TSKT-ORAM: A Two-Server K-ary Tree ORAM for Access Pattern Protection in Cloud Storage

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

***Abstract*—This paper proposes TSKT-ORAM, a two-server k-ary tree-based Oblivious RAM construction, to hide a client's access pattern to outsourced data. TSKT-ORAM is proved to hide the access pattern with a failure probability of $2^{-\lambda}$, where $k = \log N$ (N is the number of outsourced data items) and λ is a security parameter. Under the same configuration, TSKT-ORAM has an asymptotical communication cost of $O(\frac{\log N}{\log \log N} \cdot B)$ (B is the size of a data block) if the number of recursion levels on meta data is $O(1)$, or $O(\frac{\log^2 N}{\log \log N} \cdot B)$ if the number of recursion levels is $O(\log N)$. Asymptotical analysis and detailed implementation-based comparisons are conducted to compare the performance of TSKT-ORAM with state-of-the-art ORAM schemes.**

I. Introduction

Cloud storage services such as Amazon S3 and Dropbox have been popularly used by business and individual clients to host 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 insufficient for data security, because data secrecy can still be exposed if a client's access pattern to the data is revealed [1].

Oblivious RAM (ORAM) [2]–[12] is a well-known security-provable approach for hiding a client's access pattern to the data stored in remote storage. In the literature, the most communication-efficient ORAM construction is the constant ORAM (C-ORAM) [13], which only requires $O(1)$ blocks to be accessed from the server for every query issued by the client. However, C-ORAM incurs expensive computational cost at the server. For example, to retrieve a block of 100 KB, it takes 7 minutes and most of time is spent on computation; the high computational cost results in high query delay, which overshadows the improvement in bandwidth efficiency. Among the constructions that do not incur high computational cost, Path-ORAM [7] proposed by Stefanov et al. is the most efficient one. Its communication cost is $O(B \cdot \log N)$ per query, when the total number of exported data blocks is N and each data block is of size $B \geq N^{\epsilon}$ for some constant $0 < \epsilon < 1$. Though Path-ORAM has achieved better communication efficiency than prior works, further reducing the communication and other costs is still desirable to make the ORAM construction more feasible to implement in cloud storage systems. In addition, some recent researchers studied ORAMs from different perspectives. Bindschaedler et. al. proposed an ORAM system, CURIOUS [14], to study the ORAM performance for cloud application with $O(B \cdot \sqrt{N})$ client-side storage and data block size of no more than 64KB. Later, Sahin et. al. proposed a TaoStore [15] based on Path-ORAM to further resolve the asynchronicity issue in CURIOUS.

In this paper, we propose TSKT-ORAM, in which data blocks are outsourced to two independent servers where the data blocks are stored in k-ary trees and evictions are delayed and aggregated to reduce its cost. The design can further reduce the costs of data access pattern protection and simultaneously accomplish the following performance goals: (i) *communication efficiency* - $O(\frac{\log N}{\log \log N} \cdot B)$ communication cost per query when the block size $B \geq N^{\epsilon}$ bits for some constant $0 < \epsilon < 1$; (ii) *storage efficiency* - $O(B)$ storage space at the client side and $O(B \cdot N)$ storage space at the cloud server side; and (iii) *computational efficiency* - the server only needs to perform simple XOR operations.

We have conducted detailed analysis to show the security of the proposed TSKT-ORAM. We have also implemented TSKT-ORAM and Path-ORAM in a system of remotely connected servers and clients, and conduct experiments to compare their performance. In the comparison, we have measured the actual communication costs per query and query and access delay for each data query. The results show that the communication cost of TSKT-ORAM is 25% to 33% of that of Path-ORAM in real-world. In terms of the access delay which includes the time for both a query and its follow-up eviction, TSKT-ORAM also significantly outperforms Path-ORAM.

In the following, Section II defines the problem. Section III describes our proposed TSKT-ORAM construction. The security analysis and Implementation-based comparisons are reported in Section IV and V, respectively. Finally, Section VI concludes the paper.

II. Problem Definition

Assume that a client selects two cloud storage servers (for example Amazon S3 and Google Drive) which are independent and do not collude with each other, and exports two identical copies of N encrypted and equal-size data blocks to them. Later on, the client may access the exported data every now and then, and wishes to keep the pattern of her accesses private from the servers.

The client's private data request to the remote storage is one of the following types: (i) read a data block D of unique ID i from the storage, denoted as $D = (read, i)$; (ii) write/modify a data block D of unique ID i to the storage, denoted as $(write, i, D)$. For simplicity, we also denote the type of each request as (op, i, D) where op is either $read$ or $write$.

978-1-5090-3781-0/16/$31.00 ©2016 IEEE

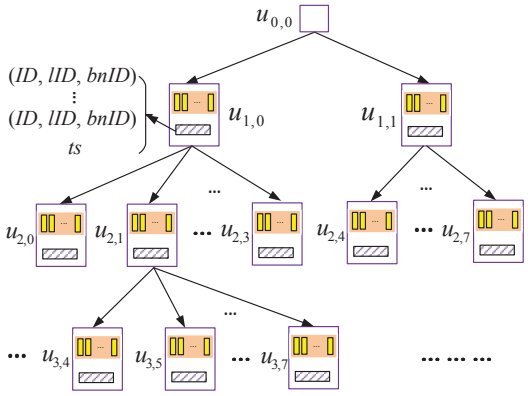

Figure 1. An example TSKT-ORAM scheme with a quaternary-tree storage structure.

To accomplish a private data request, the client needs to access some location(s) of the remote storage. Each location access, which is observable by the server, is one of the following types: (i) read a data block D from a location l at the remote storage, denoted as $D = (read, l)$; (ii) write a data block D to a location l at the remote storage, denoted as $(write, l, D)$. For simplicity, we also denote the type of each request as (op, l, D) where op is either $read$ or $write$.

Following the threat models of existing ORAM constructions [7], [9], we also consider each remote storage server to be honest but curious; i.e., it stores data and serves the client's requests according to our defined ORAM protocol, but it may attempt to figure out the client's access pattern. We assume the network connection between the client and server is secure; in practice, this can be achieved using techniques such as SSL [16].

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 sequence of the client's private 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 location 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 $\vec{y}$, 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 no greater than $2^{-\lambda}$, where λ is a system parameter.

III. The Proposed TSKT-ORAM Construction

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

A. Server-side Storage

At each server, the data storage is organized as a *special* k-ary tree of height $H_k = \lceil \frac{\log N+1}{\log k} \rceil + 1$, where the root node only stores one data block and has only two children nodes, while each of the other nodes stores $3c(k-1)$ data blocks and can have up to k child nodes. We call each node on this k-ary tree as *k-node*. For simplicity, we set the system parameter k to a power of two. As shown in Figure 1, each non-root k-node can be mapped to a binary subtree with $k-1$ binary tree nodes (called b-nodes); hence, a whole k-ary tree can be mapped to a binary tree. Each k-node $u_{l,i}$ (where $l = 0, \cdots, H_k - 1$) has the following components:

- *Data Array (DA)*: a data container with the capacity of either 1 (for root k-node) or $3c(k-1)$ (for any non-root k-node) data blocks, where $c = 7$ is a default system parameter.
- *Encrypted Index Block (EI)*: a metadata block with 1 or $3c(k-1)$ 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: (i) ID - ID of the block; (ii) lID - ID of the leaf k-node that the block is mapped to; (iii) $bnID$ - 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.

B. 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). This table can be exported to the server, just as in T-ORAM [17], P-PIR [18], Path ORAM [7], etc. To simplify presentation of the design in this section, however, we assume the table is maintained locally at the client side. Note that, 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 temporary buffer*: a buffer used to temporarily store 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 EIs.
- $\mathcal{C}$: a counter counting the number of queries that the client has issued to the server.

C. System Initialization

To initialize the system, the client first encrypts each real data block d_i to D_i, assigns it to a randomly-selected leaf k-node at each server, and constructs dummy data blocks to fill the rest space on the tree.

The client initializes the root node to empty. For each k-node, the client initializes its EI entries to record the information of blocks stored in the node. Specifically, for each real data block, its ID field records the block ID, its lID field records the ID of the leaf k-node that the block is assigned to, and its $bnID$ field records the ID of an arbitrary leaf b-node of the binary subtree that the k-node of ID lID maps to; for

each dummy data block, its block ID is marked as "-1", while its lID and $bnID$ fields are filled with arbitrary values. The ts field of the EI is initialized to 0.

For the client-side storage, the client intializes the index table $\mathcal{I}$ to record the mapping from each real data block to the leaf k-node that it is assigned to, stores the keys for data and index table encryption in the permanent storage space, and initializes the counter $\mathcal{C}$ to 0.

D. Data Query

The client queries data block D_t of ID t as follows.

- The client increments the counter $\mathcal{C}$, and checks the index table to find out the leaf k-node (denoted as $u_{H_k-1,f}$) which D_t maps to. Hence, a path from the root to $u_{H_k-1,f}$ on the k-ary tree is identified. For simplicity, we denote the path as $\overrightarrow{u} = (u_0, \cdots, u_{H_k-1})$.
- For each k-node u_l $(1 \le l \le H_k - 1)$ on $\overrightarrow{u}$, the client retrieves the EI, and checks if D_t is in the node. Note that the root k-node never stores a real block at the query time. Then, the client constructs two $3c(k-1)$-bit query vectors, denoted as $\overrightarrow{Q}_1^l$ and $\overrightarrow{Q}_2^l$. If D_t is on certain position m of the node, the client sets the two bit vectors as follows:

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

  where each r_i is a bit randomly selected from $\{0, 1\}$. If D_t is not in the node,

$$\overrightarrow{Q}_1^l = \overrightarrow{Q}_2^l = (r_0, r_1, \cdots, r_m, \cdots, r_{3c(k-1)-1}). \tag{2}$$

  Then, the client sends $\overrightarrow{Q}_1^l$ to $\mathcal{S}_1$ and $\overrightarrow{Q}_2^l$ to $\mathcal{S}_2$.
- Each server $\mathcal{S}_j$, where $j \in \{0, 1\}$, computes $\hat{D}_j$ as

$$\bigoplus_{1 \le l \le H_k - 1} \left(\bigoplus_{\forall D \in \{D \text{ on pos } i \text{ of } u_l | 0 \le i \le 3c(k-1)-1, Q_j^l[i]=1\}} D \right), \tag{3}$$

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

After D_t has been accessed, the client randomly maps the block to a leaf k-node, re-encrypts the block, and then uploads it to the root node of each server.

E. Data Eviction

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

1) Overview: Data eviction in TSKT-ORAM is conducted over its logical binary tree, and a sketch of the process is as follows. First, one or two nodes from each layer of the logical binary tree are randomly selected. Specifically, if a layer has only one node, the node is selected; otherwise, two nodes 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. 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 2, the eviction from $v_{3,3}$ to its child nodes is an intra k-node eviction, as $v_{3,3}$ and its child nodes belong to the same k-node $u_{2,3}$. On the other hand, the eviction from $v_{2,0}$ to its child nodes is an inter k-node eviction, as $v_{2,0}$ 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.

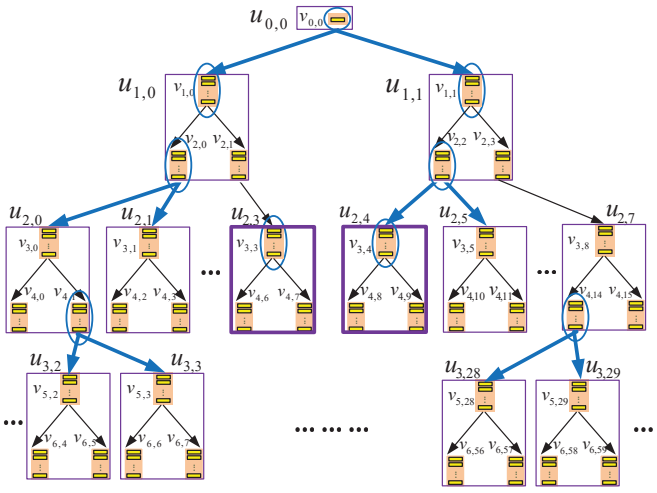

Figure 2. An example data eviction process in TSKT-ORAM with a quaternary-tree storage structure. The b-nodes selected to evict data blocks are circled. The k-node scheduled with delayed evictions (i.e., $u_{2,3}$) is highlighted with bold boundaries.

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 are not selected to evict any data blocks. In Figure 2, $u_{2,3}$ is an example 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 2, evictions from b-nodes $v_{3,3}$ can be delayed. Later on, when $u_{2,3}$ is accessed, the delayed evictions shall be executed before any other updates.

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

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 ($l \in \{\log k, 2\log k, \cdots, (\lceil \frac{\log N+1}{\log k} \rceil - 1) \cdot \log k\}$). 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 2, b-nodes $v_{2,0}$ and $v_{2,2}$, 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.

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 III-D, 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) 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 IV, 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

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

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 IV 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.

IV. Security Analysis

In the security analysis which can be found in our technical report [19], we first show that with proper setting of parameters, TSKT-ORAM construction fails with a probability of $2^{-\lambda}$ 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 TSKT-ORAM, if each DA stores $3c(k-1)$ data blocks, given $N \geq 2^{16}$, $k = \log N$ and $c = 7$, the probability for the DA of any k-node in the k-ary tree to overflow is $2^{-\lambda}$, where $\lambda > 2k$.

Lemma 2: Any query process in TSKT-ORAM accesses k-nodes from each layer of the k-ary tree, uniformly at random.

Lemma 3: Any eviction process in TSKT-ORAM accesses a sequence of k-nodes independently of the client's private data request.

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.

Theorem 1: TSKT-ORAM is secure under Definition II, if $N \geq 2^{16}$, $k = \log N$ and $c = 7$.

Due to page limit, please refer to [19] for the proofs of the above lemmas and theorem.

V. Performance Comparison

This section compares TSKT-ORAM with state-of-the-art ORAMs including B-ORAM [5], T-ORAM [17], G-ORAM [10], Path ORAM [7], SCORAM [12], and P-PIR [18].

A. Asymptotical Results

Table I shows the asymptotical comparison between TSKT-ORAM and existing ORAM schemes. We consider two scenarios: the index structure is exported and accessed in $O(1)$ recursion levels or in $O(\log N)$ recursion levels. From the table we can see that, TSKT-ORAM is the most communication efficient among the schemes, and can achieve a factor of $O(\log \log N)$ improvement in efficiency compared to the most communication-efficient state-of-the-art schemes such as P-PIR and Path-ORAM. Meanwhile, TSKT-ORAM requires smaller server-side storage than T-ORAM and P-PIR, and on the same level as other compared ORAM schemes.

For fairness, the system parameters of the compared schemes have been set carefully to make the schemes to have the same or similar levels of failure probability: specifically, TSKT-ORAM fails with probability $2^{-\lambda}$, T-ORAM and P-PIR fails with probability $O(N^{-c})$ $(c > 1)$, Path-ORAM fails with probability $O(N^{-\omega(1)})$.

B. Implementation Results

Experiments have been conducted based on our implementations of TSKT-ORAM and Path-ORAM to compare the communication performance between them. Both implementations are written in Java and each of them runs on a Linux PC with an Intel Core i5 Processor of 2.4GHz and dual cores. Three metrics are used to measure the communication performance: (i) communication cost, i.e., the amount of data transferred between the server and the client per query; (ii) query delay, i.e., the time elapsed from when the client issues a query for a data block till when the client can access the plaintext of the queried data block; (iii) access delay, which is defined as the time elapsed from when the client issues a query for a block till when the server finishes both the query and the follow-up eviction processes. We use the similar settings as in the simulations: In each round of experimentation, the number of exported data blocks N varies from 2^{16} to 2^{20}, and 2^{15} sequential and random data queries are conducted. We also vary the network bandwidth between the client and the server from 1 MB/s to 10 MB/s to evaluate the impact of network bandwidth on the performance.

Table II compares the communication costs between TSKT-ORAM and Path-ORAM. Specifically, it shows the actual number of data blocks transferred between the client and the server per query. As we can see, the communication cost of TSKT-ORAM is 25% to 33% of that of Path-ORAM.

Table II
Communication cost comparisons between TSKT-ORAM and Path-ORAM. The communication cost is measured in the unit of data block.

Comm Cost	B = 4KB	B = 16KB	B = 64KB	B = 256KB
Path-ORAM ($N = 2^{16}$)	170	170	170	170
TSKT-ORAM ($N = 2^{16}$)	43.1	37.8	36.4	36.1
Path-ORAM ($N = 2^{20}$)	210	210	210	210
TSKT-ORAM ($N = 2^{20}$)	52.8	46.2	44.6	44.1

Figure 3 compares the query and access delays between TSKT-ORAM and Path-ORAM in two scenarios, i.e., the client-server network bandwidth is 1MB/s or 10MB/s. As we can see from the figures, TSKT-ORAM has much shorter query delay than Path-ORAM. This is due to the following reasons: During a data query process, only one data block is retrieved from each of the two servers; while in Path-ORAM, $10 \log N$ data blocks are retrieved from the server. Hence, TSKT-ORAM spends less time for data transmission than Path-ORAM. The retrieved data blocks should be decrypted before the target data block can be accessed; therefore, TSKT-ORAM also spends less time for decryption than Path-ORAM.

Table I
ASYMPTOTICAL COMPARISONS. N IS THE TOTAL NUMBER OF DATA BLOCKS AND B IS THE SIZE OF EACH BLOCK IN THE UNIT OF BITS. $k = \log N$ AND $c = 7$ FOR TSKT-ORAM, AND $k = \log N$ FOR G-ORAM.

ORAM	Communication Cost with $O(\log N)$ Recursion Levels	Communication Cost with $O(1)$ Recursion Levels	Client-Side Storage Cost	Server-Side Storage Cost
B-ORAM [5]	$\Omega(\log^3 N \cdot B)$ for $N \leq 2^{37}$;	Same as the left column	$O(B)$	$O(N \cdot B)$
T-ORAM [17]	$O(\log^3 N \cdot B)$	$O(\log^2 N \cdot B)$	$O(B)$	$O(N \log N \cdot B)$
G-ORAM [10]	$O(\frac{\log^3 N}{\log \log N} \cdot B)$	$O(\frac{\log^2 N}{\log \log N} \cdot B)$	$O(\log^2 N \cdot B)$	$O(N \cdot B)$
Path ORAM [7] SCORAM [12]	$O(\log^2 N \cdot B)$	$O(\log N \cdot B)$	$O(\log N \cdot B)$	$O(N \cdot B)$
P-PIR	$O(\log^2 N \cdot B)$	$O(\log N \cdot B)$	$O(B)$	$O(N \log N \cdot B)$
TSKT-ORAM	$O(\frac{\log^2 N}{\log \log N} \cdot B)$	$O(\frac{\log N}{\log \log N} \cdot B)$	$O(B)$	$O(N \cdot B)$

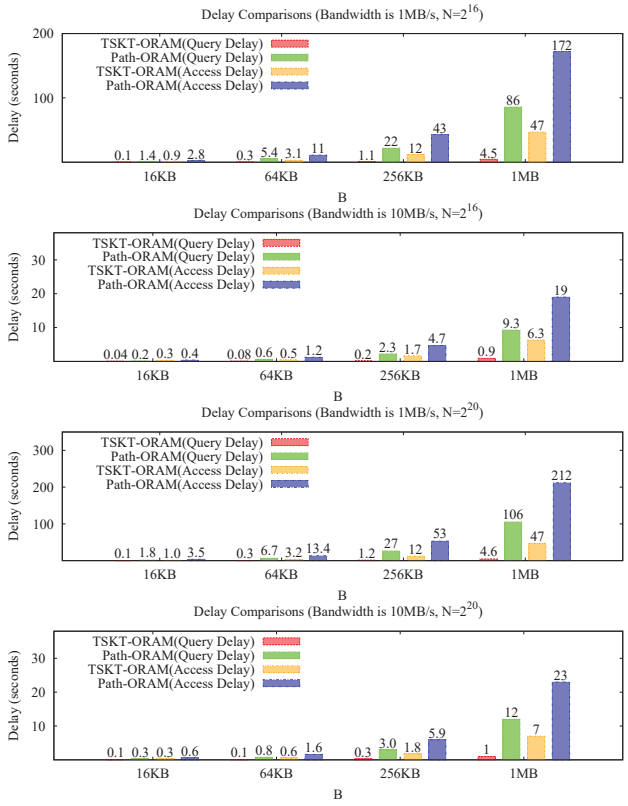

Figure 3. Performance comparison of query and access delays between TSKT-ORAM and Path-ORAM. Query delay is measured as the time elapsed from when a client issues a request to when the user can access the decrypted data. Access delay is measured as the time elapsed from when a client issues a request to the time when the server finishes eviction. Settings: the block size B ranges from 4KB to 256KB; the client-server network bandwidth is 1MB/s or 10MB/s.

TSKT-ORAM also has shorter access delay than Path-ORAM, because it incurs lower communication cost.

VI. CONCLUSION

This paper proposes TSKT-ORAM, which exports data blocks to two servers each using a k-ary tree to store the data blocks. It also adopts a novel delayed eviction technique to optimize the eviction process. TSKT-ORAM is proved to hide the data access pattern with a failure probability of $2^{-\lambda}$, where $k = \log N$ (N is the number of exported data itmes) and λ is the security parameter. Implementation-based comparisons have also been conducted to compare its performance with state-of-the-art ORAM schemes.

ACKNOWLEDGEMENT

This work was partly supported by NSF under grant CNS-1422402.

REFERENCES

[1] M. Islam, M. Kuzu, and M. Kantarcioglu, "Access pattern disclosure on searchable encryption: ramification, attack and mitigation," in *Proc. NDSS*, 2012.
[2] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious RAMs," *Journal of the ACM*, vol. 43, no. 3, May 1996.
[3] M. T. Goodrich and M. Mitzenmacher, "Mapreduce parallel cuckoo hashing and oblivious RAM simulations," in *Proc. CoRR*, 2010.
[4] ——, "Privacy-preserving access of outsourced data via oblivious RAM simulation," in *Proc. ICALP*, 2011.
[5] 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.
[6] P. Williams, R. Sion, and A. Tomescu, "PrivateFS: a parallel oblivious file system," in *Proc. CCS*, 2012.
[7] E. Stefanov, M. V. Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: an extremely simple oblivious RAM protocol," in *Proc. CCS*, 2013.
[8] E. Stefanov and E. Shi, "ObliviStore: high performance oblivious cloud storage," in *Proc. S&P*, 2013.
[9] E. Stefanov, E. Shi, and D. Song, "Towards practical oblivious RAM," in *Proc. NDSS*, 2011.
[10] 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.
[11] E. Stefanov and E. Shi, "Multi-cloud oblivious storage," in *Proc. CCS*, 2013.
[12] X. S. Wang, Y. Huang, T.-H. H. Chan, A. Shelat, and E. Shi, "SCORAM: oblivious RAM for secure computation," in *Proc. CCS*, 2014.
[13] T. Moataz, T. Mayberry, and E.-O. Blass, "Constant Communication ORAM with Small Blocksize," in *Proc. CCS*, 2015.
[14] V. Bindschaedler, M. Naveed, X. Pan, X. Wang, and Y. Huang, "Practicing Oblivious Access on Cloud Straoge: the Gap, the Fallacy, and the New Way Forward," in *Proc. CCS*, 2015.
[15] C. Sahin, V. Zakhary, A. E. Abbadi, H. Lin, and S. Tessaro, "TaoStore: Overcoming Asynchronicity in Oblivious Data Storage," in *Proc. S&P*, 2016.
[16] A. Freier, P. Karlton, and P. Kocher, "The secure sockets layer (SSL) protocol version 3.0," in *RFC 6101*, 2011.
[17] 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.
[18] T. Mayberry, E.-O. Blass, and A. H. Chan, "Efficient private file retrieval by combining ORAM and PIR," in *Proc. NDSS*, 2014.
[19] J. Zhang, Q. Ma, W. Zhang, and D. Qiao, "TSKT-ORAM: A two-server k-ary tree ORAM for access pattern protection in cloud storage." [Online]. Available: http://www.cs.iastate.edu/~wzhang/TSKTORAMfull.pdf