KT-ORAM: An Efficient ORAM Built on k-ary Tree of PIR Nodes

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

Abstract. This paper proposes KT-ORAM, a new hybrid ORAM-PIR construction, to protect a client's access pattern to outsourced data. KT-ORAM organizes the server storage as a k-ary tree with each node acting as a fully-functional PIR storage, and adopts a novel delayed eviction technique to optimize the eviction process. KT-ORAM is proved to protect the data access pattern privacy at a failure probability negligible in N (N is the number of exported data blocks), when system parameter $k = \log N$. Under the same configuration, KT-ORAM has an asymptotical communication cost of $O(\frac{\log N}{\log \log N} \cdot B)$ when the recursion level on meta data is of $O(1)$ depth, which can be achieved if block size $B = N^{\epsilon}$ ($0 < \epsilon < 1$), or $O(\frac{\log^2 N}{\log \log N} \cdot B)$ when the number of recursion levels is $O(\log N)$. The costs of KT-ORAM are compared with those of several state-of-the-art ORAMs. The results show that, KT-ORAM achieves the best communication, storage and client-side computational efficiency simultaneously, at the price of requiring $O(\frac{\log^2 N}{\log \log N} \cdot B)$ computational cost at the storage server for each data query.
Key words: Cloud storage, Access pattern privacy, Oblivious RAM, and PIR.

1 Introduction

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 user's access pattern to the data is revealed [17].

The private information retrieval (PIR) [4,1,3,9,10,21,2,22,19,18] and the oblivious RAM (ORAM) [11,12,15,13,14,20,24,31,32,33,25,26,28,29,8,27,6,30] are two well-known security-provable approaches to protect a client's access pattern to 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 [31], the ORAM approach still appears 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 Path-ORAM [26] proposed by Stefanov et al. and P-PIR [23] proposed by Mayberry et al., both consuming $O(B \cdot \log N)$ bandwidth for each data query, when the total number of exported data blocks is N and each data block is of size $B = N^{\epsilon}$ bits for constant $0 < \epsilon < 1$.

Though Path-ORAM and P-PIR have 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.

Our Results. In this paper, we propose a new ORAM construction, named KT-ORAM, to further reduce the costs of data access pattern protection and simultaneously accomplish the following performance goals:

- *communication efficiency* - $O(\frac{\log N}{\log \log N} \cdot B)$ communication cost per data query when $B = N^{\epsilon}$ bits for constant $0 < \epsilon < 1$;
- *storage efficiency* - $O(B)$ storage space at the client side and $O(B \cdot N)$ storage space at the cloud server side; and
- *computational efficiency* - $O(\frac{\log N}{\log \log N} \cdot B)$ computational cost at the client side and $O(\frac{\log^2 N}{\log \log N} \cdot B)$ computational cost at the server side per data query.

To the best of our knowledge, there is no other ORAM constructions can simultaneously achieve the same or better level of communication, storage and client-side computational efficiency. Compared to P-PIR [23] and Path-ORAM [26], which are the most communication-efficient ORAM constructions with and without constant size client-side storage respectively, KT-ORAM reduces the communication cost by a factor of $O(\log \log N)$. Regarding other costs, Path-ORAM requires $O(B \cdot \log N)$ storage space at the client side, while KT-ORAM requires only $O(B)$; P-PIR requires $O(B \cdot N \cdot \log N)$ storage space at the server side, while KT-ORAM only needs $O(B \cdot N)$.

Though the server-side computational cost introduced by KT-ORAM is higher than several state-of-the-art ORAM constructions, the cost is $O(\log \log N)$ times lower than that required by P-PIR. As justified by Mayberry et al. [23] in the evaluation of P-PIR, reducing communication cost at the price of increasing computational cost is favorable to both clients and cloud server in practice: On one hand, because public cloud providers charge CPU time with much lower rate than that for communication, the communication/computation tradeoff exploited by KT-ORAM can reduce the monetary cost for clients. Particularly, the evaluation of P-PIR [23] has shown P-PIR to be very competitive among related works in terms of monetary cost; since KT-ORAM has lower communication and computational costs than those for P-PIR, the monetary cost of KT-ORAM should be even more competitive. On the other hand, just like P-PIR, the computation required by KT-ORAM can be performed in a highly-parallel, efficient manner by massive nodes and CPUs of the cloud data center.

Our Methodology. Similar to P-PIR [23], KT-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 [25], 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, KT-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. [8] also have used k-ary tree in their designed ORAM construction which we call *G-ORAM* hereafter. However, KT-ORAM and G-ORAM use different eviction algorithms to evict actual data blocks within the tree. Also, they have different requirements on the size of each tree node: to achieve the same level of failure probability $O(2^{-k})$, G-ORAM requires each tree node to contain $O(k^2)$ data blocks while KT-ORAM only needs $O(k)$ blocks due to its new eviction algorithm. Both differences have contributed to the different levels of communication efficiency that they can accomplish: when security parameter $k = \log N$, G-ORAM requires to transfer $O(\frac{\log^2 N}{\log \log N})$ data blocks per query, while KT-ORAM only requires $O(\frac{\log N}{\log \log N})$ blocks.

KT-ORAM should use an eviction algorithm different from the one used in G-ORAM due to following reasons: The AH operation-based PIR primitives can help to reduce communication cost only 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 were applied, the number of blocks moved out/in from/to a node could be as large as $O(\log^2 N)$, which is the total number of blocks in a node; hence, to obliviously perform these movements, all the blocks in the node have to be downloaded and re-uploaded, no matter PIR primitives are used or not.

For KT-ORAM, we design a novel eviction algorithm that moves only one data block out/in from/to each selected tree node during an eviction process, so as the PIR primitives can be utilized to further reduce communication cost. Specifically, the algorithm treats the physical k-ary tree as a logical binary tree, where each k-ary tree node mapped to a logical binary subtree. Over the logical tree, a binary tree-based eviction algorithm, similar to the one used in Tree ORAM [25] and P-PIR [23], is logically simulated but not directly executed. Instead, an idea of *delayed eviction* is further employed to defer and aggregate as many as possible the logical eviction operations to ensure that: (i) for each eviction process, only $O(\frac{\log N}{\log k})$ 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.

Organization of the Paper. In the rest of the paper, Section 2 presents the problem definition. Section 3 reviews the related works of ORAM and PIR. Section 4 introduces the preliminary techniques, which is followed by detailed descriptions of the construction in Section 5. Sections 6 and 7 report the security and cost analysis. Finally, Section 8 concludes the paper.

2 Problem Definition

We consider a system as follows. A client exports N equal-size data blocks to a remote storage server. He/she 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 $(read, i, D)$; or (ii) write/modify a data block D of unique ID i to the storage, denoted as a 3-tuple $(write, i, D)$.

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 $(read, l, D)$; or (ii) upload (i.e., write) a data block D to a location l at the remote storage, denoted as a 3-tuple $(write, l, D)$.

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 [7].

Following the security definition of ORAMs [11,26,29], we specify the security of our proposed ORAM as follows.

Definition 1. *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 negligible in N.*

3 Related Work

Oblivious RAM. According to the data lookup technique adopted, existing ORAMs can be classified into two categories, namely, hash-based ORAMs and index-based ORAMs. Hash-based ORAMs [11,12,15,13,14,20,24,31,32,33] require data structures such as buckets or stashes to deal with hash collisions. Among them, the Balanced ORAM (B-ORAM) [20] 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 [25,26,28,29,8,27] 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) [25], Path ORAM [6], and SCORAM [30]. 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 incurs $O(B \cdot \log N) \cdot \omega(1)$ communication cost with $O(N^{-\omega(1)})$ failure probability and $O(B \cdot \log N) \cdot \omega(1)$ client-side storage.

Private Information Retrieval (PIR). PIR protocols were proposed mainly to protect the pattern in accessing *read-only* data at remote storage. There are two flavors: information-theoretic PIR (iPIR) [4,1,9,10], assuming multiple non-colluding servers each holding one replica of the shared data; computational PIR (cPIR) [21,2,22,3], where cPIR 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 [21]. Designed based on the hardness of quadratic residuosity decision problem, the scheme has $O(n^c)$ ($0 < c < 1$) communication cost. Since then, several other single-server cPIRs [2,22] 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 [19,18] have been proposed to achieve satisfactory performance in practice, when database size is small. Due to the property of partial homomorphic encryption, [16,23] shows that these cPIR schemes can also be adapted for data updating.

Hybrid ORAM-PIR Designs Designs based on a hybrid of ORAM and PIR techniques have emerged recently. Among them, P-PIR [23] has the best-known performance. As our proposed KT-ORAM design shares some similar ideas with P-PIR, more detailed discussions of P-PIR and comparisons between KT-ORAM and P-PIR will be given in the following sections.

4 Preliminaries

Our proposed KT-ORAM employs the additively homomorphic encryption [19,18] primitives and shares some basic ideas with P-PIR [23]. Hence, this section provides an overview of additively homomorphic encryption primitives and P-PIR.

4.1 Additively Homomorphic Encryption

Additively Homomorphic encryption (AH encryption) [19,18] is a fundamental primitive used in our proposed design of KT-ORAM. Letting A and B be two data items, and $\mathcal{E}(*)$ denote an AH encryption (which is also a probabilistic encryption), the following properties hold:

$$\begin{aligned} &\mathcal{E}(A) \oplus \mathcal{E}(B) = \mathcal{E}(A + B), \\ &\mathcal{E}(A) \odot B = \mathcal{E}(A \cdot B). \end{aligned} \tag{1}$$

Here, $+$ and $\cdot$ are regular addition and multiplication operations between two data items; $\oplus$ stands for a homomorphic addition between two homomorphically-encrypted data items; the "homomorphic" multiplication (denoted as $\odot$) between a homomorphically-encrypted data item $\mathcal{E}(A)$ and a data item B represents the homomorphic summation of B identical copies of $\mathcal{E}(A)$, i.e., $\oplus_{i=1}^{B}\mathcal{E}(A)$.

Based on an AH encryption, primitives PIR-read and PIR-write have been defined in AH-based PIR constructions [23]. As they are also used in KT-ORAM, we introduction their definitions below. Suppose a client exports to a storage server w double-encrypted data blocks, denoted as $\overrightarrow{\mathcal{E}(E(D))} = (\mathcal{E}(E(D_1)), \cdots, \mathcal{E}(E(D_w)))$, where $E(*)$ represents a symmetric encryption such as AES [5]. Primitives PIR-read and PIR-write are defined as follows.

PIR-read(m) When the client wishes to query data block D_m without exposing D_m's position m to the server, it should issue a PIR-read(m) request as follows: (i) The client first constructs a query vector $\overrightarrow{q}$ of w entries, in which only the m^{th} entry is $\mathcal{E}(1)$ while each of the other entries is $\mathcal{E}(0)$. (ii) The vector $\overrightarrow{q}$ is then sent to the server.

Upon receiving the request, the server performs the following homomorphic encryption operation for each entry q_i of $\overrightarrow{q}$:

$$c_i = q_i \odot \mathcal{E}(E(D_i)) = \begin{cases} \mathcal{E}(0), & \text{if } i \neq m; \\ \mathcal{E}(\mathcal{E}(E(D_i))), & \text{otherwise.} \end{cases} \tag{2}$$

Then, the server calculates

$$c_1 \oplus \cdots \oplus c_w = \mathcal{E}(\mathcal{E}(E(D_m))). \tag{3}$$

Lastly, this result is returned to the user, who will decrypt it to obtain D_m.

PIR-write(m, ΔD) When the client wishes to replace D_m with D'_m without exposing the change to the server, it should issue a PIR-write(m, ΔD) request as follows: (i) The client first computes $\Delta D = E(D'_m) - E(D_m)$. (ii) Then, it constructs a writing vector $\overrightarrow{q}$ of w entries, in which only the m^{th} entry is $\mathcal{E}(1)$ while each of the other entries is $\mathcal{E}(0)$. (iii) Finally, ΔD and $\overrightarrow{q}$ are both sent to the server.

Upon receiving the request, the server conducts the following computations for each $i \in \{1, \cdots, w\}$:

$$\mathcal{E}(\Delta D_i) = q_i \odot \Delta D = \begin{cases} \mathcal{E}(0), & \text{if } i \neq m; \\ \mathcal{E}(\Delta D), & \text{otherwise.} \end{cases} \tag{4}$$

$$\begin{aligned} \mathcal{E}(E(D_i)) &= \mathcal{E}(E(D_i)) \oplus \mathcal{E}(\Delta D_i) \\ &= \begin{cases} \mathcal{E}(E(D_i)), & \text{if } i \neq m; \\ \mathcal{E}(E(D'_m)), & \text{otherwise.} \end{cases} \end{aligned} \tag{5}$$

Note that, the effect of the above operations is to change only the m^{th} block to $\mathcal{E}(E(D'_m))$ while other blocks remain intact. Also, if $m = \perp$, it means the write operation is a dummy write, and therefore all entries of $\overrightarrow{q}$ are set to $\mathcal{E}(0)$.

4.2 Overview of P-PIR

The design of P-PIR is summarized in the following from the aspects of storage organization, data query process, and data eviction process.

Storage Organization Assuming N data blocks are exported by the client to a storage server. The server-side storage of P-PIR is organized as a binary tree with $L = \log N + 1$ layers, the same as in T-ORAM [25]. Each node can store $\log N$ blocks. As the capacity of the storage is larger than the N *real* data blocks, *dummy* blocks are introduced to fill up the rest of the storage.

A real data block is first encrypted with symmetric encryption and then re-encrypted with homomorphic encryption before it is stored to a position in the node; that is, each data block D_i is stored as $\mathcal{E}(E(D_i))$ in a node. Each node also contains an 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 1 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 number and i is the node index on the layer.

P-PIR 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 [25], the index table can be exported to the server as well; hence, the user-side storage is of constant size and only needs to store at most two data blocks and some secret information such as encryption keys.

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 $\overrightarrow{v}$ from the root to $v_{L-1,f}$ is identified.
- For each node on the path $\overrightarrow{v}$, the client first retrieves the encrypted index block from it, and checks if D_t is in the node. If D_t is at a certain position m of the node, the process PIR-read(m) (as defined in Section 4.1) is launched to retrieve D_t; otherwise, the client launched process PIR-read(x) where x is a randomly-picked position in the node.
- After D_t has been retrieved and accessed, it is re-encrypted and inserted into the root node $v_{0,0}$.

An example is given in Figure 1, 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 retrieved. Finally, block D_t is found at node $v_{5,10}$. After being accessed, it is re-encrypted and added to root node $v_{0,0}$. Therefore, the client downloads and uploads totally $2 \log N$ index and data blocks for each query.

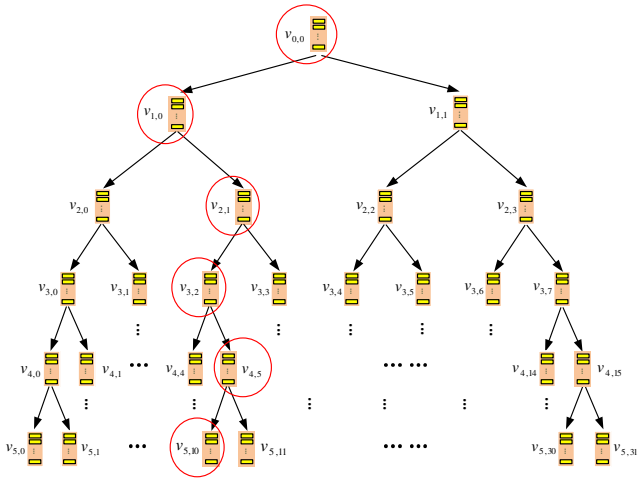


Fig. 1. P-PIR's server-side storage structure. Circled nodes represent the ones accessed by the client during a query process when the target data block is mapped to leaf node $v_{5,10}$.

Data Eviction Process To prevent any node on the tree from overflowing, the following data eviction process is conducted by the client after every query. Firstly, for each non-bottom layer l, two nodes are randomly selected. Note that, a single node $v_{0,0}$ is selected from the top layer as it only contains a single root node. Then, for each selected node $v_{l,i}$, there are two cases:

- If node $v_{l,i}$ contains at least one real data block, one such real block is selected and evicted to the child node which is on the path that the selected block is mapped to; meanwhile, a dummy eviction to another child of $v_{l,i}$ is performed to hide the actual pattern of eviction. Primitives PIR-read and PIR-write are employed together for the evictions. Specifically, the index blocks of $v_{l,i}$ and its two child nodes (denoted as $v_{l+1,j}$ and $v_{l+1,k}$) are first retrieved; based on the index information, it can be determined that a certain real block D_e in $v_{l,i}$ should be evicted to one child node (say, $v_{l+1,j}$). Then, D_e in $v_{l,i}$, a dummy block D' in $v_{l+1,j}$, and an arbitrary block D'' in $v_{l+1,k}$ are retrieved with primitive PIR-read. After that, process PIR-write(m,$E(D_e)-E(D')$) (where m is the location of D' in $v_{l+1,j}$) is performed for $v_{l+1,j}$ to obliviously update D' to D_e, and dummy process PIR-write($\perp$,x) (where x is an arbitrary value) is performed for node $v_{l+1,k}$ to pretend an update at the node. Finally, three index blocks are updated, re-encrypted, and uploaded.
- If node $v_{l,i}$ does not contain any real data block, two dummy evictions are performed to the two child nodes of $v_{l,i}$.

Figure 2 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 $v_{3,5}$.

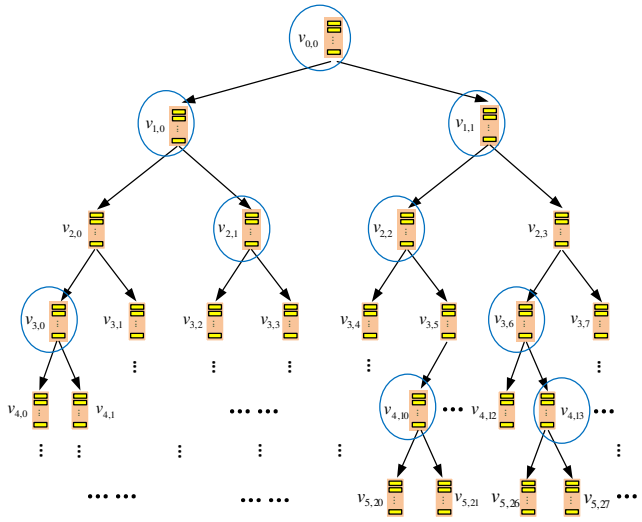
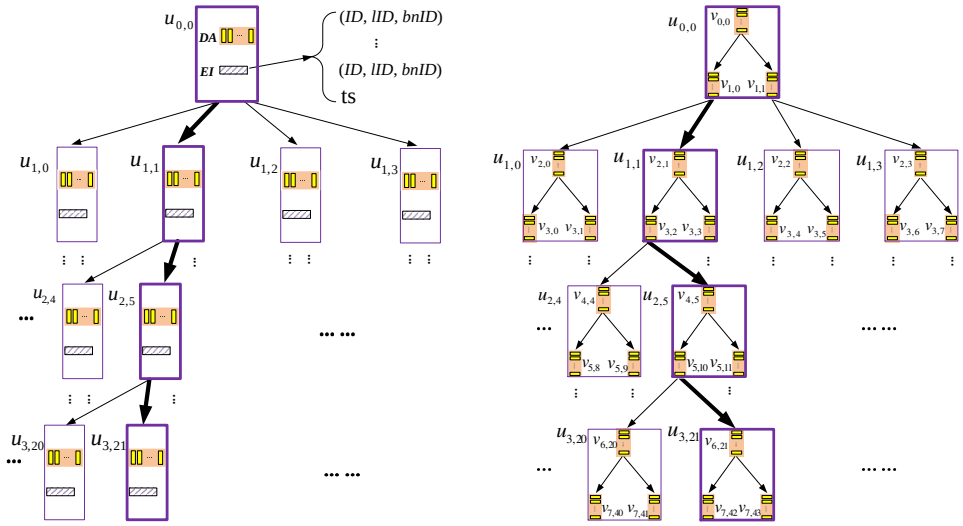


Fig. 2. An example of the eviction process in P-PIR.

5 The Proposed KT-ORAM Scheme

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

5.1 Storage Organization



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

Fig. 3. An example KT-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 $u_{3,21}$.

Server-side Storage At the server side, data storage is physically organized as a k-ary tree. For simplicity, we set the system parameter k to be a power of two. Each node in the tree (called a k-node) is a PIR storage. As shown in Figure 3, each k-node $u_{l,i}$ has the following components:

- *Data Array (DA)*: a data container that stores $c(k-1)$ data blocks, where $c \geq 1$ is another system parameter.
- *Encrypted Index Table (EI)*: a table of $c(k-1)$ entries recording the control 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 time stamp indicating when this block was accessed in the last time.

Each k-node can be mapped to a binary subtree of $k-1$ binary nodes (called b-nodes). For example, k-node $u_{0,0}$ in Figure 3(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 3(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).

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 KT-ORAM, the table can be exported to the server, just as in T-ORAM [25] and P-PIR [23]; 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 [6] and SCORAM [30], 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.

5.2 System Initialization

To initialize the system, the client acts as follows. It first prepares each real data block D_i by encrypting it with a symmetric key and then homomorphically encrypting it to get $\mathcal{E}(E(D_i))$, and then randomly assigns it to a leaf k-node on the k-ary tree maintained at the server-side storage. 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 should 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 should 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.

5.3 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 following operations are performed:

- The encrypted index table (EI) in k-node u is retrieved and decrypted. Then, delayed evictions are first executed, of which the details will be explained in Section 5.4 Phase II.
- According to the decrypted EI, the following operations are executed:
  - If block D_t is found at a certain location m of the DA in u, process PIR-read(m) will be launched by the client to retrieve $\mathcal{E}(E(D_t))$, and then decrypt and access D_t. After the access, D_t will be temporarily stored locally, and the entry for D_t in the downloaded EI is updated to mark the block as a dummy.
  - On the other hand, if D_t can not be found in u, the client will launch process PIR-read(x), where x is an arbitrary location at the DA in u, to pretend retrieving a data block, and the retrieved data block will be discarded without processing.
- Finally, if u is the root k-node, the downloaded EI is temporarily saved locally; else, the downloaded EI is re-encrypted and uploaded back to u.

After all k-nodes on the path have been processed, the retrieved D_t is re-encrypted to $\mathcal{E}(E(D_t))$ and then inserted to the root k-node $u_{0,0}$, Note that this encrypted block appears differently from the one downloaded earlier as the AH encryption $\mathcal{E}(*)$ is probabilistic. Specifically, the insertion is implemented in the following steps:

- From the downloaded EI of the root k-node $u_{0,0}$, a location m' that currently stores a dummy block is identified. Note that, if such a location cannot be found, the root k-node is said to *overflow*, which is a failure of the KT-ORAM system; but as we prove in the Section 6, the probability for such failure to occur is negligibly small with proper setting of system parameters.
- The client launches process PIR-read(m') to obliviously retrieve and decrypt dummy block D' from location m'.
- The client launches process PIR-write(m', $E(D_t)-E(D')$) to obliviously replace the dummy block at location m' with $\mathcal{E}(E(D_t))$.
- The EI of the root k-node is updated to reflect the change in position m'. In addition, the lID field in the tuple corresponding to location m' should be updated to store the ID of a uniform randomly selected leaf k-node, which now becomes the current leaf k-node that D_t is mapped to. Then, the EI is re-encrypted and uploaded back to the root k-node.

As shown in Figure 3(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 $u_{3,21}$ should be 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 should be 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 $u_{0,0}$.

5.4 Data Eviction

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 should be launched in KT-ORAM immediately after each query.

Overview Logically, data eviction in KT-ORAM is conducted in its logical view similarly as binary tree ORAM eviction. In a nutshell, two nodes from each layer of the logical binary tree are randomly selected. In each of the selected nodes, if there is a real data block, the block will be evicted to one of its child b-node 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 4, 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 need PIR-read or PIR-write and could be performed more efficiently than inter k-node evictions.

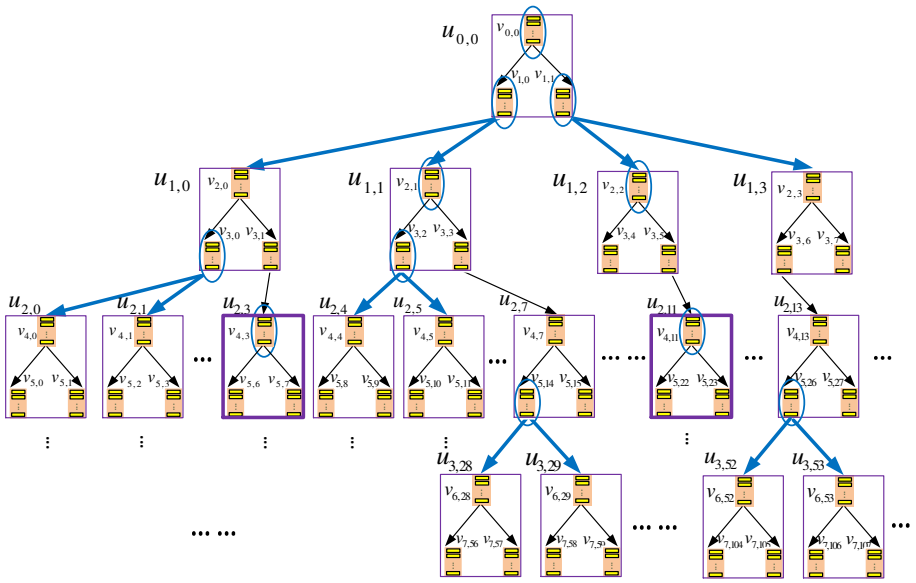


Fig. 4. An example data eviction process in KT-ORAM with a quaternary-tree storage structure. The b-nodes that are selected to evict data blocks are circled. The k-nodes scheduled with delayed evictions (i.e., $u_{2,3}$ and $u_{2,11}$) are highlighted with bold boundaries.

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 4, $u_{2,3}$ 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 4, 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.

Phase I: Selecting k-nodes for Eviction At the beginning of an eviction process, the client uniform randomly picks up two b-nodes from each binary-tree layer l ($l \in \{\log k - 1, 2\log k - 1, \cdots, (\lceil \frac{\log N+1}{\log k} \rceil - 1) \cdot \log k - 1\}$). These layers are the bottom binary subtree layers inside non-bottom k-nodes, and so evicting data blocks from the b-nodes on these layers to their child b-nodes are inter k-node evictions that should 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 4, 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, should 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 that can be delayed.

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 5.3, 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 should 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 intra k-node evictions.

Then, for each of these delayed eviction round, 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 the delayed intra k-node evictions have all been executed, the ts of k-node $u_{l,i}$ is updated to $\mathcal{C}$.

Phase III: Execution of Inter k-node Evictions For each b-node selected in Phase I, after the k-nodes that contain 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.

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.

Recall that, to perform intra k-node eviction, the EIs of $u_{l',x'}$, $u_{l'+1,y'}$, and $u_{l'+1,z'}$ have been downloaded and updated. After the intra k-node evictions, if $v_{l,x}$ stores at least one real data blocks, one of them is downloaded by using the PIR-read primitive. Let the downloaded real block be d and without loss of generality, assume k-node $u_{l'+1,y'}$ is on the path from the root to the leaf k-node that d records in its lID. Then, one dummy block d' will be downloaded from k-node $u_{l'+1,y'}$ and an arbitrary block will be downloaded from k-node $u_{l'+1,z'}$, both using the PIR-read primitive.

After that, $\mathcal{E}(E(d))$ will be written to k-node $u_{l'+1,y'}$ to replace dummy block d' by using the PIR-write primitive, and block d becomes a data block stored in the root b-node within k-node $u_{l'+1,y'}$. Simultaneously , a dummy PIR-write process is launched to update a block in k-node $u_{l'+1,z'}$ as well to hide the fact that this k-node is not actually involved in eviction. Finally, the EIs of the three k-nodes are updated to reflect the movement of block d from k-node $u_{l',x'}$ to $u_{l'+1,y'}$, and then re-encrypted and uploaded back to the server.

If $v_{l,x}$ does not have any real data block, three arbitrary blocks will be retrieved from the above three k-nodes, respectively, with the PIR-read primitive. Then, two dummy PIR-write processes will be launched to update two blocks in k-nodes $u_{l'+1,y'}$ and $u_{l'+1,z'}$, respectively. Finally, the EIs of the three k-nodes will be re-encrypted and uploaded back to the server.

6 Security Analysis

In this section, we first show that KT-ORAM construction fails with a probability of $O(2^{-k})$ through proving the DA of each k-node overflows with a probability of $O(2^{-k})$. When k is larger enough, for example, $k = (\log N)^{c'}$ for $c' \geq 1$, the probability is $O(\frac{1}{N^{c'}})$ which is

negligible with large N. Then, we show that both the 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. *When $5 \leq c \leq k-2$, the probability for the DA of any k-node in the k-ary tree to overflow is $O(2^{-k})$.*

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

Lemma 3. *An eviction process in KT-ORAM accesses a sequence of k-nodes independently of the client's private data request.*

Please refer to Appendices for proofs of the lemmas.

Theorem 1. *Assuming PIR-read and PIR-write are both oblivious operations, KT-ORAM is secure under Definition 1.*

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.
- Finally, both PIR-read and PIR-write operations are oblivious. Hence, each PIR-read or PIR-write operation does not expose which data block within a k-node is actually read or written, or what has been written in the case of write operation.

Also, according to Lemma 1, the KT-ORAM construction fails with a probability of $O(2^{-k})$, which is considered negligible when k is large enough. For example, when $k = (\log N)^{c'}$ for $c' \geq 1$, the probability is $O(\frac{1}{N^{c'}})$ which is negligible with large N.

7 Cost Analysis

We analyze the costs of KT-ORAM, and compares KT-ORAM with state-of-the-art ORAMs. To simplify the analysis and comparison, we set system parameters $k = \log N$ and $c = 5$. Also, we assume block size $B = \Omega(N^{\epsilon})$ bits for constant $0 < \epsilon < 1$. Not that, this assumption ensures that, if the client exports the index structure (metadata) to the server, the recursion level is $O(1)$ [6]; the assumption is also consistent with practical scenarios [23,29], for example, when $N \leq 2^{40}$ and $B \geq 64$ K bytes, $B \geq N^{1/2}$ bits.

7.1 Costs of KT-ORAM

We analyze the performance of KT-ORAM in terms of communication, storage and computational costs. The following notations and assumptions are used in the analysis:

- b: size of an additively homomorphic encryption cipher-text, in the unit of bits.

- H_k and H_b: heights of the k-ary and binary trees, respectively. According to the scheme, $H_b = \lceil \log N \rceil$ and $H_k = \lceil \frac{H_b}{\log \log N} \rceil = \lceil \frac{\log N}{\log \log N} \rceil$.
- S_{EI}: size of the EI at a node. According to the scheme, $S_{EI} = c \cdot (k-1) \cdot \{\log N + \log(N/k) + \log[c \cdot (k-1)]\}$, which is less than $10 \log^2 N$ bits due to $c = 5$ and $k = \log N$.
- We assume the client stores the index table $\mathcal{I}$ locally and thus there is no recursion. Note that, when recursion is considered, the overall communication and computational costs can be obtained by simply multiplying the costs computed in this subsection with the depth of recursion.

Communication Cost Per Query During a query process, one k-node is accessed from each layer of the k-ary tree. The client needs to (i) download the EI of the k-node (which is S_{EI} bits); (ii) send one PIR-read vector (i.e., $5b \cdot (\log N - 1)$ bits); (iii) upload the EI (i.e., S_{EI} bits). After all PIR-reads have been executed by the server, the server gets H_k data blocks. The client does not retrieve these H_k blocks immediately. Instead, another PIR-read is launched on these blocks to fetch only one of them. This PIR-read requires the client to send one PIR-read vector (i.e., $H_k \cdot b$ bits) and download the target data block (i.e., B bits). To wrap up a query process, the target data block is obliviously written back to the root k-node using one PIR read and one PIR-write Note that, as EI has been retrieved before, these PIR-read and PIR-write only need to transfer the read and write vectors and two data blocks; hence, the cost is $10b \cdot (\log N - 1) + 2B$ bits. Therefore, the communication cost per query is:

$$Qu(N) \leq 3B + H_k \cdot (2S_{EI} + 5b \cdot \log N - 4b) + 10b \cdot (\log N - 1) = O(B + \log^3 N) \text{ bits.} \quad (6)$$

According to the assumption of $B = \Omega(N^\epsilon)$ for constant $0 < \epsilon < 1$,

$$Qu(N) = O(B). \quad (7)$$

During an eviction process, up to two k-nodes for each non-bottom layer in the k-ary tree are selected for eviction, each of which requires one PIR-read. The four child k-nodes of these selected evicting k-nodes are also accessed, each requiring one PIR-read and one PIR-write. The EIs of these six k-nodes are retrieved and uploaded back after the eviction operations. All these need transferring $12S_{EI}$ bits. Optimization techniques can be applied here. Though two child k-nodes of each selected evicting k-node are accessed, only one of them has a block updated; hence, only one block needs to be downloaded from each pair of child k-nodes. Therefore, the client uses PIR-read to retrieve up to four data blocks (two evicting blocks from the two evicting k-nodes plus two blocks from the four child k-nodes of the evicting k-nodes), which need transferring up to $4B + 30b \cdot (\log N - 1)$ bits. Similarly, when PIR-write is used to update child k-nodes, only one data block is written to each pair of child k-nodes; hence, only two data blocks need to be uploaded. Therefore, the communication cost for each eviction process is

$$Ev(N) \leq (H_k - 1) \cdot (12S_{EI} + 30b \cdot \log N + 6B) = O(\frac{\log N}{\log \log N} \cdot (\log^2 N + B)) \text{ bits.} \quad (8)$$

Due to the assumption of $B = \Omega(N^\epsilon)$ for $0 < \epsilon < 1$,

$$Ev(N) = O(B \cdot \frac{\log N}{\log \log N}). \quad (9)$$

Therefore, the overall communication cost per query is

$$Qu(N) + Ev(N) = O(B \cdot \frac{\log N}{\log \log N}). \quad (10)$$

Storage Costs According to the design, the client-side storage cost is dominated by the size of a constant number of data blocks, i.e., $O(B)$. The storage at the server side needs to store $2c \cdot N$ data blocks, hence its size is $O(N \cdot B)$.

Computational Costs per Query The computational costs at the server and clients are analyzed in the following.

Server-side Computational Cost. During a query process, a PIR-read operation is conducted on each accessed k-node. As we analyze previously, the total number of accessed k-node is H_k. Each k-node has $c \cdot (k-1) = 5(\log N - 1)$ blocks and each block has B bits. Each block is processed by AH operations in the unit of piece (i.e., b bits). Hence, the server conducts $Comp_{Mul} = 5(\log N - 1) \cdot B/b$ AH multiplications and $Comp_{Add} = [5(\log N - 1) - 1] \cdot B/b$ AH additions. Therefore, the computational cost for a query process is $H_k \cdot (Comp_{Mul} + Comp_{Add}) = O(\frac{\log^2 N}{\log \log N} \cdot \frac{B}{b})$ AH operations. During an eviction process, at most one PIR-read and one PIR-write operations are conducted on each accessed k-node. The number of accessed k-nodes is bounded by $6H_k$ and the cost of PIR-write is similar to that of PIR-read except that the number of AH additions is $2(\log N - 1) \cdot B/b$. Therefore, the computational cost for an eviction process is also $O(\frac{\log^2 N}{\log \log N} \cdot \frac{B}{b})$ AH operations. In total, the server-side computational cost is

$$O(\frac{\log^2 N}{\log \log N} \cdot \frac{B}{b}) \tag{11}$$

AH operations per query.

Client-side Computational Cost. During a query process, the client needs to generate one PIR-vector for each layer on the k-ary tree, where each vector contains $c(k-1) = O(\log N)$ elements. Then, for the retrieved target data block, it needs to decrypt and re-encrypt it with both homomorphic and regular encryption and decryption algorithms. Hence, the cost for the query process is

$$O([\frac{\log^2 N}{\log \log N} + \frac{B}{b}] \cdot C_{AH} + B \cdot C_{reg}), \tag{12}$$

which is $O(\frac{B}{b} \cdot C_{AH})$ because (i) homomorphic encryption/decryption of a block is more costly than regular encryption/decyption of it, and (ii) $B = \Omega(N^{\epsilon})$ $(0 < \epsilon < 1)$ is assumed. During an eviction process, the client needs to generate one PIR-vector for each accessed k-nodes, and there are up to six such blocks for each layer. Also, up to two data blocks from each non-bottom layer are downloaded, decrypted, re-encrypted and uploaded. Hence, the cost for the eviction process is

$$O(\frac{\log N}{\log \log N} \cdot [\log N + \frac{B}{b}] \cdot C_{AH} + \frac{\log N}{\log \log N} \cdot B \cdot C_{reg}), \tag{13}$$

which is

$$O(\frac{\log N}{\log \log N} \cdot \frac{B}{b} \cdot C_{AH}). \tag{14}$$

This is also the total per-query computational cost at client-side.

7.2 Comparisons with Existing ORAMs

Detailed comparisons between KT-ORAM and several state-of-the-art ORAMs including B-ORAM [20], T-ORAM [25], G-ORAM [8], Path ORAM [6], SCORAM [30], and P-PIR [23] are reported in this section.

Communication Costs We compare the communication costs of different ORAM constructions asymptotically and through numeric simulations.

Table 1. Asymptotical Communication Costs per Query. N is the total number of data blocks and B is the size of each block in the unit of bit. $k = \log N$ and $c = 5$ for KT-ORAM, and $k = \log N$ for G-ORAM. The column of "Communication Cost*" assumes the constructions (except for B-ORAM) export recursively index structures and the recursion depth is $O(\log N)$. The column of "Communication Cost**" assumes the constructions (except for B-ORAM) either not export index structures or export them recursively and the recursion depth is $O(1)$ due to the assumption of $B = \Omega(N^\epsilon)$ bits for $0 < \epsilon < 1$.

ORAM	Communication Cost*	Communication Cost**
B-ORAM [20]	$\Omega(\log^3 N \cdot B)$ for $N \leq 2^{37}$; $O(\frac{\log^2 N}{\log\log N} \cdot B)$ for $N > 2^{37}$	Same as the left column
T-ORAM [25]	$O(\log^3 N \cdot B)$	$O(\log^2 N \cdot B)$
G-ORAM [8]	$O(\frac{\log^3 N}{\log\log N} \cdot B)$	$O(\frac{\log^2 N}{\log\log N} \cdot B)$
Path ORAM [6], SCORAM [30]	$O(\log^2 N \cdot B)$	$O(\log N \cdot B)$
P-PIR	$O(\log^2 N \cdot B)$	$O(\log N \cdot B)$
KT-ORAM	$O(\frac{\log^2 N}{\log\log N} \cdot B)$	$O(\frac{\log N}{\log\log N} \cdot B)$

Asymptotical Comparisons. Table 1 shows that KT-ORAM is the most communication efficient among the ORAM constructions. Particularly, compared to P-PIR, Path-ORAM and SCORAM, which are the most communication-efficient state-of-the-art constructions, a factor of $\log\log N$ improvement in the efficiency is accomplished by KT-ORAM.

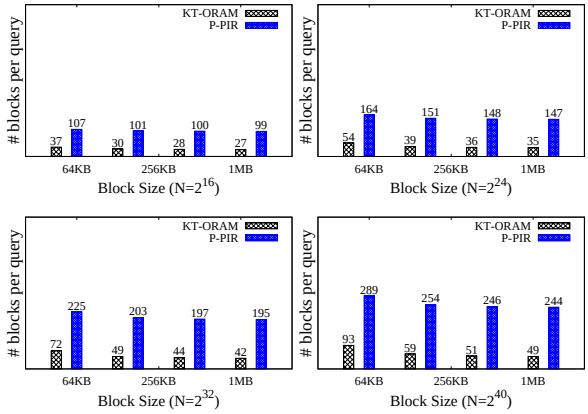


Fig. 5. Numerical comparison of communication cost in practical scenarios. $k = \log N$ and $c = 5$ for KT-ORAM. The number of blocks N ranges from 2^{16} to 2^{40} and the block size B ranges from 64 K bytes to 1 M bytes.

Numeric Comparisons. Figure 5 shows the results of numeric comparisons between KT-ORAM and P-PIR in practical scenarios [23,29]. As we can see, the cost of KT-ORAM is only $1/3$ to $1/5$ of that of P-PIR. Note that, we do not numerically compare KT-ORAM with other state-of-the-art ORAM constructions, as [23] has conducted the comparison and showed P-PIR to be more efficient than the others.

Table 2. Storage Costs. N is the total number of data blocks and B is the size of each block in the unit of bit. $k = \log N$ and $c = 5$ for KT-ORAM, and $k = \log N$ for G-ORAM.

ORAM	Client-side Storage Cost	Server-side Storage Cost
B-ORAM [20]	$O(B)$	$O(N \cdot B)$
T-ORAM [25]	$O(B)$	$O(N \log N \cdot B)$
G-ORAM [8]	$O(\log^2 N \cdot B)$	$O(N \cdot B)$
Path ORAM [6], SCORAM [30]	$O(\log N \cdot B)$	$O(N \cdot B)$
P-PIR	$O(B)$	$O(N \log N \cdot B)$
KT-ORAM	$O(B)$	$O(N \cdot B)$

Storage Costs Table 2 compares KT-ORAM with state-of-the-art ORAM constructions in terms of the client-side and server-side storage costs. As we can see, KT-ORAM does not require more client-side or storage-side space than any state-of-the-art ORAM construction. Particularly, it requires only $O(B)$ bits at the client side and $O(N \cdot B)$ bits at the server side.

Table 3. Computational Costs per Query. N is the total number of data blocks and B is the size of each block in the unit of bit. $k = \log N$ and $c = 5$ for KT-ORAM. $B = \Omega(N^\epsilon)$ bits for $0 < \epsilon < 1$. C_{Reg} denotes the cost for a regular (e.g., AES) encryption or decryption. C_{AH} denotes the cost for an additive homomorphic operation.

ORAM	Client-side Computational Cost	Server-side Computational Cost
Path ORAM [6]	$O(\log N \cdot B \cdot C_{Reg})$	N/A
P-PIR	$O(\log N \cdot \frac{B}{b} \cdot C_{AH})$	$O(\log^2 N \cdot \frac{B}{b} \cdot C_{AH})$
KT-ORAM	$O(\frac{\log N}{\log \log N} \cdot \frac{B}{b} \cdot C_{AH})$	$O(\frac{\log^2 N}{\log \log N} \cdot \frac{B}{b} \cdot C_{AH})$

Computational Costs Table 3 shows that KT-ORAM reduces the computational costs of P-PIR by a factor of $O(\log \log N)$. Though KT-ORAM requires higher server-side computational computational costs than other state-of-the-art ORAM constructions, it achieves higher communication efficiency. As argued in Section 1 and in [23], the tradeoff is favorable to both client and user in practice.

8 Conclusion

This paper proposes a new, security-provable hybrid ORAM-PIR construction called KT-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. KT-ORAM is proved to protect the data access pattern privacy at a failure probability negligible in N (N is the number of exported data blocks), when $k = \log N$. KT-ORAM has an asymptotical communication cost of $O(\frac{\log N}{\log \log N} \cdot B)$ when the recursion level on meta data is of $O(1)$ depth with uniform block size $B = N^\epsilon$ ($0 < \epsilon < 1$), or $O(\frac{\log^2 N}{\log \log N} \cdot B)$ when the number of recursion levels is $O(\log N)$. Compared to state-of-the-art ORAM constructions, KT-ORAM achieves the best communication, storage and client-side computational efficiency simultaneously, at the price of requiring $O(\frac{\log^2 N}{\log \log N} \cdot B)$ computational cost at the storage server per data query.

References

1. Beimel, A., Ishai, Y., Kushilevitz, E., Raymond, J.F.: Breaking the $O(n^{\frac{1}{2k-1}})$ barrier for information-theoretic private information retrieval. In: In Proc. FOCS (2002)
2. Cachin, C., Micali, S., Stadler, M.: Computationally private information retrieval with polylogarithmic communication. In: In Proc. Eurocrypt (1999)
3. Chor, B., Gilboa, N.: Computationally private information retrieval. In: In Proc. Theory of Computing (2000)
4. Chor, B., Goldreich, O., Kushilevitz, E., Sudan, M.: Private information retrieval. In: In Proc. FOCS (1995)
5. Daemen, J., Rijmen, V.: The Design of Rijndael. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2002)
6. Emil, S., van Dijk Marten, Elaine, S., Christopher, F., Ling, R., Xiangyao, Y., Srinivas, D.: Path ORAM: an extremely simple oblivious RAM protocol. In: Proc. CCS (2013)
7. Freier, A.O., Karlton, P., Kocher, P.C.: The secure sockets layer (SSL) protocol version 3.0. In: RFC 6101 (2011)
8. Gentry, C., Goldman, K., Halevi, S., Julta, C., Raykova, M., Wichs, D.: Optimizing ORAM and using it efficiently for secure computation. In: Proc. PETS (2013)
9. Gertner, Y., Ishai, Y., Kushilevitz, E., Malkin, T.: Protecting data privacy in private information retrieval schemes. In: In Proc. STOC (1998)
10. Goldberg, I.: Improving the robustness of private information retrieval. In: In Proc. S&P (2007)
11. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious RAM. Journal of the ACM 43(3) (May 1996)
12. Goodrich, M.T., Mitzenmacher, M.: Mapreduce parallel cuckoo hashing and oblivious RAM simulations. In: Proc. CoRR (2010)
13. Goodrich, M.T., Mitzenmacher, M.: Privacy-preserving access of outsourced data via oblivious RAM simulation. In: Proc. ICALP (2011)
14. Goodrich, M.T., Mitzenmacher, M., Ohrimenko, O., Tamassia, R.: Oblivious RAM simulation with efficient worst-case access overhead. In: Proc. CCSW (2011)
15. Goodrich, M.T., Mitzenmacher, M., Ohrimenko, O., Tamassia, R.: Privacy-preserving group data access via stateless oblivious RAM simulation. In: Proc. SODA (2012)
16. Helger, L., Bingsheng, Z.: Two new efficient PIR-writing protocols. In: Zhou, J., Yung, M. (eds.) Applied Cryptography and Network Security, Lecture Notes in Computer Science, vol. 6123. Springer Berlin Heidelberg (2010), http://dx.doi.org/10.1007/978-3-642-13708-2_26
17. Islam, M., Kuzu, M., Kantarcioglu, M.: Access pattern disclosure on searchable encryption: ramification, attack and mitigation. In: Proc. NDSS (2012)
18. Jeffrey, H., Jill, P., JosephH, S.: NTRU: A ring-based public key cryptosystem. In: Buhler, J. (ed.) Algorithmic Number Theory, Lecture Notes in Computer Science, vol. 1423, pp. 267–288. Springer Berlin Heidelberg (1998), http://dx.doi.org/10.1007/BFb0054868
19. Jonathan, T., Andy, P.: Efficient computationally private information retrieval from anonymity or trapdoor groups. In: Burmester, M., Tsudik, G., Magliveras, S., Ili, I. (eds.) Information Security, Lecture Notes in Computer Science, vol. 6531, pp. 114–128. Springer Berlin Heidelberg (2011), http://dx.doi.org/10.1007/978-3-642-18178-8_10
20. Kushilevitz, E., Lu, S., Ostrovsky, R.: On the (in)security of hash-based oblivious RAM and a new balancing scheme. In: Proc. SODA (2012)
21. Kushilevitz, E., Ostrovsky, R.: Replication is not needed: single database, computationally-private information retrieval (extended abstract). In: Proc. FOCS (1997)
22. Lipmaa, H.: An oblivious transfer protocol with log-squared communication. In: In Proc. ISC (2005)
23. Mayberry, T., Blass, E.O., Chan, A.H.: Efficient private file retrieval by combining ORAM and PIR. In: Proc. NDSS (2014)
24. Pinkas, B., Reinman, T.: Oblivious RAM revisited. In: Proc. CRYPTO (2010)
25. Shi, E., Chan, T.H.H., Stefanov, E., Li, M.: Oblivious RAM with $O((\log N)^3)$ worst-case cost. In: Proc. ASIACRYPT (2011)
26. Stefanov, E., van Dijk, M., Shi, E., Fletcher, C., Ren, L., Yu, X., Devadas, S.: Path ORAM: an extremely simple oblivious RAM protocol. In: Proc. CCS (2013)

27. Stefanov, E., Shi, E.: Multi-cloud oblivious storage. In: In Proc. CCS (2013)
28. Stefanov, E., Shi, E.: ObliviStore: high performance oblivious cloud storage. In: Proc. S&P (2013)
29. Stefanov, E., Shi, E., Song, D.: Towards practical oblivious RAM. In: Proc. NDSS (2011)
30. Wang, X.S., Huang, Y., Chan, T.H.H., Shelat, A., Shi, E.: SCORAM: oblivious RAM for secure computation. In: Proc. CCS (2014)
31. Williams, P., Sion, R.: Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In: Proc. CCS (2008)
32. Williams, P., Sion, R., Tomescu, A.: PrivateFS: a parallel oblivious file system. In: Proc. CCS (2012)
33. Williams, P., Sion, R., Tomescu, A.: Single round access privacy on outsourced storage. In: Proc. CCS (2012)

Appendix I: Proof of Lemma 1.

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 KT-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 KT-ORAM completely simulates the eviction process of T-ORAM and P-PIR over the logical binary tree, their results [25] 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 $l \geq 2$, an unique stationary distribution exists for the Chain; that is,

$$\pi_l(i) = \rho_l^i(1 - \rho_l), \tag{15}$$

where

$$\rho_l = \frac{\alpha_l(1 - \beta_l)}{\beta_l(1 - \alpha_l)} = \frac{2^l - 2}{2(2^l - 1)} \in \left[\frac{1}{3}, \frac{1}{2}\right). \tag{16}$$

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 KT-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

$$\mathbf{Pr}\left[Y_u = t\right] = \mathbf{Pr}\left[X_{v_1} + \cdots + X_{v_{k-1}} = t\right]. \tag{17}$$

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:

$$\begin{aligned}
\mathbf{Pr}\left[Y_u = t\right] &\leq \binom{t+k-2}{k-2} \prod_{i=1}^{k-1} \left[\left(\frac{1}{2}\right)^{t_i} \left(\frac{2}{3}\right)\right] && (18)\\
&\leq \left(\frac{(t+k-2)\cdot e}{k-2}\right)^{k-2} \left(\frac{1}{2}\right)^{t} \left(\frac{2}{3}\right)^{k-1} && (19)\\
&< \left(\frac{(t+k-2)\cdot e}{k-2}\right)^{k-1} \left(\frac{1}{2}\right)^{t} \left(\frac{2}{3}\right)^{k-1}\\
&\leq \left(\frac{2(t+k-2)\cdot e}{3(k-2)}\right)^{k-1} \left(\frac{1}{2}\right)^{t}.
\end{aligned}$$

Here, Equation (18) 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 (15). Inequality (19) 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:

$$\begin{aligned}
\mathbf{Pr}[Y_u = t] \ &\leq \ [\frac{2}{3} \cdot e \cdot (c+1+\frac{c}{k-2}) \cdot (\frac{1}{2})^c]^{k-1} && (20)\\
&< \ [2 \cdot (c+2) \cdot (\frac{1}{2})^c]^{k-1} && (21)\\
&< \ (\frac{1}{2})^{k-1} = \ (\frac{1}{2})^{t/c}. && (22)
\end{aligned}$$

Here, Inequality (20) is due to $t = c(k-1)$ in the scheme, Inequality (21) is due to $c \leq k-2$, and Inequality (22) is because $c \geq 5$ and therefore $2(c+2) < 2^{c-1}$.

Then, the following inequalities follows:

$$\begin{aligned}
\mathbf{Pr}\left[Y_u \geq t\right] \ &= \ \sum_{i=0}^{\infty} \mathbf{Pr}\left[Y_u = t+i\right]\\
&< \ \sum_{i=0}^{\infty} [(\frac{1}{2})^{1/c}]^{t+i}\\
&= \ \frac{4}{1-2^{-1/c}} \cdot 2^{-k} \ = \ O(2^{-k}). && (23)
\end{aligned}$$

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 N/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

$$X_i = \begin{cases} 1 \text{ the } i^{\text{th}} \text{ ball is thrown into this bin,} \\ 0 \text{ otherwise.} \end{cases} \quad (24)$$

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

$$E[X] = E\left[\sum_{i=1}^{N} X_i\right] = \sum_{i=1}^{N} E[X_i] = N \cdot \frac{k}{N} = k. \quad (25)$$

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

$$\begin{aligned}
&\mathbf{Pr}\left[\text{at least } j \text{ balls in this bin}\right]\\
= \ &\mathbf{Pr}\left[X \geq j\right] < \left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right)^k < \left(\frac{e^\delta}{(2e)^\delta}\right)^k = 2^{-k\delta}. && (26)
\end{aligned}$$

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

$$\mathbf{Pr}\left[\text{at least } c(k-1) \text{ balls in this bin}\right] < 2^{-ck+c+k} = 2^c \cdot 2^{(c-1)k} = O(2^{-k}), \quad (27)$$

due to $c \geq 5$.

Appendix II: Proof of Lemma 2.

(sketch) In KT-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.