2016

Data access pattern protection in cloud storage

Jinsheng Zhang
Iowa State University

Data access pattern protection in cloud storage

by

Jinsheng Zhang

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Science

Program of Study Committee:

Wensheng Zhang, Major Professor

Daji Qiao

Ying Cai

Giora Slutzki

Zhengyuan Zhu

Iowa State University

Ames, Iowa

2016

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Cloud-based storage service has been popular nowadays. Due to the convenience and unprecedent cost-effectiveness, more and more individuals and organizations have utilized cloud storage servers to host their data. However, because of security and privacy concerns, not all data can be outsourced without reservation. The concerns are rooted from the users' loss of data control from their hands to the cloud servers' premise and the infeasibility for them to fully trust the cloud servers. The cloud servers can be compromised by hackers, and they themselves may not be fully trustable.

As found by Islam et. al. [39], data encryption alone is not sufficient. The server is still able to infer private information from the user's *access pattern*. Furthermore, it is possible for an attacker to use the access pattern information to construct the data query and infer the plaintext of the data. Therefore, Oblivious RAMs (ORAM) have been proposed to allow a user to access the exported data while preserving user's data access pattern. In recent years, interests in ORAM research have increased, and many ORAM constructions have been proposed to improve the performance in terms of the communication cost between the user and the server, the storage costs at the server and the user, and the computational costs at the server and the user.

However, the practicality of the existing ORAM constructions is still questionable: Firstly, in spite of the improvement in performance, the existing ORAM constructions still require either large bandwidth consumption or storage capacity. Secondly, these ORAM constructions all assume a single user mode, which has limited the application to more general, multiple user scenarios.

In this dissertation, we aim to address the above limitations by proposing four new ORAM constructions:

- S-ORAM, which adopts piece-wise shuffling and segment-based query techniques to improve the performance of data shuffling and query through factoring block size into design;
- KT-ORAM, which 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;
- GP-ORAM, a general partition-based ORAM that can adapt the number of partitions to the available user-side storage and can outsource the index table to the server to reduce local storage consumption; and
- MU-ORAM, which can deal with stealthy privacy attack in the application scenarios where multiple users share a data set outsourced to a remote storage server and meanwhile want to protect each individual's data access pattern from being revealed to one another.

We have rigorously quantified and proved the security strengths of these constructions and demonstrated their performance efficiency through detailed analysis.

CHAPTER 1. INTRODUCTION

1.1 Motivations

Cloud-based storage service has been popular nowadays. Due to the convenience and unprecedent cost-effectiveness, more and more individuals and organizations have utilized cloud storage servers to host their data. However, because of security and privacy concerns, not all data can be outsourced without reservation.

The concerns are rooted from the users' loss of data control from their hands to the cloud servers' premise and the infeasibility for them to fully trust the cloud servers. The cloud servers can be compromised by hackers, as evidenced by more and more frequent reports in the media [54, 11]. Even the cloud servers could be technically enhanced to become more secure against the hackers, they themselves may not be fully trustable; for example, it has been revealed that some service providers sold customers' information for their profits [11].

1.2 Objectives

As in many cloud storage systems [17], data is stored in the unit of block. Before outsourcing, each data block can be encrypted using some probabilistic encryption method such as AES [12] with CBC encryption mode, to prevent the data content from being exposed to the storage server. When the user needs to access the outsourced data, she downloads and decrypts the data, accesses them, and re-encrypts them before uploading them back. Encryption alone, however, is not sufficient. The server is still able to infer private information from the user's *access pattern*, for example, the sequences of accessed locations, the orders of accessed locations on the server, etc. As illustrated by Islam et. al. [39], it is possible for an attacker to use the access ordering information to construct the data query and infer the plaintext of the data.

Therefore, researchers have been exploring ways to protect users' access patterns and various schemes have been proposed in the literature. Among them, Oblivious RAM (ORAM) [27, 70, 71, 29, 57, 30, 31, 73, 74, 32, 40, 45, 22, 20, 72, 61, 65, 24, 66, 51, 59, 68, 78, 69, 48, 52, 64, 58, 53, 14, 49, 21, 75, 8, 16, 15, 76, 63, 50, 60, 47, 13, 46] and Private Information Retrieval (PIR) [3, 62, 56, 10, 5, 43, 25, 26, 37, 44, 9, 7, 67, 41, 23] are two categories of security-provable methods. PIR schemes are applicable to the scenarios where data are read only, while ORAM schemes are more flexible as they allow a user to perform both read and write operations on the data. Hence, in this dissertation, we focus on improving the existing ORAM schemes.

Intuitively, an ORAM system is considered secure if the server cannot learn anything about a user's data access pattern. The formal definition can be referred to Definition 2.2 in Chapter 2. In recent years, interests in ORAM research have increased, and many ORAM constructions have been proposed to improve the performance in terms of the communication cost between the user and the server, the storage costs at the server and the user, and the computational costs at the server and the user. However, the practicality of the existing ORAM constructions is still questionable: Firstly, in spite of the improvement in performance, these ORAM constructions still require either large bandwidth consumption or storage capacity in practice. Secondly, these ORAM constructions all assume a single user mode, which has limited the application to more general, multiple user scenarios. In this dissertation, we aim to address these limitations with the following approaches.

1.3 Overview of Our Approaches

We have proposed four new ORAM constructions to achieve better performance than the state of the arts. In the following, we provide a brief overview of the motivations, key design ideas, and performance of these constructions.

1.3.1 S-ORAM: A Segmentation-based ORAM

The design is motivated by the observation that a large-scale storage system (e.g., a cloud storage system such as Amazon S3 [2], Google Drive [33], Dropbox [17]) usually stores data

in large blocks [65], but most of the existing ORAM constructions treat data blocks as atomic units for query and shuffling, and do not factor block size into their designs.

S-ORAM is designed to make better use of the large block size by introducing two *segmentation-based* techniques, namely, *piece-wise shuffling* and *segment-based query*, to improve the efficiency in data shuffling and query.

- With piece-wise shuffling, data can be shuffled across a larger range of blocks in a limited user-side storage; this way, the shuffling efficiency can be improved, and the improvement gets more significant as the block size increases.

- With segment-based query, S-ORAM organizes the server-side data storage as a hierarchy of single-segment and multi-segment layers, and an encrypted index block is introduced to each segment. The introduction of segmentation allows the adoption of hashing and indexing combined technique to locate query target in the server-side storage, which can accomplish higher efficiency than the prior hash-based ORAM schemes that only use hashing technique.

Extensive security proofs have been conducted to demonstrate that the security of S-ORAM. Particularly, we have shown that the scheme can make the observable location access sequences of any two private equal-length data query sequences to be computationally indistinguishable, with a failure probability of only $O(N^{-\log N})$, where N is the total number of outsourced data blocks.

In terms of communication and storage costs, S-ORAM outperforms the Balanced ORAM (B-ORAM) [40] and the Path ORAM [66], which are the best known theoretical hash-based and practical index-based ORAMs, respectively, that can work under small user-side storage. Particularly, under practical settings [65] where the number of data blocks N ranges from 2^{20} to 2^{36} and the block size ranges from 32 KB to 256 KB, the communication cost of S-ORAM is 12 to 23 times less than B-ORAM when they are given the same constant-size user-side storage; S-ORAM consumes 80% less server-side storage and around 60% to 72% less bandwidth than Path ORAM when they are given the similar logarithmic-size user-side storages.

1.3.2 KT-ORAM: An ORAM Built on A K-ary Tree of PIR Nodes

In most of existing ORAM schemes, the server is simply a storage, which does not perform any computation. In practice, however, cloud storage providers usually maintain data centers to provide their services, and a data center is a collection of not only storage but also rich computation resources. Hence, we propose a new ORAM construction called KT-ORAM, based on the motivation of leveraging the available computation capacity of the storage server to reduce the cost of user-server communication.

More specifically, this design of KT-ORAM is based on the following ideas:

- Firstly, to combine the merits of ORAM and PIR, we organize the server-side storage as a tree in which each node acts as a fully-functional PIR storage. The PIR-read and PIR-write primitives are implemented based on additive homomorphic (AH) encryption, addition and multiplication operations. Combined with PIR, the communication cost becomes determined only by the height of the tree, because only one data block is transferred from/to each accessed tree node along a root to leaf path in the tree. Meanwhile, the PIR primitives can be performed efficiently because they process only a small fraction of the dataset stored on the tree.

- Secondly, we use a k-ary tree instead of a binary tree for KT-ORAM, 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.

- Thirdly, we propose a *delayed eviction* mechanism which can defer and aggregate as many eviction operations as possible to reduce the data block access frequency, and thus further reduce the communication and computational costs.

Through rigorous security analysis and extensive evaluation, we show that KT-ORAM meets the security requirement of an ORAM construction, and meanwhile accomplishes the following performance goals simultaneously:

- *communication efficiency* - Its communication cost is $O(\frac{\log N}{\log \log N} \cdot B)$ bits per query, as long as the block size B is N^{ϵ} bits for some constant $0 < \epsilon < 1$.

- *storage efficiency* - The scheme requires $O(B)$ storage space at the user side and $O(B \cdot N)$ storage space at the cloud server side.

- *computational efficiency* - The scheme incurs $O(\frac{\log N}{\log \log N} \cdot B)$ computational cost at the user 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, no other ORAM constructions can simultaneously achieve the same or better level of communication, storage and user-side computational efficiency.

1.3.3 GP-ORAM: A Generalized Partition ORAM

Existing ORAM constructions have been designed based on the assumptions that the server-side storage capacity is constant [27, 57, 30, 40, 61, 52], $O(\log N)$ blocks [66, 24, 68, 69], or $O(\sqrt{N})$ blocks [70, 22, 65, 64, 14, 63]. In practice, users of a cloud storage service may have different local storage capacities. It is ideal if the available local storage capacities can be fully utilized to achieve a high level of performance efficiency in accessing data outsourced to the cloud storage. The GP-ORAM construction is designed for this purpose.

More specifically, GP-ORAM is built based on the Partition ORAM (P-ORAM) construction [65], a state-of-the-art communication-efficient ORAM construction. P-ORAM requires a user to have a fixed local storage of capacity of $O(\sqrt{N})$ blocks; however, GP-ORAM allows smaller and adjustable number of partitions, fully utilizes the available user-side storage to reduce communication cost, and can efficiently export the index table to the server. As a result, GP-ORAM incurs low bandwidth cost (i.e., $O(\log N)$ data blocks per query in practice) and has significantly less user-side storage cost than P-ORAM. We demonstrate the security and practicality of GP-ORAM through extensive security and performance analysis.

1.3.4 MU-ORAM: Dealing with Stealthy Privacy Attacks in Multi-User Data Outsourcing Services

At last, we consider a general data outsourcing model, in which multiple users share a set of data blocks outsourced to a cloud storage. For a multi-user data outsourcing setting, the users become vulnerable to stealthy privacy attacks targeted at revealing the data access patterns

of innocent users, even if only one curious or compromised user colludes with the storage server. To study the feasibility and costs of overcoming the above limitation, we propose a new ORAM construction called Multi-User ORAM (MU-ORAM), which is resilient to stealthy privacy attacks. The key ideas in the design are (i) introducing a chain of proxies to act as a common interface between users and the storage server, (ii) distributing the shares of the system secrets delicately to the proxies and users, and (iii) enabling a user and/or the proxies to collaboratively query and shuffle data. Through extensive security analysis, we quantify the strength of MU-ORAM in protecting the data access patterns of innocent users from attacks, under the assumption that the server, users, and some but not all proxies can be curious but honest, and even colluding. Cost analysis has also been conducted to quantify the extra cost incurred by the MU-ORAM design.

1.4 Organization of the Dissertation

The rest of the dissertation is organized as follows. In Chapter 2, we first describe the ORAM system and threat model. Then, we give the formal security definition of ORAM system. In Chapter 3, we review the state-of-the-art Oblivious RAM schemes. In Chapters 4, 5, 6, and 7, we present our proposed four schemes. In Chapter 8, we conclude this dissertation with a summary of our main contributions and future research plans.

CHAPTER 2. PROBLEM STATEMENT

In this chapter, we present the system model, threat model, security definitions and design goal of an Oblivious RAM (ORAM) construction.

2.1 System Model

When designing an ORAM system, we consider a system consisting of two parties: users and a remote storage server. Depending on the number of users, the ORAM construction can be categorized as a single-user ORAM or a multi-user ORAM.

The users export a large amount of data to store at the server, and wish to hide from the server the pattern of their accesses to the data. Data are assumed to be stored and accessed in the unit of blocks and the data block size (denoted as B) can range from several kilo bytes to several mega bytes. We also use N to denote the total number of data blocks outsourced by the users.

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

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

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

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

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

2.2 Threat Model and Security Definition

In a single-user ORAM system, the purpose is to protect the data access pattern of the user under the assumption that the remote server is not trusted. Particularly, the server is assumed to be honest but curious; that is, it behaves correctly in storing data and serving the user's data access requests, but it may attempt to figure out the user's access pattern. In all the proposed work, we do not consider attacks such as timing attacks [6] that are based on other side-channel information, as they can be addressed separately. The network connection between the user and the server is assumed to be secure; in practice, this can be achieved by using well-known techniques such as SSL [35] and HTTPS [34].

We inherit the common security definition of a single-user ORAM [27], and rephrase it as follows:

Definition Let $\vec{x} = \langle\ (op_1, i_1, D_1), (op_2, i_2, D_2), \cdots\ \rangle$ denote a private sequence of the user's intended data requests, where each op is either a *read* or *write* operation. Let $A(\vec{x}) = \langle\ (op'_1, l_1, D'_1), (op'_2, l_2, D'_2), \cdots\ \rangle$ denote the sequence of the user's accesses to the remote storage (observed by the server), in order to accomplish the user's intended data requests. An ORAM system is said to be secure if (i) for any two equal-length private sequences $\vec{x}$ and $\vec{y}$ of the intended data requests, their corresponding observable access sequences $A(\vec{x})$ and $A(\vec{y})$ are computationally indistinguishable; and (ii) the probability that the ORAM system fails to operate is negligibly small, i.e., $O(2^{-\lambda})$, where λ is a security parameter.

The above threat model and security definition will be applied to our first three proposed schemes, but the model and security definitions for a multi-user ORAM system will be presented in Chapter 7.

2.3 Design Goal

The design goal of an ORAM system includes the following aspects:

- The proposed scheme must be secure according to the ORAM security definition. Specifically, a user's access pattern should be indistinguishable from a random access pattern of the same length; the failure probability of the scheme is upper bounded by a small probability.

- The costs of the proposed scheme should be as low as possible, which is measured by the following metrics: (i) the communication cost (bandwidth consumption) between the user and the server; (ii) the computational costs for both the user and the server; (iii) the user-side storage cost; (iv) the server-side storage cost; and (v) data query and access latency.

CHAPTER 3. LITERATURE REVIEW

The concept of Oblivious RAM (ORAM) was first introduced by Goldreich and Ostrovsky [27], which enables users to export their data to a remote storage and access the remote data storage without exposing the data access pattern. Since then, various ORAM constructions have been proposed, including single-user ORAMs and multi-user ORAMs. In this chapter, we survey the state-of-the-art of the ORAM research. Here, we use N to denote the total number of data blocks outsourced by the user to the storage server and B to denote the data block size in bits.

3.1 Single-user ORAMs

According to the adopted data lookup techniques, single-user ORAMs have two major classes, namely, hash-based ORAMs (hORAMs) and index-based ORAMs (iORAMs).

In hORAMs [27, 70, 71, 29, 57, 30, 31, 74, 40, 45, 20, 72], the server-side storage is usually organized as a hierarchy of layers and each layer is associated with a hash function to locate each data block on this layer. The hash function is kept secret from the server. Data blocks on each layer is distributed according to the hash function. During data query, the user requests data blocks from the locations according to the hash functions. After obtaining the target data block, the user re-encrypts and uploads the block back to the top layer on the server. To avoid layer overflowing, when any layer is full, all data blocks on this layer will be obliviously shuffled and dumped into the next larger layer.

As the first ORAM solution, Bucket Hash ORAM (BH-ORAM [27]) uses one normal hash function for each of its $\log N$ layers. Thus, the server-side storage for each layer is a hash table where each entry of the hash table is a bucket that can store up to $\log N$ data blocks

to avoid hash collision. When data blocks are shuffled to a specific layer, all buckets on this layer must be fully occupied by adding additional dummy data blocks. Therefore, each data query retrieves all data blocks in one selected bucket from each non-empty layer. Bucket Hash ORAM incurs a communication cost of $O(\log^3 N \cdot B)$ bits per query with constant user-side storage.

The efficiency of Bucket Hash ORAM has been improved by two follow-up proposals, namely, Bloom Filter ORAM (BF-ORAM) by Williams et. al. [73] and Cuckoo Hash ORAM (CH-ORAM) by Pinkas et. al. [57] and Goodrich et. al. [32, 30, 29, 31]. Bloom Filter ORAM uses one collision-free Bloom Filter at each layer to replace the fixed-size hash bucket in Bucket Hash ORAM. Each bit of the Bloom Filter is encrypted and exported to the server. Thus, each data query retrieves and checks the Bloom Filter for the target data block and only one data block is retrieved from each non-empty layer. Compared to Bucket Hash ORAM, the communication cost is reduced by a factor of $\log N$, which is $O(\log^2 N \cdot B)$ bits per query. In Cuckoo Hash ORAM, a Cuckoo hash function is utilized such that each layer is organized as a Cuckoo hash table. Due to Cuckoo hash function, each data query only retrieves two data blocks from each layer. Thus, the communication cost is reduced to $O(\log^2 N \cdot B)$ bits per query under constant user-side storage.

Furthermore, Kushilevitz et. al. [40] proposed a hybrid ORAM solution, called B-ORAM, to balance the communication cost of data query and data shuffling. B-ORAM incurs $O(\frac{\log^2 N}{\log \log N} \cdot B)$ bits communication cost per query with constant user-side storage.

In iORAMs [61, 65, 24, 66, 51, 59, 68, 78, 69, 48, 52, 64, 58, 53, 14, 49, 21, 75, 8, 16, 15, 76, 63, 50, 60, 46], index is used to locate a user's desired data on the remote server. Due to the obliviousness requirement, index should be either stored at the user side or outsourced to the storage server as an oblivious data structure (e.g. index can be recursively built up at the server side similarly as that of data blocks).

The first iORAM construction was proposed by Shi et. al. [61] with $O(\log^3 N \cdot B)$ bits per query, given a constant user-side storage. In that work, the server-side storage is organized as a binary tree, where each node on the tree is a small bucket to hold up to $\log N$ data blocks. The obliviousness of the scheme is accomplished through distributing each data block to a randomly-

Table 3.1 Comparisons of State-of-the-art Oblivious RAM Constructions. N denotes the total number of exported data blocks, B denotes the size of each data block.

ORAM	Communication Cost	User-side Storage	Server-side Storage
BH-ORAM [27]	$O(\log^3 N \cdot B)$	$O(B)$	$O(N \log N \cdot B)$
CH-ORAM [57]	$O(\log^2 N \cdot B)$	$O(B)$	$O(N \cdot B)$
BF-ORAM [70]	$O(\log^2 N \log\log N \cdot B)$	$O(B)$	$O(N \cdot B)$
B-ORAM [40]	$O(\frac{\log^2 N}{\log\log N} \cdot B)$	$O(B)$	$O(N \cdot B)$
T-ORAM [61]	$O(\log^3 N \cdot B)$	$O(B)$	$O(N \log N \cdot B)$
Path ORAM [66]	$O(\log N \cdot B) \cdot \omega(1)$	$O(\log N \cdot B) \cdot \omega(1)$	$O(N \cdot B)$
G-ORAM [24]	$O(\frac{\log^2 N}{\log\log N} \cdot B) \cdot \omega(1)$	$O(\log^2 N \cdot B) \cdot \omega(1)$	$O(N \cdot B)$
P-PIR [61]	$O(\log^2 N \cdot B)$	$O(B)$	$O(N \log N \cdot B)$
P-ORAM [65]	$O(\log N \cdot B)$	$O(\sqrt{N} \cdot B)$	$O(N \cdot B)$

selected path on the tree. A data eviction process is launched after every query to make the node overflow probability small. The construction was later improved to Path ORAM [66] by reducing the size of each node and adding a stash at the user-side storage to deal with node overflowing. The evaluation of Path ORAM shows that its per-query communication cost is $O(\log N \cdot B)$ bits with a stash size of $O(\log N \cdot B)$ bits. According to the Path ORAM, numerous ORAM constructions have been further proposed. For example, the construction proposed by Ren et. al. [60], makes integrity checking available in Path ORAM.

Concurrently, another iORAM called Partition ORAM (P-ORAM) [65] was proposed based on the assumption that the user-side storage size is $O(\sqrt{N} \cdot B)$ bits. The key idea of the scheme is to split the server storage into "smaller" partitions such that each partition is a fully functional ORAM; leveraging the user-side storage, data blocks are obliviously transferred between the partitions. Due to the reduced size of each partition ORAM, this scheme incurs a $\log N \cdot B$ bits communication cost per query in practice.

In Table 3.1, we compare several representative state-of-the-art ORAM constructions.

3.2 Multi-user ORAMs

Most of the existing multi-user ORAMs assume all the users trust each other and they do not collude with the storage server. Hence, they only need to protect their data access patterns from the storage server (without the need to protect one user's access pattern from other users).

Based on such assumption, all stateless ORAMs (i.e., ORAMs that do not require local storage of data) [22, 27, 61, 70, 73, 57, 32, 30, 29, 31, 77, 49, 47] can be extended to support such scenarios.

Particularly, PrivateFS [73] has been proposed to support parallel accesses from multiple users. In PrivateFS, the server-side storage is organized as a layered structure and a log file is stored on the first layer. In order to achieve parallel access, the log file is shared by all the users in addition to the target data those users try to access. For each access, both the log file and the data blocks are accessed by all the users to achieve parallelism.

In Delegation ORAM (D-ORAM) [22], the problem of ORAM delegation was studied. It proposes a scheme with which the data owner can delegate controlled access to third parties for the outsourced data, while preserving the access pattern privacy. However, this scheme can only be applied to the square-root ORAM [27].

The recently proposed Group ORAM (GRP-ORAM) [47] considers the following scenario: A data owner outsources a dataset to a semi-honest cloud storage server, via which the data is shared with a group of untrusted users who may be malicious. The storage server is assumed not to collude with any user. The design goal is to employ the storage server to enforce that a user can only access the data that it is authorized to, and meanwhile preserve the obliviousness of the users' access from the server. The obliviousness property for GRP-ORAM is defined as follows: Assuming the server is not allowed to collude with any users in the system, the access pattern of any user is protected against the server.

CHAPTER 4. S-ORAM: SEGMENTATION-BASED OBLIVIOUS RAM

In the first work, we proposed a novel ORAM scheme, called *segmentation-based Oblivious RAM (S-ORAM)*, aiming to bring theoretical ORAM constructions one step closer to practical applications. This work is motivated by the observation that a large-scale storage system (e.g., a cloud storage system such as Amazon S3 [2]) usually stores data in blocks and such a block typically has a large size [65], but most existing ORAM constructions treat data blocks as atomic units for query and shuffling, and do not factor block size into their designs.

S-ORAM is designed to make better use of the large block size by introducing two *segment-based* techniques, namely, *piece-wise shuffling* and *segment-based query*, to improve the efficiency in data shuffling and query. With piece-wise shuffling, data can be perturbed across a larger range of blocks in a limited user-side storage; this way, the shuffling efficiency can be improved, and the improvement gets more significant as the block size increases. With segment-based query, S-ORAM organizes the data storage at the server side as a hierarchy of single-segment and multi-segment layers, and an encrypted index block is introduced to each segment. With these two techniques at the core, together with a few supplementary algorithms for distributing blocks to segments, S-ORAM can accomplish efficient query with only $O(\log N)$ communication cost and a constant user-side storage, while existing ORAM constructions have to use a larger user-side storage to achieve the same level of communication efficiency in query.

Extensive security analysis has been conducted to verify the security of the proposed S-ORAM. Particularly, S-ORAM has been shown to be a provably highly secure solution that has a negligible failure probability of $O(N^{-\log N})$ according to Definition 2.2, which is no higher than that of existing ORAM constructions.

In terms of communication and storage costs, S-ORAM outperforms the B-ORAM [40] and the Path ORAM [66], which are the best known theoretical hash-based and practical index-

based ORAMs under small local storage assumption, respectively. Particularly, under practical settings [65] where the number of data blocks N ranges from 2^{20} to 2^{36} and the block size is 32 KB to 256 KB, (i) the communication cost of S-ORAM is 12 to 23 times less than B-ORAM when they have the same constant-size user-side storage; (ii) S-ORAM consumes 80% less server-side storage and around 60% to 72% less bandwidth than Path ORAM when they have the similar logarithmic-size user-side storage.

4.1 Intuition

The design of S-ORAM is motivated by the observation that a large-scale storage system usually stores data in blocks and such a block typically has a large size. To the best of our knowledge, most existing ORAM constructions treat data blocks as atomic units for query and shuffling, and do not factor block size into their designs. The recently proposed index-based ORAM constructions [48, 65, 64, 63] have used large-size blocks to store indices to improve index search efficiency; still, more opportunities wait to be explored to fully utilize this feature.

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

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

- *Segment-based Query.* In order to improve query efficiency, S-ORAM organizes the data storage at the server side as a hierarchy of single-segment and multi-segment layers. In each segment, an encrypted index block (with the same size as a data block) is introduced to maintain the mapping between data block IDs and their locations within the segment.

This way, when a user needs to access a block in a segment, he/she only needs to access two blocks - the index block and the intended block. By adopting this technique together with supplementary algorithms for distributing blocks to segments, S-ORAM can accomplish efficient query with only $O(\log N)$ communication cost and a constant user-side storage, while existing ORAM constructions have to use a larger user-side storage to achieve the same level of communication efficiency in query.

The details of the proposed S-ORAM are elaborated in the rest of this chapter.

4.2 Scheme

The presentation of S-ORAM consists of the storage organization and system initialization, data query procedure and data eviction procedure.

4.2.1 Storage Organization and Initialization

4.2.1.1 Data Block Format

Similar to existing ORAMs, S-ORAM stores data in blocks, and a data block is the basic unit for read/write operations by the user. A plain-text data block can be split into *pieces* and each piece is $z = \log N$ bits long, where N is the total number of data blocks. The first piece contains the ID of the data block, say i, which is also denoted as $d_{i,1}$. The remaining pieces store the content of the data block, denoted as $d_{i,2}$, $d_{i,3}$, $\cdots$, $d_{i,P-1}$. Before being exported to the remote storage server, the plain-text data block is encrypted piece by piece with a secret key k, as shown in Figure 4.1:

$$c_{i,0} = E_k(r_i), \text{ where } r_i \text{ is a random number;}$$

$$c_{i,1} = E_k(r_i \oplus d_{i,1});$$

$$c_{i,2} = E_k(c_{i,1} \oplus d_{i,2});$$

$$\cdots,$$

$$c_{i,P-1} = E_k(c_{i,P-2} \oplus d_{i,P-1}).$$

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

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

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

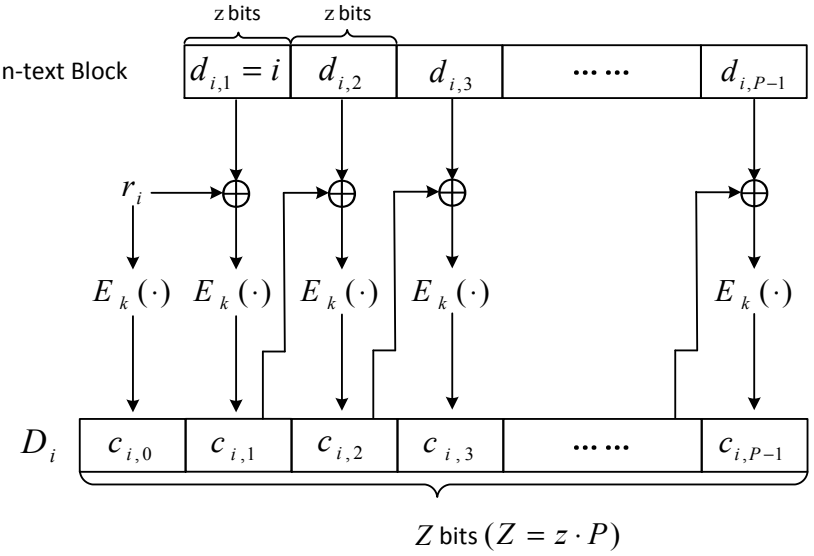


Figure 4.1 Format of a data block in S-ORAM.

4.2.1.2 Server-side Storage

S-ORAM stores data at the remote server in a pyramid-like structure as shown in Figure 4.2. The top layer, called *layer 1*, is an array containing at most four data blocks. The rest of the layers are divided into two groups as follows.

T1 (Tier 1) Layers: Single-Segment Layers. T1-layers refer to those between (inclusive) layer 2 and layer $L_1 = \lfloor 2 \log \log N \rfloor$. As illustrated in Figure 4.3, each T1-layer consists of a single segment, which includes an encrypted index block I_l and 2^{l+1} data blocks. Among the data blocks, at most half of them are real data blocks as formatted in Figure 4.1, while the rest are dummy blocks each with ID 0 and randomly-stuffed content. The index block has 2^{l+1} entries; each entry corresponds to a data block in the segment which consists of three fields: *ID of the data block*, *location of the data block in the segment*, and *access bit* indicating whether the block has been accessed since it was placed to the location.

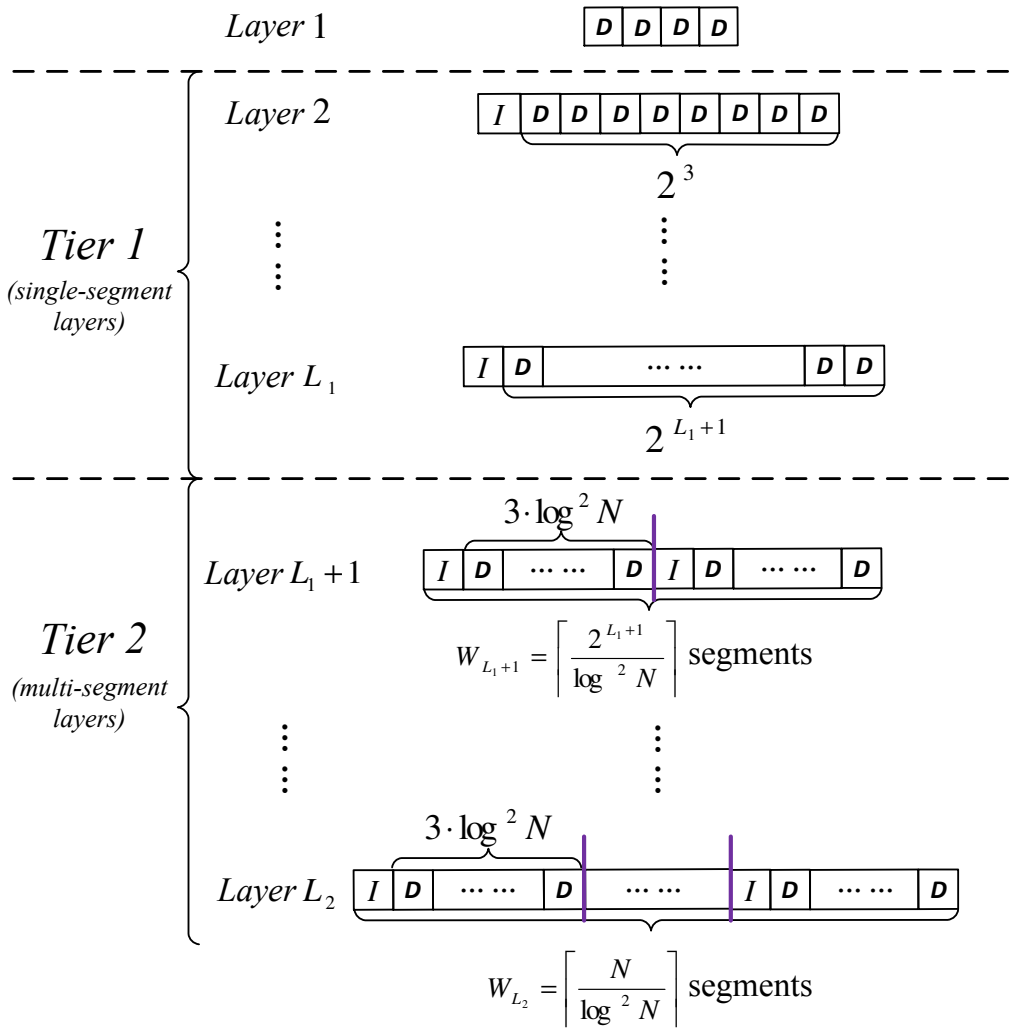


Figure 4.2 Organization of the server-side storage.

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

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

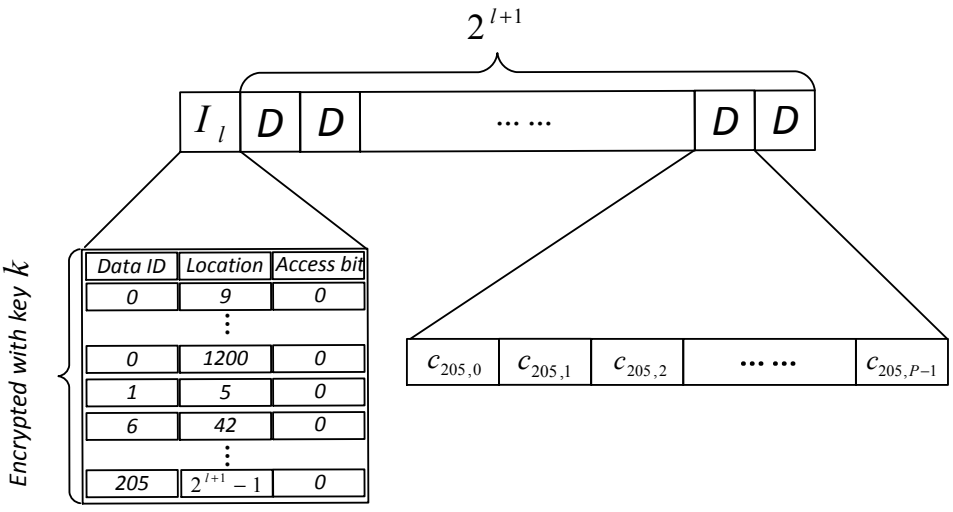


Figure 4.3 Structure of a T1-layer.

index block is less than 32 KB, which can fit into a typical data block assumed in the existing studies of practical ORAM schemes [65].

4.2.1.3 User-side Storage

The user organizes its local storage into two parts: *cache (temporary storage)* and *permanent storage*. Cache is used to buffer and process (including encrypt and decrypt) data blocks downloaded from the server. We assume that the size of the cache is αZ bits where α is a constant. In the S-ORAM design presented in this section, we set $\alpha = 2$. This design can be conveniently adapted to other configurations of cache size, as will be discussed in Section 4.4.1.

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

4.2.1.4 Storage Initialization

The user initializes the S-ORAM system as follows:

- It randomly selects a secret key k and a one-way hash function $H_{L_2}(\cdot)$ of layer L_2, i.e., the bottom layer.
- N plain-text data blocks are encrypted into blocks D_i where $i = 1, \cdots, N$ with the secret key k in the format illustrated by Figure 4.1. In addition, $2N$ dummy blocks are randomly generated and encrypted also with key k.
- N real data blocks and $2N$ dummy blocks are uploaded to layer L_2 of the server storage in a delicate manner to ensure that (i) each real data block D_i of unique ID i is distributed to segment $H_{L_2}(i)$ at layer L_2, (ii) each segment is assigned with exactly $3\log^2 N$ data blocks, and (iii) data blocks distributed to the same segment are randomly placed within the segment. Note that, a process like *data shuffling* elaborated in Section 4.2.3.3 can be adopted to distribute and place the data blocks to satisfy the above properties.

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

4.2.2 Data Query

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

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

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

- If D_t has not been found at any layer prior to layer l and I_l indicates that D_t is at layer l, record the location where D_t resides, set the access bit of the location to 1, and re-encrypt

and upload I_l to save cache space. Then, retrieve D_t. Meanwhile, the server makes a copy of user uploaded dummy block D to the location where D_t was retrieved.

- Otherwise, the location of a dummy block $D_{t'}$ whose access bit in I_l is 0 (i.e., it has not been accessed since last time it was distributed to its current location) is randomly picked and recorded. After the block's access bit is set to 1 in I_l, I_l is re-encrypted and uploaded. Then, $D_{t'}$ is retrieved and discarded. The server also makes a copy of dummy block D to fill in this location.

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

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

- If D_t has already been found at a layer prior to layer l, a segment is randomly selected from layer l and the user randomly selects a dummy block $D_{t'}$ in this segment whose access bit in I_l^s is 0. After the access bit of $D_{t'}$ is set to 1, I_l^s is re-encrypted and uploaded. Then, $D_{t'}$ is retrieved, discarded, and a copy of dummy block D is filled in $D_{t'}$'s original location.

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

4.2.3 Data Shuffling

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

4.2.3.1 Preliminary: A Segment-Shuffling Algorithm

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

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

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

$$2m^2 \cdot [\log(2m^2) + 3z] \leq \alpha Z. \tag{4.1}$$

Data shuffling in S-ORAM is based on a *segment-shuffling algorithm* (as shown in Algorithm 2). It is able to shuffle n ($\leq 3\log^2 N$) data blocks with a communication cost of $O(n)$ data blocks, by setting the system parameter m to $\sqrt{1.5}\log N$, under the following practical assumptions: (1) $N \leq 2^{36}$ which is considered large enough to accommodate most practical

applications [65]; (2) the size of Z is between 32 KB and 256 KB which is typically assumed in practical ORAM schemes [65]; and (3) $\alpha = 2$ meaning that a small local cache of two data blocks is assumed. It is easy to verify that, under these assumptions, Equation (4.1) holds. Moreover, as $n \leq 3\log^2 N = 2m^2$, π is large enough to store a permutation of the IDs of n data blocks, and B_0, B_1, and B_2 are large enough to store n data pieces, which are required in the algorithm.

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

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

4.2.3.2 Shuffling a T1-layer l ($2 \leq l \leq L_1$)

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

4.2.3.3 Shuffling a T2-layer l ($L_1 < l < L_2$)

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

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

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

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

Round II: Oblivious Sorting. All the labeled blocks are sorted obliviously (using the *oblivious data sorting* scheme presented in Section 4.2.3.5 in the non-descending order based on the tag-tuple. Particularly, a block with a smaller first-tag should precede ones with larger first-tags; blocks with the same first-tag are sorted in the non-descending order based on the second-tag. This way, real data blocks are sorted to precede dummy blocks.

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

- If the block is the very first one or it has a different first-tag from its immediate predecessor, it becomes the first one of a new segment, and the counter is reset to 1.

- Otherwise: If the counter is less than $3\log^2 N$, the counter is incremented by 1. If the counter reaches $3\log^2 N$, the block is considered redundant and hence its first-tag is relabeled as ∞, which means this block is a redundant dummy blocks.

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

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

4.2.3.4 Shuffling the Bottom Layer L_2

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

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

4.2.3.5 Oblivious Data Sorting

Existing oblivious sorting techniques for ORAMs with constant local storage either incurs high asymptotical cost (for example, Batcher's sorting network [19] incurs $O(n\log^2 n)$ communication cost) or large hidden constant behind the big-O notations (e.g., AKS sorting network [1] incurs $c \cdot n\log n$ communication cost with $c \geq 10^3$ and randomized shellsort [28] incurs $> 24 \cdot n\log n$ cost), which significantly impede their practical efficiency. Hence, a more practically efficient sorting method is needed.

In S-ORAM, we develop an m-way oblivious sorting scheme based on the m-way sorting algorithm [42]. It sorts data in pieces rather than blocks, which exploits the user cache space more efficiently and thus achieves a better performance than the afore-mentioned algorithms, particularly when the block size is relatively large (which is common in practice [65]). Modifications have also been made to the original m-way sorting algorithm to ensure the obliviousness of data sorting. The proposed m-way oblivious sorting scheme is shown in Algorithm 3. To sort a set $\mathcal{D}$ of n blocks, the m-way oblivious sorting algorithm works recursively as follows: if $n \leq 2m^2$, a *segment-sorting algorithm* similar to the segment-shuffling algorithm is applied to sort the n blocks at the communication cost of $O(n)$ blocks; otherwise, the n blocks are split into m subsets each of $\frac{n}{m}$ blocks, the m-way oblivious sorting algorithm is applied to sort each of the subsets, and finally a *merging algorithm* is used to merge the sorted subsets into a sorted set of n blocks.

Next, we describe the segment-sorting algorithm (Algorithm 4) and the merging algorithm (Algorithm 5) The segment-sorting algorithm is based on the segment-shuffling algorithm (Algorithm 2) with the following revisions: (1) The segment-sorting algorithm sorts blocks that are labeled with tags. The format of a labeled block is slightly different from the one shown in Figure 4.1; particularly, the encrypted tag is inserted as an extra piece before the encrypted block ID. (2) While the segment-shuffling algorithm can randomly pick a permutation function to shuffle pieces and blocks, the segment-sorting algorithm must permute pieces and blocks according to the non-decreasing order of tags. (3) The segment-sorting algorithm does not need to re-construct index blocks.

Finally, Algorithm 5 formally presents the merging algorithm.

4.3 S-ORAM Security Analysis

According to the security definition of ORAM, the security of S-ORAM can be proved through Theorem 1. Before the proof of the theorem, we describe three lemmas before presenting the main theorem.

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

Proof. When shuffling a T2-layer l as in Section 4.2.3.3, up to 2^l real data blocks are mapped (by a hash function) to $\lceil \frac{2^l}{\log^2 N} \rceil$ segments uniformly at random. In the following proof, we first assume the number of real data blocks is 2^l and compute the probability that there exists a segment with at least $1.5\log^2 N$ real blocks.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

A data shuffling process for layer l fails only if: (S1) layer overflow occurs, i.e., the process tries to store more data blocks to the layer than its capacity; or (S2) segment overflow occurs when layer l is a T2-layer, i.e., the process tries to store more than $3\log^2 N$ real data blocks to a segment. As discussed in Section 4.2.3.2, case (S1) will not occur when shuffling a T1-layer l because the total number of blocks to be shuffled is 2^{l+1}, which is the capacity of the layer. According to Section 4.2.3.3, case (S1) will not occur when shuffling a T2-layer l, because Round IV of the shuffling algorithm marks and removes redundant blocks to make the total number of blocks less than the capacity of the layer.

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

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

When case (S2) occurs on a T2-layer, there must be at least one segment of the layer distributed with more than $3\log^2 N$ blocks. The probability that this case occurs is smaller than the probability that at least one segment of the layer is distributed with at least $1.5\log^2 N$ blocks, which is $O(N^{-\log N})$. Hence, the probability for case (S2) to occur is also $O(N^{-\log N})$.

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

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

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

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

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

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

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

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

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

Also, according to Lemma 2, the S-ORAM construction fails with probability $O(N^{-\log N})$, which is considered negligible and no larger than the failure probability of existing ORAMs [27, 29, 32, 30, 31, 40, 57, 70, 61, 66, 64, 65]. □

4.4 S-ORAM Cost Analysis and Evaluations

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

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

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

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

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

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

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

We have the following recursive relation:

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

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

uploading of each block. Hence, each data block should be retrieved and uploaded for

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

times during the entire shuffling process.

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

- Each T1-layer l $(2 \leq l \leq L_1)$ is shuffled once every time when an odd multiple of 2^l queries have been made, and each of the 2^l data blocks at T1-layer l is scanned once for every shuffling. Hence, the amortized cost is $S_l(N) = \frac{2^{l+1}}{2^{l+1}} = 1$ block scanning per query.

- Each T2-layer l $(L_1 < l < L_2)$, except the bottom layer L_2, is shuffled also once every time when an odd multiple of 2^l queries have been made, and two rounds of oblivious sorting are performed on $4 \cdot 2^l$ data blocks. Hence, the amortized cost is $S_l(N) = \frac{2 \cdot 4 \cdot 2^l \cdot T(4 \cdot 2^l)}{2^{l+1}} = 4 \cdot T(4 \cdot 2^l)$ block scannings per query.

- The bottom layer L_2 is shuffled every time when a multiple of N queries have been made, and two rounds of oblivious sorting are performed. The first oblivious sorting is performed on $w < 6N$ blocks and second one is performed on $4N$. Hence, the amortized cost is at most $S_{L_2}(N) = \frac{6N \cdot T(6N)}{N} + \frac{4N \cdot T(4N)}{N} = 6 \cdot T(6N) + 4 \cdot T(4N)$ block scannings per query.

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

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

To summarize, the bandwidth consumption for S-ORAM is

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

4.4.1 Cost Comparison

We now compare the performance of S-ORAM with that of B-ORAM and Path ORAM from both theoretical and practical aspects. The theoretical results of bandwidth, user-side storage and server-side storage costs are denoted as T_b, T_c, and T_s, and the practical results as

P_b, P_c, and P_s, respectively. The practical settings used here are as follows: the number of data blocks N ranges from 2^{20} to 2^{36} and the block size ranges from 32 KB to 256 KB, which are similar to the practical settings adopted by Stefanov et. al. [65]. In the comparisons, system parameter α in S-ORAM may be set to a value other than 2. If $\alpha \neq 2$, the scheme presented in Section 4.2 can be modified to accommodate this by simply setting parameter m to the largest integer satisfying Equation (4.1).

Table 4.1 Performance Comparison: S-ORAM vs. B-ORAM

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

Table 4.2 Theoretical Performances: S-ORAM vs. Path ORAM

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

Table 4.3 Practical Performances: S-ORAM vs. Path ORAM

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

4.4.1.1 S-ORAM vs. B-ORAM

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

As shown in Table 4.1, the bandwidth consumption of S-ORAM is 12 to 23 times less than that of B-ORAM under practical settings, while the server-side storage cost of S-ORAM is about 75% of that of B-ORAM. The improvement in bandwidth efficiency is attributed to two factors: (i) the query cost of S-ORAM is only $2 \log N$ blocks while the cost of B-ORAM is $\frac{2 \log^2 N}{\log \log N}$; and (ii) the shuffling algorithm of S-ORAM is more efficient than that of B-ORAM. In addition, the failure probability S-ORAM is $O(N^{-\log N})$, which is asymptotically lower than that of B-ORAM which is $O(N^{-\log \log N})$ [40].

4.4.1.2 S-ORAM vs. Path ORAM

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

Table 4.2 shows the theoretical performances of both S-ORAM and Path ORAM and Table 4.3 is the practical performance comparison of these two ORAMs.

In Table 4.3, it can be seen that S-ORAM outperforms Path ORAM in both bandwidth efficiency and server-side storage efficiency. It requires 80% less server-side storage and consumes around 60% to 72% less bandwidth than Path ORAM.

4.5 Summary

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

Algorithm 1 Query data block D_t of ID t.

1: $found \leftarrow false$
   /* *Phase I: access layer 1* */
2: Retrieve & decrypt blocks in layer 1
3: **if** D_t is found in layer 1 **then** $found \leftarrow true$
   /* *Phase II: access T1-layers* */
4: **for** each non-empty layer $l \in \{2, \cdots, L_1\}$ **do**
5: Retrieve & decrypt I_l – index block of the layer
6: **if** $(found = false \wedge t \in I_l)$ **then**
7: Set the access bit of D_t to 1 in I_l
8: Re-encrypt & upload I_l
9: Retrieve & decrypt D_t
10: $found \leftarrow true$
11: **else**
12: Randomly pick a dummy $D_{t'}$ with access bit 0
13: Set the access bit of $D_{t'}$ to 1 in I_l
14: Re-encrypt & upload I_l
15: Retrieve & discard $D_{t'}$
16: **end if**
17: **end for**
   /* *Phase III: access T2-layers* */
18: **for** each non-empty layer $l \in \{L_1 + 1, \cdots, L_2\}$ **do**
19: **if** $(found = false)$ **then**
20: $s \leftarrow H_l(t)$
21: **else**
22: s is randomly picked from $\{0, \cdots, W_l - 1\}$
23: **end if**
24: Retrieve & decrypt I_l^s – index block of segment s
25: **if** $(found = false \wedge t \in I_l^s)$ **then**
26: Set the access bit of D_t to 1 in I_l^s
27: Re-encrypt & upload I_l^s
28: Retrieve & decrypt D_t
29: $found \leftarrow true$
30: **else**
31: Randomly find a dummy $D_{t'}$ with access bit 0
32: Set the access bit of $D_{t'}$ to 1 in I_l^s
33: Re-encrypt & upload I_l^s
34: Retrieve & discard $D_{t'}$
35: **end if**
36: **end for**
   /* *Phase IV: wrap up* */
37: **if** (D_t is found in layer 1) **then**
38: Encrypt an extra dummy D in local storage
39: **else**
40: Re-encrypt D_t in local storage
41: **end if**
42: Upload all blocks in local storage back to layer 1

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

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

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

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

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

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

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

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

CHAPTER 5. KT-ORAM: K-ARY TREE OBLIVIOUS RAM

In the second work, we proposed a new hybrid ORAM-PIR construction called KT-ORAM [76]. Suppose N denotes the number of outsourced data blocks, B denotes the data block size. Given $k = \log N$, KT-ORAM achieves: (i) $O(\frac{\log N}{\log \log N} \cdot B)$ communication cost when the recursion level on metadata is of $O(1)$ depth with uniform block size $B = N^{\epsilon}$ ($\epsilon < 0$) such that the metadata part is absorbed into the data block part, which outperforms all existing solutions with the same assumption; (ii) $O(\frac{\log^2 N}{\log \log N} \cdot B)$ communication cost when the number of recursion levels is $O(\log N)$ with $B = \Omega(\log^2 N \log \log N)$; (ii) $O(N^{-\log \log N})$ failure probability during ORAM operations, which is better than or equivalent to all existing ORAM schemes with tree structure; (iii) $O(1)$ user storage, which is the best scheme among those ORAMs with constant user storage assumptions.

Our proposed KT-ORAM construction shares the similar idea of P-PIR [48], which builds 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 ORAM operations, 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 for data block (not including the metadata) from $O(\log N \cdot B)$ to $O(\frac{\log N}{\log k} \cdot B)$; (ii) mapping the k-ary tree to a logical binary tree and executing evictions on the binary tree; and (iii) delayed evictions to reduce the eviction cost for data block (not including the metadata) from $O(\log N \cdot B)$ to $O(\frac{\log N}{\log k} \cdot B)$. Compared to P-PIR, KT-ORAM has the same asymptotical user-side and the server-side computational costs when $k = \log N$ and $B = \Omega(\log^2 N \log \log N)$.

Comprehensive security analysis has been conducted to analyze the performance of KT-ORAM. The results show that the construction can preserve a user's data access pattern with a negligibly-small failure probability of $O(N^{-\log \log N})$ according to Definition 2.2.

Theoretical and numerical analysis has been conducted to evaluate KT-ORAM and compare it with several existing ORAM constructions. Results show that, under practical scenarios, where N ranges from 2^{16} to 2^{40} and $k = \log N$, KT-ORAM yields about 1/2 to 1/5 of communication cost of P-PIR, and it saves even more compared with other ORAM schemes.

5.1 Preliminaries

Our proposed KT-ORAM employs the additively homomorphic encryption [67, 38] primitives and shares some basic ideas with P-PIR [48]. Hence, this section starts with an overview of additively homomorphic encryption primitives and P-PIR, which is followed 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.

5.1.1 Additively Homomorphic Encryption

Additively Homomorphic encryption (AH-encryption) [67, 38] 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:

$$\mathcal{E}(A) \oplus \mathcal{E}(B) = \mathcal{E}(A + B),$$

$$\mathcal{E}(A) \odot B = \mathcal{E}(A \cdot B).$$

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 [48]. As they are also used in KT-ORAM, we introduction their definitions below. Suppose a user 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 [12]. Primitives PIR-read and PIR-write are defined as follows.

PIR-read(m) When the user 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 user 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 $\overrightarrow{q}$:

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

Then, the server calculates

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

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

PIR-write(m, ΔD) When the user 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 user 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 $i \in \{1, \cdots, w\}$:

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

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

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 $\mathcal{E}(0)$.

5.1.2 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.

5.1.2.1 Storage Organization

Assuming N data blocks are exported by the user 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 [61]. 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 5.1 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 user 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 [61], 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.

5.1.2.2 Data Query Process

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

- The user 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 user 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 5.1.1) is launched to retrieve D_t; otherwise, the user launched process PIR-read(x) where x is a randomly-picked position in the node.

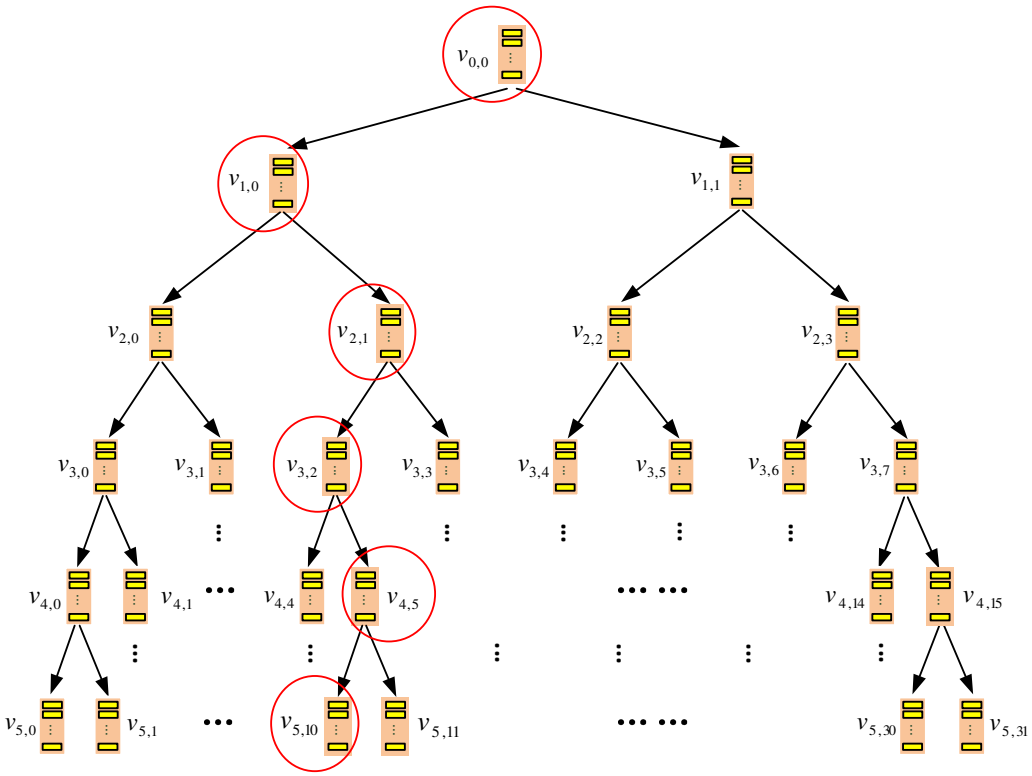


Figure 5.1 P-PIR's server-side storage structure. Circled nodes represent the ones accessed by the user during a query process when the target data block is mapped to leaf node $v_{5,10}$.

- 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 5.1, 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 user needs to download $2 \log N$ index and data blocks for each query.

5.1.2.3 Data Eviction Process

To prevent any node on the tree from overflowing, the following data eviction process is conducted by the user 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 5.2 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 $v_{3,5}$.

5.1.3 Limitation of P-PIR

Though P-PIR was proposed to reduce the communication cost, the overall communication cost is still as high as $O(\log N)$ data blocks per query (metadata recursion is of $O(1)$ depth). 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 [48], fetching 1 MB data incurs nearly 200 MB communication cost. Thus, 5 queries would result in almost

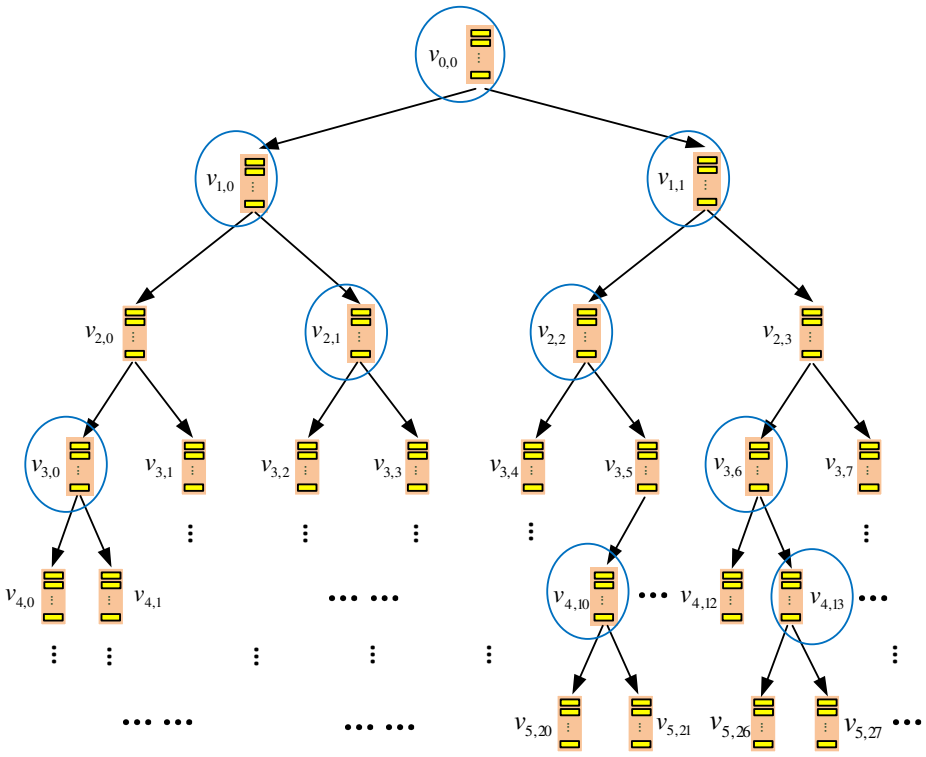


Figure 5.2 An example of the eviction process in P-PIR.

1 GB data transfer between the user 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 [65].

5.1.4 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 $O(\log N - \log \alpha)$ blocks per query (with $O(1)$ metadata recursions).

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 $O(\frac{\log N}{\log k})$. 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 $O(k \cdot \frac{\log N}{\log k})$. Consequently, the overall communication cost becomes $O(k \cdot \frac{\log N}{\log k})$ per query, which is higher than that of P-PIR.

5.1.5 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 (with $O(1)$ metadata recursions):

- *Replacement of the binary tree-based ORAM storage with a k-ary tree-based storage.* As we discussed in Section 5.1.4, adopting this idea can reduce the height of the tree structure and thus reduce query cost over the data block tree (excluding the recursions on the metadata) from $O(\log N)$ to $O(\frac{\log N}{\log k})$.

- *Execution of binary-tree eviction in a k-ary tree.* As also discussed in Section 5.1.4, directly implementing an eviction process on the k-ary tree causes a high cost of $O(k \cdot \frac{\log N}{\log k})$ 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.

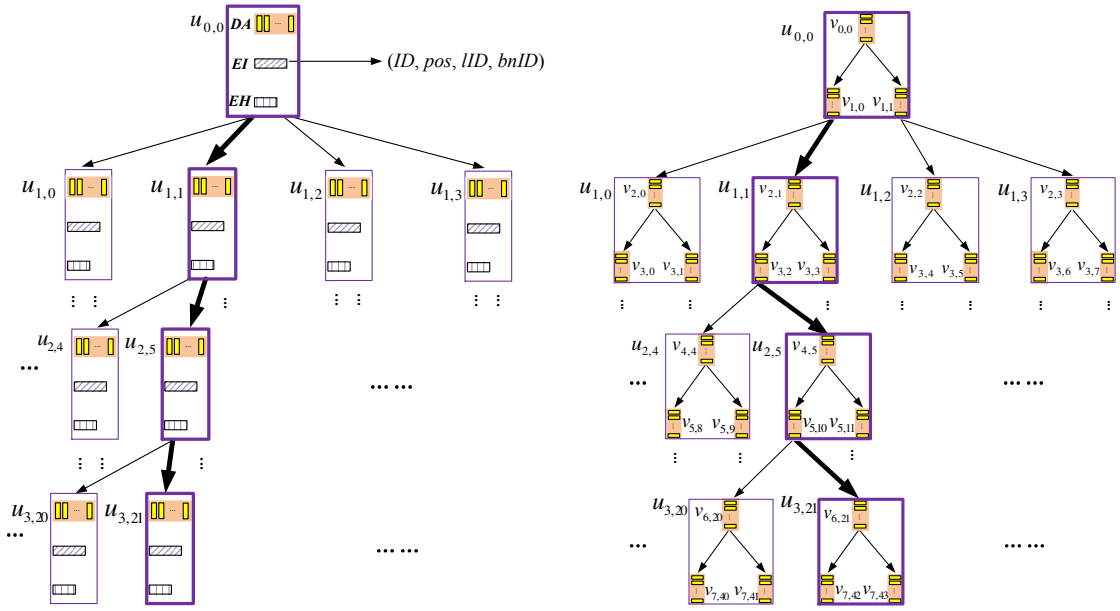
- *Delayed evictions.* This is a unique process in the proposed KT-ORAM. The key idea is that evictions between b-nodes within the same k-node may not be executed immediately by the user; instead, they may be recorded by the storage server in a data structure called *eviction history (EH)*, and multiple such recorded evictions may be executed at a later time in a batch to reduce the communication cost.

5.2 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.

5.2.1 Storage Organization

5.2.1.1 Server-side Storage



(a) physical view of the server-side storage (b) logical view of the server-side storage

Figure 5.3 An example KT-ORAM scheme with a quaternary-tree storage structure. Bold boxes represent the k-nodes accessed when a user queries a target data block stored at k-node $u_{3,21}$.

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 5.3, each k-node can be mapped to a binary subtree of $k-1$ nodes. For example, k-node $u_{0,0}$ in Figure 5.3(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 5.3(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 stores $2(k-1)\log\log N$ data blocks.

- *Encrypted Index Table (EI)*: a table of $2(k-1)\log\log N$ 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 $2\log N\log\log N$ records.

5.2.1.2 User-side Storage

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

- *A user-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 [61] and P-PIR [48]; to simplify presentation of the design in this section, however, we assume the table is maintained locally at the user side. Note that, similar to Path ORAM [66] and SCORAM [69], 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 user's secrets such as the keys used for data encryption and decryption.

5.2.2 System Initialization

To initialize the system, the user 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 user-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.

5.2.3 Data Query

To query a data block D_t with ID t, the user 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 and

then the EH is cleared. The details of this step will be explained later in Section 5.2.5, 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 user 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 user-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 user 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 5.3, the probability for such failure to occur is negligibly small.
- The user launches process PIR-read(m') to obliviously retrieve and decrypt dummy block D' from location m'.
- The user 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 5.3(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 $u_{3,21}$.

5.2.4 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 5.1.5, 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.

5.2.4.1 Phase I: Scheduling of Evictions for Logical Binary Tree

At the beginning of an eviction process, the user 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 b-nodes randomly picked from level l on the binary tree. Note that, v_{l,i_l} and v_{l,j_l} can be the same node. In this case, e_l becomes a single value v_{l,i_l}.

5.2.4.2 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 user 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 5.4, 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.

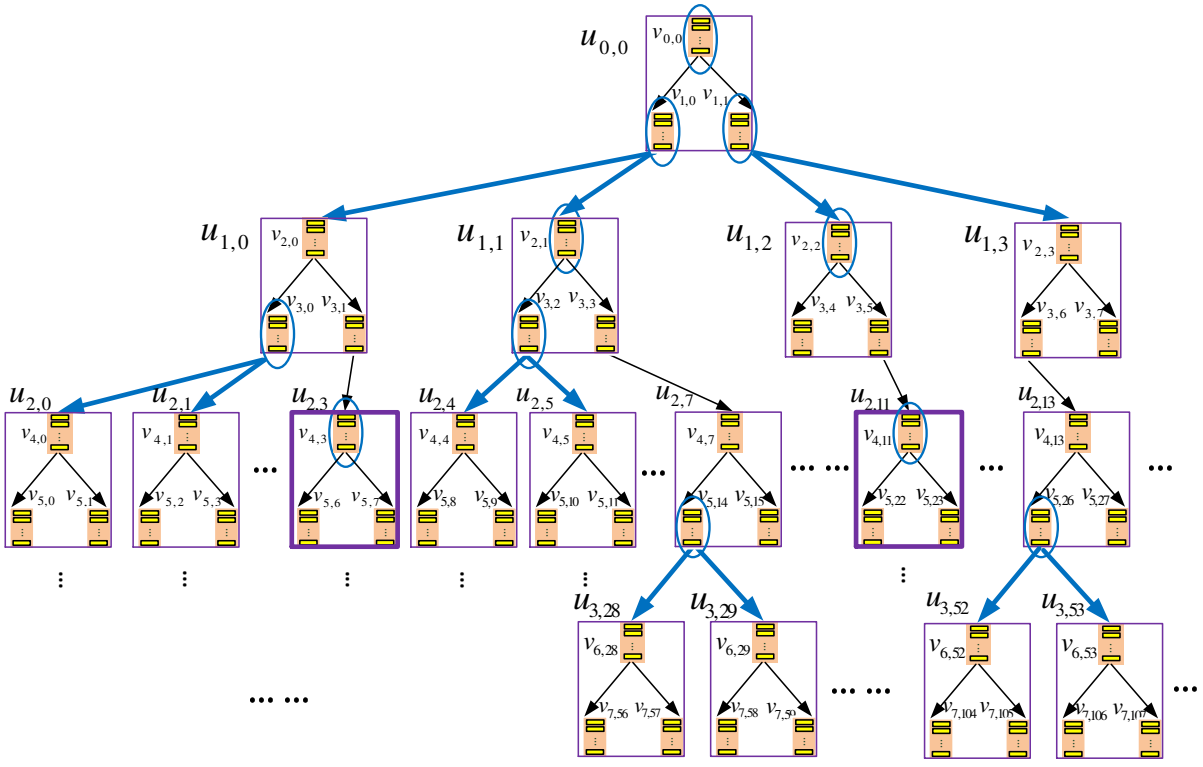


Figure 5.4 An example data eviction process in KT-ORAM with a quaternary-tree storage structure. The b-nodes that are selected to evict data blocks are circled. The k-nodes scheduled with delayed evictions (i.e., $u_{2,3}$ and $u_{2,11}$) are highlighted with bold boundaries.

Opportunities to Delay Intra k-node Evictions 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 5.4, $u_{2,3}$ 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 user 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 cost, 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 5.4, 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 5.2.5, the recorded evictions shall be executed first before any other updates.

5.2.4.3 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 5.2.5 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.

5.2.5 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.

5.3 Security Analysis

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

Lemma 4. *Assume $k \geq \log N$. The DA of any k-node in the k-ary tree has a probability of $O(N^{-\log\log N})$ to overflow.*

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

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

In the first step, we consider the binary tree that a k-ary tree in 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 [61] 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,

$$\pi_l(i) = \rho_l^i(1 - \rho_l), \tag{5.1}$$

where

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

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

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

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 (5.1), we have:

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

Note that, the first inequality is due to the following fact: for any l $(1 \leq l \leq \log N)$,

$$\pi_l(i) = \rho_l^i(1-\rho_l) \leq \rho_l^i \cdot \frac{2}{3} < \left(\frac{1}{2}\right)^i \cdot \frac{2}{3}.$$

The second inequality is due to $\binom{n}{k} \leq \left(\frac{n\cdot e}{k}\right)^k$ for all $1 \leq k \leq n$. The last inequality is due to

the fact that $t = 2(k-1)\log\log N$, $k \geq \log N$, and $\frac{2(t+k-2)\cdot e}{3(k-2)} \leq (\frac{1}{2})^{t/2}$. Therefore, we have:

$$\begin{aligned} &\mathbf{Pr}\left[Y_u \geq t\right] = \sum_{i=0}^{\infty} \mathbf{Pr}\left[Y_u = t+i\right] \\ < &\sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^{(t+i)/2} = \frac{(\frac{1}{2})^{t/2}}{1-(\frac{1}{2})^{1/2}} \leq 4 \cdot \left(\frac{1}{2}\right)^{t/2} \\ \leq &\ 4 \cdot \left(\frac{1}{2}\right)^{\log N \log\log N}. \end{aligned} \tag{5.2}$$

Given $t = 2\log N \log\log N$, Equation (5.2) renders a negligible probability of $O(N^{-\log\log N})$ 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

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

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

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

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

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

By applying the union bound, we obtain:

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

Further considering $j = 2(k-1)\log\log N$ and $k \geq \log N$, we have

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

To sum up, the number of data blocks in any k-node is bounded by $O(\log N \log\log N)$ with probability $1 - O(N^{-\log\log N})$. □

Lemma 5. *Any query process in KT-ORAM accesses k-nodes from each layer of the k-ary tree, uniformly at random.*

Proof. (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. □

Lemma 6. *An eviction process in KT-ORAM accesses a sequence of k-nodes independently of the user's private data request.*

Proof. (sketch) During an eviction process, the accessed sequence of k-nodes is independent to the user'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 user'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 user's private data requests. □

Theorem 2. *Assuming PIR-read and PIR-write are both oblivious operations, KT-ORAM is secure under Definition 2.2.*

Proof. Given any two equal-length sequence $\vec{x}$ and $\vec{y}$ of the user'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 user'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 5, the sequence of locations (i.e., k-nodes) accessed by each query process are uniformly random and thus independent of the user's private data request.
- According to Lemma 6, the sequence of locations (i.e., k-nodes) accessed by each eviction process after a query process is also independent of the user'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 4, the KT-ORAM construction fails with probability $O(N^{-\log\log N})$, which is considered negligible and no higher than the failure probability of existing ORAMs. □

5.4 Cost Analysis and Evaluations

This section analyzes the costs of KT-ORAM, and compares KT-ORAM with state-of-the-art ORAMs. To simplify presentation, we assume $k = \log N$.

5.4.1 Costs of KT-ORAM

The server-side storage of KT-ORAM is $O(N \log\log N \cdot B)$. Before analyzing the communication and computational costs of KT-ORAM, we introduce the following notations:

- B: size of a single data block in the system. We assume $B > \max\{6\log^2 N \log\log N), 2b \cdot \log N \log\log N\}$ bits. This assumption is commonly used in modern ORAM/file systems [65, 48], i.e., a moderate data block size ranges from 64 KB to 4 MB. For example, in P-PIR, it uses 1 MB; in Dropbox system [17], each data block size is 4 MB.
- H_k and H_b: heights of the k-ary and binary trees. Obviously, $H_b = \lceil \log N \rceil$ and $H_k = \lceil \frac{H_b}{\log\log N} \rceil = \lceil \frac{\log N}{\log\log N} \rceil$.
- b: size of an additively homomorphic encryption cipher-text, in the unit of bits. In practice, $b \ll B$. For example, in the NTRU implementation, $b = 2048$ bits, while $B = 1$ MB.

- S_{EH}: size of an EH. According to Lemma 7, the size of the EH is no more than $2\log^2 N \log\log N$ bits as there are $2\log N \log\log N$ records and each of them takes at most $\log N$ bits. In practice, this is no more than 3 KB when $N \leq 2^{40}$ and is at least one magnitude less than the data block size.

- S_{EI}: size of an EI. Each EI is $2\log N \log\log N \cdot \{2\log N + \log[2\log N \log\log N] + \log(\log N - 1)\}$ bits and this is no more than 5 KB given $N \leq 2^{40}$ and is at least one magnitude less than the data block size.

Lemma 7. *For any k-node in KT-ORAM, the probability that the EH of the k-node has more than* $2\log N \log\log N$ *records is* $O(N^{-\log\log N})$.

Proof. 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 $2\log N \log\log N$. Hence, we assume u is on layer l ($l > 0$) of the k-ary tree, and let $m = k^l \geq 2^l$ 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 $X_{l'}$) that are the IDs of b-nodes on level l' in the EH.

After every eviction process, $X_{l'}$ may increase by 1 or 2 if k-node u is not accessed by the user but some intra k-node evictions have been appended; or, it may decrease to 0 if it has been accessed by the user 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 $X_{l'}$. Hence, $X_{l'}$ can be modeled as a Markov Chain as shown in Figure 5.5.

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

Transition from $X_{l'} = i$ to 0 occurs when u is accessed by the user 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

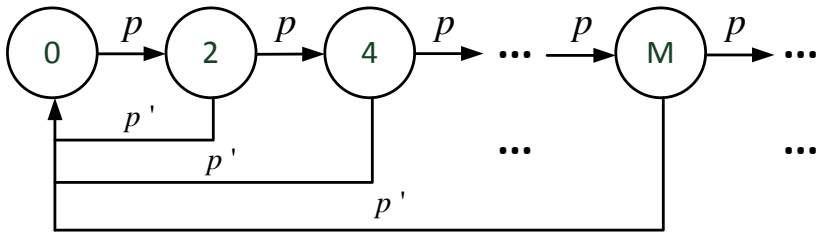


Figure 5.5 Markov Chain for random variable $X_{l'}$ (i.e., the number of EH entries from layer l').

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 $X_{l'} = i$ to $X_{l'} = i+2$ 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

$$\pi_i = \left(\frac{1}{2}\right)^{i/2+1}. \tag{5.3}$$

Since u has $\log k - 1$ such layers in its binary subtree, let's denote the number of entries in EH for each such layers as X'_l $(1 \le l' \le \log k - 1)$ and $Y_u = X_1 + \cdots + X_{\log k - 1}$.

Assume t is the maximum number of entries for the k-node u, there are $\binom{t + \log k - 2}{\log k - 2}$ different combinations of $X_i = t_i$ $(i = 1, \cdots, \log k - 1)$ such that $t_1 + \cdots + t_{\log k - 1} = t$. Hence, according to Equation (5.3), we have:

$$\begin{aligned} & \mathbf{Pr}\left[Y_u = t\right] = \mathbf{Pr}\left[X_1 + \cdots + X_{\log k - 1} = t\right] \\ \le\ & \binom{t + \log k - 2}{\log k - 2} \prod_{i=1}^{\log k - 1} \left[\left(\frac{1}{2}\right)^{t_i/2+1}\right] \\ \le\ & \left(\frac{(t + \log k - 2) \cdot e}{\log k - 2}\right)^{\log k - 2} \left(\frac{1}{2}\right)^{t/2 + \log k - 1}. \end{aligned} \tag{5.4}$$

Given $k = \log N$, $2^{16} \leq N \leq 2^{40}$, and $t = 2 \log N \log \log N$ we can simplify Equation (5.4) as:

$$\begin{aligned} \mathbf{Pr}\ [Y_u = t] &< \left(\frac{(t + \log k - 2) \cdot e}{\log k - 2}\right)^{\log k} \left(\frac{1}{2}\right)^{t/2 + \log k} \\ &< \left(\frac{t \cdot e}{2}\right)^{\log k} \left(\frac{1}{2}\right)^{t/2} < 6t^{\log k} \left(\frac{1}{2}\right)^{t/2} < 6 \left(\frac{1}{2}\right)^{t/4}. \end{aligned}$$

The last inequality can be proved based on $t^{\log k} \leq 2^{t/4}$, which can be easily obtained through the given conditions on N, k and t. Hence, we have the following equation holds:

$$\begin{aligned} \mathbf{Pr}\ [Y_u \geq t] &= \sum_{i=0}^{\infty} \mathbf{Pr}\ [Y_u = t + i] < \sum_{i=0}^{\infty} 6 \left(\frac{1}{2}\right)^{(t+i)/4} = 6 \frac{(\frac{1}{2})^{t/4}}{1 - (\frac{1}{2})^{1/4}} \\ &< 40 \left(\frac{1}{2}\right)^{\log N \log \log N/2} = O(N^{-\log \log N}). \end{aligned}$$

Thus, the maximum number of entries of EH of any k-node u can be bounded by $t = 2 \log N \log \log N$ with overwhelming probability $1 - O(N^{-\log \log N})$. □

5.4.1.1 Per-query Communication Cost

During a query process, one k-node is accessed from each layer of the k-ary tree. The user needs to (i) download the EI and EH of the k-node (needing $S_{EI} + S_{EH}$ bits); (ii) send one PIR-read vector (needing $2 \log N \log \log N \cdot b$ bits); (iii) upload the EI (needing S_{EI} bits). Note that, after all PIR-reads have been executed by the server, there are H_k data blocks on the server. At this time, the user does not retrieve those data blocks. Instead, s/he launches another PIR-read on these H_k data blocks to fetch only one data block. This PIR-read requires the user to send one PIR-read vector (needing $H_k \cdot b$ bits) and download the target data block (needing B bits). To wrap up a query, the target data block will be obliviously written back to the root k-node using one PIR read and PIR-write (EI has been retrieved before, so, the PIR-read and PIR-write here only transfers the read and write vectors and two data blocks. Thus, this is $4 \log N \log \log N \cdot b + 2B$ bits). Therefore, the communication cost per query is:

$$Qu(N) = 3B + H_k \cdot (2S_{EI} + S_{EH}) + (H_k + (2H_k + 4) \log N \log \log N) \cdot b.$$

During an eviction process, at most two k-nodes for each non-bottom layer will be selected for actual eviction, each of which requires one PIR-read. Meanwhile, four k-nodes are selected

as the children of previous layer, where each k-node requires one PIR-read and one PIR-write. All the EI and EH of these six k-nodes will be retrieved and the EI will be uploaded back after the eviction operations (needing $12S_{EI} + 6S_{EH}$ bits). Two optimization techniques similar to P-PIR can be applied here. First, when two child k-nodes of the same parent are accessed, only one of them contains the block that is required by the user and the other one is dummy. Thus, the server can further add the two block from this two child k-nodes into one, thus, the user retrieves 2 data blocks from 2 selected parent k-node and 2 data blocks from 4 children k-node (needing $4B + 12 \log N \log\log N \cdot b$ bits). Second, when PIR-write are executed on the two child k-nodes of the same parent, only one data block is uploaded to the server and the PIR-write vector is the same as that of the PIR-read vector (needing $2B$). Hence, the total bandwidth consumption is bounded by

$$Ev(N) = (H_k - 1) \cdot (6S_{EH} + 12S_{EI} + 12 \log N \log\log N \cdot b + 6B).$$

Due to the assumptions of B, when the index table is stored at the user storage, we have the communication cost of KT-ORAM is: $O(\frac{\log N}{\log\log N} \cdot B)$. If the user-side index table is exported recursively, the overall bandwidth consumption per query is

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

which is $O(\frac{\log^2 N}{\log\log N} \cdot B)$.

5.4.1.2 Per-query Computational Cost

Server-side Computational Cost The server-side computational cost comes from the homomorphic addition and multiplication operations; hence, we only count such operations on data block tree (except the metadata part).

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 $2(\log N - 1) \log\log N$ blocks each with B bits, there are B/b data pieces operate-able by AH operations, each PIR-read operation on a k-node requires $Comp_{Mul} = 2(\log N - 1) \log\log N \cdot B/b$ AH multiplications and $Comp_{Add} = [2(\log N - 1) \log\log N - 1] \cdot B/b$ AH additions. Therefore,

the computational cost for a query process is

$$H_k(Comp_{Mul} + Comp_{Add}) = O(\log^2 N \cdot \frac{B}{b})$$

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 $2(\log N - 1)\log\log N \cdot B/b$. Therefore, the computational cost for an eviction process is $O(\log^2 N \cdot \frac{B}{b})$ AH operations.

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

User-side Computational Cost The computational cost at the user side is mainly contributed by decrypting and re-encrypting downloaded blocks (data blocks and EI), where each block needs normal encryption (e.g., AES) and/or homomorphic decryption/re-encryption (except the metadata part).

For each PIR-read of one block from w blocks, the user needs to calculate and send the query vector to the server. Thus, the calculation takes w AH operations. For each PIR-write of one block from w blocks, the user will send one query vector of w AH operations and one data block with B/b normal encryption.

The number of data blocks accessed per query is $O(1)$ with the optimization in Section 5.4.1.1. For data query, the user needs to send the query vector, which takes $H_k \cdot 2(\log N - 1)\log\log N = O(\log^2 N)$ AH operations. For data eviction, the user sends vectors taking $6H_k \cdot 2(\log N - 1)\log\log N = O(\log^2 N)$ AH operations and $12B/b$ normal encryption operations for data block encryption. Therefore, the computational cost for the user is $O(\max\{\log^2 N, B/b\})$.

5.4.2 Comparisons with Existing ORAMs

Detailed comparisons between KT-ORAM and several state-of-the-art ORAMs including B-ORAM [40], T-ORAM [61], G-ORAM [24], Path ORAM [66], SCORAM [69], and P-PIR [48]

are reported in this section.

5.4.2.1 Asymptotical Comparisons

First, we show the asymptotical comparisons in terms of the communication, user storage costs and failure probability.

Table 5.1 Asymptotical comparisons. $k = \log N$ for both KT-ORAM and G-ORAM and $B = O(N^{\epsilon})$ $(0 < \epsilon < 1)$.

ORAM	Comm. Cost	User Storage	Server Storage	Failure Probability
B-ORAM [40]	$\Omega(\log^3 N \cdot B)$ $(N \leq 2^{37})$ | $O(\frac{\log^2 N}{\log\log N} \cdot B)$ $(N > 2^{37})$	$O(N \cdot B)$	$O(B)$	$O(N^{-\log\log N})$
T-ORAM [61]	$O(\log^2 N \cdot B)$	$O(B)$	$O(N \log N \cdot B)$	$O(N^{-c})$
G-ORAM [24]	$O(\frac{\log^2 N}{\log\log N} \cdot B)$	$O(\log^2 N \cdot B) \cdot \omega(1)$	$O(N \cdot B)$	$O(N^{-\omega(1)})$
Path ORAM [66] SCORAM [69]	$O(\log N \cdot B) \cdot \omega(1)$	$O(\log N \cdot B) \cdot \omega(1)$	$O(N \cdot B)$	$O(N^{-\omega(1)})$
P-PIR	$O(\log N \cdot B)$	$O(B)$	$O(N \log N \cdot B)$	$O(N^{-c})$
KT-ORAM	$O(\frac{\log N}{\log\log N} \cdot B)$	$O(B)$	$O(N \log\log N \cdot B)$	$O(N^{-\log\log N})$

From Table 5.1, the communication cost of KT-ORAM is asymptotically lower than or equal to the state-of-the-art constructions, when they have the same level of failure probability $O(N^{-\log\log N})$. In addition, KT-ORAM requires only a constant user storage. Note that, B-ORAM incurs $O(\frac{\log^2 N}{\log\log N})$ when the database size is extremely large, but it is degraded to $\Omega(\log^3 N)$ when $N \leq 2^{37}$ [48]. Also, there is a large constant behind the big-O notation.

Table 5.2 shows the computational comparison between KT-ORAM and P-PIR, which shows that KT-ORAM does not increase the asymptotical computational cost compared to P-PIR.

5.4.2.2 Comparisons under Practical Settings

Next, the communication cost is compared between KT-ORAM, SCORAM, and P-PIR. Note that, T-ORAM and G-ORAM are outperformed by Path ORAM according to [66]. Path ORAM is outperformed by SCORAM and P-PIR according to [69] and [48]. We assume b is fixed to 2048 bits same as P-PIR and N ranges from 2^{16} to 2^{40}. When data block size B varies between 64 KB and 4 MB, as shown in Figure 5.6, KT-ORAM outperforms P-PIR and SCORAM in all the studied scenarios. Particularly, the communication cost introduced by KT-ORAM is only about 1/2 to 1/5 of that by P-PIR and 1/4 to 1/8 of that by SCORAM.

Table 5.2 Computational cost comparisons with $O(1)$ recursion levels.

ORAM	User-side	Server-side
P-PIR	$O(\max\{\log^2 N, B/b\})$	$O(\log^2 N \cdot \frac{B}{b})$
KT-ORAM	$O(\max\{\log^2 N, B/b\})$	$O(\log^2 N \cdot \frac{B}{b})$

5.4.2.3 Communication Cost before Target Data Access

Communication Cost before Target Data Access The communication occurring before the user can access its target data has major impact on data query latency. Hence, similar to [48], we also analyze the communication cost before target data access.

In KT-ORAM, the following communication occurs before the user can access its target data : (i) $2\log N/\log\log N$ EIs of size S_{EI} sent from the server to the user or uploaded back to the server, for which the amount of communication is $2\log N/\log\log N \cdot S_{EI}$ bits; (ii) one PIR vector shared by all k-nodes on the retrieved path sent from the user to the server to perform PIR-read primitives, for which the amount of communication is $c(\log N - 1) \cdot b$ bits; (iii) another PIR vector sent to the server to retrieve the only target data, for which the amount of communication is $\log N/\log\log N \cdot b$ bits; (iv) the target data block sent from the server to the user (needing B bits). Thus, the cost of communication occurring before the user can access its target data in KT-ORAM is

$$[c(\log N - 1) + \frac{\log N}{\log\log N}] \cdot b + \frac{2c\log^2 N}{\log\log N} \cdot (2\log N + \log c) + B,$$

which is $O(B + \log N \cdot b)$ based on the data block size assumption.

To compare the query latency caused by different ORAM constructions, similar to [48], we further compare KT-ORAM and P-PIR in terms of the communication cost before target data access. Note that, we do not compare KT-ORAM with other state-of-the-art ORAM constructions, as [48] has conducted the comparison and showed P-PIR outperforms others in terms of this metric.

Asymptotical Comparisons Table 5.3 shows the result of asymptotical comparison between KT-ORAM and P-PIR [48, 65]. As we can see, KT-ORAM has the same level of communication cost before target data access, compared to P-PIR. Similar to 5.4.1.1, we consider

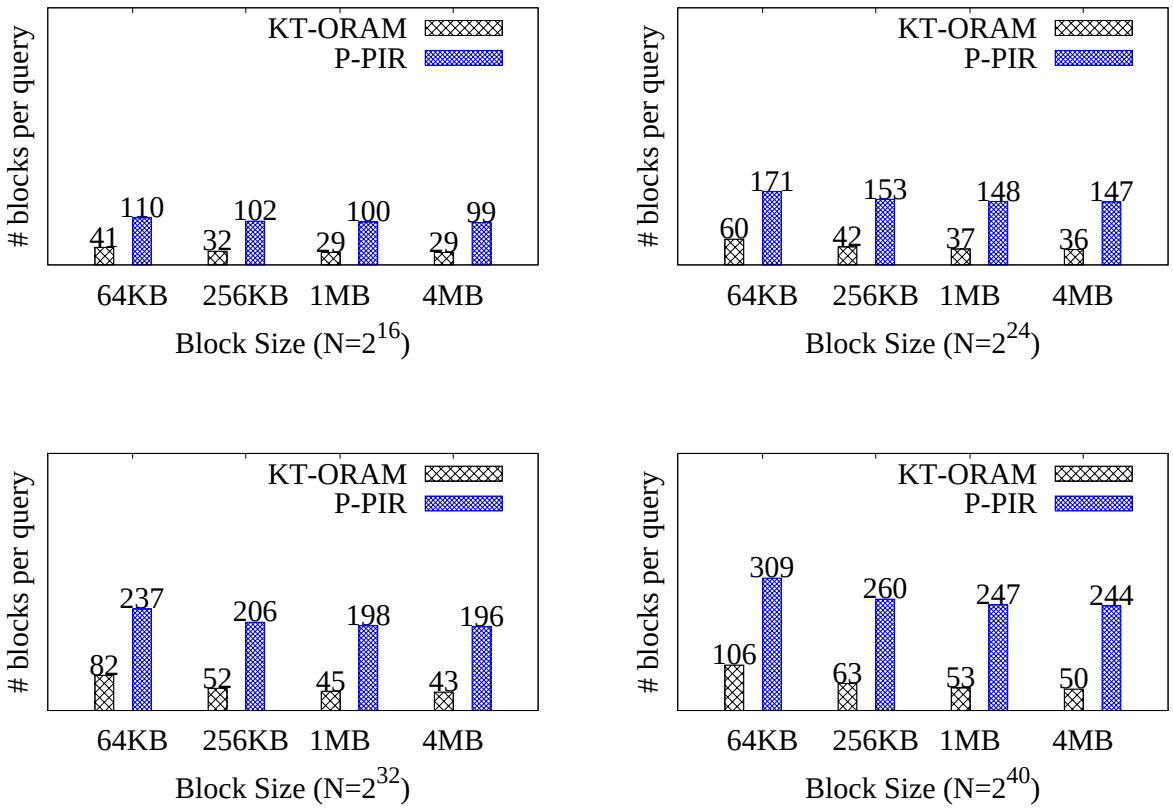


Figure 5.6 Communication cost comparisons. The above comparisons show the communication cost to transfer user's data block part. For KT-ORAM, the k-ary tree node size is set to be $2(\log N - 1)\log\log N$, tree height is $\lceil \frac{\log N}{\log\log N} \rceil$, and user's storage stores $O(1)$ data blocks. For Path-PIR, the binary tree node size is set to be $\log N$, tree height is $\log N$, and user's storage stores $O(1)$ data blocks. In SCORAM, the binary tree node size is set to be $Z = 5$, the tree height is $\log N$ and user's storage stores $O(\log N) \cdot \omega(1)$ data blocks.

two scenarios in the compared ORAMs: the index structure is exported and accessed in $O(1)$ recursion levels or in $O(\log N)$ recursion levels.

Numeric Comparisons Figure 5.7 shows the results of numeric comparisons between KT-ORAM and P-PIR. As we can see, the communication before target data access of KT-ORAM is similar to that of P-PIR. Comparing Figure 5.7 and Figure 5.6, we can also see that, the communication cost before target data access accounts for only 1/20 to 1/50 of the total communication cost in KT-ORAM. Most of the communication can be performed at the background.

Table 5.3 Asymptotical Communication before Target Data Access. 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 $c = 7$ for KT-ORAM.

ORAM	Communication Cost with $O(\log N)$ Recursion Levels	Communication Cost with $O(1)$ Recursion Levels
P-PIR	$O(\log N(\log N \cdot b + B))$	$O(\log N \cdot b + B)$
KT-ORAM	$O(\log N(\log N \cdot b + B))$	$O(\log N \cdot b + B)$

5.5 Summary

In the second work, we proposed 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 preserve the data access pattern privacy with a negligibly-small failure probability of $O(N^{-\log\log N})$ where N is the number of exported data blocks. With a constant-size user storage and $k = \log N$, KT-ORAM has an asymptotical communication cost of $O(\frac{\log N}{\log\log N} \cdot B)$ when the recursion level on metadata 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 $O(\log N)$. In addition, KT-ORAM outperforms all these constructions in terms of communication and user-side storage costs, under practical scenarios.

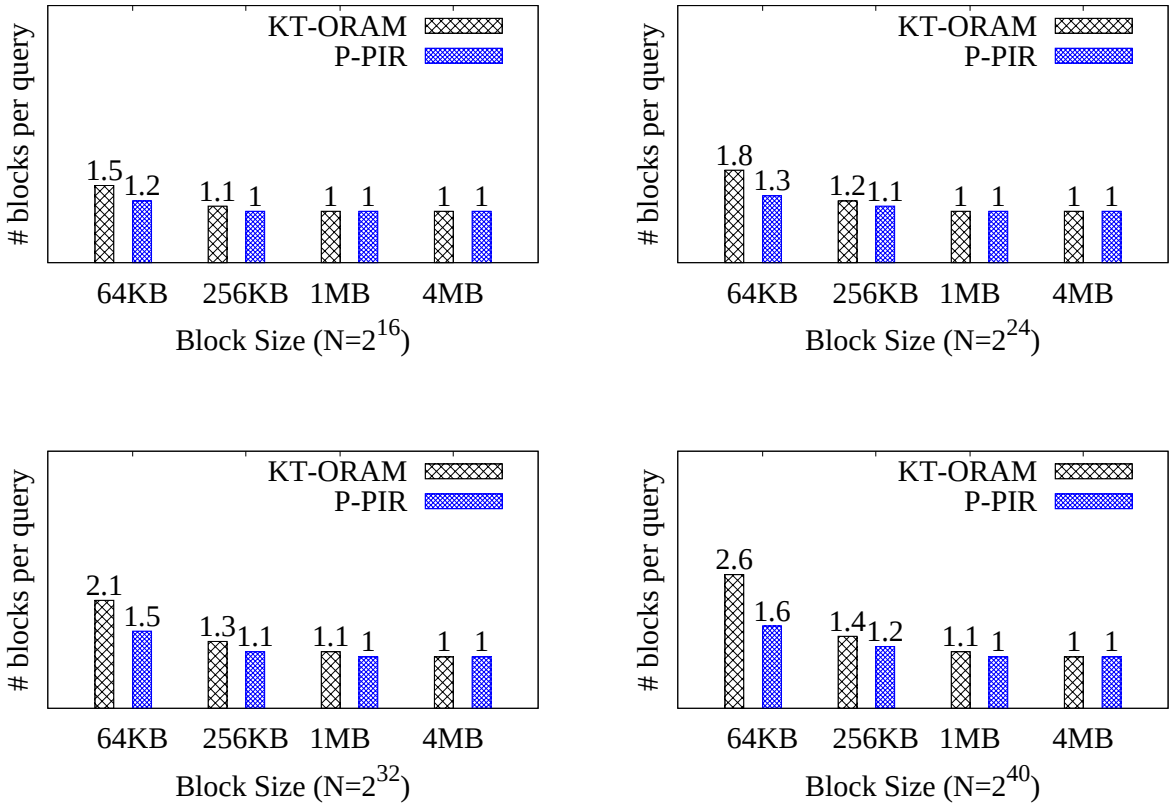


Figure 5.7 Numerical comparison of communication before target data access in practical scenarios. $k = \log N$ and $c = 7$ for KT-ORAM. The number of blocks N ranges from 2^{16} to 2^{40} and the block size B ranges from 64 K bytes to 4 M bytes.

CHAPTER 6. GP-ORAM: A GENERALIZED PARTITION ORAM

In the third work, we proposed a generalized partition ORAM, GP-ORAM [78], which is motivated by the design of P-ORAM [65]. P-ORAM construction was designed to achieve a low and thus practically acceptable communication cost. Specifically, the server-side storage of P-ORAM is organized as $\sqrt{N}$ partitions, assuming N is the number of exported data blocks, and each partition is an ORAM. The user-side storage includes an index table recording the location of each block, a shuffling buffer that can store and shuffle all data blocks of any ORAM partition, and $\sqrt{N}$ stash slots. With such a storage arrangement, it has been shown that the communication cost for data query and shuffling is as low as $\log N$ data blocks per query. Compared to other state-of-the-art ORAM constructions[66, 77, 48], P-ORAM achieves higher communication efficiency.

However, P-ORAM design has its limitations. First of all, it requires a large and fixed local storage to store the index table and facilitate shuffling. For example, when $N = 2^{32}$ and block size is 64 KB, 31 GB local storage is needed. Second, the index table cannot be efficiently exported to the server. According to our evaluation, if the index structure is exported to the server, in order to query just a single block, more than 1000 data blocks on average have to be retrieved. In addition, the user's accesses to data blocks have to be entirely sequential in order to compress the index table.

To address the above limitations of P-ORAM, while inheriting its nice feature of low communication cost, this work proposes a generalized version of P-ORAM, called GP-ORAM. There are a few key improvements of GP-ORAM over P-ORAM. First, the number of partitions is adjustable in GP-ORAM. This way, even with a smaller local storage than what P-ORAM requires, GP-ORAM may still achieve a low communication cost via properly adjusting the number of partitions. Second, each ORAM partition in GP-ORAM is redesigned (different

from that in P-ORAM) to enable efficient query and shuffling. Finally, the index structure in GP-ORAM is also redesigned to enable efficient exportation of it and accommodate the above changes.

Rigorous security analysis has been conducted to prove that the proposed GP-ORAM construction can preserve a user's access pattern and the construction fails with only a probability of $O(N^{-\log\log N})$ according to Definition 2.2. Extensive cost analysis has also been conducted to show that GP-ORAM is a more practical construction than P-ORAM. Particularly, the local storage demanded by the recursive version of our proposed GP-ORAM scheme is only 2.5%∼0.14% of that by the non-recursive version of the P-ORAM scheme (note: as shown in Section 6.3, the recursive version of the P-ORAM scheme is impractical due to its extremely high communication cost, and therefore is not considered), while GP-ORAM only yields 1 to 3 times higher communication cost than P-ORAM.

6.1 Intuition

As GP-ORAM is generalized from P-ORAM, we first review the key ideas and limitations of P-ORAM. As shown in Figure 6.1, the server-side storage of P-ORAM is organized as $\sqrt{N}$

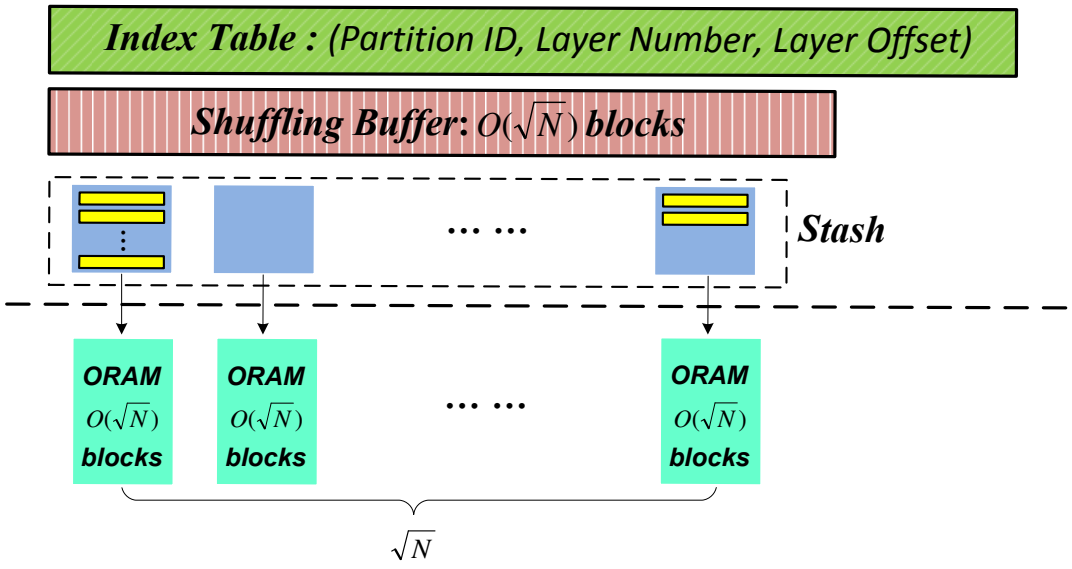


Figure 6.1 P-ORAM Storage Organization.

ORAM partitions, while the user-side storage includes an index table recording the location

(i.e., partition ID, layer number and layer offset) of each block, a shuffling buffer that can store and shuffle $O(\sqrt{N})$ data blocks and $\sqrt{N}$ stash slots each corresponding to one partition. To query one data block, it needs to retrieve one data block from each layer of an ORAM partition on the server, which results in $O(\log N)$ data blocks of communication cost, and the query target block is relocated to a randomly selected stash slot. Each query is followed by a background eviction, in which some data blocks are evicted from stash slots into their corresponding ORAM partitions; the evictions cause the ORAM partitions to be gradually reshuffled, and shuffling causes $O(\log N)$ data blocks of communication cost per query, on average. To summarize, as bandwidth is usually more expensive than storage, P-ORAM was designed to achieve a low communication cost at the cost of increased local storage.

However, P-ORAM has the following limitations. First, P-ORAM requires a large local storage ($O(\sqrt{N}B)$ bits), due to $\sqrt{N}$ stash slots and a shuffling buffer with a capacity of $O(\sqrt{N})$ blocks. This limits P-ORAM's practical applicability as it is impossible to implement P-ORAM if the user has less local storage than required. Second, the index table cannot be efficiently outsourced to the server. Each entry of the table has three fields: partition ID, layer number, and layer offset. The layer number and layer offset need to be updated during both query and shuffling processes. If the index table is outsourced to the server, the query and shuffling processes need to frequently query and update the index table, which leads to impractically high communication cost. Third, the user's data accesses have to be entirely sequential in order to compress the index table.

Motivated by P-ORAM and also to overcome its limitations, we present GP-ORAM as a new framework to assemble multiple ORAM partitions together. It has the following key ideas. First, the number of partitions is not fixed so that the user can adjust the number of partitions according to the available local storage. Second, the index table is re-designed so that it can be outsourced to the server efficiently. Third, to make full use the available local storage, each ORAM partition is based on a revised S-ORAM [77] construction. As a result, GP-ORAM inherits the security property and the communication efficiency of P-ORAM while being able to work with and fully utilize a wide range of available local storage.

6.2 Scheme

We elaborate the design of GP-ORAM in terms of storage organization, system initialization, query process, and background eviction process. To simplify the presentation, we assume the user stores index entries of all outsourced data blocks locally. In practice, to save the user's local storage, the index entries can be recursively exported to the storage server, following the same ideas used in tree ORAM [61] and Path-ORAM [66], which is described in Section 6.3.

6.2.1 Storage Organization

GP-ORAM stores both real blocks (i.e., user's N actual data blocks outsourced to the server) and dummy blocks (i.e., faked data blocks with random padding). When a block is in plain-text, it can be split into *pieces* and the size of each piece is $b = \log N$ bits. For each real block, the block ID i is contained in its first piece, denoted as $d_{i,1}$, while the first piece of each dummy block is set to -1. The remaining pieces store the content of that block, denoted as $d_{i,2}$, $d_{i,3}$, $\cdots$, $d_{i,\eta-1}$.

Before being exported to the remote storage server, the plain-text block is encrypted using CTR encryption mode (counter encryption mode) [55] piece by piece with a secret key k. Specifically, the ciphertext of each block D_i contains η pieces, denoted as $c_{i,0}, \cdots, c_{i,\eta-1}$, where

$c_{i,0} = E_k(ctr)$, where ctr is a nounce generated by a pseudo-random function;

$c_{i,1} = E_k(ctr + 1) \oplus d_{i,1}$;

$\cdots$;

$c_{i,\eta-1} = E_k(ctr + \eta - 1) \oplus d_{i,\eta-1}$.

Thus, the encrypted block (denoted as D_i) is $D_i = (c_{i,0}, c_{i,1}, c_{i,2}, \cdots, c_{i,\eta-1})$.

6.2.1.1 Server Storage

The server-side storage is divided into P smaller fully-functional ORAM partitions, where P is a system parameter. Each partition can hold $1.1N/P$ real blocks. As shown in Lemma 8 (Section 6.4), given that $\log N \log\log N \leq P \leq \sqrt{N}$, the number of real blocks in each partition is upper bounded by $1.1N/P$ with a probability of $1 - O(N^{-\log\log N})$.

In GP-ORAM, each ORAM partition is a revised version of the S-ORAM [77] construction. Specifically, each partition is organized as a pyramidical structure shown in Figure 6.2, where the total number of layers is denoted as $L_2 = \lceil \log(N/P) \rceil$. The top layer, i.e., layer 1, is an array containing up to four blocks. Each of the rest layers is organized as one or multiple segments. These layers are further divided into single-segment layers (i.e., T1-layers, including layers 2 to $L_1 = \lfloor \log(3\log^2 N) \rfloor - 1$) and multi-segment layers (i.e., T2-layers, including layers $L_1 + 1$ to L_2).

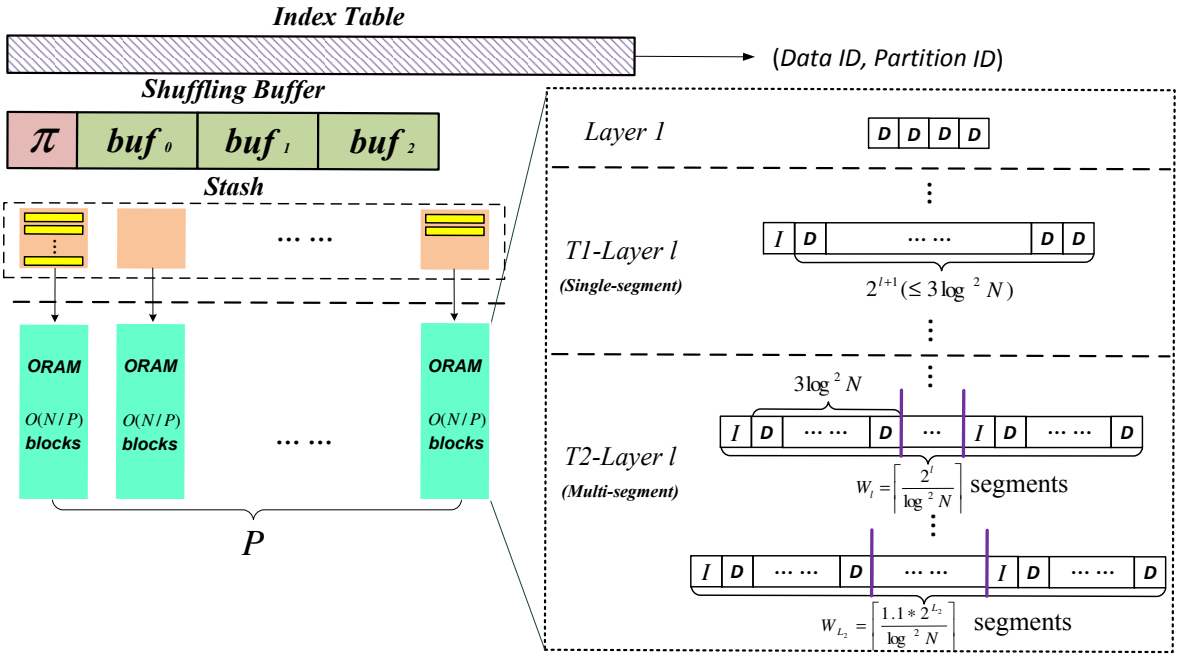


Figure 6.2 Organization of the server-side storage.

Each T1-layer l has a single segment. The segment stores 2^{l+1} blocks, at most half of which are real blocks, and one encrypted index block I_l with 2^{l+1} entries. Each entry of I_l corresponds to a block in the segment and consists of three fields: *ID of the block*, *location of the block in the segment*, and *access bit* indicating whether the block has been accessed since it was placed to the segment.

For each T2-layer $l < L_2$, it is composed of $W_l = \lceil 2^l / \log^2 N \rceil$ segments, while the bottom layer (i.e., layer L_2) contains $W_{L_2} = \lceil 1.1 * 2^{L_2} / \log^2 N \rceil$ segments. The bottom layer has slightly more segments, because it should be able to accommodate $1.1N/P$ real data blocks. A T2-layer segment has the same format as a T1-layer segment except that it needs to contain exactly

$3\log^2 N$ data blocks. Having $3\log^2 N$ data blocks per segment is to ensure the security property of the design and it has been proved by Zhang et. al. [77].

Inside each segment, there is an index block with at most $3\log^2 N$ entries and each entry contains three fields: *ID of the block* (needing $\log N$ bits), *location of the block in the segment* (needing $\log(3\log^2(1.1N/P))$ bits), and *access bit* (needing 1 bit). Thus, an index block needs at most $3\log^2 N[\log N + \log(3\log^2(1.1N/P)) + 1]$ bits. In practice, with $N \leq 2^{32}$ which is considered large enough to accommodate most practical applications, the size of an index block is less than 32 KB, which can fit into a typical block assumed in P-ORAM [65].

In addition, each ORAM partition p maintains a counter C_p to keep track of the times that the partition has been queried.

6.2.1.2 User Storage

The user-side storage consists of the following components. (i) **Stash with P slots**: each stash slot corresponds to one of the ORAM partitions; that is, it buffers the blocks that should be written to the corresponding partition later. (ii) **Shuffling buffer**: the shuffling buffer (with the capacity of S blocks) is used for data shuffling process. (iii) **Index table**: the index table records the information of each block. Specifically, it has N entries and each entry (p_i, l_i) has two fields; the block is in partition p_i and the block is latest stored on layer l_i. (iv) **Secret storage**: it stores all secrets including cryptographic keys for encryption and authentication, and its size is negligible compared to the other components.

6.2.2 System Initialization

To initialize, the user first selects a data encryption key, denoted as k. Then, each real block is encrypted and randomly assigned to one of the P partitions; the local index table is initialized to reflect the assignment.

After the above assignment, the user initializes each partition p_i as follows. For each of the real blocks D_j assigned to partition p_i, the user selects a secure hash function, denoted as $H_{p_i,L_2}(*)$, for the bottom layer L_2, and assign D_j to segment $H_{p_i,L_2}(j)$. Then, the user adds dummies to ensure each segment contains exactly $3\log^2 N$ blocks. For each segment, the user

randomly permutes all blocks inside it and builds an encrypted index block for it. Finally, the index and data blocks are uploaded to the server.

6.2.3 Data Query

To query a data block D_t, the user first searches the index table to get partition ID p_t and layer number l_t for D_t. Then, the user searches the stash slot of p_t. If D_t is not found, the user will launch a query for D_t in partition p_t; otherwise, a dummy query to p_t will be launched.

Algorithm 6 $Query(D_t, p_t)$

1: $\mathcal{L} \leftarrow$ the set of non-empty layers of partition p_t
2: Retrieve C_p from partition p_t
3: **if** (D_t is a dummy block) **then**
4: $\quad \mathcal{S} \leftarrow \{seg_l | \forall\ l \in \mathcal{L}, seg_l$ is a randomly-selected segment of layer $l\}$
5: $\quad$ Retrieve the index block of each segment in $\mathcal{S}$
6: $\quad$ From each segment in $\mathcal{S}$, retrieve a dummy block that has not been accessed
7: $\quad$ Update, re-encrypt & upload the retrieved index block
8: **else**
9: $\quad$ Find layer $\hat{l}_t$ where D_t is located; $seg_{\hat{l}_t} \leftarrow H_{p_t,\hat{l}_t}(t)$
10: $\quad$ //Secure hash function $H_{p_t,\hat{l}_t}(t)$ decides which segment of layer $\hat{l}_t$ in partition p_t stores D_t
11: $\quad \mathcal{S} \leftarrow \{seg_l | \forall\ l \in \mathcal{L} \setminus \{\hat{l}_t\}, seg_l$ is a randomly-selected segment of layer $l\}$
12: $\quad$ Retrieve the index blocks of segments in $\mathcal{S} \cup \{seg_{\hat{l}_t}\}$
13: $\quad$ From each segment $s \in \mathcal{S} \cup \{seg_{\hat{l}_t}\}$, retrieve a dummy block that has not been accessed if $s \in \mathcal{S}$, or D_t otherwise
14: $\quad$ Update, re-encrypt & upload the retrieved index block
15: **end if**

The algorithm for querying D_t in partition p_t, i.e., $Query(D_t, p_t)$, is revised from the query algorithm in S-ORAM [77] and formally presented in Algorithm 6. In the algorithm, the layer $\hat{l}_t$ where D_t is located is found as follows: First, based on the query counter C_{p_t}, the most recently shuffled layer l' can be inferred. Then, $\hat{l}_t \leftarrow l'$ if $l' \geq l_t$ because D_t must have been shuffled to l' during the most recent shuffling process; otherwise, $\hat{l}_t \leftarrow l_t$.

6.2.4 Background Eviction

After each data query, a background eviction process as described in Algorithm 7 should be launched to avoid stash overflowing. Similar to P-ORAM, this process could be sequential or

random. For simplicity, we adopt the sequential approach. Suppose ψ records the last evicted stash slot and λ denotes the eviction rate (i.e., the number of stash slots that should be evicted after each data query). The eviction operation essentially pushes one data block from its stash slot to layer 1 of its corresponding partition. As the capacity of layer 1 is limited, every four eviction operations performed on a partition could result in layer 1 overflow and thus should trigger a data shuffling of that partition.

Algorithm 7 Sequential Background Eviction (λ)

1: **for** $k = 1$ to λ **do**
2: $\psi \leftarrow (\psi + 1) \mod P$
3: **if** (stash slot[ψ] does not contain real block) **then** write a dummy to layer 1 of p_ψ
4: **else**remove a real block from stash slot[ψ] and write it to layer 1 of p_ψ
5: **end if**
6: $C_{p_\psi} \leftarrow C_{p_\psi} + 1$
7: **if** ($C_{p_\psi} \bmod 4 = 0$) **then**
8: Shuffle partition p_ψ
9: **end if**
10: **end for**

Different from P-ORAM, GP-ORAM shuffles data in *pieces* instead of *blocks*, as in S-ORAM [77]. To shuffle a certain x number of blocks in the unit of piece, only bx bits of local storage is needed, while Bx bits of local storage would be needed if shuffling these blocks in the unit of block. Hence, GP-ORAM can utilize the shuffling buffer more efficiently than P-ORAM. To facilitate fine-grained shuffling, the shuffling buffer is split into the following two components (as shown in Figure 6.2): (i) π, which is a buffer to store a *permutation* of up to $2m^2$ inputs and thus needs $2m^2 \log(2m^2)$ bits, where m is a system parameter; (ii) buf_0, which is used to temporarily store up to $2m^2$ data pieces. Recall that each data piece has b bits and the capacity of the shuffling buffer is S bits. In GP-ORAM, we set the shuffling buffer size to

$$S = 4.4 \cdot \frac{N}{P} \cdot (\log(4.4 \cdot \frac{N}{P}) + b). \tag{6.1}$$

The purpose is to ensure that, for any layer of each partition, each block is downloaded and uploaded for only once during a shuffling process. The shuffling process is the same as in S-ORAM [77], and thus is skipped here due to space limitation.

6.3 Recursive GP-ORAM

In the construction presented in Section 6.2, the user needs to maintain an index table in local storage. To reduce the cost, we can adopt recursive construction to outsource the index table to the server. Specifically, letting GP-ORAM$_1$ denote the original GP-ORAM used to store data blocks, a new GP-ORAM$_2$ can be introduce to store the index table of GP-ORAM$_2$; furthermore, another GP-ORAM called GP-ORAM$_3$ may be introduced to store the index table of GP-ORAM$_2$, and so on and so forth. Suppose one block in GP-ORAM$_{i+1}$ can store up to α index entries of GP-ORAM$_i$ ($1/\alpha$, therefore, is the compression rate, which is the ratio of GP-ORAM$_{i+1}$'s capacity to GP-ORAM$_i$'s capacity). Then, in each block of GP-ORAM$_{i+1}$, $\log(\frac{N}{\alpha^i})$ bits are needed to represent a sequence of α blocks in GP-ORAM$_i$ and $\alpha \cdot \log P_i$ bits are needed to record the partitions which these blocks should be stored to, where P_i is the number of partitions in GP-ORAM$_i$. Therefore, the relation between α, N, B and P_i is as Equation (6.2):

$$\log(\frac{N}{\alpha^i}) + \alpha \cdot \log P_i \leq B. \tag{6.2}$$

With the recursive construction, the local storage can be greatly reduced while the extra communication cost is insignificant. This is analyzed in detail in Section 6.5.

6.4 Security Analysis

To show that GP-ORAM is secure according to Definition 2.2 in Section 2.2, we develop a proof in two parts: (1) GP-ORAM generates a random access pattern independent of user's actual access pattern, and (2) GP-ORAM fails with only a negligible probability. For the second part, there are three aspects to be proved in detail: (i) the stash overflows with a negligible probability of $O(N^{-\log\log N})$, (ii) any partition overflows with a negligible probability of $O(N^{-\log\log N})$, and (iii) any layer of any partition overflows during data shuffling with a negligible probability of $O(N^{-\log N})$.

Lemma 8. *Given that $P \geq \log N \log\log N$, the total number of real blocks in the stash at any time during data queries is upper bounded by $2P(1-2/P)$ with a probability of $1-O(N^{-\log\log N})$.*

Proof. The stash capacity can be computed by summing up the number of blocks in all P slots. Thus, we can focus on the analysis for a single slot. Note that, blocks are loaded to slots in a uniformly random fashion, and are evicted through the background eviction procedure with eviction rate λ (we use $\lambda = 2$ in the following analysis).

For any single slot, a real block enters this slot with probability $p = 1/P$ and leaves with probability $q = 2/P$. Then, the number of real blocks in any slot is a Discrete Time Markov Chain (DTMC) starting with state 0 (no blocks in the slot), each state i (i blocks in the slot) has forward probability $p_f = p(1-q)$ to state $i+1$ and backward probability $p_b = q(1-p)$ to state $i-1$. Since $\rho = p_f/p_b \leq 1/2$. The stationary distribution for each state i is $\pi_i = \rho^i(1-\rho)$ ($i = 0, 1, 2, \cdots$) and the expectation of the stationary distribution is $\rho/(1-\rho)$. Thus, the expected number of real blocks in the entire stash is

$$\chi = P \cdot \frac{\rho}{1-\rho} = P(1 - \frac{2}{P}).$$

Now, let's observe the upper bound on the stash capacity. Suppose Z_i denotes the number of real blocks in slot i ($1 \leq i \leq P$). Then, Z_i's are negatively associated [18] and Z_i's are geometric random variables with parameter ρ as mentioned before. Hence, the upper tail bound for $Z = \sum_{i=1}^{P} Z_i$ [18, 65] is:

$$\mathbf{Pr}[Z \geq \Psi] \leq e^{-\frac{P \cdot (\Psi/\chi - 1)^2}{4}} = N^{-\frac{P}{4 \ln N}} = O(N^{-\log\log N}),$$

where $\Psi = 2\chi$ and $P \geq \log N \log\log N$. Therefore, given $P \geq \log N \log\log N$, the stash size can be bounded by Ψ with overwhelming probability $1 - O(N^{-\log\log N})$. □

Lemma 9. *Given that* $\log N \log\log N \leq P \leq \sqrt{N}$*, the total number of real blocks for any partition at any time during data queries is upper bounded by* $\Phi = 1.1N/P$ *with a probability of* $1 - O(N^{-\log\log N})$.

Proof. For every individual partition, we consider the partition together with its corresponding stash slot as a bin. Thus, the partition capacity can be upper bounded by the bin capacity. Note that, since the snapshot of any moment for the whole system can be seen as a particular distribution of randomly throwing N blocks into P bins, the following results can be deduced on bin capacity:

Consider a particular bin, and define $X_1, \cdots, X_N$ as random variables such that

$$X_i = \begin{cases} 1, & \text{the } i^{\text{th}}\text{real block is mapped to the bin,} \\ 0, & \text{otherwise.} \end{cases}$$

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

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

According to the multiplicative form of Chernoff bound, for any $\Phi \geq E[X] = N/P$, it holds that

$$\mathbf{Pr}[\text{a specific bin has more than } \Phi \text{ real blocks}] < e^{-\frac{N(\Phi P/N-1)^2}{3P}}.$$

By applying the union bound, we can obtain

$$\mathbf{Pr}[\ \exists \text{ any bin with more than } \Phi \text{ real blocks}] < P \cdot e^{-\frac{N(\Phi P/N-1)^2}{3P}} = O(N^{-\log\log N}),$$

where $\Phi = 1.1N/P$ and $\log N \log\log N \leq P \leq \sqrt{N}$. Note that any partition capacity is upper bounded by the bin capacity, it holds immediately that any partition capacity is also upper bounded by Φ with overwhelming probability $1 - O(N^{-\log\log N})$. □

Theorem 3. *GP-ORAM is secure under the security definition in Section 2.2.*

Proof. According to Definition 2.2, we will first show that, given any two equal-length sequence $\vec{x}$ and $\vec{y}$ of private data requests, their corresponding observable access sequences $A(\vec{x})$ and $A(\vec{y})$ are computationally indistinguishable.

Note that, for the k^{th} access $x_k = (op_k, i_k, D_k)$, the observable sequence $A(x_k)$ consists of two parts: $(read, p, \vec{D})$ which is data query; $(write, p', \vec{D})$ which is background eviction.

- First, for data query, x_k (or y_k) introduces a read operation on a random partition p_x (or p_y). Then, the background eviction incurs a sequential of write operations on pre-defined partition p' for both x_k and y_k. Hence, $A(x_k)$ and $A(y_k)$ are computationally indistinguishable with each other, because their first parts follow a uniform random distribution and their second parts are the same to each other.

- Second, accesses to individual ORAM partition are oblivious.
  - The read operation to a selected partition accesses locations from each non-empty layer (except layer 1) randomly and non-repeatedly;
  - When a data block is evicted to this partition, it is re-encrypted and appended to the first layer of this partition.

Also, we have proved GP-ORAM fails with a probability of $O(N^{-\log\log N})$ based on Lemma 8 and Lemma 9. Therefore, it is proved that GP-ORAM construction is secure. □

6.5 Cost Analysis and Evaluations

In this section, we analyze the costs of non-recursive and recursive GP-ORAM constructions, and compare them to P-ORAM [65], Path-ORAM [66] and S-ORAM [77], which are the most communication-efficient state-of-the-art ORAM constructions.

Cost Analysis for Non-recursive GP-ORAM The communication cost includes query and background eviction costs. Each data query retrieves two blocks (i.e., one index block and one data block) from and uploads only the index block to each non-empty layer of the server. As there are $L_2 = \lceil \log(N/P) \rceil$ layers, query cost on average is:

$$C_{\text{query}} < 1.5 \cdot \log(\frac{N}{P}) \cdot B.$$

As for the background eviction cost, after each query, λ blocks are written to λ consecutive partitions at the server. Thus, P/λ queries result in all P partitions being accessed once. Therefore, for each partition, layer l $(1 < l < L_2)$ is involved in a shuffling process every $2 \cdot 2^l \cdot P/\lambda$ queries, while layer L_2 is shuffled every $2^{L_2} \cdot P/\lambda$ queries. Recall that shuffling a T1-layer l involves $2 \cdot 2^l$ blocks, shuffling a T2-layer l involves $4 \cdot 2^l$ blocks, and shuffling layer L_2 involves $5.3 \cdot 2^{L_2}$ blocks. Hence, the amortized shuffling cost is

$$C_{\text{shuffle}} = (\sum_{l=2}^{L_1} \frac{2 \cdot 2^l \cdot P}{2 \cdot 2^l \cdot P/\lambda} + \sum_{l=L_1+1}^{L_2-1} \frac{4 \cdot 2^l \cdot P}{2 \cdot 2^l \cdot P/\lambda} + \frac{4.4 \cdot 2^{L_2} \cdot P}{2^{L_2} \cdot P/\lambda})B,$$

Therefore, the communication cost for non-recursive GP-ORAM is

$$C_{\text{GP-ORAM(NR)}} = C_{\text{query}} + C_{\text{shuffle}} = (1.5 + 2\lambda) \log \frac{N}{P} \cdot B - \lambda(\log\log N - 2.8) \cdot B.$$

For storage cost, as stated in Lemmas 8 and 9, the user needs to maintain the following amount of storage space:

$$2P(1-\frac{2}{P})B + S + N \cdot (\log N + \log\log \frac{1.1N}{P}),$$

where $P \geq \log N \log\log N$. The size of the stash is $2P(1-2/P)B$, the size of the shuffling buffer is S, and the size of the index table is $N \cdot (\log N + \log\log \frac{1.1N}{P})$, respectively. Note that, the shuffling buffer storage is temporary, while the stash and index table spaces are permanently needed. For server storage, each partition contains at most $5.3N/P$ blocks. Thus, the server storage is less than $5.3NB$.

Cost Analysis for Recursive GP-ORAM Suppose there are ϕ levels of recursion in the recursive construction, and the i^{th} level of recursion is implemented by GP-ORAM$_i$. Thus, GP-ORAM$_1$, which is used to store the user's data blocks, requires a stash of size $2P(1-2/P)B$ and a shuffling buffer of size S in the user's local storage, while the index table is exported to the server as GP-ORAM$_2$. The compression rate for GP-ORAM$_2$ can be smaller than 2^{-13} (i.e., the size of GP-ORAM$_2$ can be less than $\frac{1}{2^{13}}$ of that of GP-ORAM$_1$) when $N \leq 2^{44}$ and $B \geq 64\ KB$, which covers the practical scenarios considered by Stefanov et. al. [65]. Therefore, parameter ϕ is no more than 4; that is, no more than 4 levels of recursion are needed in practice.

Since GP-ORAM$_1$ has much larger capacity than other GP-ORAMs, the extra communication cost introduced by recursion can be computed as $O(\sum_{i=1}^{\phi} \log(\alpha^{-i}N) \cdot B)$ in practice. For the extra local storage cost, it mainly comes from the stashes for extra GP-ORAMs (note that the shuffling buffer for GP-ORAM$_1$ can be reused for other smaller GP-ORAMs), and the total size of these stashes is much less than that for GP-ORAM$_1$. Specifically, a stash of size $3P(1-2/P)B$ is enough for recursive constructions. At last, the extra cost on server storage is $O(\sum_{i=1}^{\phi} \alpha^{-i}N \cdot B)$.

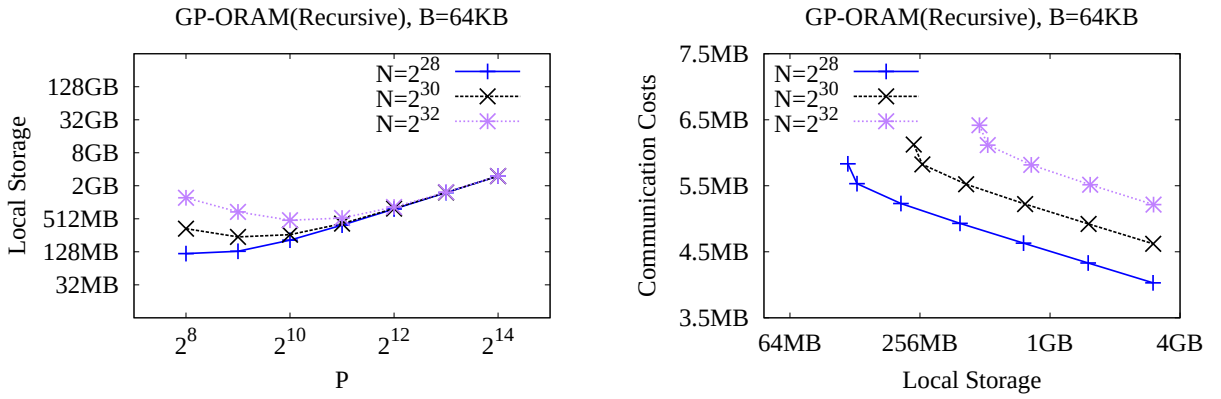
Tradeoff between Local Storage Capacity and Communication Cost in GP-ORAM Suppose a user exports N data blocks each of B bits, and the local storage capacity is $\mathcal{S}_l$. The user could find an optimal P (i.e., number of partitions) for GP-ORAM to minimize the communication cost.

According to $C_{\text{GP-ORAM(NR)}}$ in the non-recursive GP-ORAM cost analysis, the larger is P, the smaller is the communication cost. Hence, the optimal P should be the largest P without incurring a local storage cost higher than $\mathcal{S}_l$. Formally:

$$\begin{aligned}
&\text{Maximize} \quad P, \\
&\text{subject to} \quad 2P(1-\frac{2}{P})B + S + \frac{NB}{\alpha} \leq \mathcal{S}_l \quad \text{for non-recursive GP-ORAM}, \\
&\text{subject to} \quad 3P(1-\frac{2}{P})B + S \leq \mathcal{S}_l \quad \text{for recursive GP-ORAM}.
\end{aligned}$$

The example plotted in Figure 6.3(a) shows the relation between P and local storage consumption in the recursive GP-ORAM. Recall that, the local storage includes shuffling buffer and stash. As we can see from Figure 6.3(a), when P is small, local storage consumption decreases as P increases; when P becomes large, local storage consumption increases as P increases. This phenomenon can be explained as follows.

- When P is small, the size of each partition is large; hence, the shuffling buffer dominates the local storage. As P increases, shuffling buffer decreases which causes the local storage to decrease as well.
- When P is large, the number of partitions gets large and so the stashes dominates the local storage. As P increases, the size of stashes increases which causes the local storage to increase too.



(a) P vs. Local storage consumption

(b) Local storage capacity vs. Minimal communication cost

Figure 6.3 Examples illustrating the relation between P, local storage, and minimal communication cost.

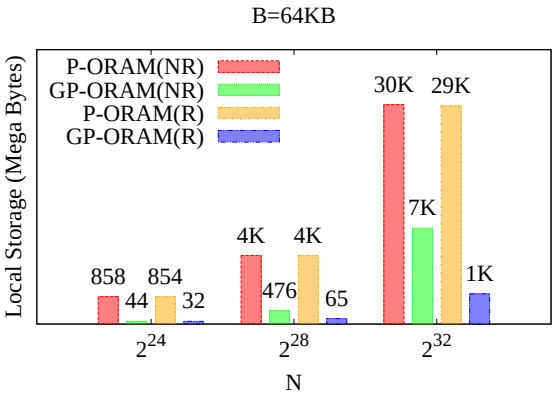
Based on the relation plotted in Figure 6.3(a), the user can find a range of P, with which the required local storage does not exceed $\mathcal{S}_l$. Because the communication cost decreases as P increases, the maximum P within the range becomes the optimal P that minimizes the communication cost. This way, for any given $\mathcal{S}_l$, the communication cost corresponding to the optimal P can be found. Figure 6.3(b) plots an example to illustrate the relation between local storage capacity and minimal communication cost in the recursive GP-ORAM.

GP-ORAM VS. P-ORAM Table 6.1 compares GP-ORAM with P-ORAM in terms of asymptotical performance. From the table, we have the following observations: (i) When P is set to N^c ($c < 0.5$) and S is set as in Equation (6.1), the communication costs for both non-recursive and recursive GP-ORAM can be re-written as $O(\log N \cdot B)$, which is comparable to the cost for non-recursive P-ORAM and much lower than that for recursive P-ORAM. (ii) The local storage costs for non-recursive P-ORAM and GP-ORAM are both $O(NB)$, as the costs are dominated by the index table. The local storage cost for recursive GP-ORAM is $O(PB + S)$, which is asymptotically smaller than $O(\sqrt{N}B)$ as $P < \sqrt{N}$.

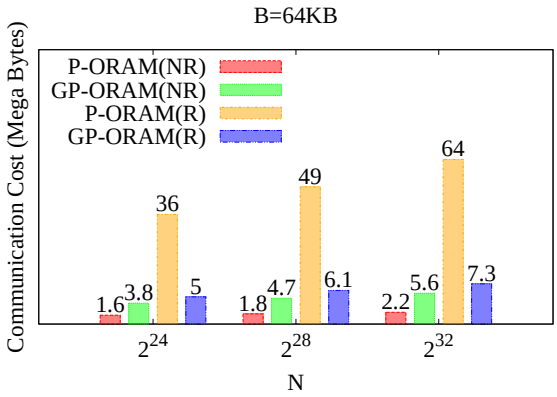
Table 6.1 Asymptotical Performance Comparison.

Scheme	Bandwidth Cost	User Storage	Server Storage	Failure Prob.
P-ORAM (NR)	$O(\log N \cdot B)$	$O(NB)$	$< 4NB$	$O(\frac{1}{N^c})$
P-ORAM (R)	$O(\log^2 N \cdot B)$	$O(\sqrt{N}B)$	$< 8NB$	$O(\frac{1}{N^c})$
GP-ORAM (NR)	$O(\frac{\log^3(N/P)}{\log^2 S} \cdot B)$	$O(NB)$	$< 5.3NB$	$O(N^{-\log\log N})$
GP-ORAM (R)	$O(\frac{\log^3(N/P)}{\log^2 S} \cdot B)$	$O(PB + S)$	$< 5.3NB$	$O(N^{-\log\log N})$

Figures 6.4 and 6.5 compare the performance of GP-ORAM with P-ORAM under the practical system settings used by Stefanov et. al. [65] (i.e., block size ranging from 64 KB to 1 MB; the number of blocks ranging from 2^{24} to 2^{32}). From the figures, we have the following observations: (i) The local storage demanded by recursive GP-ORAM is only 2.5%∼0.14% of that by non-recursive P-ORAM, while GP-ORAM only yields about 1 to 3 times higher communication cost than P-ORAM. (ii) Recursive P-ORAM is impractical due to its extremely high communication cost.

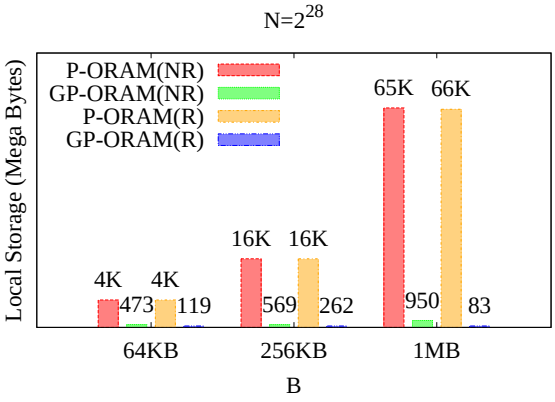


(i) Local Storage Size under Different N

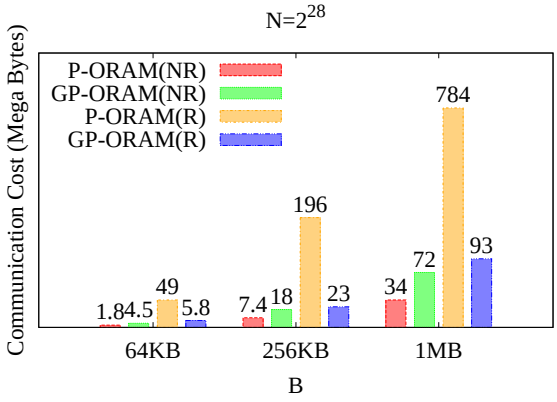


(ii) Bandwidth Cost under Different N

Figure 6.4 Comparing local storage and communication cost when $B = 64$ KB.



(i) Local Storage Size under Different Block Size



(ii) Bandwidth Cost under Different Block Size

Figure 6.5 Comparing local storage and communication cost when $N = 2^{28}$.

Comparing GP-ORAM, Path-ORAM and S-ORAM Table 6.2 shows the asymptotical performance comparisons between GP-ORAM, Path-ORAM and S-ORAM. Compared to S-ORAM and Path-ORAM, GP-ORAM introduces one adjustable system parameter P, which makes it more tunable.

The performance comparison between GP-ORAM and Path-ORAM under practical scenarios [65] is shown in Table 6.3. From the table, it can be seen that GP-ORAM can fully utilize the local storage to achieve better communication efficiency, and it incurs lower server-side storage cost.

Figure 6.6 shows the performance comparison between GP-ORAM and S-ORAM under

Table 6.2 Asymptotical Performance Comparison.

Scheme	Bandwidth Cost	User Storage	Server Storage	Failure Prob.
S-ORAM	$O(\frac{\log^3 N}{\log^2 S} \cdot B)$	$O(S)$	$< 6NB$	$O(N^{-\log N})$
Path-ORAM (NR)	$O(\log N \cdot B)$	$O(NB)$	$10NB$	$N^{-\omega(1)}$
Path-ORAM (R)	$O(\log^2 N \cdot B)$	$O(\log N \cdot B) \cdot \omega(1)$	$> 10NB$	$N^{-\omega(1)}$
GP-ORAM (NR)	$O(\frac{\log^3(N/P)}{\log^2 S} \cdot B)$	$O(NB)$	$< 5.3NB$	$O(N^{-\log\log N})$
GP-ORAM (R)	$O(\frac{\log^3(N/P)}{\log^2 S} \cdot B)$	$O(PB + S)$	$< 5.3NB$	$O(N^{-\log\log N})$

Table 6.3 Practical Performance Comparison.

Scheme	Bandwidth Cost	User Storage	Server Storage
Path-ORAM (NR)	$10 \log N \cdot B$	$N \log N + \log N \cdot B \cdot \omega(1)$	$10NB$
Path-ORAM (R)	$10 \log^2 N \cdot B$	$\log N \cdot B \cdot \omega(1)$	$20NB$
GP-ORAM (NR)	$< 4 \log N \cdot B$	$N \log N + PB + S$	$< 5.3NB$
GP-ORAM (R)	$< 6 \log N \cdot B$	$PB + S$	$< 5.3NB$

practical scenarios [65]. From the figure, we can see that S-ORAM is not fully tunable as local storage increases. Especially when the local storage is large enough, the communication cost cannot be further reduced. For example, when $N = 2^{32}$, $B = 64$KB and the local storage size has exceeded 1.2 GB, the communication remains the same regardless of the increase in local storage size, while GP-ORAM can achieve 50%-60% savings in communication cost as the local storage gets larger.

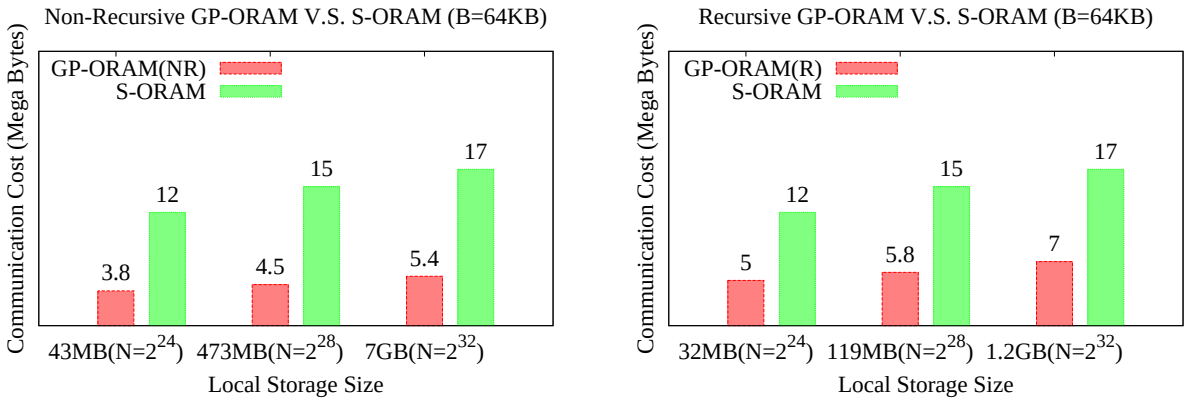


Figure 6.6 GP-ORAM vs. S-ORAM with same given local storage.

6.6 Summary

In the third work, we proposed a new ORAM construction, called Generalized Partition ORAM (GP-ORAM). GP-ORAM utilizes a new shuffling method, adjusts the number of partitions according to the available user-side local storage, and outsources the index table to the server. Through these techniques, it achieves low bandwidth cost ($O(\log N)$) and has significantly less user-side storage cost than P-ORAM. We demonstrate the effectiveness of GP-ORAM via extensive security and cost analysis.

CHAPTER 7. MU-ORAM: DEALING WITH STEALTHY PRIVACY ATTACKS IN MULTI-USER DATA OUTSOURCING SERVICES

Most of existing ORAM constructions assume only a single user to interact with the storage server; therefore, the user's device holds all the system secrets about how the outsourced data are encrypted, placed and scrambled in the server's storage. As it is also popular for multiple users to share outsourced data, such constructions [47, 22, 32, 73] have been proposed to extend the single-user ORAM to support parallel accesses from multiple users. In these proposals, however, the users essentially work together as a single user, because either the users need to go through a single proxy which holds the system secrets and interacts with the server on behalf of all users, or each of the users should hold the same system secrets and interacts with the server directly. In either case, the single proxy or any one user becomes a single point of security failure. If it is malicious or compromised, attacks can be launched from the inside, and the security of the whole system can be easily brought down. Observable attacks (e.g., illegitimate deletion or modification of data) launched by the insider attacker can be detected, and the attacker can be identified with some accountability mechanisms (e.g., auditing the logs), but *detecting stealthy attacks targeted at privacy is much more difficult.* A curious or compromised user can collude with the storage server (if the server is also curious or compromised) to reveal the access patterns of all other users; meanwhile, the attackers can keep their attacks stealthy, because they still follow the ORAM protocols without extending any anomaly observable by others.

For instance, a hospital may wish to export the encrypted information of all its patients, to a remote storage organized as an ORAM. To allow each doctor to access the data of any patient who has visited the hospital, all the doctors should share the same secret keys. With such a system, if a doctor is curious or the account of a doctor is compromised by an attacker,

the adversary (i.e., the curious doctor or the attacker) may be able to observe the accesses made by all other doctors, through colluding with the storage server which is also curious or compromised, without launching any observable attack to the ORAM.

In the fourth work, we study the feasibility and cost of overcoming the above limitation of existing ORAM constructions, we propose, design, and analyze a new ORAM construction called *Multi-User ORAM (MU-ORAM)* [79]. The construction has two design goals. First, it shall support multiple users to share data outsourced to a remote storage. Second, it shall be resilient to the afore-described *stealthy privacy* attacks, in which the curious or compromised insider attackers do not extend observable misbehaviors, but collude stealthily to reveal the data access patterns of innocent users. To the best of our knowledge, this is the first effort aiming to attain these goals.

To tolerate stealthy privacy attacks, the basic principle is to distribute the shares of the system secrets among the users, instead of letting every user to hold all the system secrets. This way, any single user alone will not have sufficient secrets to locate and decrypt a data block of interest to access; collaboration between the users is required. However, when a user needs to access a data block, it is not realistic to require other users to be online and available for collaboration. Hence, the key idea in our design is to introduce a chain of collaborative but mutually independent *proxies* between users and the storage server. These proxies are always online, like the storage server. The shares of the system secrets are distributed delicately to the proxies and the users. When a user needs to query a data block, its request and the storage server's replies shall pass through and be processed by the proxies before they reach the destination.

In practice, the proxies can be implemented as mutually independent hardware components (e.g., computers) or software components (e.g., virtual machines) provided in public or private domains. For instance, in the afore-mentioned "hospital" example, the proxies can be implemented as several physical/virtual machines running in the premise of the hospital or some cloud providers independent of the remote storage server.

Within this architecture, (i) users do not need to hold all the system secrets as they do not interact directly with the storage server; (ii) each user can set up a secure and logically isolated

communication channel with the chain of proxies; (iii) multiple proxies, with each holding an independent share of the system secrets, work together to act as a common interface between users and the storage server. They also take non-user-specific workload (e.g., data shuffling). Due to the above features, users are securely isolated from each other, and compromising some but not all proxies cannot capture the system secrets. Thus, the system becomes more resilient to the stealthy privacy attacks.

We propose formal security definitions to quantify the security strength of MU-ORAM in protecting an innocent user's data access patterns against stealthy attacks, and conduct extensive analysis:

- First, we have shown that, like existing single-user ORAMs, MU-ORAM can fully protect the access pattern privacy of each individual user against an semi-honest storage server with a failure probability of $O(N^{-\log\log N})$, where N is the total number of exported data blocks.

- Second, assuming that the server, some users and some but not all proxies are semi-honest and colluding, we study the security strength of MU-ORAM under different scenarios. Particularly, we have shown that, the collusive coalition has an advantage of less than 2ϵ within time period t to reveal an innocent user's access to data that the coalition is not authorized to access,

  if the Modified Matching Diffie-Hellman (MMDH) problem G_p cannot be solved with an advantage of at least ϵ within the same time period t.

Note that, as our design aims at dealing with stealthy privacy attacks, the threat model of our security analysis assumes that the attackers are semi-honest (i.e., the attackers honestly follow the protocols that they are expected to execute, but may take extra actions to reveal the data access patterns of innocent users).

Cost analysis has been conducted to quantify the costs incurred to provide the protection. The results show that, the communication cost introduced by MU-ORAM is $O(\log^2 N)$ data blocks per query for the user and $O(\log^2 N \log\log N)$ for the proxies. Meanwhile, MU-ORAM does not store any dummy data blocks, which makes the server-side storage to be $O(N)$.

7.1 Preliminaries

This section presents the system model, the architecture of our proposed MU-ORAM, and the formal definitions of security.

7.1.1 System Model

We consider a system where multiple users share N data blocks exported to a storage server. Let F_p be a finite field with p distinct elements, where p is a prime number and $N \ll p$. For example, $\log p$ is usually 128 or larger, while in practice $\log N$ is seldom greater than 40. Let G_p be a multiplicative, cyclic group with also p distinct elements. Each data block, denoted as D_i, consists of two components: (i) unique data ID denoted as g_i which is an element of G_p; (ii) data content that is a sequence of pieces each being an element of G_p. As the operations on each piece of the data content are the same, we use a single element denoted as d_i to represent the sequence unless stated otherwise. Hereafter, each data block D_i is represented as

$$(g_i,\ d_i) \text{ where } g_i \in G_p \text{ and } d_i \in G_p. \tag{7.1}$$

Each data request from a user, which shall be kept confidential, is one of the following two types: (i) read a data block d_i of unique ID g_i from the storage, denoted as a 3-tuple $(read, g_i, d_i)$; or (ii) write/modify a data block d_i of unique ID g_i to the storage, denoted as a 3-tuple $(write, g_i, d_i)$.

To accomplish a confidential data request, the user may need to access the remote storage multiple times. Each access to the remote storage can be observed by the server and its collusive coalition, and is one of the following two types: (i) retrieve (i.e., read) a data block d_i from a location l at the remote storage, denoted as a 3-tuple $(read, l, d_i)$; or (ii) upload (i.e., write) a data block d_i to a location l at the remote storage, denoted as a 3-tuple $(write, l, d_i)$.

Also, we assume there is a trusted system initialization server. This server is not involved in data access, but only responsible for initializing the system and providing public information for a user when the user joins the system. Note that, once the system initialization finishes, all system secrets are removed from this server. Therefore, we assume the server is immune from attacks.

7.1.2 Proposed Architecture

MU-ORAM is designed to protect the data access patterns of individual users against stealthy privacy attacks launched by collusive parties in the system. To attain this goal, we propose a new architecture (as shown in Figure 7.1) composed of a hierarchical storage server, multiple users, and a chain of *proxies* as a bridge between users and the storage server. In practice, proxies can be implemented as mutually independent hardware components (e.g., computers) or software components (e.g., virtual servers). These proxies can be deployed in the premise of the users or some cloud providers independent of the provider of the storage server.

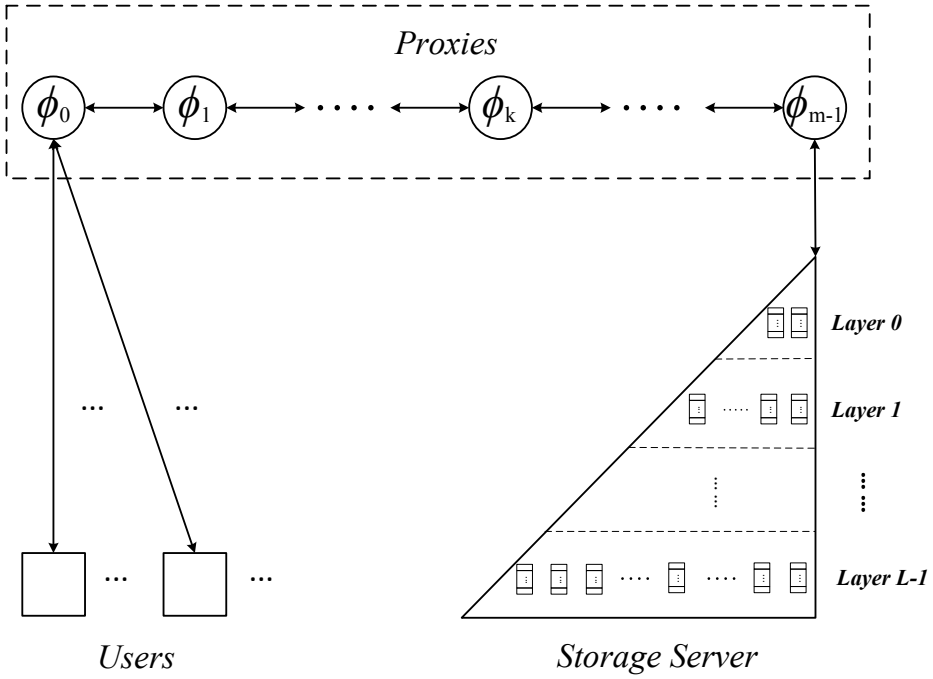


Figure 7.1 System overview.

Specifically, the introduced chain of proxies serves as a common interface for all users to access data at the storage server as follows.

- When a user needs to access a certain data block, the request and the data replied from the storage server shall pass through and be processed (i.e., encrypted or decrypted) by all the proxies before they reach either the storage server or the user.

- By introducing proxies to protect users from direct interactions with the storage server,

each individual user does not need to maintain the information about storage locations or encryption keys of the data shared with other users. Without exposing such knowledge to individual users, it becomes possible to prevent a user from learning other users' data access patterns through colluding with the storage server or observing their interactions with the storage server.

- Such an architecture also allows each user to establish a secure and logically isolated communication channel with the chain of proxies, which makes it possible to prevent a user from learning other users' data access patterns through observing their interactions with the proxies.

- As all the user/server interactions must go through the entire chain of independent proxies, the user's access pattern privacy is protected, as long as not all of the proxies are compromised and collude with the storage server.

Under this proposed architecture, appropriate algorithms must be designed to guide the interactions between the storage server, proxies, and users. We will present these algorithms in Section 7.2.

7.1.3 Security Definitions

As the major goal of our design is to protect individual users' access pattern privacy from stealthy attacks, we assume the storage server, users, and proxies in the system are *honest but curious* or called *semi-honest*. Specifically:

- In response to a data query from a user, the user, the proxies and the storage server follow the query protocol honestly to process the query.

- At the time when data shuffling shall be conducted, we assume that the storage server and the proxies all follow the shuffling protocol honestly to shuffle the data.

- The storage server, each proxy, and each user may be curious to find out the access pattern of other users. To do so, they may collude. However, we assume no collusive coalition will include all proxies.

7.1.3.1 Security against semi-honest storage server

As a baseline, we first consider the scenario that the storage server does not collude with any user or proxy. Following the security definition of ORAMs [27, 66, 65], we define the security of an MU-ORAM against an honest but curious storage server as follows.

Definition 1. (*Security against semi-honest storage server*). Let $\vec{x} = \langle\ (op_1, i_1, d_1), (op_2, i_2, d_2), \cdots\ \rangle$ denote a private sequence of a user's intended data requests, where each *op* is either a *read* or *write* operation. Let $A(\vec{x}) = \langle\ (op'_1, l_1, d'_1), (op'_2, l_2, d'_2), \cdots\ \rangle$ denote the sequence of the user's accesses to the remote storage (observed by the server), in order to accomplish the user's private data requests. MU-ORAM 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 MU-ORAM fails to operate is $O(N^{-\log\log N})$.

7.1.3.2 Security against collusive coalition

Next, we consider the more general scenario that the storage server colludes with some users and some (but not all) proxies.

Depending on whether the collusive users have authorized access to the data blocks queried by an innocent user, the security strength of MU-ORAM can be very different. This is because, when the collusive users have access to the data accessed by the innocent user, the collusive users can check if some data blocks have been changed after the innocent user's access to infer the innocent user's access pattern; however, this approach cannot be applied when the collusive users are not authorized to access the data accessed by the innocent user. Hence, we study two cases separately as follows.

Case 1: Users with same access privileges to data In a system where users have the same access privileges to outsourced data, we study the security strength of MU-ORAM in protecting an innocent user's access pattern to the data that can also be accessed by the collusive users.

To facilitate the study, we define a game between an adversary (i.e., the collusive coalition) and a challenger (i.e., the rest of the system) in the following. Intuitively, the game models the attacks that can be launched by the adversary: it can launch queries and observe how these queries are handled; it can observe the interactions between the innocent user and the server and proxies; it can compromise and thus obtain the secrets of some but not all proxies; it can inspect data stored on the storage server. The adversary is said to have won the game (i.e., defeated the MU-ORAM) if the following happens: the innocent user first selects two data blocks uniformly at random to query; the user is then asked to randomly choose one of these two data blocks to query again; the adversary is able to find out the user's choice.

Definition 2. A game $\mathcal{G}_1(\mathcal{M}, p, N, m, n_{cq})$ between a *challenger* and an *adversary* is defined as follows (Here, $\mathcal{M}$ denotes an MU-ORAM construction):

- *Initialization Phase.* The challenger initializes the storage server and the chain of m proxies, according to the algorithm of $\mathcal{M}$. Here, N data blocks $\{(g_i, d_i)|i = 0, \cdots, N-1; g_i \in G_p; d_i \in G_p\}$ are exported to the storage server. The adversary has access to all the data block IDs.

- *Query Phase I.* The adversary can make any number of queries of the following types.

  - *Proxy Compromising.* The adversary requests to get the information (e.g., secrets) owned by any compromised proxy. We restrict that at most $m-1$ proxies can be compromised.

  - *Proxy and Server Transcript Inspection.* The adversary requests to get the input/output of any compromised proxy and the storage server.

  - *Data Query.* Two types of queries can be requested:

    * Type I (controlled queries) - The adversary selects an ID and acts as a user to start querying the data block of this ID. In response, if the number of Type I query has exceeds n_{cq}, the request is denied; otherwise, the proxies and the server follow the $\mathcal{M}$ protocol to process the query request.

    * Type II (random queries) - The adversary requests an innocent user to start a query. In response, the challenger secretly selects an ID from the pool of IDs uniformly at random, and then acts as a user to start querying the data block of this ID. The proxies and the server follow the $\mathcal{M}$ protocol to process the query. Note that, this selected ID is unknown to the adversary.

  – *Storage Inspection.* The adversary asks the storage server to return the data blocks in a specified bucket.

- *Selection Phase I.* The challenger secretly selects a data block ID denoted as θ_0 from the pool of IDs uniformly at random, and queries it. Note that, θ_0 is known only by the challenger.

- *Query Phase II.* The phase is the same as Query Phase I, except that the following rule should be added when processing a Type I data query: the challenger aborts the game and declares failure if the queried data ID is θ_0, θ_0 was queried by the adversary before the Selection Phase I, and there is no Type II query for θ_0 between the adversary's last and current query for θ_0. This is because, when the above conditions are satisfied, the adversary will find that the content of data block θ_0 was changed after the Selection Phase I, and thus find θ_0 was queried in the Selection Phase I; therefore, it will know which of θ_0 and θ_1 is selected in the later Challenge Phase by simply querying θ_0 right after the Challenge Phase.

- *Selection Phase II.* The challenger secretly selects another data block ID denoted as θ_1 ($\theta_0 \neq \theta_1$), and queries it.

- *Query Phase III.* The phase is the same as Query Phase I, except that the following rule should be added when processing a Type I data query: the challenger aborts the game and declares failure if either (i) the queried data ID is θ_0, θ_0 was queried by the adversary before the Selection Phase I, and there is no Type II query for θ_0 between the adversary's last and current query for θ_0; or (ii) the queried data ID is θ_1, θ_1 was queried by the adversary before the Selection Phase II, and there is no Type II query for θ_1 between

the adversary's last and current query for θ_1. This change is due to the same reason explained in Query Phase II.

- *Challenge Phase.* The challenger decides a binary bit b uniformly at random. Then, it queries the data block of ID θ_b.

- *Query Phase IV.* The phase is the same as Query Phase I, except that the following rule should be added when processing a Type I query: if θ_0 or θ_1 is queried, the challenger aborts the game and declares failure. Note that, the adversary may or may not find out the query target chosen in the Selection Phases if it requests to query θ_0 or θ_1. Hence, by this rule we may under-estimate the security strength of MU-ORAM.

- *Response Phase.* The adversary returns a binary bit b' as a guess of the b.

- *Result.* The adversary wins the game if the challenger declares failure or $b' = b$; otherwise, it loses the game. The advantage for the adversary to win the game is defined as the probability that it wins the game minus 1/2.

An MU-ORAM construction $\mathcal{M}$ is considered secure against a collusive coalition, if it is *hard* for an adversary with limited computational capability to win the above game. To quantify this notation, we introduce the following definition:

Definition 3. *((ϵ, t, n_{cq})-security against collusive coalition)* An MU-ORAM construction $\mathcal{M}$, in which all users have the same access privileges to the outsourced data, is said to be (ϵ, t, n_{cq})-secure against a collusive coalition of semi-honest storage server, users and some (but not all) proxies if: no adversary can win the game $\mathcal{G}_1(\mathcal{M}, p, N, m, n_{cq})$ with an advantage of at least ϵ under the time complexity of t and the restriction that the adversary cannot make more than n_{cq} Type I data queries (i.e., controlled queries) during the game.

Case 2: Users with different access privileges to data In a system where users have different access privileges to data, we study the security strength of MU-ORAM in protecting an innocent user's access pattern to the data blocks that cannot be accessed by the collusive users. In the following, we present new game and security definitions.

Definition 4. A game $\mathcal{G}_2(\mathcal{M}, p, N, N', m)$ between a *challenger* and an *adversary* is defined similarly to $\mathcal{G}_1$ (in Definition 2) except for the following differences:

- In the Initialization Phase: the adversary is given only N' IDs from the totally N IDs.
- In the Query Phases I, II and III: there is no limitation on the number of Type I data queries that the adversary can make.
- In the Selection Phase I and II: θ_0 and θ_1 are two distinct IDs selected uniformly at random from the set of IDs that are unknown to the adversary.

To quantify the security strength of MU-ORAM in Case 2, we introduce the following definition:

Definition 5. *((ϵ, t)-security against collusive coalition)* An MU-ORAM construction $\mathcal{M}$, in which users have different access privileges to the outsourced data, is said to be (ϵ, t)-secure in protecting an innocent user's access to the data that a collusive coalition of semi-honest storage server, users and some (but not all) proxies are not authorized to access, if no adversary can win the game $\mathcal{G}_2(\ \mathcal{M},\ p,\ N,\ N',\ m)$, where $N' \leq N - 2$, with an advantage of at least ϵ within time period t.

7.2 Scheme

This section elaborates our proposed MU-ORAM design, which includes storage structure, system initialization, data query, and data shuffling. Figure 7.2 illustrates the overall workflow of data query and shuffling.

7.2.1 Storage Structure

MU-ORAM server organizes its storage as a hierarchy of buckets, and each bucket can store up to $\log N$ data blocks:

- The hierarchy consists of $L = \lceil \log N - \log \log N \rceil$ layers.

- Each layer l ($l = 0, \cdots, L-1$) has $n_l = 2^{l+1} \cdot \log N$ buckets. Hence, the top layer of the hierarchy (i.e., layer 0) has $2 \log N$ buckets, while the bottom layer of the hierarchy (i.e., layer $L-1$) has N buckets.

- Each layer l is associated with a public hash function, denoted as $H_l(*)$, which maps each element of group G_p to one bucket at layer l.

- Each layer l has a bitmap to record whether each bucket at this layer is empty or not.

Note that, in MU-ORAM, there is no dummy data in its storage.

7.2.2 System Initialization

A trusted authority, which we call system initialization server, is responsible for initializing the system. The system initialization includes *proxy initialization*, *storage initialization*, and *user initialization*.

- The initialization server first picks z from $F_p \setminus \{0\}$ uniformly at random.

- Suppose there are m proxies, denoted as $\phi_0, \cdots, \phi_{m-1}$ in the system. For each ϕ_k ($k = 0, \cdots, m-1$), it is preloaded by the initialization server with the following keys: $x_k(l)$, $y_k(l)$ and $\Delta z_k(l)$ for each layer $l \in \{0, \cdots, L-1\}$, which are randomly picked from $F_p \setminus \{0\}$. These keys are used for encrypting data block IDs and contents. To facilitate presentation, we introduce the following notations:

$$x(l) = \prod_{k=0}^{m-1} x_k(l); \quad y(l) = \prod_{k=0}^{m-1} y_k(l); \quad \Delta z(l) = \prod_{k=0}^{m-1} \Delta z_k(l); \quad z(l) = z + \Delta z(l). \tag{7.2}$$

- The initialization server exports all the N data blocks to the bottom layer (i.e., Layer $L-1$) as follows: for each data block (g_i, d_i), it is encrypted to $(g_i^{x(L-1)}, (g_i^{-z(L-1)} d_i)^{y(L-1)})$, and stored to bucket $H_{L-1}(g_i^{x(L-1)})$.

- For each user, when s/he joins the system, the initialization server preloads to him/her the public hash function $H_l(*)$ for each layer $l \in \{0, \cdots, L-1\}$. For each data block D_i that this user is authorized to access, the user is preloaded with tuple $(g_i, g_i' = g_i^{-z})$, where g_i is the ID of the data block D_i.

7.2.3 Data Query

When a user wants to query the data block of ID g_i from the storage server, she first needs to randomly select another ID denoted as g_j and also query the data block of ID g_j. Then, for each of the IDs g_i and g_j, the *data request*, *data reply* and *data uploading* phases shall be run sequentially for each of the non-empty layers from the top to the bottom of the storage hierarchy. As the processes for querying g_i and g_j are similar, in the following we only present how these phases are executed for non-empty layer l when g_i is queried.

7.2.3.1 Phase 1: Data Request

In this phase, the user determines a bucket on layer l and sends a request to retrieve data blocks from the bucket. The phase includes the following steps.

[Q1: Obtain Encrypted ID of the Query Target Data] The goal of this step is to compute the encrypted ID of the query target data block. As MU-ORAM uses the product of all proxies' secret keys as the encryption key, [Q1] requires a collaboration between the user and proxies, as shown in Figure 7.3. It consists of two sub-steps as follows.

[Q1.1] In the first sub-step, the user sends the following message to proxy ϕ_0:

$$\left\langle g_i^{r_0}, g_i^{r_0}, (g_i')^{r_1} \right\rangle,$$

where r_0 and r_1 are two nonces randomly picked from $F_p \setminus \{0\}$.

[Q1.2] Upon receiving the message, each proxy ϕ_k $(k = 0, \cdots, m-2)$ updates it and forwards to ϕ_{k+1}:

$$\left\langle (g_i^{r_0})^{\prod_{t=0}^{k} x_t(l)}, (g_i^{r_0})^{\prod_{t=0}^{k} \Delta z_t(l) y_t(l)}, \left((g_i')^{r_1}\right)^{\prod_{t=0}^{k} y_t(l)} \right\rangle.$$

Note that $x_k(l), y_k(l)$ and $\Delta z_k(l)$ are secrets preloaded to ϕ_k. After the message has traversed the entire proxy chain, it becomes

$$\left\langle (g_i^{r_0})^{\prod_{t=0}^{m-1} x_t(l)}, (g_i^{r_0})^{\prod_{t=0}^{m-1} \Delta z_t(l) y_t(l)}, \left((g_i')^{r_1}\right)^{\prod_{t=0}^{m-1} y_t(l)} \right\rangle = \left\langle g_i^{r_0 x(l)}, g_i^{r_0 \Delta z(l) y(l)}, (g_i')^{r_1 y(l)} \right\rangle,$$

according to Equation (7.2). Then, the message is returned to the user by ϕ_{m-1}. Upon receiving the message, the user can obtain

$$g_i^{x(l)} = \left(g_i^{r_0 x(l)}\right)^{1/r_0}$$

and

$$g_i^{y(l)z(l)} = \left(g_i^{r_0 \Delta z(l) y(l)}\right)^{1/r_0} \cdot \left((g_i')^{r_1 y(l)}\right)^{1/r_1},$$

respectively, as r_0 and r_1 are its self-generated nonces. Note that, $g_i^{x(l)}$ is the ID of the query target data block encrypted with the product of all proxies' secret keys, which will be used in [Q2]. $g_i^{y(l)z(l)}$ is stored locally at the user and will be used in [Q5: Data Reply].

[Q2: Compute Bucket for Access] Based on $g_i^{x(l)}$, the user computes the position *pos* of the bucket that may contain the target data block:

$$pos \leftarrow H_l\left(g_i^{x(l)}\right).$$

[Q3: Bitmap Retrieval] This step is to retrieve the bitmap that will be used by the user to decide the buckets to request. This is to avoid the situation where the user may attempt to retrieve an empty bucket at layer l; if this happens, the server would know for sure that D_t is not at this layer, thus leaking the information about D_t.

[Q4: Bucket Request] The user selects the bucket to request based on the retrieved bitmap as follows:

- If D_i has already been found at layer $l' < l$, the user randomly picks a non-empty bucket according to the bitmap.
- Otherwise, the user checks if the bucket at position *pos* is empty or not. If it is empty, the user randomly picks a non-empty bucket to access; else, the user accesses bucket *pos*.

Note that in [Q3], the user needs to retrieve a bitmap of $2^l \log N$ bits; when l is large, it is infeasible for the user to do so. To deal with this issue, the bitmap can be stored in a recursive manner. For example, suppose there are α bits in the bitmap. The server can create two bitmaps instead of one. In the first bitmap, it stores α bits and each bit indicates whether the corresponding bucket is empty or not. The second bitmap stores $\sqrt{\alpha}$ bits and each bit i ($0 \leq i \leq \sqrt{\alpha} - 1$) is set to 0 if all buckets from $\sqrt{\alpha} \cdot i$ to $\sqrt{\alpha} \cdot (i+1) - 1$ are empty. This way, [Q4] becomes:

- If D_i has already been found at layer $l' < l$, the user first requests the second bitmap of $\sqrt{\alpha}$ bits. According to the retrieved bitmap, the user randomly selects a "1" bit, say,

at position P. Then, the corresponding segment indicated by P in the first bitmap is retrieved. At last, the user randomly picks a non-empty bucket from the segment.

- Otherwise, the user downloads the second bitmap and checks if the bit of position $P' = \lfloor \frac{pos}{\sqrt{\alpha}} \rfloor$ is 1. If it is 0, the user randomly picks a "1" bit (say, at position P) from the second bitmap; else, let $P = P'$. Next, the user retrieves the segment from the first bitmap that corresponds to P: (1) if bucket at position pos is not empty, it is selected; (2) otherwise, a non-empty bucket is randomly selected.

This way, the communication cost is reduced to $O(\sqrt{\alpha})$ bits. Indeed, the communication cost can be reduced further with more recursive levels introduced in the bitmap.

7.2.3.2 Phase 2: Data Reply

In response to the bucket request from the user, the storage server returns all the data blocks at the requested buckets to the user in two sub-steps: [Q5.1: From Server to User] and [Q5.2: From User to Proxies and back to User], as shown in Figure 7.4.

[Q5.1] The storage server returns all encrypted data blocks in the requested buckets to the user. Each data block has the following format:

$$\left(g_{i'}^{x(l)}, (g_{i'}^{-z(l)} d_{i'})^{y(l)}\right).$$

If $g_{i'}^{x(l)} = g_i^{x(l)}$ for a data block, it is the target data block. In this case, the data content part $\hat{d} = d_i^{y(l)}$ is encrypted by multiplying $(g_{i'}^{-z(l)} d_{i'})^{y(l)}$ with $g_i^{y(l)z(l)}$ obtained in step [Q1.2]; then the following step [Q5.2] is executed to decrypt $\hat{d}$ and obtain d_i.

Otherwise (i.e., none of the returned data blocks is the query target), the user randomly selects $\hat{d}$ from G_p and then starts step [Q5.2] to also pretend the decryption process.

[Q5.2] The user randomly picks r_2 from $F_p \setminus \{0\}$, and sends $\hat{d}^{r_2}$ to proxy ϕ_0. Then, each proxy ϕ_k $(k = 0, \cdots, m-2)$ updates it and forwards to ϕ_{k+1}:

$$\hat{d}^{\frac{r_2}{\Pi_{t=0}^{k} y_t(l)}}.$$

After the message has traversed the entire proxy train, it becomes

$$\hat{d}^{\frac{r_2}{\prod_{t=0}^{m-1} y_t(l)}} = \hat{d}^{\frac{r_2}{y(l)}},$$

and is then returned to the user.

If $\hat{d} = d_i^{y(l)}$, the returned message is $d_i^{r_2}$ and the user can obtain d_i and access it. Otherwise, the returned message is simply discarded.

7.2.3.3 Phase 3: Data Uploading

In Phase 2, one bucket is downloaded from each non-empty layer of the storage server. After data access, only one data block from each bucket, which must include the query target, need to be uploaded to the shuffling buffer, while other downloaded data blocks are discarded. The storage server updates the corresponding buckets and the bit map to reflect the changes. Note that, the content of the query target data block may have been changed after access. For simplicity, the following description will still use d_i to denote each data block. The *data uploading* phase uploads each of the selected data blocks, denoted as $(g_i^{x(l)}, (g_i^{-z(l)} d_i)^{y(l)})$, to a temporary buffer at the storage server as follows.

Each proxy ϕ_k $(k = 0, \cdots, m-1)$ picks x_k^{temp}, y_k^{temp} and Δz_k^{temp} randomly from $F_p \setminus \{0\}$. We introduce x^{temp}, y^{temp}, Δz^{temp} and z^{temp} as follows:

$$x^{\text{temp}} = \prod_{k=0}^{m-1} x_k^{\text{temp}}; \quad y^{\text{temp}} = \prod_{k=0}^{m-1} y_k^{\text{temp}}; \Delta z^{\text{temp}} = \prod_{k=0}^{m-1} \Delta z_k^{\text{temp}}; \quad z^{\text{temp}} = z + \Delta z^{\text{temp}}.$$

The user sends $\left\langle g_i^{r_3 x(l)},\ \hat{d}_i^{r_4} = (g_i^{-z(l)} d_i)^{r_4 y(l)}, \Delta d_i^{r_5} \right\rangle$ to proxy ϕ_0, which updates it to

$$\langle\ g_i^{r_3 x(l) \cdot \frac{x_0^{\text{temp}}}{x_0(l)}},\ g_i^{r_3 x(l) \cdot \frac{\Delta z_0(l) y_0^{\text{temp}}}{x_0(l)}}, g_i^{r_3 x(l) \cdot \frac{-\Delta z_0^{\text{temp}} y_0^{\text{temp}}}{x_0(l)}},\ \hat{d}_i^{r_4 \frac{y_0^{\text{temp}}}{y_0(l)}}, (\Delta d_i^{r_5})^{y_0^{\text{temp}}}\ \rangle,$$

and sends it to ϕ_1. Here, r_3, r_4 and r_5 are three random numbers picked by the user from $F_p \setminus \{0\}$ and $\Delta d_i = d_i'/d_i$ if d_i is the target data block (where d_i' denotes the content of the target data after the access), otherwise, Δd_i is randomly selected from G_p.

Upon receiving the message, each proxy ϕ_k $(k = 1, \cdots, m-2)$ updates it and forwards the following to ϕ_{k+1}:

$$\langle \; g_i^{r_3 x(l) \cdot \frac{\prod_{t=0}^{k} x_t^{\text{temp}}}{\prod_{t=0}^{k} x_{t(l)}}}, \; g_i^{r_3 x(l) \cdot \frac{\prod_{t=0}^{k} \Delta z_{t(l)} y_t^{\text{temp}}}{\prod_{t=0}^{k} x_{t(l)}}}, \; g_i^{r_3 x(l) \cdot \frac{-\prod_{t=0}^{k} \Delta z_t^{\text{temp}} y_t^{\text{temp}}}{\prod_{t=0}^{k} x_{t(l)}}}, \; \hat{d}_i^{r_4 \frac{\prod_{t=0}^{k} y_t^{\text{temp}}}{\prod_{t=0}^{k} y_{t(l)}}}, (\Delta d_i^{r_5}) \prod_{t=0}^{k} y_t^{\text{temp}} \; \rangle.$$

After proxy ϕ_{m-1} updates, the message becomes:

$$\langle \; g_i^{r_3 x(l) \cdot \frac{\prod_{t=0}^{m-1} x_t^{\text{temp}}}{\prod_{t=0}^{m-1} x_{t(l)}}}, \; g_i^{r_3 x(l) \cdot \frac{\prod_{t=0}^{m-1} \Delta z_{t(l)} y_t^{\text{temp}}}{\prod_{t=0}^{m-1} x_{t(l)}}}, \; g_i^{r_3 x(l) \cdot \frac{-\prod_{t=0}^{m-1} \Delta z_t^{\text{temp}} y_t^{\text{temp}}}{\prod_{t=0}^{m-1} x_{t(l)}}}, \; \hat{d}_i^{r_4 \frac{\prod_{t=0}^{m-1} y_t^{\text{temp}}}{\prod_{t=0}^{m-1} y_{t(l)}}}, (\Delta d_i^{r_5}) \prod_{t=0}^{m-1} y_t^{\text{temp}} \; \rangle,$$

which is equal to

$$\langle \; g_i^{r_3 x^{\text{temp}}}, \; g_i^{r_3 \Delta z(l) y^{\text{temp}}}, \; g_i^{-r_3 \Delta z^{\text{temp}} y^{\text{temp}}}, (g_i^{-z(l)} d_i)^{r_4 y^{\text{temp}}}, (\Delta d_i^{r_5})^{y^{\text{temp}}} \; \rangle.$$

Then, the message is sent to the user and the user removes r_3, r_4 and r_5 and calculates

$$g_i^{\Delta z(l) y^{\text{temp}}} \cdot g_i^{-\Delta z^{\text{temp}} y^{\text{temp}}} \cdot (g_i^{-z(l)} d_i)^{y^{\text{temp}}} = (g_i^{-z^{\text{temp}}} d_i)^{y^{\text{temp}}}.$$

If the computed entry is the target data block, the user will further multiply Δd_i^{temp} to the data content field to get

$$(g_i^{-z^{\text{temp}}} d_i')^{y^{\text{temp}}}.$$

Without loss of generality, we still use d_i to denote the content of each data block including the target data block.

Then, the user uploads

$$\langle \; g_i^{x^{\text{temp}}}, \; (g_i^{-z^{\text{temp}}} d_i)^{y^{\text{temp}}} \; \rangle$$

to the shuffling buffer at the storage server.

7.2.4 Data Shuffling

After every data query, data shuffling is performed. First, the layer which data blocks should be shuffled to needs to be determined. As a rule, data should be shuffled to layer $l' > 0$ if the total number of data blocks in the temporary buffer and at layers $0, \cdots, l'-1$ is greater

than or equal to the total number of buckets at layer $l'-1$, but less than the total number of buckets at layer l'. Otherwise, shuffling should be performed at layer 0 only. For simplicity, we use l' to denote the layer that data blocks are shuffled to.

Data shuffling in MU-ORAM is conducted in the following main steps: (i) Scrambling Round I (oblivious scrambling data blocks that have been uploaded during Phase 2 and thus are already in the temporary buffer before data shuffling); (ii) Data Updating and Appending (updating data blocks at layers $0, \cdots, l'$ that also need to be shuffled and appending them to the temporary buffer); (iii) Scrambling Round II (oblivious scrambling all the data blocks); and (iv) Data Mapping (assigning all the data blocks in the temporary buffer to layer l' according to a hash function). The first three steps are performed through the collaborations between the proxies while the last step is conducted only by the storage server.

To facilitate data shuffling, each proxy maintains a cache that can store $c \cdot \sqrt{N \log N} \cdot \log p$ bits, where $c \geq 1$ is a system parameter and $\log p$ bits is the size of each data block ID or each piece of the data content. Also, each proxy ϕ_k $(k = 0, \cdots, m-1)$ selects keys x_k^{new}, y_k^{new} and $\Delta z_k^{\text{new}}(l')$ for layer l', as well as x_k^{shuf} and y_k^{shuf} for the temporary buffer. All these keys are selected from $F_p \setminus \{0\}$ uniformly at random.

7.2.4.1 Scrambling Round I

The purpose of this round is to re-encrypt and obliviously scramble the data blocks that are in the server's temporary buffer immediately after the data uploading phase ends. Let n_I denote the number of these data blocks. As the total number of layers is L and at most two data blocks are moved from each non-empty layer to the temporary buffer during the query process, at most $2L$ data blocks need to be re-encrypted and scrambled in this round. Hence, $n_I \leq 2L$.

Firstly, each proxy ϕ_k $(k = 0, \cdots, m-1)$ determines a permutation function $\pi_k^{n_I}$ that permutes a sequence of n_I elements. The proxy also prepares a local cache with size $3n_I \log p$ bits; note that $\log p$ bits is the size of each data block ID or each piece of data block content.

Secondly, the proxies collaborate in scrambling and re-encrypting the IDs of the data blocks in the temporary buffer of the server. The process is as follows.

Proxy ϕ_0 fetches the encrypted IDs of all the data blocks in the server's temporary buffer, to its own cache and scrambles these IDs using permutation function $\pi_0^{n_I}$. Then, each encrypted ID, denoted as $g_i^{x^{\text{temp}}}$, is updated (i.e., re-encrypted) to tuple

$$\langle \; g_i^{x^{\text{temp}} \cdot \frac{x_0^{\text{shuf}}}{x_0^{\text{temp}}}}, g_i^{x^{\text{temp}} \cdot \frac{\Delta z_0^{\text{temp}} y_0^{\text{shuf}}}{x_0^{\text{temp}}}}, g_i^{x^{\text{temp}} \cdot \frac{-\Delta z_0^{\text{new}}(l') y_0^{\text{shuf}}}{x_0^{\text{temp}}}} \; \rangle,$$

and sent to proxy ϕ_1.

Upon receiving the n_I tuples from proxy ϕ_{k-1}, each proxy ϕ_k ($k = 1, \cdots, m-2$) scrambles the tuples using permutation function $\pi_k^{n_I}$, and then updates each tuple to the following and forwards it to ϕ_{k+1}:

$$\langle \; g_i^{x^{\text{temp}} \cdot \frac{\prod_{t=0}^{k} x_t^{\text{shuf}}}{\prod_{t=0}^{k} x_t^{\text{temp}}}}, g_i^{x^{\text{temp}} \cdot \frac{\prod_{t=0}^{k} \Delta z_t^{\text{temp}} y_t^{\text{shuf}}}{\prod_{t=0}^{k} x_t^{\text{temp}}}}, g_i^{x^{\text{temp}} \cdot \frac{-\prod_{t=0}^{k} \Delta z_t^{\text{new}}(l') y_t^{\text{shuf}}}{\prod_{t=0}^{k} x_t^{\text{temp}}}} \; \rangle.$$

After proxy ϕ_{m-1} scrambles the tuples that it has received and updates them, each tuple becomes:

$$\langle \; g_i^{x^{\text{temp}} \cdot \frac{\prod_{t=0}^{m-1} x_t^{\text{shuf}}}{\prod_{t=0}^{m-1} x_t^{\text{temp}}}}, g_i^{x^{\text{temp}} \cdot \frac{\prod_{t=0}^{m-1} \Delta z_t^{\text{temp}} y_t^{\text{shuf}}}{\prod_{t=0}^{m-1} x_t^{\text{temp}}}}, g_i^{x^{\text{temp}} \cdot \frac{-\prod_{t=0}^{m-1} \Delta z_t^{\text{new}}(l') y_t^{\text{shuf}}}{\prod_{t=0}^{m-1} x_t^{\text{temp}}}} \; \rangle.$$

which is equal to

$$\langle \; g_i^{x^{\text{shuf}}}, \; g_i^{\Delta z^{\text{temp}} y^{\text{shuf}}}, \; g_i^{-\Delta z^{\text{new}}(l') y^{\text{shuf}}} \; \rangle,$$

where x^{shuf}, y^{shuf}, Δz^{temp} and $\Delta z^{\text{new}}(l')$ are defined as

$$x^{\text{shuf}} = \prod_{k=0}^{m-1} x_k^{\text{shuf}}; \quad y^{\text{shuf}} = \prod_{k=0}^{m-1} y_k^{\text{shuf}}; \Delta z^{\text{temp}} = \prod_{k=0}^{m-1} \Delta z_k^{\text{temp}}; \quad \Delta z^{\text{new}}(l') = \prod_{k=0}^{m-1} \Delta z_k^{\text{new}}(l').$$

Proxy ϕ_{m-1} saves the sequence of re-encrypted IDs (i.e., $g_i^{x^{\text{shuf}}}$) back to the server's temporary buffer, but stores the sequence of $\{g_i^{(\Delta z^{\text{temp}} - \Delta z^{\text{new}}(l')) y^{\text{shuf}}}\}$ to its local cache.

Thirdly, the proxies scramble and re-encrypt the contents of the n_I data blocks, piece by piece. As the operations for pieces are similar, we only present the operations on the first piece of the data blocks in the following.

Proxy ϕ_0 fetches the first pieces of all the data blocks from the server's buffer to its own cache, and scramble these pieces using permutation function $\pi_0^{n_I}$. Then, each piece, denoted as $\hat{d}_i = (g_i^{-z^{\text{temp}}} d_i)^{y^{\text{temp}}}$, is updated (i.e., re-encrypted) to $\hat{d}_i^{\frac{y_0^{\text{shuf}}}{y_0^{\text{temp}}}}$, and sent to the next proxy

ϕ_1. The following proxies conduct the similar scrambling, re-encryption, and forwarding. After scrambling and re-encryption have been completed in ϕ_{m-1}, each of the n_I pieces in the sequence is in the form of

$$\hat{d}_i^{\frac{\prod_{k=0}^{m-1} y_k^{\text{shuf}}}{\prod_{k=1}^{m-1} y_k^{\text{temp}}}},$$

which is equal to

$$\hat{d}_i^{\frac{y^{\text{shuf}}}{y^{\text{temp}}}} = (g_i^{-z^{\text{temp}}} d_i)^{y^{\text{temp}} \cdot \frac{y^{\text{shuf}}}{y^{\text{temp}}}} = g_i^{-(z+\Delta z^{\text{temp}}) y^{\text{shuf}}}.$$

Finally, proxy ϕ_{m-1} multiplies the piece with its locally-stored $g_i^{(\Delta z^{\text{temp}} - \Delta z^{\text{new}}(l')) y^{\text{shuf}}}$ to obtain $(g_i^{-z^{new}(l')} d_i)^{y^{\text{shuf}}}$, and saves it back to the server's temporary buffer.

7.2.4.2 Data Updating and Appending

For each data block D_i on layer l ($l = 0, \cdots, l'$), which needs to be shuffled to layer l', it should be updated to

$$\langle \ g_i^{x^{\text{shuf}}}, \ (g_i^{-z^{\text{new}}(l')} d_i)^{y^{\text{shuf}}} \ \rangle.$$

The updating is performed collaboratively by the proxies, similar to Phase 2 (Data Uploading). Different from Phase 2, no any user is involved in the process. Hence, the first proxy ϕ_0 directly updates based on $\langle g_i^{x(l)}, \hat{d}_i = (g_i^{-z(l)} d_i)^{y(l)} \rangle$. After the last proxy ϕ_{m-1} has completed its update, it appends the updated data block to the server's temporary buffer. Therefore, at the end of this step, all the data blocks that should be shuffled to layer l' are stored in the server's temporary buffer.

7.2.4.3 Scrambling Round II

This round is to re-encrypt and scramble all the data blocks in the server's temporary buffer. Let n_{II} denote the total number of these data blocks. As the total number of data blocks stored at the server is N, it holds that $n_{II} \leq N$. Our proposed algorithm for this round is based on the idea of piece-wise shuffling proposed by Zhang et al. [77] and the data scrambling algorithm proposed by Williams et al. [72]. Our algorithm requires the capacity of each proxy's local cache to be $c\sqrt{N \log N} \log p$ bits and incurs the communication cost of $O(N \log \log N)$ data blocks on average.

When $n_{II} \leq \sqrt{N}$, the scrambling round operates as follows.

Initially, each proxy ϕ_k for $k = 0, \cdots, m-1$ determines a secret permutation function $\pi_k^{n_{II}}$ which permutes a sequence of n_{II} elements; therefore, the storage requirement of this function is $n_{II} \log n_{II}$. The proxy also randomly picks new keys $x_k^{\text{new}}(l')$ and $y_k^{\text{new}}(l')$ for layer l'.

Then, proxy ϕ_0 downloads the encrypted IDs of the n_{II} data blocks, and performs the following steps sequentially:

- Re-encryption. Each encrypted ID denoted as $g_i^{x^{\text{shuf}}}$ is re-encrypted to $(g_i^{x^{\text{shuf}}})^{\frac{x_k^{\text{new}}(l')}{y_k^{\text{shuf}}}}$.
- Scrambling. All the n_{II} re-encrypted data IDs are scrambled using permutation function $\pi_0^{n_{II}}$.
- Forwarding. The encrypted IDs are forwarded to the next proxy, which also performs the re-encryption and scrambling using its own key and secret permutation function, and forwards them to its next proxy. The last proxy stores the encrypted IDs back to the server's temporary storage.

In a similar way, the data contents of the n_{II} blocks are also re-encrypted using key $y_k^{\text{new}}(l')$ and scrambled using permutation function $\pi_k^{n_{II}}$ sequentially by each proxy ϕ_k, piece by piece.

When $n_{II} > \sqrt{N}$, the data blocks are also re-encrypted and scrambled sequentially by all the proxies, piece by piece. As different pieces of the same data block are processed in the similar way (the only difference is, the first piece, i.e., the encrypted ID, is re-encrypted with key $x_k^{\text{new}}(l')$ while the content pieces are re-encrypted with key $y_k^{\text{new}}(l')$ by each proxy ϕ_k), we present only the processing of the first pieces (i.e., encrypted IDs) of all the n_{II} data blocks. Furthermore, the processing by different proxies are also similar, except that they use different keys and permutation functions for re-encryption and permutation. Hence, in the following we only elaborate how proxy ϕ_0 processes the encrypted IDs of the data blocks.

As formally presented in Algorithms 8 and 9, the data blocks are processed through multiple sub-rounds. In the first sub-round, each of the n_{II} data blocks in the server's temporary buffer forms a single-element group, and every $n_{II}^{1/2}$ groups are randomly merged together, re-encrypted, and uploaded back to the server's temporary buffer. In the second sub-round, these

Algorithm 8 Re-encrypt&Scramble($\{I_0, \cdots, I_{n_{II}-1}\}$): re-encryption and scrambling of encrypted IDs I_0, $\cdots$, $I_{n_{II}-1}$ by proxy ϕ_0.

1: $n \leftarrow n_{II}$
2: $k_{old} = x_0^{\text{shuf}}$
3: **while** $n > 1$ **do**
4: split $\{I_0, \cdots, I_{n_{II}-1}\}$ evenly to n groups: $\hat{g}_0, \cdots, \hat{g}_{n-1}$
5: $n' \leftarrow \lfloor \sqrt{n} \rfloor$
6: split local cache evenly to n' segments: $S_0, \cdots, S_{n'}$
7: **if** $n' > 1$ **then**
8: select k_{new} randomly from $F_p \setminus \{0\}$
9: $k_{old} = k_{old} * k_{new}$
10: **else**
11: $k_{new} = x_0^{\text{new}}(l')/k_{old}$
12: **end if**
//Merge n groups into n' larger groups
13: **for** $i := 0$ to $n' - 1$ **do**
14: Re-encrypt&Merge(k_{new}, $\{\hat{g}_{i*n'}, \cdots, \hat{g}_{(i+1)*n'-1}\}$)
15: **end for**
16: $n \leftarrow n'$
17: **end while**

data blocks form $n_{II}^{1/2}$ groups with $n_{II}^{1/2}$ pieces in each group. Then, every $n_{II}^{1/4}$ of such groups are randomly merged together, re-encrypted, and uploaded back to the server. Such merging and re-encryption repeat until all the pieces are merged together.

7.2.4.4 Data Mapping

In this step, the server assigns each of the n_{II} data blocks, which is in the form of $(g_i^{x^{\text{new}}(l')}, \hat{d}_i)$ into bucket $H_{l'}(g_i^{x^{\text{new}}(l')})$ of layer l'.

7.3 Security Analysis

This section presents the security analysis of the proposed MU-ORAM. First, we show that MU-ORAM is secure against honest but curious storage server. Then, we show that MU-ORAM is secure against a collusive coalition of honest but curious server, proxies and users.

Algorithm 9 Re-encrypt&Merge(k_{new}, $\{\tilde{g}_0, \cdots, \tilde{g}_{n-1}\}$): merge pieces in groups $\tilde{g}_0, \cdots, \tilde{g}_{n-1}$ and re-encrypt them with key k_{new}

$s \leftarrow c\sqrt{N \log N}/n$ //calculate the capacity of each segment
//fill in half of each segment
for $i := 0$ to $n-1$ **do**
    for $j := 0$ to $S/2$ **do**
        download one piece from $\tilde{g}_i$ (if any) to segment S_i
    end for
end for
//scramble and re-encrypt the pieces
while segments are not empty **do**
    if groups are not empty **then**
        for $i := 0$ to $n-1$ **do**
            download one piece from $\tilde{g}_i$ (if any) to S_i
        end for
    end if
    for $i := 0$ to $n-1$ **do**
        randomly select r from $\{j | S_j$ is not empty $\}$
        re-encrypt the first piece in S_r with k_{new} & upload it to the server's temporary buffer
    end for
end while

7.3.1 Security against Curious Server

MU-ORAM follows the framework of hash-based ORAMs [27] with the following major differences: (i) no dummy data block in the system; (ii) during each query process, two data blocks from each non-empty level are removed from its bucket and uploaded to the top layer; (iii) empty bucket will never be accessed due to the bitmap. In the following, we first find the upper bound of the failure probability (i.e., bucket overflow probability) and then prove that MU-ORAM is secure against an honest but curious server according to Definition 7.1.3.1.

Lemma 10. *(Probability of bucket overflow).* $\forall 0 \leq l \leq L-1$,

$$\boldsymbol{Pr}[\text{A bucket overflows on layer } l] \leq O(N^{-\log\log N}).$$

Proof. In MU-ORAM, there are at most $n_l = 2^{l+1} \log N$ data blocks to be distributed into n_l buckets. Then, according to a standard balls and bins model, we could have the following analysis:

Let us consider a particular bucket buc_j, and for each data block, define $X_1, \cdots, X_{n_l}$ as random variables such that

$$X_i = \begin{cases} 1 & \text{the } i^{\text{th}}\text{data block mapped to } buc_j, \\ 0 & \text{otherwise.} \end{cases}$$

Note that, $X_1, \cdots, X_{n_l}$ are independent of each other, and hence for each X_i, $\mathbf{Pr}[X_i = 1] = \frac{1}{n_l}$. Then, the probability that buc_j has more than t data blocks is:

$$\mathbf{Pr}[\#\text{ data blocks in } buc_j \geq t] \leq \binom{n_l}{t}\left(\frac{1}{n_l}\right)^t \leq \left(\frac{e \cdot n_l}{t}\right)^t \left(\frac{1}{n_l}\right)^t = \left(\frac{e}{t}\right)^t \tag{7.3}$$

Note that the second inequality of Equation (7.3) is due to $\binom{n}{k} \leq \left(\frac{e \cdot n}{k}\right)^k$ for all $k < n$. Further considering the fact that $n_l \leq N$ and $t = \log N$, we apply the union bound of all buckets on layer l:

$$\begin{aligned} &\mathbf{Pr}[\ \exists \text{ a bucket with more than } t \text{ data blocks}] \\ \leq\ & n_l \cdot \left(\frac{e}{t}\right)^t \leq N \cdot \left(\frac{e}{\log N}\right)^{\log N} = O(N^{-\log\log N}). \end{aligned}$$

Therefore, we have for any layer l in MU-ORAM, the probability of any buckets to have more than $\log N$ data blocks is negligible in N, which is $O(N^{-\log\log N})$. □

Theorem 4. *MU-ORAM is secure against an honest but curious server.*

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

- Firstly, according to the query algorithm, sequences $A(\vec{x})$ and $A(\vec{y})$ should have the same format; that is, they contain the same number of accesses, and each pair of corresponding accesses have the same format.

- Secondly, all data blocks in MU-ORAM are randomly encrypted and each data block is re-encrypted after each access. Hence, the two sequences could not be distinguished based on the appearance of data blocks.

- Thirdly, according to the query algorithm, the j-th accesses ($j = 1, \cdots, |A(\vec{x})|$) of the $A(\vec{x})$ and $A(\vec{y})$ are from the same non-empty layer of the storage. Also, according to the MU-ORAM design, the buckets accessed from each layer are either selected uniformly at random, or determined by a hash function (which is also uniformly random); hence, they are uniformly random in both sequences.

Furthermore, according to Lemma 10, a bucket overflows (i.e., MU-ORAM fails) with probability $O(N^{-\log\log N})$. Therefore, according to Definition 7.1.3.1, MU-ORAM is secure against an honest but curious storage server. □

7.3.2 Security against Collusive Coalition

To quantify the security strength of MU-ORAM against a collusive coalition, we first introduce the *Modified Matching Diffie-Hellman (MMDH) problem* as follows:

Definition 4. (*Modified Matching Diffie-Hellman (MMDH) Problem*). Let G_p be a multiplicative cyclic group of order p and generator g. The MMDH problem is defined as: given g^{a_0}, g^{a_1}, g^c, and $(g^{a_b c}, g^{a_{1-b} c})$, for some unknown a_0, a_1 and c randomly picked from F_p and an unknown binary bit b randomly picked from $\{0, 1\}$, find out the value of b.

Similar to the proofs by Handschuh et. al. [36] and Bao et. al. [4], it can be shown that MMDH is a computational hard problem as the Decisional Diffie-Hellman and Matching Diffie-Hellman problems.

We study the security strength of MU-ORAM in the following two cases, as described in Section 7.1.3.2.

Case 1: Users with same access privileges to data For this case, we study the security strength of MU-ORAM in protecting an innocent user's access pattern to the data that can also be accessed by the collusive users. Specific, we have proved the following theorem based on the game $\mathcal{G}_1$ and the (ϵ, t, n_{cq})-security notion defined in Section 7.1.3.2:

Theorem 5. *If the MMDH problem is (ϵ, t)-hard (i.e., there is no algorithm can solve the MMDH problem with an advantage of at least ϵ within time period t), MU-ORAM is* $(1.5n_{cq}/N+$

$(1-3n_{cq})2\epsilon/N, t, n_{cq})$-*secure against a collusive coalition of semi-honest storage server, users and some (but not all) proxies, in the scenario that the collusive users can access all the data accessed by any innocent user.*

Proof. The proof includes two parts: In the first part, we develop an algorithm $\mathcal{B}$ to play as the challenger in game $\mathcal{G}_1$. Note that, there can be two consequences of the game: (i) *Consequence I*: $\mathcal{B}$ aborts the game and claims failure because adversary $\mathcal{A}$ succeeds in finding a data ID chosen in a Selection Phase. (ii) *Consequence II*: $\mathcal{B}$ does not "abort the game and declare failure". In this case, $\mathcal{B}$ will attempt to solve the MMDH problem if $\mathcal{A}$ succeeds in the end of the game. In the second part, we analyze the probabilities of the above two consequences respectively, and the probability for $\mathcal{B}$ to succeed in solving the MMDH problem when Consequence II occurs.

Part (1): Algorithm $\mathcal{B}$.

$\mathcal{B}$ acts as the challenger in the game $\mathcal{G}_1(\mathcal{M}, p, N, m, n_{cq})$. $\mathcal{B}$ is given g, g^{a_0}, g^{a_1}, g^c, and $(g^{a_b c}, g^{a_{1-b}c})$, where $b \in \{0,1\}$ and a_0, a_1 and c are randomly picked from F_p.

Initialization Phase - $\mathcal{B}$ simulates to construct and initialize m proxies and the data storage of N encrypted data blocks according to Sections 7.2.1 and 7.2.2: A hierarchical storage structure as described in Section 7.2.1 is constructed and initialized. Three integers are randomly selected from $F_p \setminus \{0\}$, and denoted as z, α and β respectively. For each layer $l \in \{0, \cdots, L-1\}$ in the hierarchical structure, a random oracle (hash function) H_l is introduced to map encrypted data block IDs to buckets. Each proxy ϕ_k $(k = 0, \cdots, m-1)$ is preloaded with keys $x_k(l)$, $y_k(l)$ and $\Delta z(l)$ for $l \in \{0, \cdots, L-1\}$, which are all randomly selected from $F_p \setminus \{0\}$. N data blocks are initialized as N distinct (ID, content) pairs as follows: Let u and v be two distinct integers selected from $\{0, \cdots, N-1\}$ uniformly at random. Data blocks D_u and D_v are set to (g^{a_0}, g^{y_u}) and (g^{a_1}, g^{y_v}) respectively, where y_u and y_v are randomly selected from $F_p \setminus \{0\}$. For each of the rest data blocks $D_i = (g_i, d_i)$ where $i \in \{0, \cdots, N-1\} \setminus \{u, v\}$, $g_i = g^{x_i}$ and $d_i = g^{y_i}$ where x_i and y_i are randomly selected from $F_p \setminus \{0\}$. The IDs, i.e., g_i for $i = 0, \cdots, N-1$, are provided to $\mathcal{A}$. Each D_i is then encrypted into $(g_i^{x(L-1)}, (g_i^{-z(L-1)} d_i)^{y(L-1)})$, where $z(L-1) = z + \Delta z(L-1)$, and uploaded to bucket $H_{L-1}(g_i^{x(L-1)})$ of layer $L-1$.

Query Phase I - This phase consists of multiple requests that can be made by $\mathcal{A}$. We describe $\mathcal{B}$'s response to each type of $\mathcal{A}$'s requests as follows:

- *Data Query* - Depending on the type of query requests made by $\mathcal{A}$, the responses are different.
  - For Type I query (i.e., controlled query), if the number of such type of query exceeds n_{cq}, the game aborts. Otherwise, $\mathcal{A}$ simulates the behavior of the user, and $\mathcal{B}$ simulates the behavior of the proxies and the server. They both follow MU-ORAM's query and shuffling algorithms.
  - For Type II query (i.e., random query), $\mathcal{B}$ simulates the querying user, the proxies and the server, following MU-ORAM's algorithms. Note that, $\mathcal{A}$ does not know which ID is selected by $\mathcal{B}$ to query, but it can observe the process through requesting transcripts from the server and compromised proxies.

  Without loss of generality, in this proof we assume that the content of the queried target data block is always changed before it is uploaded.

- *Proxy Compromise* - Upon $\mathcal{A}$ queries to compromise a proxy, the secret keys $x_k(l)$, $y_k(l)$ and $\Delta z_k(l)$, where $l = 0, \cdots, L-1$, are returned to $\mathcal{A}$.

- *Proxy and Server Transcript Checking* - Upon $\mathcal{A}$ queries to check the transcript of a proxy's or the server's certain operations, the input and the output of the operations are returned to $\mathcal{A}$.

- *Storage Inspection* - $\mathcal{A}$ may request to inspect the bitmap of a particular layer or the content of a particular bucket of some layer. As a result, the bitmap and/or the content of a bucket are returned.

Selection Phase I - In this phase, $\mathcal{B}$ launches the process of querying data block D_u, i.e., the data block of ID g^{a_0}.

A new game instance (which we call *Game Instance 1*) is forked from the current game (which we call *Game Instance 0*). In both game instances, the query for data block D_u is executed following the querying and shuffling algorithms described in Sections 7.2.3 and 7.2.4. However, the data content of D_u is changed differently in these instances:

- In Game Instance 0, after D_u is queried, the content of D_u, i.e., d_u, is changed from its current value to g^w where w is randomly selected from $F_p \setminus \{0\}$.

- In Game Instance 1, after D_u is queried, d_u is changed to $g_u^{z+\alpha \cdot c} g^\beta$. Also, from this point, the Δz used in the uploading phase and the shuffling phase should always follow the format of $\Delta z = \alpha \cdot c + \gamma$ where $\gamma \in F_p \setminus \{0\}$ and can vary. This way, the data content of D_u will be encrypted into

$$g_u^{-(z+\Delta z)} g_u^{z+\alpha \cdot c} g^\beta = g_u^{-\gamma} g^\beta,$$

  which can be computed without knowing the answer to the MMDH problem.

Query Phase II - In this phase, the requests are handled in the same way as in the Query Phase I in both instances of the game, except for the following scenarios. (i) When data block D_u, which was queried in the Selection Phase I, is queried by $\mathcal{A}$ as Type I query request: The rule specified in the definition of $\mathcal{G}_1$ is applied, and $\mathcal{B}$ will abort the game and declare failure if the specified conditions are satisfied. (ii) When data block D_u is queried again in the response to Type II query request: The current Game Instance 1 is aborted, and the current Game Instance 0 forks a new game instance which we call Game Instance 1. In both instances, the query for D_u is processed following the querying and shuffling algorithms described in Sections 7.2.3 and 7.2.4, but the content of D_u is changed differently:

- In Game Instance 0, after D_u is queried, d_u is changed from its current value to g^w where w is randomly selected from $F_p \setminus \{0\}$.

- In Game Instance 1, after D_u is queried, d_u is changed from its current value to $g_u^{z+\alpha \cdot c} g^\beta$. And, from this point, the Δz used in the uploading phase and the shuffling phase should always follow the format of $\Delta z = \alpha \cdot c + \gamma$ where $\gamma \in F_p \setminus \{0\}$ and can vary. Note that, this is the same as in Selection Phase I.

Selection Phase II - In this phase, $\mathcal{B}$ launches the process of querying data block D_v, i.e., the data block of ID g^{a_1}, in both instances of the game. Then, each of the current game instance forks a new game instance. Game Instance 0 forks a new Game Instance 2, and Game Instance

1 forks a new Game Instance 3. All these four game instances follow the same data querying and shuffling algorithms as above, but the content of D_v is changed differently:

- In Game Instances 0 and 1, after D_v is queried, d_v is changed from its current value to g^w where w is randomly selected from $F_p \setminus \{0\}$.
- In Game Instances 2 and 3, after D_v is queried, d_v is changed to $g_v^{z+\alpha \cdot c} g^{\beta}$.

Furthermore, in Game Instance 3, if D_u and D_v are in the same bucket when the query is launched, the game aborts; otherwise, the following operations should be conducted:

- In the data query phase, both D_u and D_v (i.e., the buckets that contain these two data blocks) should be selected to download. Note that this is attainable because the querying algorithm downloads two buckets from each layer that has two or more buckets, and D_u and D_v are in different buckets. Also this process is oblivious because both blocks are randomly distributed to the buckets.
- In the data uploading phase, both D_u and D_v should be selected to upload to the temporary buffer of the server. This is attainable because the uploading algorithm uploads one data block from each downloaded bucket.
- In the data shuffling phase immediately after the query, D_u and D_v are scrambled during the *Scrambling Round I*, in which $\Delta z^{new}(l')$ (note: l' is the layer that the data blocks should be shuffled to) is set to $r_0 \cdot c$ and x^{shuf} is set to $r_1 \cdot c$ where r_0 and r_1 are randomly selected from $F_p \setminus \{0\}$. Hence, D_u and D_v are encrypted to

$$\left(g^{a_0 c \cdot r_1}, (g^{-a_0 c \cdot (r_0 - \alpha)} g^{\beta})^{y^{shuf}}\right)$$

and

$$\left(g^{a_1 c \cdot r_1}, (g^{-a_1 c \cdot (r_0 - \alpha)} g^{\beta})^{y^{shuf}}\right),$$

respectively; further after random scrambling, they become

$$\left((g^{a_b c})^{r_1}, (g^{-a_b c})^{(r_0 - \alpha) \cdot y^{shuf}} g^{\beta \cdot y^{shuf}}\right)$$

and

$$\left((g^{a_{1-b}c})^{r_1}, (g^{-a_{1-b}c})^{(r_0-\alpha)\cdot y^{shuf}} g^{\beta \cdot y^{shuf}}\right),$$

where b is either 0 or 1 with the same probability.

- In the rest lifetime of this game instance, the key $x(l)$ should always be some $r \cdot c$, where r is selected randomly from $F_p \setminus \{0\}$ and can vary.

Query Phase III - The requests in this phase are handled in the same way as in Query Phase I, except when D_u or D_v is queried. (i) When D_u or D_v is queried by $\mathcal{A}$ as Type I query request, the specified rule is checked to determine if $\mathcal{B}$ should abort the game and declare failure. (ii) When D_u or D_v is queried as $\mathcal{B}$'s response to Type II query request, it is handled as follows.

When D_u is queried, Game Instances 1 and 3 abort; meanwhile, Game Instance 0 forks a new Game Instance 1, and Game Instance 2 forks a new Game Instance 3. These four game instances follow the same data querying and shuffling algorithms as above, but the content of D_u is changed differently:

- In Game Instances 0 and 2, after D_u is queried, d_u is changed from its current value to g^w where w is randomly selected from $F_p \setminus \{0\}$.

- In Game Instances 1 and 3, after D_u is queried, d_u is changed to $g_u^{z+\alpha \cdot c} g^{\beta}$.

Furthermore, in Game Instance 3, if D_u and D_v are in the same bucket when the query is launched, the game aborts; otherwise, the operations as described in the Selection Phase II should be conducted as well.

When D_v is queried, Game Instances 2 and 3 abort; meanwhile, Game Instance 0 forks a new Game Instance 2, and Game Instance 1 forks a new Game Instance 3. These four game instances follow the same data querying and shuffling algorithms as above, but the content of D_v is changed differently:

- In Game Instances 0 and 1, after D_v is queried, d_v is changed from its current value to g^w where w is randomly selected from $F_p \setminus \{0\}$.

- In Game Instances 2 and 3, after D_v is queried, d_v is changed to $g_v^{z+\alpha \cdot c} g^{\beta}$.

Furthermore, in Game Instance 3, if D_u and D_v are in the same bucket when the query is launched, the game aborts; otherwise, the operations as described in the Selection Phase II should be conducted as well.

Challenge Phase - If Game Instance 3 does not exist, the game will abort. Otherwise, $\mathcal{B}$ launches the process of querying the data block with ID g^{a_b} in this game instance. Though $\mathcal{B}$ does not know g^{a_b} because b is unknown, the query can be implemented as follows: to execute the first step of data query phase for layer l where $x(l) = rc$ for some $r \in F_p$, $\mathcal{B}$ picks γ_1 and γ_2 from G_p uniformly at random, and then sends out $(\gamma_1, \gamma_1, \gamma_2)$ to the simulated proxies which will execute the algorithm as specified in MU-ORAM design; no matter what are returned from the proxies, $\mathcal{B}$ sets $(g_{a_b})^{x(l)} = (g^{a_b c})$. Then, the rest part of the data query and shuffling algorithms can be implemented trivially.

Query Phase IV - The requests in this phase are handled in the same way as in Query Phase I, except that: (i) If $\mathcal{A}$ requests to query ID g_u or g_v, $\mathcal{B}$ will abort the game and declare failure. (ii) In respond to $\mathcal{A}$'s type II query, $\mathcal{B}$ will randomly select one of the IDs to query. In this case, if the selected ID is g_u, the data block with ID g^{a_b} is queried instead; if the selected ID is g_v, the data block with ID $g^{a_{1-b}}$ is queried instead.

Response Phase - $\mathcal{A}$ responds with b'. Algorithm $\mathcal{B}$ uses b' as the solution to the MMDH problem.

Part (2): Analysis of $\mathcal{B}$.

First, we analyze the probability for Consequence I to occur. Note that, Consequence I refers to the case that $\mathcal{B}$ aborts the game and declares failure according to the rules in processing Type I data queries in Query Phase II and III. As explained in the definition of $\mathcal{G}_1$, such failure of $\mathcal{B}$ is due to that $\mathcal{A}$ finds the ID chosen in a Selection Phase and thus can discover the access pattern. Specifically, there are three sub-cases for such failure to occur: Sub-case (i): $\mathcal{A}$ finds the ID chosen in Selection Phase I (i.e., g_u - ID of D_u); Sub-case (ii) $\mathcal{A}$ finds the ID chosen in Selection Phase II (i.e., g_v - ID of D_v); Sub-case (iii): $\mathcal{A}$ requests to query D_u or D_v in Query Phase IV. Sub-case (i) occurs if D_u as a Type I data query is requested both before and after Selection Phase I. We can compute the probability for this to occur as follows:

- Supposing x Type I queries are made before Selection Phase I, the probability for the

query of D_u to be among the x queries is $\binom{N-1}{x-1} / \binom{N}{x} = x/N$.

- Suppose y Type I queries are made after Selection Phase I. As $\mathcal{B}$ has higher probability to find the target of Selection Phase I if it only chooses the IDs that it has queried before Selection Phase I to query again (Note: this way it can detect the data block whose content is changed), the probability for D_u to be queried among the y IDs is $\binom{x-1}{y-1} / \binom{x}{y} = y/x$.

Therefore, the probability for Sub-case (i) is $(x/N) \cdot (y/x) = y/N$; further due to $x + y \leq n_{cq}$ and $y < x$, the probability is at most $n_{cq}/2N$.

Similarly, the probability for Sub-case (ii) is at most $n_{cq}/2N$. Lastly, the probability for Sub-case (iii) is at most $1 - \binom{N-2}{n_{cq}} / \binom{N}{n_{cq}} < 2n_{cq}/N$. Totally, the probability for Consequence I to occur is less than $3n_{cq}/N$.

Second, we consider Consequence II, i.e., $\mathcal{B}$ does not "abort the game and declare failure". Here, there will be two cases: (i) In the first case, the game aborts at the beginning of Challenge Phase because D_u and D_v are in the same bucket in Game Instance 3. This case occurs with a probability of at most $\frac{1}{2 \log N}$, due to the facts that each layer has at least $2 \log N$ different buckets and data blocks are randomly distributed to the buckets. (ii) In the second case, the game finishes normally, and the adversary returns a binary bit b'. In this case, if $\mathcal{A}$ wins the game (i.e., $b' = b$), $\mathcal{B}$ also obtains the correct answer (i.e., $b = b'$) to the MMDH problem and thus solve the problem.

Considering the above two cases together, if $\mathcal{A}$ can win the game under Consequence II with advantage ϵ' within time period t, $\mathcal{B}$ can solve the MMDH problem with an advantage of at least $\frac{2 \log N - 1}{2 \log N} \epsilon' > 0.5\epsilon'$ with the same time complexity. Hence, with the assumption that the MMDH problem is (ϵ, t)-secure, i.e., no algorithm can solve the MMDH problem with an advantage of at least ϵ within time t, we conclude that $\mathcal{A}$ cannot win the game under Consequence II with an advantage of at least 2ϵ within time t.

To summarize the above analysis for Consequence I and II, $\mathcal{A}$ cannot win the game with a probability greater than $3n_{cq}/N + (1 - 3n_{cq}/N) * (0.5 + 2\epsilon)$ (i.e., an advantage greater than $1.5n_{cq}/N + (1 - 3n_{cq})2\epsilon/N$), if it can issue at most n_{cq} ($n_{cq} < N/3$) Type I data queries and its

overall running time is t. That is, MU-ORAM is $(1.5n_{cq}/N + (1-3n_{cq})2\epsilon/N, t, n_{cq})$-secure. □

This theorem reveals the following intuition: As the collusive attackers have the access privileges to all data that an innocent user can access, they can attack the access pattern privacy of an innocent user through checking if some randomly selected data blocks have been changed after innocent user accessed a data block. However, to make such attack effective, the attackers need to make a number of queries that is proportional to N; specifically, to gain an advantage A, the attackers need to make $A \cdot N/3$ queries on average. Note that, this will further require the adversary to incur $O(A \cdot N \log^2 N \log\log N \cdot B)$ bits communication cost as the per query communication cost of MU-ORAM is $O(\log^2 N \cdot \log\log N \cdot B)$ bits as shown in Section 7.4.

Case 2: Users with different access privileges to data For this case, we study the security strength of MU-ORAM in protecting an innocent user's access pattern to the data that cannot be accessed by the collusive users. To quantify the strength, we have proved the following theorem based on the game $\mathcal{G}_2$ and the notion of (ϵ, t)-security defined in Section 7.1.3.2:

Theorem 6. *If the MMDH problem is (ϵ, t)-hard (i.e., there is no algorithm can solve the MMDH problem with an advantage of at least ϵ within time period t), MU-ORAM is $(2\epsilon, t)$-secure in protecting an innocent user's access pattern to the data that cannot be accessed by a collusive coalition of semi-honest storage server, users and some (but not all) proxies.*

Proof. As the proof of Theorem 6 is actually a subset of the proof of Theorem 5, we only sketch the difference as follows: This proof also includes two parts: construction of algorithm $\mathcal{B}'$ to play as the challenger in game $\mathcal{G}_2$ with the adversary $\mathcal{A}$, and the analysis of $\mathcal{B}'$.

In the first part, as the definitions of $\mathcal{G}_2$ is different from that of $\mathcal{G}_1$, we describe the major difference of $\mathcal{B}'$ from $\mathcal{B}$ in the proof of Theorem 2: (i) Initially, $\mathcal{A}$ is only provided with a subset of the complete ID sets; that is, there are totally N IDs but $\mathcal{A}$ is only given $N' \leq N-2$ of the IDs. (ii) The IDs of D_u and D_v (i.e., the two data blocks queried in Selection Phase I and II) are unknown to $\mathcal{A}$. (iii) $\mathcal{B}'$ will not "abort the game and declare failure" in the game.

In the second part, as $\mathcal{B}'$ does not "abort the game and declare failure", we only need to analyze the probability for $\mathcal{B}'$ to solve the MMDH problem if $\mathcal{A}$ wins the game. Therefore, based on the assumption that the MMDH problem is (ϵ, t)-hard, we can conclude that $\mathcal{A}$'s advantage to win the game within time t is at most 2ϵ. □

Theorem 6 reveals the intuition that, MU-ORAM is more effective in protecting an innocent user's access pattern to the data that cannot be accessed by the collusive attackers; specifically, if the advantage is negligible to solve the MMDH problem in a certain time period t, the chance for the collusive attackers to reveal the above data access pattern within time period t is also negligible.

Comparing the security strength of MU-ORAM in the above two cases, we can see that, MU-ORAM is more effective to protect data access pattern in Case 2 than in Case 1.

7.4 Cost Analysis

In this section, we analyze the storage and communication costs of the MU-ORAM with the following assumptions:

- We assume the data block size $B \geq \sqrt[4]{N}$ bits. Note that this is reasonable in practice. For example, if $N \leq 2^{32}$, B is just required to be at least 32 bytes.
- We assume the bitmap recursion depth is 4. Hence, for a layer with n buckets, the total size of the bitmap in bits is:

$$bitmap(n) = n + \sqrt[4]{n^3} + \sqrt[4]{n^2} + \sqrt[4]{n} \leq 2n,$$

  where the inequality holds as long as $n \geq 2$.
- For simplicity, the size of a piece is set to $b = 2048$ bits.

7.4.1 Storage Costs

We analyze the storage costs for the storage server, each proxy, and each user, respectively.

Storage cost at the server The storage cost at the server is no more than $N \cdot (2 + B)$ bits, which is $O(N \cdot B)$ because: (i) there is no dummy data stored on the server; (ii) the size of the bitmap in bits is

$$\sum_{l=0}^{L-1} bitmap(n_l) \leq \sum_{l=0}^{L-1} 2 \cdot n_l = \sum_{l=0}^{L-1} 2^{l+1} \log N = 2 \log N \cdot (2^L - 1) \leq 2 \log N \cdot 2^L \leq \ 2N.$$

Storage cost at each proxy Due to the need to perform data shuffling, the storage cost at each proxy is $O(\sqrt{N \log N} \cdot b)$ bits, where b is the size of each data piece. Note that, in practical settings [65] where $N \leq 2^{32}$, the cost is no larger than 2 GB.

Storage cost at each user For each user, the storage cost is only $O(B + \sqrt[4]{N})$ bits, which is $O(B)$ due to the assumption that $B \geq \sqrt[4]{N}$ bits.

7.4.2 Communication Costs

The communication costs of each user, each proxy and the server are studied in this subsection.

Communication cost of each user Each user is involved only in the query process. For step [Q3], the user needs to retrieve the bitmap from each layer. To retrieve the bitmap from layer l, $4\sqrt[4]{n_l}$ bits will be transferred between the server and the user. Thus, the total number of transferred bits for the bitmap is

$$\sum_{l=0}^{L-1} 4\sqrt[4]{n_l} = \sum_{l=0}^{L-1} 4\sqrt[4]{2^{l+1} \log N} = 4\sqrt[4]{2 \log N} \sum_{l=0}^{L-1} 2^{l/4}$$

$$= \ 4\sqrt[4]{2 \log N} \frac{2^{L/4} - 1}{2^{1/4} - 1} \leq 22\sqrt[4]{2 \log N} \cdot 2^{L/4} = 22\sqrt[4]{N}.$$

Because there are at most L non-empty layers, the cost of bitmap is at most $22L\sqrt[4]{N}$ bits. For step [Q4], the user needs to retrieve two buckets from each non-empty layer. In step [Q5], since each bucket may contain up to $\log N$ data blocks, the maximum number of data blocks retrieved is $2L \log N$ (i.e., $2L \log N \cdot B$ bits). During data uploading phase, at most $2L$ data blocks are uploaded to the proxy chain and then uploaded to the server, which incurs $2L \cdot B$ bits communication cost. Therefore, the total number of bits transferred to the user during

each data query is no more than $O(\log^2 N \cdot B + \log N \cdot \sqrt[4]{N})$ bits. According to the assumption of $B \geq \sqrt[4]{N}$ bits, the cost is $O(\log^2 N \cdot B)$ bits.

Communication cost of each proxy During data query process, each data block needs to go through each proxy. Thus, each proxy's communication cost for query is the same as a user's communication cost, which is $O(\log^2 N \cdot B)$.

Next, we analyze the communication cost for data shuffling. First of all, as the proxies local storage is large enough to scramble all data blocks from the first layer, the communication cost for data scrambling I on the first layer is $2 \log N \cdot B$.

Data shuffling for layer l is triggered when the total number of data blocks on layers 0 to $l-1$ exceeds the total number of buckets on layer $l-1$. Also, each query process moves up to $\log N$ data blocks to the top layer. Hence, the frequency for data shuffling occurring for layer l is at most once per $n_l / \log N$ queries.

The communication cost incurred to each proxy during a query process is

$$S(n_l) = n_l(\lceil \log \log n_l \rceil + 1) \cdot B.$$

Therefore, the amortized communication cost for data shuffling is bounded by:

$$\begin{aligned}
& \sum_{l=0}^{L-1} \frac{S(n_l)}{n_l / \log N} = \sum_{l=0}^{L-1} \log N \cdot (\lceil \log \log n_l \rceil + 1) \cdot B \\
\leq & \sum_{l=0}^{L-1} \log N \cdot (\log \log N + 1) \cdot B \\
< & \sum_{l=0}^{\log N} \log N \cdot (\log \log N + 1) \cdot B \\
= & \ O(\log^2 N \log \log N \cdot B).
\end{aligned}$$

Overall, data shuffling communication cost is $O(\log^2 N \log \log N \cdot B)$ bits.

Communication cost of the storage server The communication cost of the storage server is no more than the sum of the costs of each user and each proxy. Hence, the communication cost of the server is $O(\log^2 N \log \log N \cdot B)$ bits.

7.4.3 Cost Comparison

Table 7.1 compares MU-ORAM with a couple of representative ORAM constructions, namely, B-ORAM [40] and Path ORAM [66]. B-ORAM is the most communication-efficient hash-based ORAM construction; Path ORAM is the most communication-efficient index-based O-RAM that does not require the server to conduct intensive computation. Note that, other ORAM constructions have been briefly introduced and compared to MU-ORAM in Section 6.

Table 7.1 Cost Comparison. N is the total number of data blocks outsourced to the storage server, B is the size of a data block ($B \geq \sqrt[4]{N}$), and b is the size of a data piece.

Cost		B-ORAM	Path ORAM	MU-ORAM
Query Comm.	User	$O(B\frac{\log^2 N}{\log\log N})$	$O(B\log N)\omega(1)$	$O(B\log^2 N)$
	Proxy	N/A	N/A	$O(B\log^2 N)$
Shuffle Comm.	User	$O(B\frac{\log^2 N}{\log\log N})$	$O(B\log N)\omega(1)$	N/A
	Proxy	N/A	N/A	$O(B\log^2 N\log\log N)$
User Storage		$O(B)$	$O(B\log N)\omega(1)$	$O(B)$
Proxy Storage		N/A	N/A	$O(b\sqrt{N\log N})$
Server Storage		$\geq 4N\cdot B$	$20N\cdot B$	$(B+2)N$

As shown in Table 7.1, MU-ORAM incurs higher communication cost compared to both B-ORAM and Path ORAM, which is the cost to support multi-user ORAM model and deal with the stealthy privacy attacks.

7.5 Summary

In this work, we propose MU-ORAM, a new ORAM construction to deal with stealthy privacy attack in the application scenarios where multiple users share a data set outsourced to a remote storage server and meanwhile want to protect each individual's data access pattern from being revealed to one another. We propose new security definitions for MU-ORAM, design data storage, query and shuffling algorithms, and conduct extensive security and cost analysis to evaluate the security properties as well as the communication and storage costs of the design.

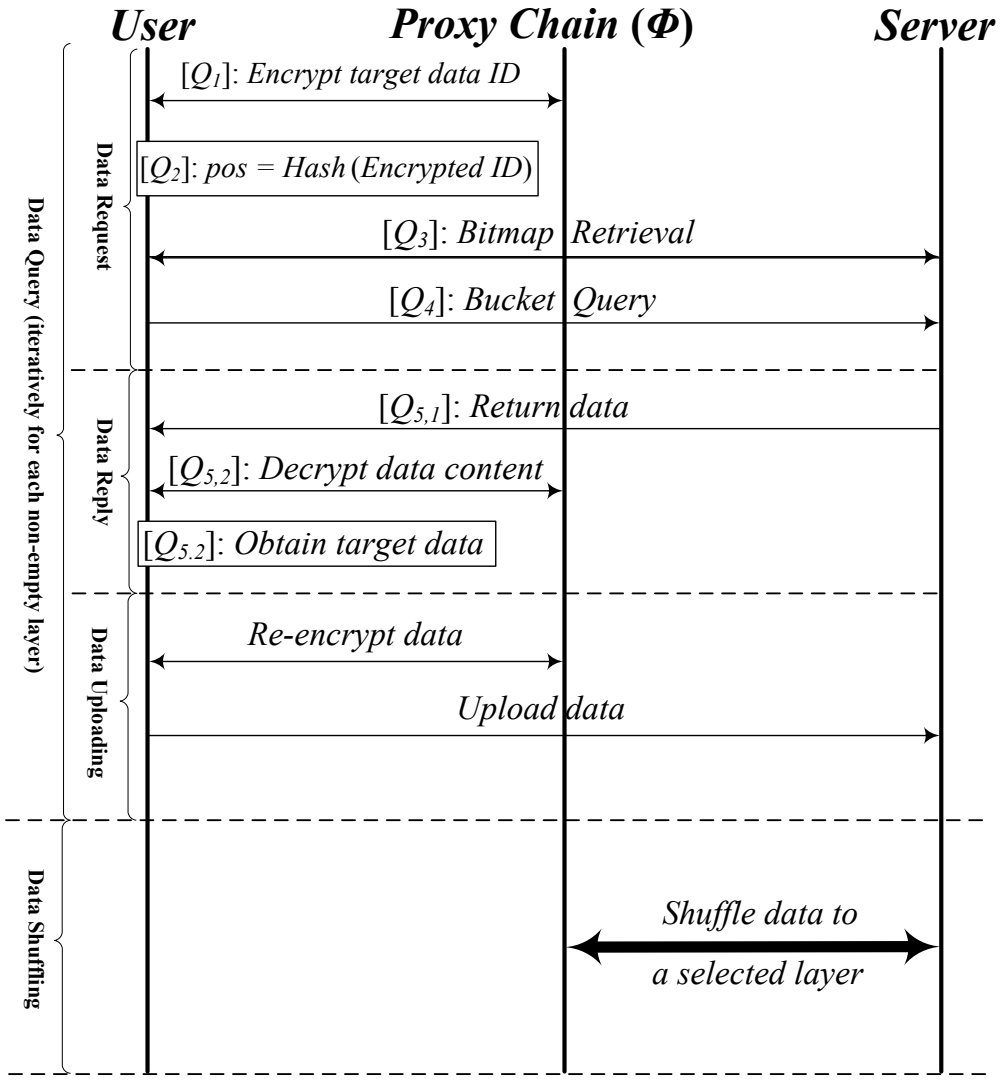


Figure 7.2 MU-ORAM Overview. The data query process includes the three phases of data request, data reply and data uploading, which is followed by the data shuffling process.

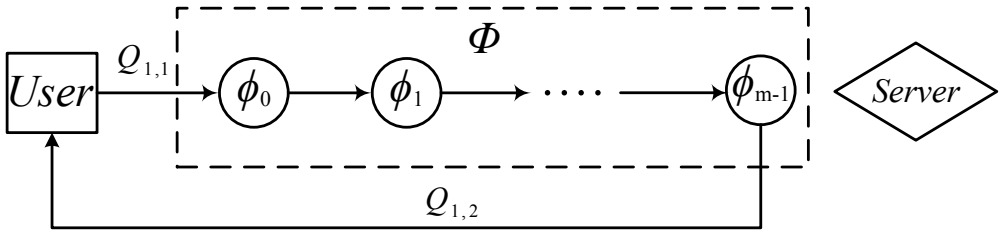


Figure 7.3 [Q1]: Obtain encrypted target data ID.

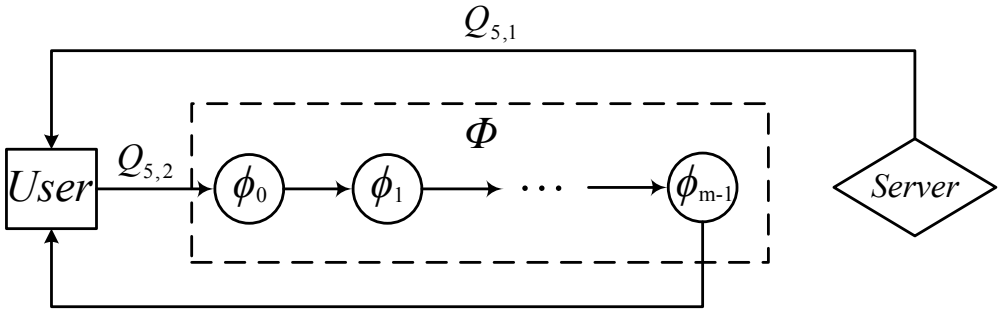


Figure 7.4 Phase 2: Data reply.

CHAPTER 8. CONCLUSIONS AND FUTURE WORKS

8.1 Conclusions

In this dissertation, we have presented four novel Oblivious RAM solutions to improve the state-of-the-art Oblivious RAM performance and study the feasibility of multi-user ORAM. We have rigorously proved their security and demonstrated their asymptotical efficiency. We have also shown their practical performances through numerical analysis. The main contributions of our work are:

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

- Secondly, we present 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 preserve the data access pattern privacy with a small failure probability of $O(N^{-\log\log N})$ where N is the number of exported data blocks. With a constant-size user storage and $k = \log N$, KT-ORAM has an asymptotical communication cost of $O(\frac{\log N}{\log\log N} \cdot B)$ when the recursion level on metadata is of $O(1)$ depth with uniform block size $B = N^{\epsilon}$ $(0 < \epsilon < 1)$. In addition, KT-ORAM outperforms

all these constructions in terms of communication and user-side storage costs, under practical scenarios.

- In the third work, a new ORAM called Generalized Partition ORAM (GP-ORAM) is presented. GP-ORAM utilizes a new shuffling method, adjusts the number of partitions according to the available user-side storage, and outsources the index table to the server. Through these techniques, it achieves low bandwidth cost ($O(\log N)$) and has significantly less user-side storage cost than P-ORAM. We demonstrate the effectiveness of GP-ORAM via extensive security and cost analysis.

- In the final work, we present MU-ORAM, a new ORAM construction to deal with stealthy privacy attack in the application scenarios where multiple users share a data set outsourced to a remote storage server and meanwhile want to protect each individual's data access pattern from being revealed to one another. We propose new security definitions for MU-ORAM, design data storage, query and shuffling algorithms, and conduct extensive security and cost analysis to evaluate the security properties as well as the communication and storage costs of the design.

8.2 Future Works

For the future work, there are multiple directions to work on. First of all, the feasibility of all existing ORAM systems in the practical cloud/data center will be a very challenging topic. For practical deployment, it is possible to have many underlying problems such as how to make backups of an existing ORAM system such that the disaster tolerance and availability can be guaranteed. In addition, the user access parallelism could be another issue. Secondly, as the cloud storage is usually physically distributed in distributed infrastructure, the multi-server ORAM systems will be more complicated and more interesting, including the issue of how to distributed workload among these servers. Thirdly, the server-side storage cost of existing ORAMs are usually high. For example, suppose the server storage cost is 10 times the storage required without ORAM systems as shown in Path ORAM [66]. When a company outsources 1 PB data, 10 PB storage space is needed for Path ORAM. It would be a very practical issue

to reduce the storage cost on the server. At last, for multi-user ORAM systems, the security strength and efficiency improvement is also a potential and challenging problem to solve.

BIBLIOGRAPHY

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

[2] Amazon (2006). Amazon S3. https://aws.amazon.com/s3/.

[3] Asonov, D. (2004). Querying databases privately: a new approach to private information retrieval. In *Springer Verlag*.

[4] Bao, F., Deng, R. H., and Zhu, H. (2003). Variations of Diffie-Hellman problem. In *LNCS*. Springer.

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

[6] Brumley, D. and Boneh, D. (2003). Remote timing attacks are practical. In *Proc. USENIX Security*.

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

[8] Chen, B., Lin, H., and Tessaro, S. (2015). Oblivious Parallel RAM: Improved efficiency and generic constructions. In *IACR Cryptology ePrint Archive*. International Association for Cryptologic Research.

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

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

[11] ComputerWeekly (2012). Investigation reveals serious cloud computing data security flaws. http://www.computerweekly.com/news/2240148943/Investigation-reveals-serious-cloud-computing-data-security-flaws.

[12] Daemen, J. and Rijmen, V. (2002). *The design of Rijndael*. Springer-Verlag New York, Inc.

[13] Damgard, I., Meldgaard, S., and Nielsen, J. B. (2011). Perfectly secure Oblivious RAM without random oracles. In *Proc. TCC*.

[14] Dautrich, J. and Ravishankar, C. (2015). Combining ORAM with PIR to minimize bandwidth costs. In *Proc. CODASPY*.

[15] Dautrich, J., Stefanov, E., and Shi, E. (2014). Burst ORAM: Minimizing ORAM response times for bursty access patterns. In *Proc. USENIX Security*.

[16] Devadas, S., van Dijk, M., Fletcher, C. W., Ren, L., Shi, E., and Wichs, D. (2015). Onion ORAM: A constant bandwidth blowup Oblivious RAM. In *IACR Cryptology ePrint Archive*. International Association for Cryptologic Research.

[17] Dropbox (2006). http://www.dropbox.com/. In *Dropbox*.

[18] Dubhashi, D. and Ranjan, D. (1996). Balls and bins: a study in negative dependence. *Random Structures and Algorithms*, 13.

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

[20] Fletcher, C., Naveed, M., Ren, L., Shi, E., and Stefanov, E. (2015). Bucket ORAM: Single online roundtrip, constant bandwidth Oblivious RAM. In *IACR Cryptology ePrint Archive*. International Association for Cryptologic Research.

[21] Fletcher, C. W., Ren, L., Kwon, A., Dijk, M. V., Stefanov, E., and Devadas, S. (2014). Tiny ORAM: A low-latency, low-area hardware ORAM controller. In *IACR Cryptology ePrint Archive*. International Association for Cryptologic Research.

[22] Franz, M., Williams, P., Carbunar, B., Katzenbeisser, S., Andreas, P., Sion, R., and Sotakova, M. (2012). Oblivious outsourced storage with delegation. In *Proc. FC*.

[23] Gasarch, W. (2004). A survey on private information retrieval. In *Online at http://crypto.stanford.edu/d̃abo/courses/cs355_fall04/pir.pdf*.

[24] Gentry, C., Goldman, K., Halevi, S., Julta, C., Raykova, M., and Wichs, D. (2013). Optimizing ORAM and using it efficiently for secure computation. In *Proc. PETS*.

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

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

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

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

[29] Goodrich, M. T. and Mitzenmacher, M. (2010). Mapreduce parallel cuckoo hashing and Oblivious RAM simulations. In *Proc. CoRR*.

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

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

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

[33] Google (2012). Google Drive. https://www.google.com/drive/.

[34] Group, N. W. (2000). HTTP Over TLS. In *RFC 2818*.

[35] Group, N. W. (2011). The secure sockets layer (SSL) protocol version 3.0. In *RFC 6101*.

[36] Handschuh, H., Tsiounis, Y., and Yung, M. (1999). Decision oracles are equivalent to matching oracles. In *Proc. PKC*.

[37] Helger, L. and Bingsheng, Z. (2010). Two new efficient PIR-writing protocols. In Zhou, J. and Yung, M., editors, *Applied Cryptography and Network Security*, volume 6123 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg.

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

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

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

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

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

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

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

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

[46] Ma, Q., Zhang, J., Zhang, W., and Qiao, D. (2016). SE-ORAM: A storage-efficient Oblivious RAM for privacy-preserving access to cloud storage. In *IACR Cryptology ePrint Archive*. International Association for Cryptologic Research.

[47] Maffei, M., Malavolta, G., Reinert, M., and Schroder, D. (2015). GORAM – Group ORAM for privacy and access control in outsourced personal records. In *IACR Cryptology ePrint Archive*. International Association for Cryptologic Research.

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

[49] Mayberry, T., Blass, E.-O., and Noubir, G. (2015). Multi-client Oblivious RAM secure against malicious servers. In *IACR Cryptology ePrint Archive*. International Association for Cryptologic Research.

[50] Moataz, T., Blass, E.-O., and Mayberry, T. (2015a). Constant communication ORAM without encryption. In *IACR Cryptology ePrint Archive*. International Association for Cryptologic Research.

[51] Moataz, T., Blass, E.-O., and Noubir, G. (2015b). Recursive trees for practical ORAM. In *Proc. FC*.

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

[53] Moataz, T., Mayberry, T., Blass, E.-O., and Chan, A. H. (2014). Resizable tree-based Oblivious RAM. In *IACR Cryptology ePrint Archive*. International Association for Cryptologic Research.

[54] News, T. H. (2014). Google Drive vulnerability leaks users' private data. http://thehackernews.com/2014/07/google-drive-vulnerability-leaks-users_9.html.

[55] NIST (2013). Block cipher modes. http://csrc.nist.gov/groups/ST/toolkit/BCM/index.html.

[56] Ostrovsky, R. and III, W. E. S. (2007). A survey of single-database PIR: techniques and applications. In *Online at http://eprint.iacr.org/2007/059.pdf*.

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

[58] Ren, L., Fletcher, C. W., Kwon, A., Stefanov, E., Shi, E., van Dijk, M., and Devadas, S. (2014a). Ring ORAM: Closing the gap between small and large client storage Oblivious RAM. In *IACR Cryptology ePrint Archive*. International Association for Cryptologic Research.

[59] Ren, L., Fletcher, C. W., Yu, X., Kwon, A., van Dijk, M., , and Devadas, S. (2014b). Unified Oblivious-RAM: Improving recursive ORAM with locality and pseudorandomness. In *IACR Cryptology ePrint Archive*. International Association for Cryptologic Research.

[60] Ren, L., Fletcher, C. W., Yu, X., van Dijk, M., and Devadas, S. (2013). Integrity verification for Path Oblivious-RAM. In *Proc. HPEC*.

[61] Shi, E., Chan, T.-H. H., Stefanov, E., and Li, M. (2011). Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *Proc. ASIACRYPT*.

[62] Sion, R. and Carbunar, B. (2007). On the practicality of private information retrieval. In *In Proc. NDSS*.

[63] Stefanov, E. and Shi, E. (2013a). Multi-cloud oblivious storage. In *Proc. CCS*.

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

[65] Stefanov, E., Shi, E., and Song, D. (2011). Towards practical Oblivious RAM. In *Proc. NDSS*.

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

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

[68] Wang, X., Chan, T.-H. H., and Shi, E. (2015). Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. In *Proc. CCS*.

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

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

[71] Williams, P. and Sion, R. (2008b). Usable PIR. In *Proc. NDSS*.

[72] Williams, P. and Sion, R. (2013). Access privacy and correctness on untrusted storage. In *Proc. TISSEC*.

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

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

[75] Yu, X., Ren, L., Fletcher, C. W., Kwon, A., van Dijk, M., and Devadas, S. (2014). Enhancing Oblivious RAM performance using dynamic prefetching. In *IACR Cryptology ePrint Archive*. International Association for Cryptologic Research.

[76] Zhang, J., Ma, Q., Zhang, W., and Qiao, D. (2014a). KT-ORAM: A bandwidth-efficient ORAM built on k-ary tree of PIR nodes. In *IACR Cryptology ePrint Archive*. International Association for Cryptologic Research.

[77] Zhang, J., Zhang, W., and Qiao, D. (2014b). S-ORAM: A segmentation-based Oblivious RAM. In *Proc. AsiaCCS*.

[78] Zhang, J., Zhang, W., and Qiao, D. (2015). GP-ORAM: A generalized Partition ORAM. In *Proc. NSS*.

[79] Zhang, J., Zhang, W., and Qiao, D. (2016). MU-ORAM: Dealing with stealthy privacy attacks in multi-user data outsourcing services. In *IACR Cryptology ePrint Archive*. International Association for Cryptologic Research.