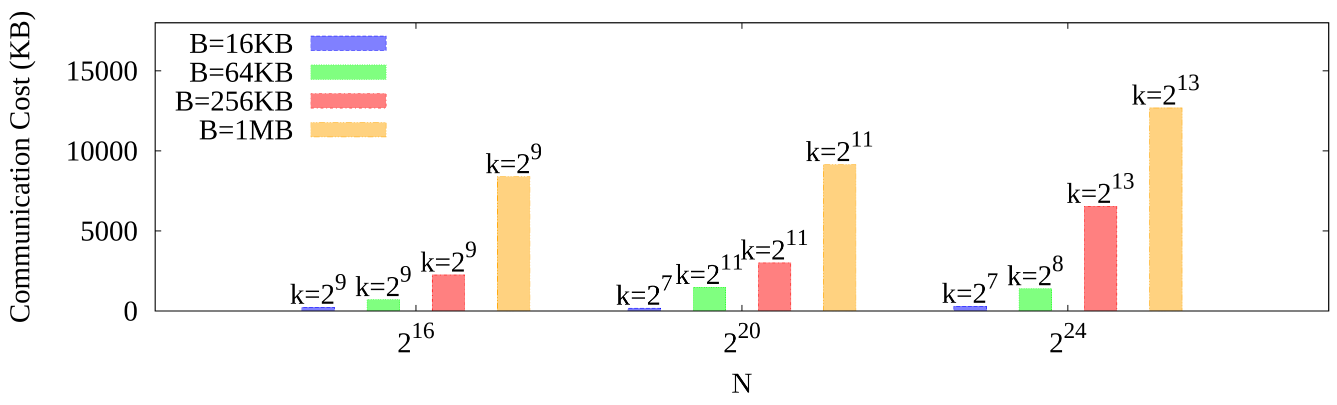


Figure 7: Communication cost. This figure shows the optimal communication cost for given a given N and B combination. Data block size B ranges from 16 KB to 1 MB and the total number of data blocks N ranges from 2^{16} to 2^{24}. The optimal k is marked above each bar.

the height of the tree is 2, no more than 10 data blocks will be transfered per data query between the client and the two servers.

6.2 Storage Costs

According to the MSKT-ORAM design, the client-side storage cost is the size of a constant number of data blocks, i.e., $O(B)$. The storage at the server side needs to store $3c(k-1)$ data blocks per k-node, which is $3c$ data blocks per b-node. Since there are $2N-1$ b-nodes, the server side storage overhead is $O(N \cdot B)$, which is $3c \cdot (2N-1)B < 24NB$ bits per server.

6.3 Computational Costs per Query

The computational cost on the client mainly consists of the following three sources: (1) Data IO cost on the server, i.e., data will be read into memory to perform XOR operations. Note that, the number of data IO operations equals the number of XOR operations. (2) Data block XOR cost, denoted as $Comp_{XOR/IO}$; (3) Data block and EI encryption/decryption cost denoted as $Comp_{Enc}$;

During data query, the client (i) decrypts the downloaded EIs of H_k k-nodes ($H_k \cdot Comp_{Enc}$); (ii) XORs and decrypts the downloaded data blocks $Comp_{XOR/IO} + Comp_{Enc}$; and (iii) re-encrypts EIs ($H_k \cdot Comp_{Enc}$) and data blocks $Comp_{Enc}$. Therefore, the total computational cost for data query on client side is

$$Comp_{Query} = (2H_k + 2)Comp_{Enc} + Comp_{XOR/IO}. \tag{22}$$

For each data eviction process, the client first XOR-downloads the evicted data blocks. Thus, the client needs to (i) download and decrypt EIs of the k-node and two of its children ($3Comp_{XOR/IO} + 3Comp_{Enc}$); (ii) XOR the downloaded blocks ($Comp_{XOR/IO}$); (iii) decrypt two data blocks from its children ($2Comp_{Enc}$); (iv) re-encrypt the three EIs and three downloaded data blocks from the servers ($6Comp_{Enc}$). Hence, the computational cost for data eviction is:

$$Comp_{Evict} = [2(H_k-1)-1] \cdot (11Comp_{Enc} + 3Comp_{XOR/IO}). \tag{23}$$

The total computational cost on the client side is:

$$Comp_{Client} = (24H_k-31)Comp_{Enc} + (6H_k-7)Comp_{XOR/IO}. \tag{24}$$

The computational cost at each server is contributed by the XOR operation in MSKT-ORAM. For each data query, XOR operations are performed on $H_k \cdot 3c(k-1)$ data blocks. Since about half of the data blocks will be XORed, the computational cost per server is:

$$Comp'_{Query} = \frac{H_k \cdot 3c(k-1)}{2} \cdot Comp_{XOR/IO}. \tag{25}$$

For each data eviction process, there are $2(H_k-1)-1$ k-nodes to perform XOR operations on. Thus, the computational cost per server is:

$$Comp'_{Evict} = \frac{[2(H_k-1)-1] \cdot 3c(k-1)}{2} \cdot Comp_{XOR/IO}. \tag{26}$$

Thus, the total computational cost is:

$$\begin{aligned} Comp_{Server} &= Comp'_{Query} + Comp'_{Evict} \\ &= \frac{9(H_k-1)c(k-1)}{2} \cdot Comp_{XOR/IO} \\ &= 18(H_k-1)(k-1) \cdot Comp_{XOR/IO}. \end{aligned} \tag{27}$$

7. COMPARISONS

In this section, we present detailed comparisons between MSKT-ORAM and several state-of-the-art ORAMs including T-ORAM [26], Path ORAM [30], P-PIR [22], C-ORAM [24], MSS-ORAM [27], and CNE-ORAM [23].

7.1 Asymptotical Comparisons

Table 1 shows the asymptotical comparison between MSKT-ORAM and some existing ORAM schemes. First, we make a comparison to single server ORAMs, which are T-ORAM, Path ORAM, P-PIR and C-ORAM. For T-ORAM, Path ORAM and P-PIR, the communication costs of these three schemes are not constant. For both C-ORAM and P-PIR, they require homomorphic encryptions in the schemes which incur a huge amount of computations on the server side. Compared to multi-server ORAMs, i.e., ObliviStore and CNE-ORAM, MSKT-ORAM only requires at least 2 non-colluding servers and no communication is required between the servers. In addition, MSKT-ORAM only requires the client to store a constant number of data blocks.

7.2 Practical Comparisons

In the state-of-the-art ORAM schemes with homomorphic encryption (such as P-PIR and C-ORAM), the computational cost and the

Table 1: Asymptotical comparisons in terms of client-server communication cost, server-server communication cost, client storage cost, server storage cost and the minimum number of non-colluding servers. N is the total number of data blocks and B is the size of each block in the unit of bits. $B = O(N^{\epsilon})$ for some $0 < \epsilon < 1$. For MSKT-ORAM, $k = O(N^{\epsilon'})$ where $0 < \epsilon' < \epsilon < 1$ and $c = 4$. For all tree structure ORAMs, suppose the index table is outsourced to the server side with $O(1)$ recursion depth. The star symbol denotes the scheme requires homomorphic encryption.

ORAM	C-S Comm. Cost	S-S Comm. Cost	Client Stor. Cost	Server Stor. Cost	Number of Servers
T-ORAM [26]	$O(\log^2 N \cdot B)$	N.A.	$O(B)$	$O(N \log N \cdot B)$	1
Path ORAM [30] SCORAM [32]	$O(\log N \cdot B)$	N.A.	$O(\log N \cdot B) \cdot \omega(1)$	$O(N \cdot B)$	1
$*$P-PIR [22]	$O(\log N \cdot B)$	N.A.	$O(B)$	$O(N \log N \cdot B)$	1
$*$C-ORAM [24]	$O(B)$	N.A.	$O(B)$	$O(N \cdot B)$	1
MS-ORAM [21]	$O(\log N \cdot B)$	$O(\log^3 N \cdot B)$	$O(B)$	$O(N \log N \cdot B)$	2
MSS-ORAM [27]	$O(B)$	$O(\log N \cdot B)$	$O(\sqrt{N} \cdot B)$	$O(N \cdot B)$	2
CNE-ORAM [23]	$O(B)$	N.A.	$O(B)$	$O(N \cdot B)$	4
MSKT-ORAM	$O(B)$	N.A.	$O(B)$	$O(N \cdot B)$	2

access delay are prohibitively high. As illustrated in C-ORAM [24], each data access will incur a delay of about 7 minutes. When the client frequently accesses the storage, access delay becomes an obstacle for these schemes. However, in MSKT-ORAM, data XOR and bit-shift operations (AES) are used to replace expensive homomorphic encryption. Thus, the major advantage of MSKT-ORAM is to reduce the computational costs on the server side and therefore the delay.

Among all the ORAMs that are considered in asymptotical comparison section, the most comparable scheme is CNE-ORAM [23]. Thus, we make a detailed comparison between CNE-ORAM and our proposed scheme. The comparisons are conducted in terms of communication cost, storage cost and access delay. The access delay comparison includes all non-implementation factors that result in the access delay. In the comparison, we consider the following parameter settings: (1) N ranges from 2^{16} to 2^{34}; (2) B is set to 1 MB for both schemes; (3) The security parameter λ ranges from 20 to 80 for CNE-ORAM and $\lambda = 80$ for MSKT-ORAM; (4) As described before, k is set to 128 in MSKT-ORAM.

Table 2: Practical Comparisons ($2^{16} \leq N \leq 2^{34}$, $B = 1$ MB). The communication cost in the table is the client-server communication cost per query. The computational cost is the number of XOR operations needed on the server side. The IO cost is the number of data block that need to be read into the memory for processing.

	CNE-ORAM	MSKT-ORAM
Comm. Cost	$> 40 \cdot B$	$22 \cdot B \sim 46 \cdot B$
Comp. Cost	$> 10(\lambda - 10) \cdot \log N$	$18(H_k - 1)(k - 1)$
IO Cost	$> 10(\lambda - 10) \cdot \log N$	$18(H_k - 1)(k - 1)$
Security Param.	$20 \leq \lambda \leq 80$	$\lambda = 80$

7.2.1 *Communication Cost Comparison*

As explained in CNE-ORAM, the communication cost is determined by the security parameter, node size and the eviction frequency. In the evaluation section, the authors of CNE-ORAM suggest a communication cost of more than $40 \cdot B$ per query. For MSKT-ORAM, a lower or comparable communication cost can be achieved under the setting of $k = 128$. As shown in Equation 21, when $2^{16} \leq N \leq 2^{20}$, the communication cost of MSKT-ORAM is about $22 \cdot B$ per query; when $2^{21} \leq N \leq 2^{27}$, the communication cost of MSKT-ORAM is $< 34 \cdot B$ per query; when $2^{28} \leq N \leq 2^{34}$, the communication cost of MSKT-ORAM is $< 46 \cdot B$ per query.

7.2.2 *Access Delay Comparison*

The access delay for both CNE-ORAM and our proposed scheme is mainly contributed by three factors: (1) Communication Delay: Delay incurred by communication between the client and the server; (2) Computational Delay: Delay incurred by server side computation; (3) IO Delay: Delay incurred by server side IO operations.

Communication Delay. Suppose the client and the server bandwidth is BW. As aforementioned, the total communication cost for CNE-ORAM is greater than $40 \cdot B$. Therefore, the communication delay of CNE-ORAM is

$$\frac{40 \cdot B}{BW}. \tag{28}$$

While for MSKT-ORAM, as shown in Table 2, the communication delay is

$$\frac{22 \cdot B}{BW} \sim \frac{46 \cdot B}{BW}. \tag{29}$$

In order to illustrate the actual communication delay. Suppose the network bandwidth between the client and the servers is 10 MB/s and data block size is $B = 1$ MB. CNE-ORAM needs about 4 seconds to transfer these data blocks while MSKT-ORAM needs about $2.2 \sim 4.6$ seconds to transfer.

Computational Delay. The computational delay is mainly contributed by server-side computation operations, i.e., the number of data to be XORed by the server from physical storage to server memory.

As described in Equation 27, the server side computational delay of MSKT-ORAM is $18(H_k - 1)(k - 1)$. Note that, since $2^{16} \leq N \leq 2^{34}$ and $k = 128$, $3 \leq H_k \leq 5$. Therefore, the total number of XOR operations ranges from 4572 to 9144.

In CNE-ORAM, the XOR operations are determined by the binary tree height $H_b = \log N$ and the node size θ. Since the node size θ of CNE-ORAM is related to its security parameter, from its regression evaluation, $\theta = 20(\lambda - 10)$. Thus, the total number of XOR operations will be performed on half of these data blocks. Note that, data eviction XOR cost is not counted here

$$0.5\theta \cdot H_b = 10(\lambda - 10) \cdot \log N. \tag{30}$$

For comparison fairness, we set the security parameter λ in CNE-ORAM to 80. Then, the total number of operations ranges from 11200 to 23800.

For illustration purpose, we choose $N = 2^{20}$ and let the security parameter λ varies between 20 to 80. The total number of operations for each server for each scheme is shown in Figure 8. In order

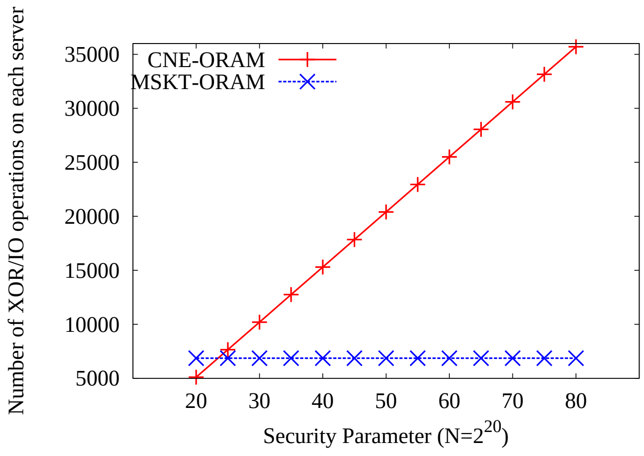


Figure 8: Comparisons of number of XOR/IO operations on the server per query. $N = 2^{20}$ **and for MSKT-ORAM,** $k = 128$**.**

to better understand the computational delay for both scheme, we suppose one XOR operation can be done with 1 millisecond for two 1 MB data blocks (Note that, as shown in [23], a 2012 Macbook Pro with 2.4 Ghz Intel i7 processor can achieve this speed). Thus, the computational delay for MSKT-ORAM is about 4.5 seconds to 9 seconds, while the computational delay for CNE-ORAM ranges from 11.2 seconds to 23.8 seconds.

IO Delay. The total number of IO operations on the server side is the same as the number of XOR operations. For each of the IO operations to read 1 MB data block to memory, it takes about 2 milliseconds on a 2012 Macbook Pro (500 MB/s read/write speed for SSD). Therefore, the MSKT-ORAM incurs a total IO delay for about 9 seconds to 18 seconds while CNE-ORAM incurs 22.4 seconds to 47.6 seconds.

Overall Delay. Putting all delay factors together, given security parameter $\lambda = 80$, CNE-ORAM takes about 37.6 $\sim$ 75.4 seconds to complete a query while MSKT-ORAM only needs 18.1 $\sim$ 31.7 seconds to complete the query.

7.2.3 Storage Cost

The storage cost of CNE-ORAM depends on the size of each node on the binary tree, θ and the number of tree nodes, which can be computed as follows:

$$NumNode = 1 + 2^1 + 2^2 + \cdot + 2^L, \tag{31}$$

where $N \leq \chi \cdot 2^{L-1}$ and $\chi = \frac{\theta}{10}$. Thus, the number of tree nodes is greater than or equal to $\frac{4N}{\chi} - 1$. Therefore, the storage cost per server in CNE-ORAM is:

$$NumNode \cdot \theta \geq 40N \cdot B. \tag{32}$$

Since there are four servers in CNE-ORAM, the storage cost in total is $160N \cdot B$. For MSKT-ORAM, the storage cost is $24N \cdot B$ per server as shown in section 6.2. Therefore, the total storage cost is $48N \cdot B$.

8. RELATED WORK

8.1 Oblivious RAM

Based on the data lookup technique adopted, existing ORAMs can be classified into two categories, namely, hash-based ORAMs and index-based ORAMs. Hash-based ORAMs [10–14, 17, 25, 33–35] require some data structures such as buckets or stashes to deal with hash collisions. Among them, the Balanced ORAM (B-ORAM) [17] proposed by Kushilevitz et al. achieves the lowest asymptotical communication cost, which is $O(B \cdot \frac{\log^2 N}{\log \log N})$, where B is the size of a data block. Index-based ORAMs [7, 26–30] use index structure for data lookup. They require the client to either store the index, or outsource the index to the server recursively in a way similar to storing data, at the expense of increased communication cost. The state-of-the-art index-based ORAMs are binary tree ORAM (T-ORAM) [26], Path ORAM [30], and SCORAM [32]. When $B = N^\epsilon$ (constant $0 < \epsilon < 1$) is assumed, the communication cost for T-ORAM is $O(B \cdot \log^2 N)$ with failure probability $O(N^{-c})$ ($c > 1$), while Path ORAM and SCORAM incur $O(B \cdot \log N) \cdot \omega(1)$ communication cost with $O(N^{-\omega(1)})$ failure probability and $O(B \cdot \log N) \cdot \omega(1)$ client-side storage.

8.2 Private Information Retrieval (PIR)

PIR protocols were proposed mainly to protect the pattern in accessing *read-only* data at remote storage. There are two variations of PIR: information-theoretic PIR (iPIR) [1, 4, 8, 9], assuming multiple non-colluding servers each holding one replica of the shared data; computational PIR (cPIR) [2, 3, 18, 19], which usually assumes single server in the system. cPIR is more related to our work, and thus is briefly reviewed in the following. The first cPIR scheme was proposed by Kushilevitz and Ostrovsky in [18]. Designed based on the hardness of quadratic residuosity decision problem, the scheme has $O(n^c)$ ($0 < c < 1$) communication cost, where n is the total number of outsourced data in bits. Since then, several other single-server cPIRs [2, 19] have been proposed based on different intractability assumptions. Even though cPIRs are impractical when database size is large, they are acceptable for small databases. Recently, several partial homomorphic encryption-based cPIRs [15, 31] have been proposed to achieve satisfactory performance in practice, when database size is small. Due to the property of partial homomorphic encryption, [20, 22] show that these cPIR schemes can also be adapted for data updating.

8.3 Hybrid ORAM-PIR Designs

Designs based on a hybrid of ORAM and PIR techniques have emerged recently. Among them, C-ORAM [24] has the best-known performance. However, as the complexity of PIR primitives, one data query would require the server to take about 7 minutes to process. In addition, data block size in C-ORAM is $O(\log^4 N)$ bits, where N is the total number of outsourced data blocks. Thus, this imposes another strict requirement on C-ORAM.

8.4 Multi-Server ORAMs

There are several multi-server ORAM schemes in literature. Among them, the first one is MS-ORAM [21]. MS-ORAM extends the idea of the hierarchical ORAM [10] and the two non-colluding servers are used to obliviously shuffle data. Following the hierarchical ORAM, the client-server communication cost is $O(\log N \cdot B)$, while the server-server communication cost is $O(\log^3 N \cdot B)$ due to its complicated shuffling process. Even though it only requires a constant local storage, the communication cost is expensive in practice.

The second scheme is MSS-ORAM [27], which follows the basic design of Partition-ORAM [29]. In their scheme, data shuffling is done between different cloud servers. The client-server communication cost is reduced to a constant number, but the server-server communication cost is $O(\log N \cdot B)$. In addition, it requires the client to store $O(\sqrt{N})$ data blocks in the local storage.

Recently, another multi-server ORAM called CNE-ORAM is proposed which incurs $O(B)$ client-server communication cost using at least 4 non-colluding servers. In CNE-ORAM, each data block is split into two parts using secret sharing techniques. Each part of one data block is further copied into two copies and each copy is stored onto 2 out of the 4 servers. The remaining part is also copied and stored onto the other 2 servers. At the server side, the storage is organized as a binary tree with of height $H = O(\log N)$ and each tree node can store θ data blocks. For each data query, the target data block is retrieved using XOR-based private information retrieval. Then client then writes ϕ data blocks to root node of each server. After χ queries, data eviction process is executed to prevent root node from overflowing. During data eviction, the client guides the servers to merge nodes on the evicting path. In post eviction process, client retrieves a block for the leaf node of the evicting path and replaces it with an empty block if it is a noise block. The computation cost is mainly contributed by data XOR operations, where for each data query, more than $0.5\theta \cdot L$ blocks are XORed and the communication cost is mainly contributed by uploading ϕ data blocks to root per query, where $L = O(\log N)$ is the height of the tree.

9. CONCLUSION

This paper proposes MSKT-ORAM, which organizes the server storage as a k-ary tree with each node acting as a fully-functional PIR storage. It also adopts a novel delayed eviction technique to optimize the eviction process. MSKT-ORAM is proved to protect the data access pattern privacy at a failure probability of 2^{-80} (N is the number of exported data blocks), when $k \geq 128$. the communication cost of MSKT-ORAM is only 22 to 46 data blocks, when N (i.e., the total number of outsourced data blocks) ranges from 2^{16} to 2^{34} and data block size $B \geq 20$ KB. Detailed asymptotical and numerical analysis, as well as complete simulation and implementation comparisons are conducted to show that MSKT-ORAM achieves better communication, storage and computational efficiency in practical scenario over these compared state-of-the-art ORAM schemes.

10. REFERENCES

[1] A. Beimel, Y. Ishai, E. Kushilevitz, and J.-F. Raymond. Breaking the $O(n^{\frac{1}{2k-1}})$ barrier for information-theoretic private information retrieval. In *Proc. FOCS*, 2002.

[2] C. Cachin, S. Micali, and M. Stadler. Computationally private information retrieval with polylogarithmic communication. In *Proc. Eurocrypt*, 1999.

[3] B. Chor and N. Gilboa. Computationally private information retrieval. In *Proc. Theory of Computing*, 1997.

[4] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *Proc. FOCS*, 1995.

[5] J. Daemen and V. Rijmen. *The Design of Rijndael*. Springer-Verlag New York, Inc., 2002.

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

[7] 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.

[8] Y. Gertner, Y. Ishai, E. Kushilevitz, and T. Malkin. Protecting data privacy in private information retrieval schemes. In *Proc. STOC*, 1998.

[9] I. Goldberg. Improving the robustness of private information retrieval. In *Proc. S&P*, 2007.

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

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

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

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

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

[15] J. Hoffstein, J. Pipher, and J. H. Silverman. NTRU: A ring-based public key cryptosystem. In *Algorithmic Number Theory*, volume 1423 of *Lecture Notes in Computer Science*, pages 267–288. Springer Berlin Heidelberg, 1998.

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

[17] 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.

[18] E. Kushilevitz and R. Ostrovsky. Replication is not needed: single database, computationally-private information retrieval (extended abstract). In *Proc. FOCS*, 1997.

[19] H. Lipmaa. An oblivious transfer protocol with log-squared communication. In *Proc. ISC*, 2005.

[20] H. Lipmaa and B. Zhang. Two new efficient PIR-writing protocols. In *Proc. ACNS*, 2010.

[21] S. Lu and R. Ostrovsky. Multi-server Oblivious RAM. In *IACR Cryptology ePrint Archive*. International Association for Cryptologic Research, 2011.

[22] T. Mayberry, E.-O. Blass, and A. H. Chan. Efficient private file retrieval by combining ORAM and PIR. In *Proc. NDSS*, 2014.

[23] T. Moataz, E.-O. Blass, and T. Mayberry. Constant Communication ORAM without Encryption. In *IACR Cryptology ePrint Archive*. International Association for Cryptologic Research, 2015.

[24] T. Moataz, T. Mayberry, and E.-O. Blass. Constant communication ORAM with small blocksize. In *Proc. CCS*, 2015.

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

[26] 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.

[27] E. Stefanov and E. Shi. Multi-Cloud Oblivious Storage. In *Proc. CCS*, 2013.

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

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

[30] 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.

[31] J. Trostle and A. Parrish. Efficient computationally private information retrieval from anonymity or trapdoor groups. In *Information Security*, volume 6531 of *Lecture Notes in Computer Science*, pages 114–128. Springer Berlin Heidelberg, 2011.

[32] X. Wang, Y. Huang, T.-H. H. Chan, A. Shelat, and E. Shi. SCORAM: Oblivious RAM for secure computations. In *Proc. CCS*, 2014.

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

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

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

Appendix II: Proof of Lemma 2.

(sketch) In MSKT-ORAM, each real data block is initially mapped to a leaf k-node uniformly at random; and after a real data block is queried, it is re-mapped to a leaf k-node also uniformly at random. When a real data block is queried, all k-nodes on the path from the root to the leaf k-node the real data block currently mapped to are accessed. Due to the uniform randomness of the mapping from real data blocks to leaf k-nodes, the set of k-nodes accessed during a query process is also uniformly at random.

Appendix III: Proof of Lemma 3.

(sketch) During an eviction process, the accessed sequence of k-nodes is independent to the client's private data request due to: (i) the selection of b-nodes for eviction (i.e. Phase I of the eviction process) is uniformly random on the fixed set of layers of the logical binary tree and thus is independent of the client's private data request; and (ii) the rules determining which evictions should be executed immediately (and hence the involved k-nodes should be accessed) and which can be delayed are also independent of the client's private data requests.