KT-ORAM: A Bandwidth-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 preserve a client's access pattern to his/her outsourced data. The construction 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 preserve the data access pattern privacy with a negligibly-small failure probability of $O(N^{-\log N})$. KT-ORAM requires only a constant-size local storage at the client side, and has an asymptotical communication cost of $O(\frac{\log^2 N}{\log\log N})$ (the best known asymptotical result of ORAM [17]) when $k = \log N$. The communication cost of KT-ORAM is also compared with two state-of-the-art ORAM constructions, B-ORAM [17] and P-PIR [20], which share the same assumption of constant-size client-side storage as KT-ORAM, in practical scenarios. The results show that, KT-ORAM outperforms these constructions.

1. INTRODUCTION

To preserve a client's access pattern to his/her data exported to a remote storage, the oblivious RAM (ORAM) [10, 11, 14, 12, 13, 17, 21, 27, 28, 29, 22, 26, 24, 25, 7, 23] and private information retrieval (PIR) [4, 1, 3, 8, 9, 18, 2, 19, 16, 15] constructions have been developed as security-provable solutions. The practicality of these constructions, however, are still questionable. Specifically, existing PIR solutions have been found infeasible when the access pattern to a large data set needs to be concealed [27]. ORAM appears to be more practical, however, the state-of-the-art constructions still incur high communication or storage costs. The Balanced ORAM (B-ORAM) [17] has the best-known asymptotical communication cost of $O(\frac{\log^2 N}{\log\log N})$, where N is the total number of exported data items, and needs only a constant-size local storage at the client. However, a big constant is hidden behind the big-O notation. For example, according to practical evaluations [20], it demands a client to retrieve more than 1000 extra data items in order to query just one useful item. Recently, Marberry et al. [20] proposed a hybrid ORAM-PIR construction named P-PIR, which organizes the server storage as a binary tree-based ORAM with each node acting as a smaller PIR storage. The asymptotical communication cost of P-PIR is $O(\log^2 N)$, higher than that of B-ORAM, but it achieves better bandwidth efficiency than B-ORAM in practice. Even though, P-PIR is still costly in communication; as their evaluation demonstrates, fetching 1 MB useful data requires the client to download/upload nearly 200 MB data from/to the server.

In this paper, we propose a new hybrid ORAM-PIR construction called KT-ORAM, to achieve: (i) an asymptotical communication cost of $O(\frac{\log^2 N}{\log\log N})$, which is on the same order as that of B-ORAM but with a much smaller constant behind the big-O notation; (ii) a better bandwidth-efficiency than B-ORAM and P-PIR in practice; and (iii) a lower failure probability than the state-of-the-art ORAM constructions and P-PIR.

Our proposed KT-ORAM construction shares the similar idea of building an ORAM storage as a tree with each node acting as a fully-functional PIR storage. However, significant redesigns have been conducted to the storage structure and the query and eviction processes, based on the following key ideas: (i) replacement of the binary tree-based ORAM storage with a k-ary tree-based storage to reduce the query cost from $O(\log^2 N)$ to $O(\frac{\log^2 N}{\log k})$; (ii) mapping the k-ary tree to a logical binary tree and executing evictions on the binary tree; and (iii) delaying evictions to reduce the eviction cost from $O(\log^2 N)$ to $O(\frac{\log^2 N}{\log k})$. Through the above redesigns, KT-ORAM can exploit the tradeoff between communication, client-side and server-side computational costs in a bolder way: it reduces both the communication cost and the client-side computational cost by a factor of $O(\log k)$ at the price of increasing the server-side computational cost by a factor of $O(\frac{k}{\log k})$. We argue that, the above tradeoff is highly beneficial in practice, because the following are common characteristics in cloud computing environments: (i) the communication bandwidth is usually much more expensive than the server-side computational resource; (ii) the client-side computational resource is usually more constrained and hence more expensive than the server; and (iii) the server usually has a high level of parallelism and hence is able to perform intensive computational task in short time.

Comprehensive security analysis has been conducted to analyze the KT-ORAM performance. The results show that the construction can preserve a client's data access pattern with a negligibly-small failure probability of $O(N^{-\log N})$, which is lower than the failure probabilities of P-PIR (i.e., $O(N^{-c})$ with c as a constant) and B-ORAM (i.e., $O(N^{-\log\log N})$).

Theoretical, numerical, and simulation-based analysis has been conducted to evaluate the cost of KT-ORAM, and compare it with P-PIR and B-ORAM. Results show that, the asymptotical communi-

cation cost of KT-ORAM is $O(\frac{\log^2 N}{\log \log N})$ when the system parameter k is set to $\log N$. In practical scenarios where N ranges from 2^{16} to 2^{40} and $k = \log N$, the communication cost of KT-ORAM is only $1/3$ to $1/5$ of that of P-PIR, and it is at least 20 times lower than that of B-ORAM.

In the rest of the paper, Section 2 presents the problem definition. Section 3 reviews the related works on ORAM and PIR. Section 4 introduces the preliminary techniques, and discusses the intuitions of KT-ORAM, which is followed by detailed description 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

Similar to existing ORAM constructions such as T-ORAM [22] and P-PIR [20], we consider a system as follows. A client exports N large, 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 [6].

Following the security definition of ORAMs [10, 26, 25], we specify the security of our proposed ORAM as follows.

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

3. RELATED WORK

This section reviews related works on oblivious RAM (ORAM) and private information retrieval (PIR).

3.1 Oblivious RAM

In the past decades, numerous ORAM constructions have been proposed as provable solutions to hide a client's access pattern to their data stored at a remote storage. According to the data lookup technique adopted, existing ORAMs can be classified into two categories, namely, hash-based ORAMs and index-based ORAMs.

Some of existing ORAMs [10, 11, 14, 12, 13, 17, 21, 27, 28, 29] are hash-based ORAMs. These ORAMs require some special data structure, for example, buckets and stashes, to deal with hash collisions. To the best of our knowledge, the Balanced ORAM (B-ORAM) [17] proposed by Kushilevitz et. al. achieves the best asymptotical communication cost, which is $O(\frac{\log^2 N}{\log \log N})$.

Other ORAMs [22, 26, 24, 25, 7, 23] use a certain index structure for data lookup. They require the client to store the index, which is feasible only if the number of data blocks is not too large ($N \leq 2^{20}$). When the client-side storage cannot afford to store the index, the index has to be outsourced to the server recursively in a way similar to storing data, at the cost of increased communication cost. Recently, Tree-based ORAM (T-ORAM) and Path ORAM (P-ORAM) schemes have been proposed. The communication cost for T-ORAM is $O(\log^3 N)$, while P-ORAM only incurs $O(\log^2 N)$ communication cost with some small constants behind the big-O notation.

3.2 Private Information Retrieval (PIR)

Private information retrieval (PIR) protocols have been proposed to preserve the pattern in accessing *read-only* data from a remote storage. There are two flavors of PIR protocols: information-theoretic PIR (iPIR) [4, 1, 8, 9], which assumes multiple non-colluding servers each holding one replica of the shared data; computational PIR (cPIR) [18, 2, 19, 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 [18]. 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, 19] have been proposed based on different intractability assumptions. Even though cPIRs are impractical when database size is large, they are still acceptable for small databases. Recently, several additively homomorphic encryption-based cPIRs [16, 15] have been proposed to achieve satisfactory performance in practice, when database size is small. Due to the property of additively homomorphic encryption, [20] shows that these cPIR schemes can also be adapted for data updating, which is to be elaborated in detail in Section 4.

3.3 Hybrid ORAM-PIR Designs

Recently, designs based on a hybrid of ORAM and PIR techniques have emerged. Among them, the most representative one is P-PIR [20]. As our proposed KT-ORAM design shares a similar idea with P-PIR, detailed discussions of P-PIR and comprehensive comparisons between KT-ORAM and P-PIR will be given next in Sections 4 and 5.

4. PRELIMINARIES

Our proposed KT-ORAM employs the additively homomorphic encryption [16, 15] primitives and shares some basic ideas with P-PIR [20]. Hence, this section starts with an overview of additively homomorphic encryption primitives and P-PIR, which is fol-

lowed by the performance limitation of P-PIR. Then, we present two straightforward methods to extend P-PIR and point out their drawbacks. Finally, we introduce the intuitions behind the design of KT-ORAM.

4.1 Additively Homomorphic (AH) Encryption

AH encryption [16, 15] 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 [20]. 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.

4.2.1 *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 [22]. 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 [22], 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.

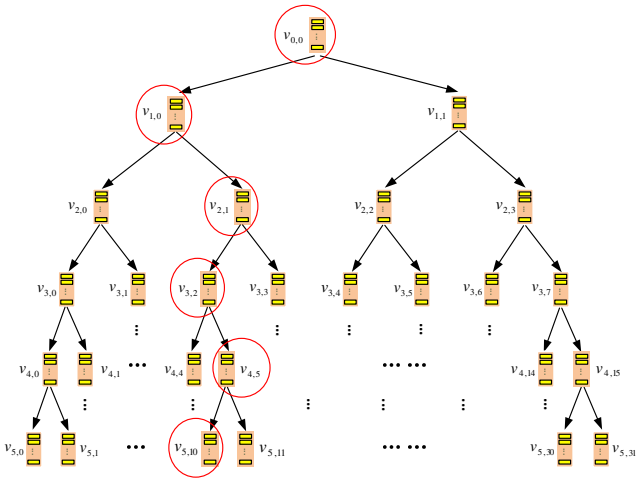


Figure 1: P-PIR's server-side storage structure. Circled nodes represent the ones accessed by the client during a query process when the target data block is mapped to leaf node $v_{5,10}$.

4.2.2 Data Query Process

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

- The client checks the index table to find out the leaf node $v_{L-1,f}$ that D_t is mapped to. Hence, a path $\overrightarrow{v}$ 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 needs to download $2 \log N$ index and data blocks for each query.

4.2.3 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 $v_{3,5}$.

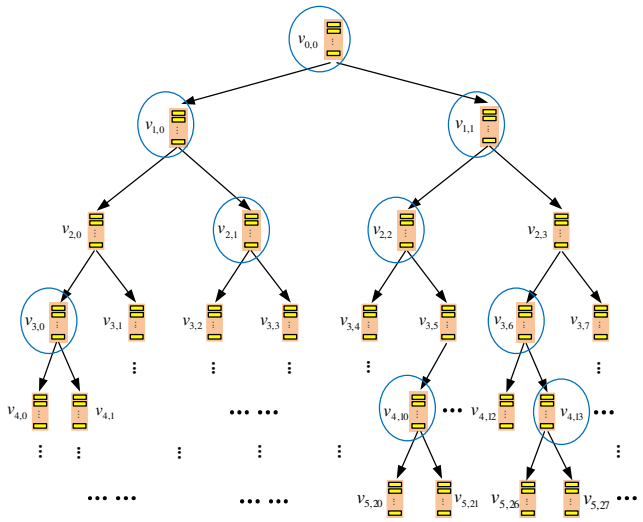


Figure 2: An example of the eviction process in P-PIR.

4.3 Limitation of P-PIR

Though P-PIR was proposed to reduce the communication cost, the overall communication cost is still as high as $O(\log^2 N)$ data blocks per query. To have a more concrete understanding of the cost, let us consider an ORAM system of 1 TB capacity and 1 MB data block size. According to the evaluation result in P-PIR [20], fetching 1 MB data incurs nearly 200 MB communication cost. Thus, 5 queries would result in almost 1 GB data transfer between the client and the server, while the requested data size is only 5 MB. Therefore, P-PIR is still expensive given the fact that bandwidth is usually more costly than computation and storage [25].

4.4 Naive Extensions of P-PIR

As the communication cost of P-PIR is mainly determined by the height of the tree structure (i.e., $\log N$), two straightforward extensions might be applied on P-PIR to reduce the tree height and hence the communication cost.

One option is to enlarge the node size. For example, let each node on the tree store $O(\alpha \log N)$ blocks, where α is an adjustable system parameter. This way, the tree height is reduced to $\log N - \log \alpha$; however, the overall communication cost is only reduced to $O(\log N(\log N - \log \alpha))$ blocks per query.

As another option, the binary tree structure used by P-PIR might be extended to a k-ary (where $k > 2$) tree structure. This way, the tree height can be reduced faster to $\frac{\log N}{\log k}$, and the communication cost for query can also be decreased to $O(\frac{\log^2 N}{\log k})$. However, oblivious eviction of a block from one single node needs to access $k+1$ nodes (i.e., the node itself and its k child nodes), which makes the communication cost of each eviction process to be $O(k \cdot \frac{\log^2 N}{\log k})$. Consequently, the overall communication cost becomes $O(k \cdot \frac{\log^2 N}{\log k})$ per query, which is higher than that of P-PIR.

4.5 Intuition of KT-ORAM

Having realized the limitations of P-PIR and its naive extensions, we propose KT-ORAM, which, similar to P-PIR, also organizes the ORAM storage as a k-ary tree (where k is a power of 2) and each node acts as a small PIR storage. However, significant redesigns

have been conducted to the storage structure and the query and eviction processes, in order to achieve a much better bandwidth efficiency. Specifically, the new ideas proposed in KT-ORAM mainly include the following:

- *Replacement of the binary tree-based ORAM storage with a k-ary tree-based storage*. As we discussed in Section 4.4, adopting this idea can reduce the height of the tree structure and thus reduce query cost from $O(\log^2 N)$ to $O(\frac{\log^2 N}{\log k})$.
- *Execution of binary-tree eviction in a k-ary tree*. As also discussed in Section 4.4, directly implementing an eviction process on the k-ary tree causes a high overhead of $O(k \cdot \frac{\log^2 N}{\log k})$ per query. To reduce the eviction cost, we propose to treat a *physical* k-ary tree as a *logical* binary tree, where every node in the k-ary tree (called *k-node* hereafter) is equivalent to a binary subtree of $k-1$ nodes (called *b-nodes* hereafter). Then, the eviction process is performed to the logical binary tree with possible *delayed evictions* described below.
- *Delayed evictions*. This is a unique process in the proposed KT-ORAM. The key idea is that evictions between b-nodes within the same k-node may not be executed immediately by the client; instead, they may be recorded by the storage server in a data structure called *eviction history (EH)*, and multiple such recorded evictions may be executed at a later time in a batch to reduce the communication cost.

5. THE PROPOSED KT-ORAM SCHEME

In this section, we present 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

5.1.1 Server-side Storage

At the server side, data storage is physically organized as a k-ary tree where k is a power of two and each node in the tree (called a k-node) is a PIR storage. As shown in Figure 3, each k-node can be mapped to a binary subtree of $k-1$ nodes. For example, k-node $u_{0,0}$ in Figure 3(a) is mapped to a 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.

In general, each k-node $u_{l,i}$ consists of the following components:

- *Data Array (DA)*: a data container that can store $(k-1)\log N$ data blocks.
- *Encrypted Index Table (EI)*: a table of $(k-1)\log N$ entries recording the control information for each block stored in the DA. Specifically, each entry is a tuple of format

  $$(ID, pos, lID, bnID)$$

  which records the following information of each block:
  - ID - ID of the block;
  - pos - position of the block in the DA;
  - 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.
- *Eviction History (EH)*: an ordered list of IDs of b-nodes. This structure is used to support *delayed evictions*, which will be elaborated later. In particular, every appearance of the ID of a b-node on the list indicates that, the b-node has been scheduled to evict a data block to its child b-node but the eviction has not been actually executed. Such a scheduled but not-yet executed eviction is called *delayed eviction*. Also, the order between the b-nodes listed on EH reflects the order in which these evictions should be executed at a later time. In KT-ORAM, EH is designed to contain up to $2\log k \log^2 N$ records.

5.1.2 Client-side Storage

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

- *A client-side index table $\mathcal{I}$*: a table of N entries, where each entry i records the ID of the leaf k-node that data block D_i is mapped to (i.e., block D_i is stored at some node on the path from the root to this k-node). In practical implementation of KT-ORAM, the table can be exported to the server, just as in T-ORAM [22] and P-PIR [20]; to simplify presentation of the design in this section, however, we assume the table is maintained locally at the client side.
- *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 data encryption and decryption.

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.

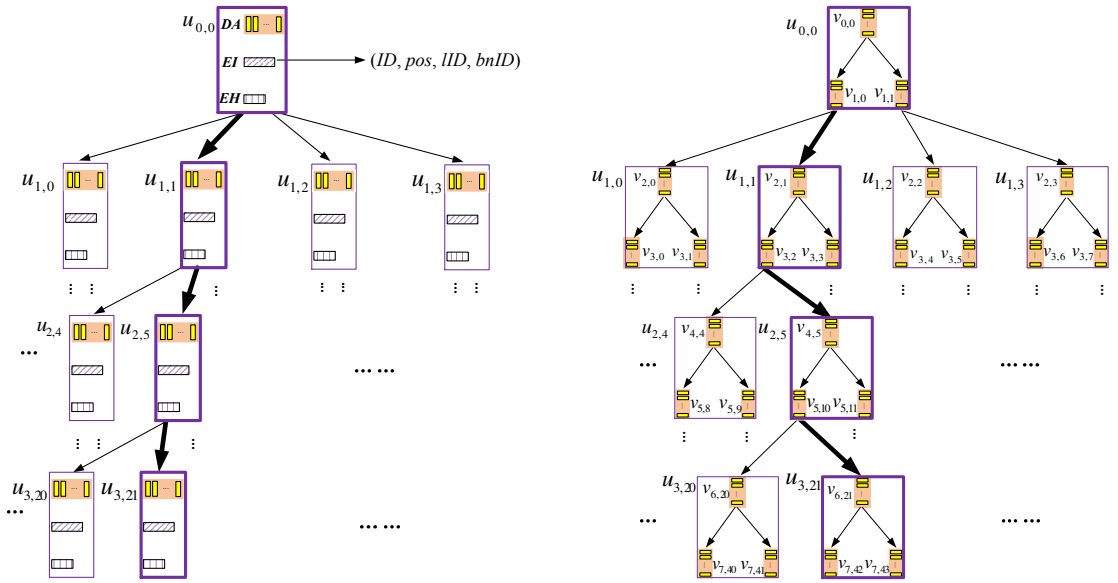
For each k-node, its EI entries are initialized to record the information of blocks stored in the node. Specifically, the entry for a real data block should record the block ID to the ID field, the ID of the assigned leaf k-node to the lID field, the position within the DA of the k-node to the pos field, 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 eviction history of the k-node is initialized to empty.

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 encryption are also recorded to a permanent storage space.

5.3 Data Query

To query a data block D_t with ID t, the client first searches the index table $\mathcal{I}$ to find out the leaf k-node that D_t is mapped to. Then, for each k-node u on the path from the root k-node to this leaf node, the following operations are performed:

- The eviction history (EH) and the encrypted index table (EI) in k-node u are retrieved and EI is decrypted. If it is non-empty, the delayed evictions recorded in EH are executed



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

Figure 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 $u_{3,21}$.

and then the EH is cleared. The details of this step will be explained later in Section 5.5, as the step would become easier to understand after the eviction process has been introduced.

- According to 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 re-mapped to another randomly-picked leaf k-node. To reflect the change, the entry for D_t in the downloaded EI should be updated to mark the block as a dummy; the entry for D_t in the client-side index table should also be updated to the ID of the newly picked leaf k-node.
  - 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.
- 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', then 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 $u_{3,21}$.

5.4 Data Eviction

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

As discussed in Section 4.5, data eviction in KT-ORAM is performed to the binary tree that the k-ary tree is logically mapped to. More specifically, the eviction process is composed of three phases as elaborated below.

5.4.1 Phase I: Scheduling of Evictions for Logical Binary Tree

At the beginning of an eviction process, the client randomly selects a list of b-nodes that should evict data blocks to their child nodes, and informs the server of the list by sending to it an eviction vector

$$\overrightarrow{e} = (e_0,\ e_1,\ \cdots,\ e_{\log N - \log k}),$$

where $e_0 = (v_{0,0})$ and for each $l \in \{1, \cdots, \log N - \log k\}$, $e_l = (v_{l,i_l}, v_{l,j_l})$ is a pair of IDs of two distinct b-nodes randomly picked from level l on the binary tree; that is, v_{l,i_l} and v_{l,j_l} should be two distinct integers randomly picked from $\{0, \cdots, 2^l - 1\}$.

5.4.2 Phase II: Identification and Recording of Delayed Evictions

Theoretically, the scheduled evictions can all be executed immediately. However, immediate execution of all of them would require the client to access $O(\log N)$ blocks, which is the same eviction cost introduced by P-PIR. To reduce the cost, we propose to delay 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, as shown in Figure 4, the scheduled 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 of the k-node to reflect the change of $bnID$ field for the evicted block. Therefore, such an eviction does not need PIR-read or PIR-write operations and could be performed more efficiently than inter k-node evictions.

Opportunities to Delay Intra k-node Evictions Opportunistically, we may find a k-node that is not involved in any other inter k-node evictions, i.e., its root b-node is not a child of any evicting b-node while 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 have been scheduled for such a k-node, they can be delayed to perform later (to update the EI of the k-node) 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 needed 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. Delayed evictions are recorded in the eviction history (EH) of the k-node in the order that they were scheduled in the eviction vector.

For example, as shown in Figure 4, evictions from b-nodes $v_{4,3}$ and $v_{4,11}$ can be delayed and hence are recorded in the EH of their k-nodes $u_{2,3}$ and $u_{2,11}$, respectively. Later on, when $u_{2,3}$ and $u_{2,11}$ are accessed, as elaborated in Section 5.5, the recorded evictions shall be executed first before any other updates.

5.4.3 Phase III: Execution of Inter k-node Evictions

All scheduled inter k-node evictions have to be executed immediately according to their appearance order in eviction vector $\overrightarrow{e}$. Specifically, the eviction for $v_{l,x}$ is performed as follows. Let $u_{l',x'}$ denote the k-node where b-node $v_{l,x}$ resides, let b-nodes $v_{l+1,y}$ and $v_{l+1,z}$ denote the two child b-nodes of $v_{l,x}$, and 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 EHs and EIs of $u_{l',x'}$, $u_{l'+1,y'}$, and $u_{l'+1,z'}$ are downloaded, and if any of the EHs are non-empty, the delayed evictions recorded in the non-empty EH shall be executed as Section 5.5 describes.

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_e 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_e is mapped to. 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, $\mathcal{E}(E(D_e))$ will be written to k-node $u_{l'+1,y'}$ to replace dummy block D' by using the PIR-write primitive, and block D_e becomes a data block stored in the root b-node within k-node $u_{l'+1,y'}$. Meanwhile, a dummy PIR-write process is launched to update a block in k-node $u_{l'+1,z'}$ as well. Finally, the EIs of the three k-nodes are updated to reflect the movement of block D_e from k-node $u_{l',x'}$ to $u_{l'+1,y'}$, re-encrypted, and uploaded back to the server.

On the other hand, if $v_{l,x}$ does not have any real data blocks, three arbitrary blocks will be retrieved from the 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.

5.5 Execution of Delayed Evictions

When a k-node is accessed during a query process or an inter k-node eviction, its eviction history (EH) may not be empty. That is, some delayed evictions may have been recorded in the EH, and these delayed evictions shall be executed before any other operations can be performed on the k-node.

Suppose the EH of an accessed k-node contains the following sequence of b-node IDs:

$$v_{l_1,i_1}, v_{l_2,i_2}, \cdots, v_{l_n,i_n},$$

which indicates that the eviction from b-node v_{l_j,i_j} ($j = 1, \cdots, n$) to one of its child b-nodes has been delayed. To execute the delay evictions, the EI of the k-node shall be updated as follows:

- If b-node v_{l_j,i_j} has at least one real data block (i.e., there is at least one real data block whose EI entry has v_{l_j,i_j} in the $bnID$ field), one of such real blocks, denoted as D_e, shall be selected. Suppose b-node $v_{l_j+1,x}$ is a child of v_{l_j,i_j} and is on the path from the root to the leaf k-node that D_e is mapped to. Then, the $bnID$ field of D_e's EI entry shall be updated to $v_{l_j+1,x}$ to indicate the eviction of D_e from v_{l_j,i_j} to $v_{l_j+1,x}$.
- On the other hand, if b-node v_{l_j,i_j} does not have any real data blocks, no change will be made to the EI as the scheduled evictions are dummy ones.
- Finally, after all the entries in the EH have been processed, the EH is cleared.

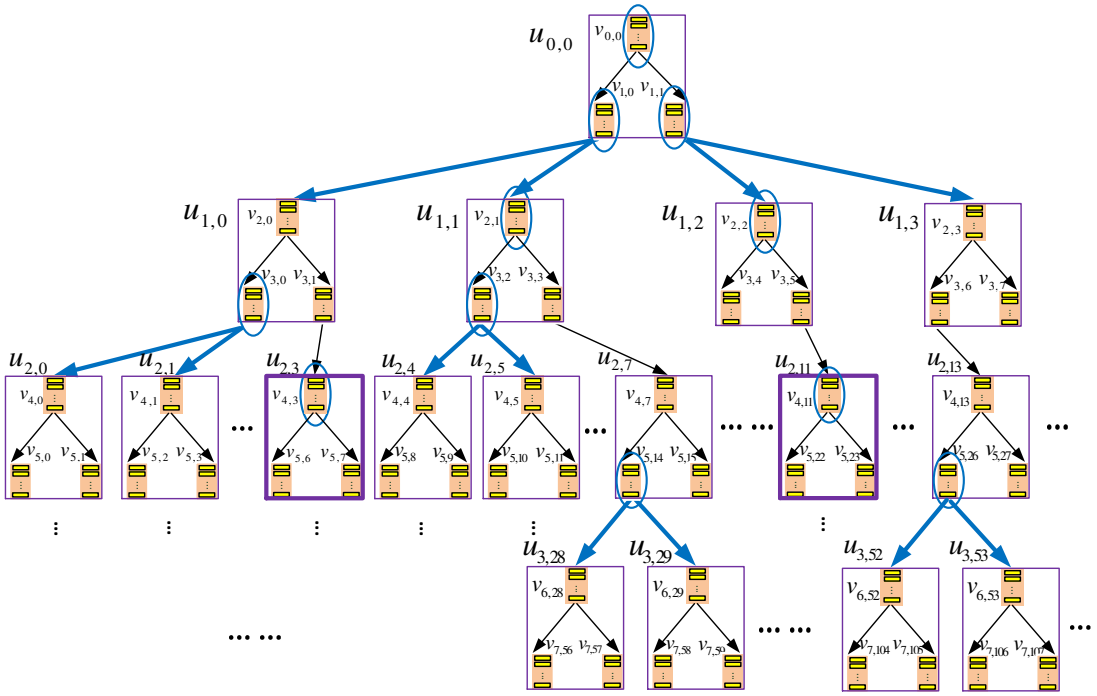


Figure 4: An example data eviction process in KT-ORAM with a quaternary-tree storage structure. The b-nodes that are selected to evict data blocks are circled. The k-nodes scheduled with delayed evictions (i.e., $u_{2,3}$ and $u_{2,11}$) are highlighted with bold boundaries.

6. SECURITY ANALYSIS

In this section, we first show that the KT-ORAM construction fails with only a negligibly small probability of $O(N^{-\log N})$ through proving (i) the DA of each k-node overflows with probability $O(N^{-\log N})$ and (ii) the EH of each k-node overflows with probability $O(N^{-\log N})$. Then, we show that the query and eviction processes both access k-nodes independently of the client's private data request. Based on the above steps, we finally present the main theorem.

LEMMA 1. *Assume $k \geq \log N$. The DA of any k-node in the k-ary tree has a probability of $O(N^{-\log N})$ to overflow.*

LEMMA 2. *In the k-ary tree, the probability that the EH of a k-node has more than $2 \log k \log^2 N$ records is $O(N^{-logN})$.*

LEMMA 3. *A query process in KT-ORAM accesses k-nodes from each layer of the k-ary tree, uniformly at random.*

LEMMA 4. *An eviction process in KT-ORAM accesses a sequence of k-nodes independently of the client's private data request.*

Due to space limitation, 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 2.*

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 3, 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 4, 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, 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 Lemmas 1 and 2, the KT-ORAM construction fails with probability $O(N^{-\log N})$, which is considered negligible and no higher than the failure probability of existing ORAMs. □

7. COST ANALYSIS

In this section, we analyze the costs of KT-ORAM, and compare KT-ORAM with the following state-of-the-art ORAM schemes: B-ORAM [17], T-ORAM [22], and P-PIR [20].

7.1 Costs of KT-ORAM

Same as T-ORAM and P-PIR, the server-side storage cost for KT-ORAM is $O(N \log N)$ data blocks. Before analyzing the communication and computational costs of KT-ORAM, we introduce the following notations:

- H_k and H_b: heights of the k-ary and binary trees. Obviously, $H_b = \log N + 1$ and $H_k = \frac{H_b}{\log k} = \frac{\log N + 1}{\log k}$.
- S_B: size of a data block in the unit of bits.
- b: size of an additively homomorphic encryption cipher-text, in the unit of bits. In practice, $b \ll S_B$.
- S_{EH}: size of an EH, which is upper bounded by $2 \log^2 k \log^2 N$ bits and is smaller than S_B in the design.
- S_{EI}: size of an EI, which is $(k-1) \cdot \log N \cdot \{2 \log N + \log[(k-1) \log N] + \log(k-1)\}$ bits and smaller than S_B in the design.

7.1.1 Per-query Communication Cost

During a query process, one k-node is accessed from each layer of the k-ary tree. For the access of each k-node, (i) its EH and EI are downloaded and then EI is uploaded after access, which consume a bandwidth of $S_{EH} + 2S_{EI}$; and (ii) a data block is read obliviously using PIR-read, which consumes $(k-1) \cdot b \cdot \log N$ bits for reading vector and S_B bits. After the above accesses, a block is obliviously written back to the root k-node using PIR-write, which consumes $(k-1) \cdot b \cdot \log N$ bits for writing vector and S_B bits. Hence, the total bandwidth consumption for a query process is

$$Qu(N) = H_k \cdot (S_{EH} + 2S_{EI}) + (H_k + 1) \cdot ((k-1) \cdot b \cdot \log N + S_B). \quad (6)$$

During an eviction process, up to six k-nodes are accessed for each layer of the k-ary tree. When a k-node is accessed, its EH and EI are downloaded and EI is uploaded after access, while at most one data block is obliviously read via PIR-read and at most one data block may be obliviously written via PIR-write. Hence, the total bandwidth consumption is bounded by

$$Ev(N) = 6H_k \cdot (S_{EH} + 2S_{EI} + 2S_B + 2 \cdot (k-1) \cdot b \cdot \log N). \quad (7)$$

Considering that the client-side index table needs to be exported recursively to keep client-side local storage constant, the overall bandwidth consumption per query is

$$\log N \cdot [Qu(N) + Ev(N)], \quad (8)$$

which is $O(\frac{\log^2 N}{\log k} \cdot S_B + \frac{\log^3 N}{\log k} \cdot b \cdot k)$ because $S_{EI} < S_B$ and $S_{EH} < S_B$. Practically, $b \ll S_B$; for example, $b = 2048$ bits while $S_B = 1$ MB in the implementation of P-PIR [20]. Hence, the overall bandwidth is $O(\frac{\log^2 N}{\log k})$ per query in practice.

7.1.2 Per-query Computational Cost

Server-side Computational Cost The server-side computational cost is dominated by the homomorphic addition (i.e., $\oplus$) and multiplication (i.e., $\odot$) operations; hence, we only count such operations.

During a query process, a PIR-read operation is conducted on each accessed k-node. As we analyzed in the previous subsection, the total number of accessed k-node is H_k. As each k-node has $(k-1) \log N$ blocks each with S_B bits and hence S_B/b data pieces operate-able by AH operations, each PIR-read operation on a k-node requires $(k-1) \log N S_B/b$ AH multiplications and $[(k-1) \log N - 1] S_B/b$ AH additions. Therefore, the computational cost for a query process is $O(\frac{k \log^2 N}{\log k} \cdot \frac{S_B}{b})$ 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. Therefore, the computational cost for an eviction process is also $O(\frac{k \log^2 N}{\log k} \cdot \frac{S_B}{b})$ AH operations.

In summary, the server-side computational cost is $O(\frac{k \log^2 N}{\log k} \cdot \frac{S_B}{b})$ AH operations per query.

Client-side Computational Cost The computational cost at the client side is mainly contributed by decrypting and re-encrypting downloaded data blocks, where each block needs both normal (e.g., AES) and homomorphic decryption/re-encryption. Since the number of data blocks accessed per query is $O(\frac{\log^2 N}{\log k})$, the numbers of normal and AH encryption/decryption operations required are both $O(\frac{\log^2 N}{\log k} \cdot \frac{S_B}{b})$.

7.2 Comparisons with Other ORAMs

The costs of KT-ORAM are compared with state-of-the-art ORAM constructions which shares the same baseline assumption, i.e., constant client-side storage, including B-ORAMand P-PIR. (Note that, we do not compare KT-ORM with ORAMs such as [7, 26], though they use a similar structure as in KT-ORAM. This is because, these constructions do not share the constant local storage assumption and it has been shown that they are outperformed by P-PIR in terms of communication cost [20].)

7.2.1 Asymptotical Comparisons

First, we show the asymptotical comparisons in terms of the communication, server-side and client-side computational costs.

From Table 1, we can see that, when $b \ll S_B$ and $k = \log N$, the communication cost of KT-ORAM is asymptotically equivalent to that of B-ORAM, a state-of-the-art ORAM construction that achieves the best asymptotical bound. However, as we will show later, KT-ORAM is much more efficient than B-ORAM in terms of communication in practical scenarios.

P-PIR and KT-ORAM both leverage the tradeoff between communication and computational costs. When bandwidth is more expensive than computation (especially server-side computational cost), it is desirable to seek low communication cost at the price of increased server-side computational cost. Comparing between them, KT-ORAM achieves an even lower communication and client-side computational costs by trading off more server-side computational cost.

7.2.2 Comparisons under Practical Settings

We now compare the communication costs of B-ORAM, P-PIR, and KT-ORAM under practical settings where b is fixed to 2048 bits as in [20], N varies from 2^{16} to 2^{40}, and block size S_B varies between 256 KB and 1 MB.

As shown in Figure 5, KT-ORAM outperforms other schemes in all the studied scenarios. Particularly, its communication cost is much lower that that of B-ORAM, though their asymptotical costs are the

Table 1: Asymptotical Cost Comparisons between B-ORAM, P-PIR, and KT-ORAM

	Communication	Client-side Computation	Server-side Computation
B-ORAM	$O(\frac{\log^2 N}{\log\log N} \cdot S_B)$	$O(\frac{\log^2 N}{\log\log N} \cdot S_B)$	N/A
P-PIR	$O(b \cdot \log^3 N + \log^2 N \cdot S_B)$	$O(\log^2 N \cdot \frac{S_B}{b})$	$O(\log^2 N \cdot \frac{S_B}{b})$
KT-ORAM	$O(b \cdot \frac{k \log^3 N}{\log k} + \frac{\log^2 N}{\log k} \cdot S_B)$	$O(\frac{\log^2 N}{\log k} \cdot \frac{S_B}{b})$	$O(\frac{k \log^2 N}{\log k} \cdot \frac{S_B}{b})$

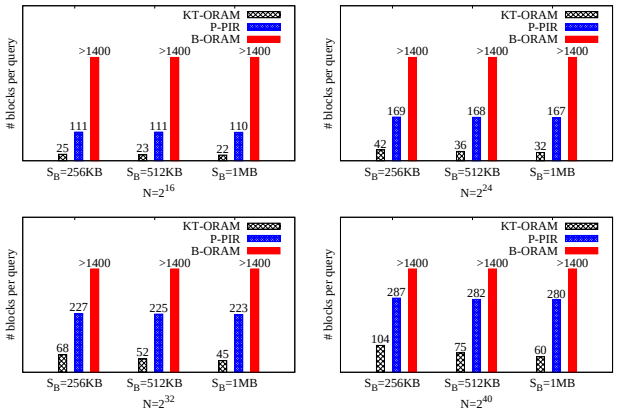


Figure 5: Comparison of communication costs under practical settings.

same. The communication cost introduced by KT-ORAM is only about $1/3$ to $1/5$ of that by P-PIR, and the improvement becomes more significant when k or S_B increases with a fixed value of N.

7.2.3 *Simulation-based Comparisons*

We simulate KT-ORAM and P-PIR using C++ and compare their performances under the settings similar to those in [20], where $S_B = 1$ MB, $b = 2048$ bits, N varies from 2^{16} to 2^{20}, and k is set to 4 or 16.

Figure 6 shows the average communication cost per query over N random requests. As we can see, the communication cost incurred by KT-ORAM is only $1/3$ to $1/5$ of that incurred by P-PIR, which conforms to the numerical results in the previous subsection.

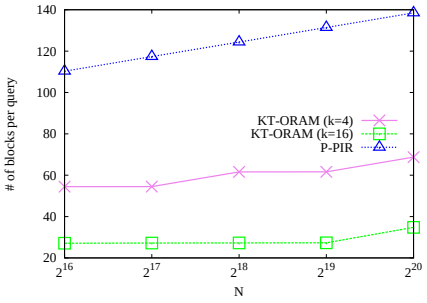


Figure 6: Comparison of communication costs using simulations.

Table 2 lists the maximum EH size when N data blocks are randomly requested. We can see that the EH size is consistently small, varying between 26 and 30 bits when $k = 4$ and between 92 and 108 bits when $k = 16$, which is significantly less than its upper bound of $2\log^2 k \log^2 N$ bits.

Table 2: Maximum EH Size in KT-ORAM (in the unit of bits)

N	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
$k = 4$	30	30	28	26	28
$k = 16$	100	92	108	100	100

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 have a negligibly-small failure probability of $O(N^{-\log N})$. Theoretically, its asymptotical communication cost is as low as $O(\frac{\log^2 N}{\log\log N})$, when $k = \log N$. In practice, it consumes significantly less bandwidth than B-ORAM and P-PIR, two state-of-the-art ORAM or hybrid ORAM-PIR solutions.

9. REFERENCES

[1] BEIMEL, A., ISHAI, Y., KUSHILEVITZ, E., AND 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., AND STADLER, M. Computationally private information retrieval with polylogarithmic communication. In *In Proc. Eurocrypt* (1999).

[3] CHOR, B., AND GILBOA, N. Computationally private information retrieval. In *In Proc. Theory of Computing* (2000).

[4] CHOR, B., GOLDREICH, O., KUSHILEVITZ, E., AND SUDAN, M. Private information retrieval. In *In Proc. FOCS* (1995).

[5] DAEMEN, J., AND RIJMEN, V. *The Design of Rijndael*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.

[6] FREIER, A. O., KARLTON, P., AND KOCHER, P. C. The secure sockets layer (SSL) protocol version 3.0. In *RFC 6101* (2011).

[7] GENTRY, C., GOLDMAN, K., HALEVI, S., JULTA, C., RAYKOVA, M., AND WICHS, D. Optimizing ORAM and using it efficiently for secure computation. In *Proc. PETS* (2013).

[8] GERTNER, Y., ISHAI, Y., KUSHILEVITZ, E., AND MALKIN, T. Protecting data privacy in private information retrieval schemes. In *In Proc. STOC* (1998).

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

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

[11] GOODRICH, M. T., AND MITZENMACHER, M. Mapreduce parallel cuckoo hashing and oblivious RAM simulations. In *Proc. CoRR* (2010).

[12] GOODRICH, M. T., AND MITZENMACHER, M. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *Proc. ICALP* (2011).

[13] GOODRICH, M. T., MITZENMACHER, M., OHRIMENKO, O., AND TAMASSIA, R. Oblivious RAM simulation with efficient worst-case access overhead. In *Proc. CCSW* (2011).

[14] GOODRICH, M. T., MITZENMACHER, M., OHRIMENKO, O., AND TAMASSIA, R. Privacy-preserving group data access via stateless oblivious RAM simulation. In *Proc. SODA* (2012).

[15] JEFFREY, H., JILL, P., AND JOSEPHH, S. NTRU: A ring-based public key cryptosystem. In *Algorithmic Number Theory*, J. Buhler, Ed., vol. 1423 of *Lecture Notes in Computer Science*. Springer Berlin

Heidelberg, 1998, pp. 267–288.

[16] JONATHAN, T., AND ANDY, P. Efficient computationally private information retrieval from anonymity or trapdoor groups. In *Information Security*, M. Burmester, G. Tsudik, S. Magliveras, and I. IliÄĞ, Eds., vol. 6531 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2011, pp. 114–128.

[17] KUSHILEVITZ, E., LU, S., AND OSTROVSKY, R. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *Proc. SODA* (2012).

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

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

[20] MARBERRY, T., BLASS, E.-O., AND CHAN, A. H. Efficient private file retrieval by combining ORAM and PIR. In *Proc. NDSS* (2014).

[21] PINKAS, B., AND REINMAN, T. Oblivious RAM revisited. In *Proc. CRYPTO* (2010).

[22] SHI, E., CHAN, T.-H. H., STEFANOV, E., AND LI, M. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *Proc. ASIACRYPT* (2011).

[23] STEFANOV, E., AND SHI, E. Multi-cloud oblivious storage. In *In Proc. CCS* (2013).

[24] STEFANOV, E., AND SHI, E. ObliviStore: high performance oblivious cloud storage. In *Proc. S&P* (2013).

[25] STEFANOV, E., SHI, E., AND SONG, D. Towards practical oblivious RAM. In *Proc. NDSS* (2011).

[26] STEFANOV, E., VAN DIJK, M., SHI, E., FLETCHER, C., REN, L., YU, X., AND DEVADAS, S. Path ORAM: an extremely simple oblivious RAM protocol. In *Proc. CCS* (2013).

[27] WILLIAMS, P., AND SION, R. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *Proc. CCS* (2008).

[28] WILLIAMS, P., SION, R., AND TOMESCU, A. PrivateFS: a parallel oblivious file system. In *Proc. CCS* (2012).

[29] WILLIAMS, P., SION, R., AND 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 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 [22] 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, $\pi_l(i) = \rho_l^i(1 - \rho_l)$, where

$$\rho_l = \frac{\alpha_l(1-\beta_l)}{\beta_l(1-\alpha_l)} = \frac{2^l - 2}{2(2^l - 1)} \in \left[\frac{1}{3}, \frac{1}{2}\right). \tag{9}$$

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

$$\mathbf{Pr}\left[Y_u = t\right] = \mathbf{Pr}\left[X_{v_1} + \cdots + X_{v_{k-1}} = t\right]. \tag{10}$$

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, according to Equation (9), we have:

$$\begin{aligned}
\mathbf{Pr}\left[Y_u = t\right] &\leq \binom{t+k-2}{k-2} \prod_{i=1}^{k-1} (\frac{1}{2})^{t_i} (1 - \frac{1}{3}) \\
&= \binom{t+k-2}{k-2} (\frac{1}{2})^t (\frac{1}{3})^{k-1} \\
&\leq \left(\frac{(t+k-2)\cdot e}{k-2}\right)^{k-2} (\frac{1}{2})^t (\frac{2}{3})^{k-1}.
\end{aligned} \tag{11}$$

Note that, the last inequality is due to $\binom{n}{k} \leq \left(\frac{n \cdot e}{k}\right)^k$ for all $1 \leq k \leq n$. Thus, given $t = (k-1)\log N$ and $k \geq \log N$, the above probability can be simplified to:

$$\begin{aligned}
&\mathbf{Pr}\left[Y_u = (k-1)\log N\right] \\
&= \left(\frac{((k-1)\log N + k - 2)\cdot e}{k-2}\right)^{k-2} (\frac{1}{2})^{(k-1)\log N} (\frac{2}{3})^{k-1} \\
&< \left(\frac{((k-1)\log N + k - 2)\cdot e}{k-2}\right)^{k-1} (\frac{1}{2})^{(k-1)\log N} (\frac{2}{3})^{k-1} \\
&\leq [e \cdot \frac{4}{3}\log N]^{k-1} (\frac{1}{2})^{(k-1)\log N} \\
&= [e \cdot \frac{4}{3}\log N (\frac{1}{2})^{\log N}]^{k-1}.
\end{aligned} \tag{12}$$

Therefore, we have:

$$\begin{aligned}
&\mathbf{Pr}\left[Y_u \geq (k-1)\log N\right] \\
&= \sum_{i=0}^{\infty} \mathbf{Pr}\left[Y_u = (k - 1 + \frac{i}{\log N})\log N\right] \\
&< \sum_{i=0}^{\infty} [e \cdot \frac{4}{3}\log N (\frac{1}{2})^{\log N}]^{k-1+i/\log N} \\
&= \frac{[e \cdot \frac{4}{3}\log N (\frac{1}{2})^{\log N}]^{k-1}}{1 - [e \cdot \frac{4}{3}\log N (\frac{1}{2})^{\log N}]^{1/\log N}}.
\end{aligned} \tag{13}$$

Equation (13) renders a negligible probability of $O(N^{-\log N})$ as long as $k \geq \log N$.

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 particular bin and let $X_1, \cdots, X_N$ be N random

variables such that

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

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

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

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

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

By applying the union bound, we obtain:

$$\begin{aligned} &\mathbf{Pr}\left[\exists \text{ a bin with at least } j \text{ balls}\right] \\ &< \frac{N}{k} \cdot 2^{-k\delta}. \end{aligned} \tag{17}$$

Further considering $j = (k-1)\log N$ and $k \geq \log N$, it follows that

$$\begin{aligned} &\mathbf{Pr}\left[\exists \text{ a bin with at least } (k-1)\log N \text{ balls}\right] \\ &< \frac{N}{\log N} \cdot 2^{-(\log^2 N - 2\log N)} \\ &= O(N^{-\log N}). \end{aligned} \tag{18}$$

Appendix II: Proof of Lemma 2.

Let us consider the EH of an arbitrary k-node u. As a root k-node is always accessed during every query and eviction process, the number of entries in its EH should never be larger than $2(\log k - 1)$, which is obviously smaller than $\log k \log^2 N$. Hence, we assume u is on layer l ($l > 0$) of the k-ary tree, and let $m = 2^l$ denote the total number of k-nodes on level l.

Since u is logically a binary subtree with $\log k$ levels, let us first consider an arbitrary binary tree level l' within u, and study the number of entries (denoted as a random variable X) that are the IDs of b-nodes on level l' in the EH.

After every eviction process, X may increase by 1 or 2 if k-node u is not accessed by the client but some intra k-node evictions have been appended; or, it may decrease to 0 if it has been accessed by the client during the eviction process. To simplify our study, we do not differentiate the cases that it increases by 1 or 2, but treat both as increasing by 2; hence, we may over-valuate X. Hence, X can be modeled as a Markov Chain as shown in Figure 7.

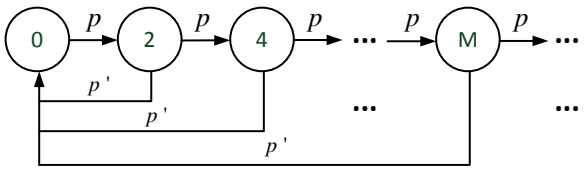


Figure 7: Markov Chain for random variable X (i.e., the number of EH entries from layer l').

Next, we compute the probability p to transition from $X = i$ to $X = i + 2$ and the probability p' to transition from $X = i$ to 0, where i is every even integer.

Transition from $X = i$ to 0 occurs when u is accessed by the client during an eviction process. This could be due to the following two cases: (i) the b-node that is the parent of the root b-node in u is selected to evict, for which the probability is $\frac{4}{m}$; (ii) a b-node on the bottom layer of the binary subtree within u is selected to evict, for which the probability is positive. So, $p' > \frac{4}{m}$ due to (i) and (ii).

Transition from $X = i$ to $X = i + 2$ occurs when one or two b-node on level l' are selected to evict. Denoting the number of b-nodes on level l' within u as n, the probability is

$$p = \frac{\binom{n}{2} + \binom{n}{1}\binom{(m-1)n}{1}}{\binom{mn}{2}} < \frac{4}{m}.$$

To further simplify the analysis, let $p' = p = \frac{4}{m}$. Note that, as p' is under-valuated and p is over-valuated, X is further over-evaluated. Then, we can find that the Markov Chain has stationary distribution $\pi = (\pi_0, \pi_2, \cdots, \pi_M)$, where $\pi_i = (\frac{1}{2})^{i/2+1}$. Hence,

$$Pr[X \geq 2\log^2 N] = (\frac{1}{2})^{\log^2 N}.$$

Node u has $\log k - 1$ such layers in its binary subtree. Its EH has more than $2\log k \log^2 N$ records, only if at least one of the layers has more than $\log^2 N$ records, for which the probability is less than $\log k \cdot (\frac{1}{2})^{\log^2 N}$, i.e., $O(N^{-\log N})$.

Appendix III: Proof of Lemma 3.

(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 IV: Proof of Lemma 4.

(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 each layer of the logical binary tree and thus is independent of the client's private data request; and (ii) the rules determining which scheduled evictions should be executed immediately (and hence the involved k-nodes should be accessed) are also independent of the client's private data requests.