Towards efficient and accountable oblivious cloud storage

Qiumao Ma
Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/etd

Part of the Computer Sciences Commons

Recommended Citation
Ma, Qiumao, "Towards efficient and accountable oblivious cloud storage" (2018). Graduate Theses and Dissertations. 17264.
https://lib.dr.iastate.edu/etd/17264

This Dissertation is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
Towards efficient and accountable oblivious cloud storage

by

Qiumao Ma

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
Yong Guan
Simanta Mitra
Daji Qiao
Lu Ruan

The student author, whose presentation of the scholarship herein was approved by the program of study committee, is solely responsible for the content of this dissertation. The Graduate College will ensure this dissertation is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University
Ames, Iowa
2018

Copyright © Qiumao Ma, 2018. All rights reserved.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF TABLES</td>
<td>v</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>vi</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>viii</td>
</tr>
<tr>
<td>CHAPTER 1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>CHAPTER 2. PROBLEM STATEMENT</td>
<td>8</td>
</tr>
<tr>
<td>CHAPTER 3. LITERATURE REVIEW</td>
<td>10</td>
</tr>
<tr>
<td>3.1 Single-server ORAMs</td>
<td>10</td>
</tr>
<tr>
<td>3.2 Multi-server ORAMs</td>
<td>13</td>
</tr>
<tr>
<td>CHAPTER 4. SE-ORAM</td>
<td>17</td>
</tr>
<tr>
<td>4.1 Research Goal and Rationales</td>
<td>17</td>
</tr>
<tr>
<td>4.1.1 Results</td>
<td>19</td>
</tr>
<tr>
<td>4.1.2 Organization</td>
<td>19</td>
</tr>
<tr>
<td>4.2 Security Definition</td>
<td>19</td>
</tr>
<tr>
<td>4.3 The SE-ORAM Construction</td>
<td>21</td>
</tr>
<tr>
<td>4.3.1 Storage Organization and Initialization</td>
<td>21</td>
</tr>
<tr>
<td>4.3.2 Data Query</td>
<td>22</td>
</tr>
<tr>
<td>4.3.3 Data Eviction</td>
<td>24</td>
</tr>
<tr>
<td>4.3.4 Extra Query-Eviction Round</td>
<td>27</td>
</tr>
<tr>
<td>4.4 Security Analysis</td>
<td>27</td>
</tr>
</tbody>
</table>
4.5 Cost Analysis ............................................................. 29
   4.5.1 Storage Overhead ................................................ 29
   4.5.2 Communication Overhead .................................... 33
   4.5.3 Performance Comparison .................................... 34
4.6 Summary ............................................................... 35

CHAPTER 5. OCTOPUS ORAM ........................................... 36
   5.1 Research Goal and Contributions .............................. 36
   5.2 Problem Definition ............................................... 36
   5.3 The Proposed ORAM Scheme ................................. 37
      5.3.1 Server-side Storage ........................................ 39
      5.3.2 Client-side Storage ....................................... 39
      5.3.3 System Initialization ..................................... 41
      5.3.4 Data Query ................................................ 41
      5.3.5 Data Eviction .............................................. 43
   5.4 Security Analysis ................................................ 46
      5.4.1 Failure Probability Analysis ............................ 46
      5.4.2 Obliviousness Analysis .................................. 52
   5.5 Performance Comparison ....................................... 54
      5.5.1 Comparison with Partition ORAM ....................... 55
      5.5.2 Comparison with Path ORAM ............................ 56
   5.6 Summary ............................................................. 59

CHAPTER 6. THREE-SERVER OCTOPUS ORAM ....................... 61
   6.1 Research Goal and Contributions .............................. 61
   6.2 System Architecture and Initialization ..................... 62
      6.2.1 Storage Organization at Server $S_0$ .................... 62
      6.2.2 Storage Organization at the Client ........................ 64
      6.2.3 System Initialization .................................... 65
6.3 Data Query Algorithm .......................................................... 65
   6.3.1 Phase I: Selecting Data Blocks to Access ...................... 66
   6.3.2 Phase II: Retrieving Query Target .............................. 68
6.4 Data Eviction Algorithm .................................................. 68
6.5 Accountability Enhancements .......................................... 71
   6.5.1 Enhancements to Storage and System Initialization .......... 72
   6.5.2 Enhancement to Data Query Algorithm ....................... 72
   6.5.3 Enhancements to Data Eviction Algorithm .................... 73
6.6 Security Analysis ............................................................ 73
   6.6.1 Obliviousness Analysis ......................................... 74
   6.6.2 Failure Analysis ................................................. 75
   6.6.3 Accountability Analysis ....................................... 77
6.7 Performance Evaluation and Comparisons ............................ 78
   6.7.1 System Settings .................................................. 79
   6.7.2 Comparison with S³ORAM ..................................... 80
6.8 Summary ................................................................. 82

CHAPTER 7. CONCLUSIONS AND FUTURE WORKS ......................... 84

BIBLIOGRAPHY ................................................................. 86
LIST OF TABLES

Table 3.1  Comparisons of State-of-the-art Single Oblivious RAM Constructions. $N$ denotes the total number of exported data blocks, $B$ denotes the size of each data block. Note that, for index-based ORAMs, we ignored the index table which takes $O(N \cdot \log N)$ bits of client-side storage. Also we only consider non-recursive version for the ORAM that supports recursive version.

Table 3.2  Comparisons of State-of-the-art Multi-Server Oblivious RAM Constructions. $N$ denotes the total number of exported data blocks, $B$ denotes the size of each data block. Note that, for index-based ORAMs, we ignored the index table which takes $O(N \cdot \log N)$ bits of client-side storage.

Table 4.1  Storage and Communication Overheads. $N$: number of data blocks; $B$: block size in bits. Server storage overhead is defined as the server’s storage consumption other than the $NB$ bits for the real data blocks. Client storage overhead is defined as the client’s storage consumption other than the index table for the exported data blocks.

Table 5.1  Practical Performance Comparisons

Table 5.2  Comparing Storage Efficiency between Octopus ORAM and Path ORAM

Table 6.1  System Costs with Varying $m$
LIST OF FIGURES

Figure 1.1 Access Pattern Attack Model .................................................. 2

Figure 4.1 Query Examples. In (a), query target $B_t$ is at node $n_{2,2}$ and has path ID $n_{3,4}$. Node $n_{3,4}$ exists on the tree and has two supplementary nodes. The client requests the server to retrieve nodes from the root to $n_{5,4}$ which is the further supplementary node of $n_{3,4}$. Then, $B_L$ obliviously replaces $B_t$; finally, as node $n_{5,4}$ becomes empty after $B_L$ has moved, the node is removed from the tree. In (b), query target $B_t$ is at node $n_{1,0}$ and has path ID $n_{3,1}$. Node $n_{3,1}$ does not exist on the tree. The client requests the server to retrieve nodes on the path from the root to $n_{4,0}$, which is the longest path that has the largest overlap with the path from the root to $n_{3,1}$. Then, the client obliviously replaces $B_t$ with $B_L$. ........................................ 23

Figure 4.2 Data eviction example. (a) Evicting data block from layer 0 to 1: Evicted block $B_e$ has path ID $n_{3,7}$ but the evicting node $n_{0,0}$ chooses to evict to left. Hence, $B_e$ is written obliviously to $n_{0,0}$, while a block with path ID $n_{3,0}$ (and therefore evictable to left) is selected from $n_{0,0}$ to the new evicted block. This is Case II in Section 4.3 E4. (b) Evicting data block from layer 1 to 2: Evicting node $n_{1,0}$ chooses to evict to right. No block (including $B_e$ and the blocks in $n_{1,0}$) is evictable to right. Hence, a dummy block is created to replace a randomly-selected real block in $n_{1,0}$, which is moved to
the cache of the client. The dummy block then becomes the new evicted block. This is Case IV in Section 4.3 E4. (c) *Evicting data blocks from layer 2 to 3:* As the evicted block $B_e$ is a dummy, it remains as the evicted block no matter whether the evicting node chooses to evict to left or right. This is Case I in Section 4.3 E4. (d) *Evicting data blocks from layer 3 to 4:* The evicting node is non-full. So the evicted block is written to it obliviously and the eviction process terminates, as explained in Section 4.3 E4.

Figure 4.3 Node State and Transition .......................................................... 31
Figure 4.4 State Transition of Dummy Block Number $N_d$ .......................... 32
Figure 5.1 Server Storage Organization .......................................................... 40
Figure 5.2 Query Examples ............................................................................ 43
Figure 5.3 Eviction Overview .......................................................................... 45
Figure 5.4 Comparing Communication Cost per Query Between Octopus ORAM and Path ORAM ................................................................. 58
Figure 5.5 Comparing Query Delay Between Octopus ORAM and Path ORAM. .... 59
Figure 5.6 Comparing the Processing Time per Query between Octopus ORAM and Path ORAM ................................................................. 60

Figure 6.1  Three-server Octopus ORAM Architecture .................................... 63
Figure 6.2 Reverse-lexicographic Order: Every eviction process involves one root-to-leaf eviction path selected in the reverse-lexicographic order. ............... 69
Figure 6.3 A High-level Illustration of Eviction Process ................................. 71
Figure 6.4 Inter-Server Communication Cost ................................................. 81
Figure 6.5 Query Delay ................................................................................... 82
Figure 6.6 Server Storage Overhead ................................................................. 83
ABSTRACT

Due to the convenience and unprecedented 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.

Though encryption helps to secure data, the server or the attacker who compromise the server is still able to infer private information from the user’s access pattern. It is possible for an attacker to use the access pattern information to reconstruct the data query and infer the plaintext of the data. Hence, a large variety of schemes based on the oblivious RAM (ORAM) model have been proposed to allow a user to access the exported data while preserving user’s data access pattern. Most of these research has focused on the communication efficiency improvement, but the storage efficiency has not received much attention. To host $N$ data blocks, in general, the state-of-the-art ORAM constructions need the storage server to also store $cN$ with $c > 3$ or $O(N \cdot \log N)$ dummy data blocks, which represents a huge storage overhead when $N$ is large. In addition to the inefficiency in server storage, most of existing ORAM constructions incur $O(\log N)$ blocks or higher client-server communication cost. Though some recent work has reduced the cost to $O(1)$ blocks by employing multiple non-colluding servers, the system could become vulnerable if some server does not follow the protocol completely.
To address the above limitations, we develop a series of new ORAM constructions, gradually towards a more practical and secure solution that can obliviously protect the data access pattern for users of cloud storage with more affordable storage, client-server communication, and server-server communication overheads. Specifically, this dissertation presents:

- **SE-ORAM**, which reduces server storage overhead to zero, but at the same time, incurs a client server communication cost of $O(\log^2 N)$ blocks;

- **Octopus ORAM**, which incurs $0.34N \cdot B$ server storage overhead, and reduces client-server communication cost to three blocks for query and about $1.5 \log N$ blocks for eviction per query;

- **Three-server Octopus ORAM**, an efficient and accountable multi-server ORAM, which incurs $0.3N \cdot B$ server storage overhead and reduces client-server communication cost to $O(1)$ blocks, at the expense of server-server communication cost at $O(\log N)$ blocks per query.

We have rigorously quantified and proved the security strengths of these constructions and demonstrated their performance efficiency through detailed analysis.
CHAPTER 1. INTRODUCTION

As cloud computing has been a common computing paradigm, cloud storage also become pervasive. In the cloud storage model, a public cloud storage provider owns and hosts a physical storage that may span over multiple physical servers and locations; a client, which could be an organization or individual, leases storage capacity to store data. Due to the convenience and unprecedented 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. Data encryption has been common for data privacy protection, but it cannot protect data access patterns for cloud storage clients. As in many cloud storage systems [12], data is stored in the unit of block. Before outsourcing, each data block can be encrypted using some probabilistic encryption method such as AES [8] 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. A curious owner or employee of a cloud storage service, or an intruder invading the storage server, can observe a client’s data access pattern. As shown in figure 1, the observed pattern and the client’s activities that could be obtained through some side channels, the attacker could develop a model relating them. Later on, the attacker may use the model and newly observed access patterns to infer or predict the client’s activities. As illustrated by Islam et. al. [27], 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, exposed data access pattern can potentially reveal some private information about cloud storage clients. Especially, military, homeland security and public safety agencies should protect these private information from the enemies; businesses should protect the information from their rivals and competitors.

Researchers have been exploring ways to protect users’ access patterns and various schemes have been proposed in the literature. Among them, Oblivious RAM (ORAM) [19, 58, 59, 20, 44, 21, 22, 62, 28, 32, 13, 60, 49, 53, 16, 54, 40, 46, 56, 67, 57, 37, 41, 52, 42, 9, 38, 14, 63, 5, 11, 64, 51, 39, 47, 33] and Private Information Retrieval (PIR) [1, 50, 43, 7, 2, 30, 17, 18, 25, 31, 6, 4, 55, 29, 15] 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 in Chapter 2. In recent years, interests in ORAM research have increased, and many ORAM constructions have been proposed targeting at making ORAM practical. In general, ORAM performance can be measured 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.

Most of these research have focused on the communication efficiency improvement, but the storage efficiency has not received much attention. The prioritization on communication efficiency over server storage efficiency is based on that, the price for communication is much higher than

![Figure 1.1 Access Pattern Attack Model](image-url)
that for storage. This is true; taking the Amazon S3 service in North America as example, the price for transferring data from Amazon to Internet is at least $0.05 per GB, while storing 1 GB data for one month only costs $0.02. Nevertheless, we observe that, when the cloud server needs to store a large amount of data, the momentary cost for storage could be comparable to or even exceed that for communication. For example, consider a client uses high-speed Internet to connect with AWS S3, and the network bandwidth is 1 Gbps. Supposing the client keeps accessing data from the server and the bandwidth is fully utilized for the accessing, the amount of data that can be transferred from the server to the client is no more than 324 TB per month, which costs about $16,200, roughly the cost for the client to store 800 TB data at the server. That is, as long as the client has data with several hundreds of TB or more to store at the server, the monetary cost for storage has been comparable to or even higher than the monetary cost for communication. The weight of storage cost could be even higher in practice, because the client may not access the outsourced data too frequently as frequently-accessed data could be cached locally. Hence, reducing the storage overhead is also imperative.

To address the problem with client-server communication, some researchers introduce multiple (at least one) non-colluding servers. For example, S⁴ORAM [26] uses at least three servers and incurs $O(B)$ bandwidth consumption for client-server communication, where $B$ is the block size in unit of bits, which is similar to the cost incurred by a non-oblivious storage system; it still requires $O(\log N \cdot B)$ bandwidth consumption for communication between the servers, but the inter-server communication occurs behind the scene and does not consume client-server bandwidth. This is based on the assumption that the bandwidth between servers is more abundant than that between client and server and the cost between servers is lower then between client and server. However, there are several limitations with these schemes. First, existing ORAM schemes require multiple servers each storing a copy of the outsourced data blocks, which significantly increases the server storage cost, for example, S⁴ORAM needs to store $12N$ blocks for every $N$ real data blocks exported. Second, these schemes assume the servers to be semi-honest, which may not be realistic in practice; a dishonest server could deviate from the designated protocol, and if not detected immediately, could
lead to big overhead to recover the system. Hence, it is also important to build an accountable multiple server ORAM scheme with server storage cost efficiency and client-server communication efficiency.

In this dissertation, we aim to improve the server storage efficiency and at the same time improve communication efficiency with the following approaches. We propose a series of three new ORAM constructions to achieve better performance than state-of-the-art. In the following, we provide a brief overview of the motivations, key design ideas, and performance of these constructions.

First, we propose SE-ORAM: A Storage-Efficient Oblivious RAM. As improving server storage efficiency is one of the most important targets of our work, the first study aims to design an ORAM construction with zero storage overhead at the server, while communication efficiency is similar or higher than state-of-the-art.

We design an storage efficient ORAM (SE-ORAM) with three novel technologies: eviction with non-uniform probabilities, On-demand Introduction of Dummies and Periodical Removal of Dummies.

- With Eviction with Non-uniform Probabilities, during eviction, the next evicting node and evicting blocks is decided based on the state of the current evicting node, i.e., a larger probability to evict data to its left child if more of its data blocks are evictable to left, and vice versa. This way, the chance could be significantly reduced for the failure situation to occur.

- With On-demand Introduction of Dummies, a dummy block (evictable to both left and right) is inserted on demand to replace a real data block, which is moved to the client’s cache. Note that, the storage server still stores the same number of data blocks, though some of the blocks become dummies.

- With Periodical Removal of Dummies, an extra query and eviction process is launched periodically to retrieve and discard a dummy from the server and evict a real data block from the client to the server. As result, we can bound the number of dummy blocks on the server as well as prevent the user side cache from overflowing.
Extensive security proofs have been conducted to demonstrate the security of SE-ORAM. SE-ORAM is secure with parameter $\lambda$, the probability is at most $(\frac{1}{N^\lambda})^{1-\frac{1}{x}}$ for the server to correctly infer a sequence with $n$ data accesses from any storage location access sequence. This is useful in practice, particularly when a large number of data blocks are outsourced. In terms of server storage efficiency, SE-ORAM stores exactly $N$ number of data blocks on server side, which has zero server storage overhead. Note that, the server stores a small number of dummy blocks, and the real block replaced by dummy block is stored in user side storage. The number of introduced dummy blocks is no more than $x \log N$ with probability $1 - \frac{1}{N^x}$, as long as $\lambda \geq 2$ and each node on the storage tree can store $4 \log N$ or more data blocks. SE-ORAM incurs communication cost of $O(\log^2 N \cdot B)$ where $B$ is block size in unit of bits, which is higher than some state-of-the-art ORAM constructions.

Second, we propose Octopus ORAM: A Storage Efficient 8-ary Tree ORAM. In SE-ORAM, we reduced the server storage overhead to zero, but the client-server communication cost is $O(\log^2 N \cdot B)$, which is higher than some of the state-of-the-art. We propose a new ORAM construction called Octopus ORAM, aiming at reducing the client-server communication cost in the cost of introducing a small amount of server storage overhead, which is much lower than existing ORAM constructions.

- With organizing the server storage as 8-ary tree instead of binary tree, we make the non-leaf nodes to have much smaller storage than the leaf nodes.

- With deliberately making the size of each leaf node to be very large, this work requires only a small fraction of redundant space to ensure a negligible probability of space overflow, hence minimize the storage space of leaf nodes.

- With the newly designed data query algorithm makes the communication cost per query to be a constant independently of either the node size or the height of the tree. Based on a carefully selected value $k$ (i.e., eight) that balances the tradeoff between storage and communication efficiency, we design an eviction scheme that runs after a certain large number of queries to evict a large batch of data blocks at once.
Extensive security proofs have been conducted to prove Octopus ORAM is secure under our security definition. In terms of performance, our proposed ORAM can significantly reduce the server storage overhead to around $0.34N$ (i.e., the server only needs to allocate $1.34N$ blocks when the client outsources $N$ blocks) while maintaining a comparable level of communication cost: to server a query, the online communication cost is 3 blocks and eviction (maintenance) communication cost is no more than $1.5 \log N$ blocks.

Third, we propose Three Servers Octopus ORAM: An efficient and Accountable $k$-ary Tree Storage Efficient ORAM. Introducing multiple servers is a well-known approach to reduce client-server communication cost. But in existing multi-server ORAM constructions, server storage is organized exactly same or almost same, which also multiply the server storage cost. For example, in $S^3$ORAM [26], the $t$ ($t \geq 3$) servers’ storage are organized in same structure, the only difference is each server stores a difference secret-shared part of a data block. As each server needs to store $4N$ data block, the overall server storage cost is tripled, i.e., the server storage cost of $S^3$ORAM is $12N \cdot B$.

To take the advantages of multi-server ORAM construction while at the same time keep the server storage efficiency, we propose a new three server ORAM, which shares some ideas as Octopus ORAM. $S_0$ stores user-data blocks as well as small amount of dummy blocks, $S_0$ is organized similar to Octopus ORAM but the number of child nodes is configurable. $S_1$ and $S_2$ only temporarily store a small number of blocks to facilitate the query and eviction processes.

- With $S_0$ organized similar to Octopus ORAM, and $S_1$ and $S_2$ only store a small number of blocks temporarily, we keep the server storage cost only a little bit higher than Octopus ORAM, instead of triple the server storage size.

- With $S_1$ and $S_2$ facilitate the query and eviction, the client only needs to transfer target block and some metadata between client and servers, which reduces client-server communication cost.
• With the authentication mechanism, the non-colluding servers can detect any malicious server, while incurring small traffic between server and client, and requiring light server computational cost.

Extensive security proofs have been conducted to prove our work can protect clients’ access pattern privacy. As for the cost, this work incurs low server storage overhead, which is around $0.3N$ blocks for every $N$ real data blocks exported, the client-server communication cost is $O(B)$ bits per query by average, and server-server communication cost is $O(\log N \cdot B)$, lower communication costs than S$^3$ORAM, the most related state-of-the-art scheme. By supporting accountability with multiple servers, the work removes the less-realistic semi-honest assumption in a multi-server oblivious storage system; Note that, our system has made full use of the available moderate level of client-side storage, but the required storage capacity is still only as small as around 0.1% of the cloud server’s storage capacity.

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 and 6, we present our proposed three schemes. In Chapter 7, we conclude this dissertation with a summary of our main contributions and future research plans.
CHAPTER 2. PROBLEM STATEMENT

We consider a distributed system that consists of a client and one or multiple cloud servers. The client has an on-premise cloud storage gateway with a moderate storage capacity, though it still much smaller than the capacity of the cloud storage servers. The cloud servers are assumed to be non-colluding, however, the servers could be malicious and some accountability mechanisms will be deployed to each server to detect the misbehavior of other servers. The client is assumed to be honest; note that, this assumption could be removed by, for example, requiring the client to electronically sign each message and data block that it sends.

Assume the client outsources \( N \) data blocks each with the same size of \( B \) bits to the cloud storage server, and then needs to access the outsourced data every now and then.

Each data access intended by the client, which should be kept private, is of two types:

- Read a data block \( D \) of unique ID \( i \) from the storage, denoted as \( (\text{read}, i, D) \);
- Write a data block \( D \) of unique ID \( i \) to the storage, denoted as \( (\text{write}, i, D) \).

To hide a private data access, the client and servers need to access multiple locations of the server-side storage and exchange some messages with each other. Each location access or message exchange, which can be observed by the servers, is one of the following types:

- Retrieve (i.e., read) a data block \( D \) from location \( l \) at the storage, denoted as \( (\text{read}, l, D) \);
- Upload (i.e., write) a data block \( D \) to location \( l \) at the storage, denoted as \( (\text{write}, l, D) \);
- Send a message from one party to another (note: a party could be the client or a server), denoted as \( (\text{send}, s, d) \) where \( s \) and \( d \) are the source and destinations.

Extending the security definition of ORAM in prior works [19, 54, 53], we define the security of our proposed oblivious storage system as follows.
Definition Let $\lambda$ be a security parameter, and $\vec{x} = \langle (op_1, i_1, D_1), (op_2, i_2, D_2), \cdots \rangle$ denote a private sequence of the client’s data accesses, where each $op$ is either a read or write. Let $A(\vec{x}) = \langle (op'_1, p_{1,1}, p_{1,2}), (op'_2, p_{2,1}, p_{2,2}), \cdots \rangle$ denote the sequence of the location accesses or message exchanges (observed by the server) in order to accomplish the data access sequence $\vec{x}$. An oblivious storage system is secure if:

1. for any two equal-length private sequences $\vec{x}$ and $\vec{y}$ of data accesses, their corresponding location access and message exchange sequences $A(\vec{x})$ and $A(\vec{y})$ are computationally indistinguishable; and

2. the system fails to operate with a probability of $O(2^{-\lambda})$. 
CHAPTER 3. LITERATURE REVIEW

The concept of Oblivious RAM (ORAM) was first introduced by Goldreich and Ostrovsky [19], 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-server ORAMs and multi-server 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-server ORAMs

According to the adopted data lookup techniques, single-server ORAMs have two major classes, namely, hash-based ORAMs and index-based ORAMs.

In hash-based ORAMs [19, 58, 59, 20, 44, 21, 22, 62, 28, 32, 13, 60], 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 [19]) 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. [61] and Cuckoo Hash ORAM (CH-ORAM) by Pinkas et. al. [44] and Goodrich et. al. [23, 21, 20, 22]. 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. [28] 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 index-based ORAMs [49, 53, 16, 54, 40, 46, 56, 67, 57, 41, 52, 42, 9, 38, 14, 63, 5, 11, 64, 51, 39, 47, 33, 36, 34, 35], 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 index-based ORAM construction was proposed by Shi et. al. [49] 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-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 [54] 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. [47], makes integrity checking available in Path ORAM.

Partition ORAM[53] organizes the server storage as $\sqrt{N}$ partitions and each partition works as an ORAM module. The client storage is utilized to contain a location map for blocks, a buffer for storing and shuffling data blocks of an ORAM partition, and $\sqrt{N}$ stash slots. Based on this storage arrangement, together with optimizations in query and shuffling algorithm, the scheme incurs a communication cost of about $1.25 \log N$ blocks per query. GP-ORAM[67] generalizes Partition ORAM by adapting the number of partitions to the available user-side storage and can outsource the index table to the server to reduce local storage consumption. Burst ORAM[10] improves upon Partition ORAM[53] by introducing a new XOR technique to reduce the online bandwidth cost to a constant, and priority scheduling algorithms to deal with request bursts. Ring ORAM[45] further improves the communication efficiency by combining the best qualities from the Partition ORAM[53] and Path ORAM [54]. CURIOUS[3] presents a partition-based ORAM framework and each partition is a small ORAM and can be organized as Path ORAM [54]. It doubles the overall communication cost of Partition ORAM, but reduces the response time.

Path-PIR[37] introduces Private Information Retrieval (PIR) technology to improve communication efficiency of ORAM. In Path-PIR, the server-side storage is organized as a binary tree with $L = \log N + 1$ layers and each node can store $\log N$ blocks. 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. During query process, PIR technology is used to each node on the query path, so for each node, the client only needs to retrieve one data block from server to save query communication cost. During eviction, If node evicting node 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 node is performed to hide the
actual pattern of eviction. The eviction is done using PIR-write and the client does not need to retrieve blocks from server during eviction. Path PIR incurs communication cost of $O(\log N \cdot B)$ bits per query by average. KT-ORAM[64] further improves Path-PIR by organizing the server storage as a $k$-ary tree and re-designed the query and eviction processes. KT-ORAM reduces the communication cost to $O(\frac{\log N}{\log \log N} \cdot B)$ when $k = \log N$.

Onion ORAM[11] is the first ORAM construction that reduces communication cost to $O(B)$ bits. In Onion ORAM, client encrypts the blocks with homomorphic encryption algorithm and stores them to server storage as a binary tree. To accommodate $N$ real data blocks, the height of tree is set as $L + 1$ layers and each node can store up to $z$ data blocks ($z$ is a system parameter and $N \leq z \cdot 2^L - 1$). Therefore, at least $4N$ data blocks (note: the $4N$ blocks include $3N$ empty blocks; the size of each block has to be expanded due to being encrypted with some homomorphic encryption algorithm) are stored on the tree. To query a data block, the client first read the index table to find the position of the block, then constructs a PIR-read vector and issues a PIR-read to retrieve the query target. After accesses the query target, client re-encrypts it and writes it back to root node using PIR-write. For every $A$ queries, client initiates an eviction process. During eviction, all data blocks in source node (i.e., current eviction node) are merged to destination node (i.e. the child node on the evicting path) using homomorphic add and copies all blocks from source node to sibling node (i.e. the child node not on the evicting path). Overall, Onion ORAM incurs a communication cost of $O(B)$ bits and server-side storage overhead is $O(N \cdot B)$ bits; but it requires expensive server computational cost.

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

3.2 Multi-server ORAMs

There are several multi-server ORAM schemes in literature. Among them, the first one is MS-ORAM[32]. MS-ORAM extends the idea of the hierarchical ORAM[19] and the two non-colluding servers are used to obliviously shuffle data. Following the hierarchical ORAM, the client-server
Table 3.1 Comparisons of State-of-the-art Single Oblivious RAM Constructions. $N$ denotes the total number of exported data blocks, $B$ denotes the size of each data block. Note that, for index-based ORAMs, we ignored the index table which takes $O(N \cdot \log N)$ bits of client-side storage. Also we only consider non-recursive version for the ORAM that supports recursive version.

<table>
<thead>
<tr>
<th>ORAM</th>
<th>Communication Cost</th>
<th>Client Storage Cost</th>
<th>Server Storage Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>BH-ORAM [19]</td>
<td>$O(\log^3 N \cdot B)$</td>
<td>$O(B)$</td>
<td>$O(N \log N \cdot B)$</td>
</tr>
<tr>
<td>CH-ORAM [44]</td>
<td>$O(\log^2 N \cdot B)$</td>
<td>$O(B)$</td>
<td>$O(N \cdot B)$</td>
</tr>
<tr>
<td>BF-ORAM [58]</td>
<td>$O(\log^2 N \log N \cdot B)$</td>
<td>$O(B)$</td>
<td>$O(N \cdot B)$</td>
</tr>
<tr>
<td>B-ORAM [28]</td>
<td>$O(\frac{\log^2 N}{\log \log N} \cdot B)$</td>
<td>$O(B)$</td>
<td>$O(N \cdot B)$</td>
</tr>
<tr>
<td>T-ORAM [49]</td>
<td>$O(\log^3 N \cdot B)$</td>
<td>$O(B)$</td>
<td>$O(N \log N \cdot B)$</td>
</tr>
<tr>
<td>G-ORAM [16]</td>
<td>$O(\frac{\log^2 N}{\log \log N} \cdot B) \cdot \omega(1)$</td>
<td>$O(\log^2 N \cdot B) \cdot \omega(1)$</td>
<td>$O(N \cdot B)$</td>
</tr>
<tr>
<td>Path-PIR [37]</td>
<td>$O(\log^2 N \cdot B)$</td>
<td>$O(B)$</td>
<td>$O(N \log N \cdot B)$</td>
</tr>
<tr>
<td>KT-ORAM [64]</td>
<td>$O(\frac{\log^2 N}{\log \log N} \cdot B)$</td>
<td>$O(B)$</td>
<td>$O(N \log N \cdot B)$</td>
</tr>
<tr>
<td>Path ORAM [54]</td>
<td>$10 \log N \cdot B$</td>
<td>$O(\log N \cdot B) \cdot \omega(1)$</td>
<td>$10N \cdot B$</td>
</tr>
<tr>
<td>Burst ORAM [10]</td>
<td>$O(\log N \cdot B)$</td>
<td>$O(\sqrt{N} \cdot B)$</td>
<td>$4N \cdot B$</td>
</tr>
<tr>
<td>Ring ORAM [45]</td>
<td>$2.2 \sim 3.7 \log N \cdot B$</td>
<td>$O(\sqrt{N} \cdot B)$</td>
<td>$6N \cdot B$</td>
</tr>
<tr>
<td>CNE-ORAM [39]</td>
<td>$O(\log N \cdot B)$</td>
<td>$O(\log N \cdot B)$</td>
<td>$O(N \cdot B)$</td>
</tr>
<tr>
<td>TSKT-ORAM [65]</td>
<td>$O(\frac{\log^2 N}{\log \log N} \cdot B)$</td>
<td>$O(N \cdot B)$</td>
<td>$24N \cdot B$</td>
</tr>
<tr>
<td>Onion ORAM [11]</td>
<td>$O(B)$</td>
<td>$O(B)$</td>
<td>$4N \cdot B$</td>
</tr>
</tbody>
</table>

Table 3.2 Comparisons of State-of-the-art Multi-Server Oblivious RAM Constructions. $N$ denotes the total number of exported data blocks, $B$ denotes the size of each data block. Note that, for index-based ORAMs, we ignored the index table which takes $O(N \cdot \log N)$ bits of client-side storage.

<table>
<thead>
<tr>
<th>ORAM</th>
<th>C-S Comm. Cost</th>
<th>Client Storage Cost</th>
<th>Server Storage Cost</th>
<th>S-S Comm. Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>MS-ORAM [32]</td>
<td>$O(\log N \cdot B)$</td>
<td>$O(B)$</td>
<td>$O(N \cdot B)$</td>
<td>$O(\log^4 N \cdot B)$</td>
</tr>
<tr>
<td>MSS-ORAM [51]</td>
<td>$O(B)$</td>
<td>$O(\sqrt{N} \cdot B)$</td>
<td>$O(N \cdot B)$</td>
<td>$O(\log N \cdot B)$</td>
</tr>
<tr>
<td>CNE-ORAM [39]</td>
<td>$O(B)$</td>
<td>$O(\sqrt{N} \cdot B)$</td>
<td>$16N \cdot B$</td>
<td>-</td>
</tr>
<tr>
<td>TSKT-ORAM [65]</td>
<td>$O(\frac{\log^2 N}{\log \log N} \cdot B)$</td>
<td>$O(N \cdot B)$</td>
<td>$24N \cdot B$</td>
<td>-</td>
</tr>
<tr>
<td>$S^3$ORAM [26]</td>
<td>$6B$</td>
<td>$O(B)$</td>
<td>$12N \cdot B$</td>
<td>$O(\log N \cdot B)$</td>
</tr>
</tbody>
</table>
communication cost is $O(\log N \cdot B)$, while the server-server communication cost is $O(\log^3 N \cdot B)$ due to its complicated shuffling process. Even though it only requires a constant local storage, the communication cost is expensive in practice.

The second scheme is MSS-ORAM\cite{51}, which follows the basic design of Partition-ORAM\cite{53}. In their scheme, data shuffling is done between different cloud servers. The client-server communication cost is reduced to a constant number, but the server-server communication cost is $O(\log N \cdot B)$. In addition, it requires the client to store $O(\sqrt{N})$ data blocks in the local storage.

CNE-ORAM\cite{39} incurs $O(B)$ client-server communication cost using at least 4 non-colluding servers. In CNE-ORAM, each data block is split into two parts using secret sharing techniques. Each part of one data block is further copied into two copies and each copy is stored onto 2 out of the 4 servers. The remaining part is also copied and stored onto the other 2 servers. At the server side, the storage is organized as a binary tree with of height $H = O(\log N)$ and each tree node can store $\theta$ data blocks. For each data query, the target data block is retrieved using XOR-based private information retrieval. Then client then writes $\phi$ data blocks to root node of each server. After $\chi$ queries, data eviction process is executed to prevent root node from overflowing. During data eviction, the client guides the servers to merge nodes on the evicting path. In post eviction process, client retrieves a block for the leaf node of the evicting path and replaces it with an empty block if it is a noise block. The computation cost is mainly contributed by data XOR operations, where for each data query, more than $0.5\theta \cdot L$ blocks are XORed and the communication cost is mainly contributed by uploading $\phi$ data blocks to root per query, where $L = O(\log N)$ is the height of the tree.

Based on KT-ORAM, TSKT-ORAM\cite{65, 66} introduce two non-colluding servers to facilitate bit-XOR read operation instead of expensive homomorphic encryption, thus reduce the server computational cost. Eviction algorithm is re-designed, where evicting node is selected uniformly at random for each layer, and the child nodes of eviction node is divided into three partitions logically and it is only known to the user. The eviction process makes sure each position of the two partitions of nodes receiving the evicted data blocks will be accessed with equal probability. TSKT-ORAM
has a communication cost of $O(\frac{\log N}{\log \log N} \cdot B)$ when $k = \log N$ and the servers only need to conduct simple XOR operation instead of expensive Homomorphic Encryption.

Hoang et al. propose S³ORAM [26] based on the utilization of multiple (at least three) non-colluding servers. Server storage is organized as that in Onion ORAM[11], and each data block is distributed to the three servers using Shamir Secret Sharing algorithm. During query, each server returns a block which is the result of XOR of selected blocks, the client then rebuild the target block. Eviction is conducted between server and client only need to send metadata to servers, thus no data blocks is transferred during eviction. S³ORAM [26] incurs $O(B)$ bandwidth consumption for client-server communication, which is similar to the cost incurred by a non-oblivious storage system; it still requires $O(\log N \cdot B)$ bandwidth consumption for communication between the servers, but the inter-server communication occurs behind the scene and does not consume client-server bandwidth.
CHAPTER 4. SE-ORAM

4.1 Research Goal and Rationales

This study [33] aims to design an single server ORAM construction with zero storage overhead at the server.

The security goal of an ORAM construction is to prevent the storage server from correctly inferring a client’s private data access sequence from the client’s storage location access sequence that the server can observe. Existing ORAM constructions target at perfect security; that is, the probability is at most \( \frac{1}{N^n} \) for the server to correctly infer a sequence with \( n \) data accesses from any observed location access sequence, since \( N^n \) is the total number of sequences with \( n \) data accesses. To attain this goal, the client’s query and shuffling operations should be fully random and independent of each other.

Particularly, let us consider the tree-based ORAM [49]. When a data block is assigned to a path of the storage tree, the path is selected uniformly at random to make the query process appear fully random. During an eviction process, nodes are randomly selected to evict data, and each selected node is dictated to evict a data block to its left or right child with the equal probability. Due to the randomness, following undesired situation may happen: a node without any real data block evictable to its left (or right) child is selected to evict data to left (or right). To deal with such situation, dummy blocks are pre-introduced into the storage when the system is initialized; and it has been shown that, \( O(N) \) or \( O(N \log N) \) dummy blocks are needed to keep a low failure probability, i.e., the probability that a node has already used up dummy blocks when it is in the afore-described undesired situation.

To address the above issue without introducing storage overhead to the server, we design a new eviction algorithm based on the following intuitions:
• **Eviction with Non-uniform Probabilities.** When a node is selected to evict data to its children, it can use different probabilities for different children; i.e., a larger probability to evict data to its left child if more of its data blocks are evictable to left, and vice versa. This way, the chance could be significantly reduced for the afore-mentioned undesired situation to occur.

• **On-demand Introduction of Dummies.** Nevertheless, the undesired situation could still occur. To deal with it, a dummy block (evictable to both left and right) is inserted on demand to replace a real data block, which is moved to the client’s cache. Note that, the storage server still stores the same number of data blocks, though some of the blocks become dummies.

• **Periodical Removal of Dummies.** As the system keeps running, more dummy blocks are inserted to the server and the client’s cache may overflow. To address this issue, an extra query and eviction process is launched periodically to retrieve and discard a dummy from the server and evict a real data block from the client to the server.

Due to the non-uniform eviction probabilities used in the eviction algorithm, perfect security is not attained. To quantify the level of security that our new ORAM construction can achieve, we propose a more generic security definition, which quantify security level by a parameter $\lambda$: if an ORAM construction is secure with parameter $\lambda$, the probability is at most $\left(\frac{1}{\sqrt[N]{n}}\right)^{1-\frac{1}{\lambda}}$ for the server to correctly infer a sequence with $n$ data accesses from any storage location access sequence. That is, the advantage for the server to discover a client’s access pattern is upper-bounded by $\left(\frac{1}{\sqrt[N]{n}}\right)^{1-\frac{1}{\lambda}} - \frac{1}{\sqrt[N]{n}}$, which decreases as $\lambda$ increases. We argue that, this notion of security can be useful in practice, particularly when a large number of data blocks are outsourced and/or protecting relatively long access patterns (i.e., $n$ is large) is the major security goal. For example, when $N = 2^{40}$ and $n = 10$ (or $N = 2^{10}$ and $n = 40$), and $\lambda = 2$, the server’s advantage is upper-bounded by $2^{-200}$, which may be considered “negligibly small” in practice. Besides, the definition allows a client of our ORAM construction to configure her desired level of security, and manage the tradeoffs between security and performance.
4.1.1 Results

Based on the new eviction algorithm and the new definition of security, we formalize a generic SE-ORAM construction with parameter $\lambda$. Through rigorous security and cost analysis, we show that the construction is secure under the definition, and the number of introduced dummy blocks is no more than $x \log N$ with probability $1 - \frac{1}{N^{2x}}$, as long as $\lambda \geq 2$ and each node on the storage tree can store $4 \log N$ or more data blocks. We also instantiate a SE-ORAM construction by setting $\lambda = 2$, analyze its performance, and compare it with the state-of-the-art ORAM constructions. To summarize, this study makes the following contributions:

- We introduce a generic security definition for ORAM constructions. It allows a client to configure a desired security level and manage the tradeoffs between security and performance.
- We propose SE-ORAM, a generic storage-efficient ORAM construction with configurable security parameter $\lambda$. Rigorous analysis shows that, SE-ORAM achieves the configured level of security, introduces zero storage overhead to the storage server (i.e., the storage server only storages $N$ data blocks), and incurs $O(\log N)$ blocks storage overhead at the client, as long as $\lambda \geq 2$ and each node on the storage tree stores $4 \log N$ or more data blocks.

4.1.2 Organization

In the rest of the chapter, Section 4.2 presents the security definition. Section 4.3 presents the basic design of SE-ORAM, which is followed by security analysis in Section 4.4 and overhead analysis and comparison in Section 4.5. Finally, Section 4.6 concludes the chapter.

4.2 Security Definition

Follow the research goal discussed in 4.1, we redefine the security of a single server ORAM for this chapter.

Let $\lambda > 1$ be a security parameter. A client exports $N$ equal-size data blocks to a remote storage server. Each data access from the client, which should be kept private, is one of the following two
types: (i) read a data block $D$ of unique ID $i$ from the storage, denoted as a 3-tuple $(\text{read}, i, D)$; (ii) write a data block $D$ of unique ID $i$ to the storage, denoted as a 3-tuple $(\text{write}, i, D)$. To accomplish each data access, the client needs to access some storage location(s) at the remote storage server. Each location access, which can be observed by the server, is one of the following types: (i) retrieve (i.e., read) a data block $D$ from a location $l$, denoted as a 3-tuple $(\text{read}, l, D)$; (ii) upload (i.e., write) a data block $D$ to a location $l$, denoted as a 3-tuple $(\text{write}, l, D)$.

We assume the remote storage server is honest but curious; that is, it stores data and serves the client’s location access requests honestly, but it may attempt to figure out the client’s data access pattern hidden behind the location accesses. The network connection between the client and the server is assumed to be secure; in practice, this can be achieved using well-known techniques such as SSL [24].

We define the security of our proposed SE-ORAM($\lambda$, $N$), which has security parameter $\lambda$ and stores $N$ real data blocks, as follows.

**Definition** In SE-ORAM($\lambda$, $N$), let $\vec{x}_n = \langle (\text{op}_1, i_1, D_1), (\text{op}_2, i_2, D_2), \ldots, (\text{op}_n, i_n, D_n) \rangle$ denote a private sequence of the client’s $n$ data accesses, where each $\text{op}_i$ is either a read or write operation; let random variable $A(\vec{x}_n)$ denote the sequence of location accesses (observable by the server) that the client uses to accomplish data access sequence $\vec{x}_n$. Note that, there may exist multiple location access sequences that can accomplish $\vec{x}_n$, each with certain probability to be used by the client as $A(\vec{x}_n)$; hence, $A(\vec{x}_n)$ is a random variable.

Let $X_n$ denote the set of all possible sequences of the client’s $n$ data accesses, and $A_n$ the set of all location access sequences that can accomplish at least one data access sequence in $X_n$.

Let $\Pr[\vec{T}_n | \vec{A}_n]$, where $\vec{A}_n \in A_n$ and $\vec{T}_n \in X_n$, denote the conditional probability of $A(\vec{T}_n) = \vec{A}_n$ given that $\vec{A}_n$ has been observed by the server.

SE-ORAM($\lambda$, $N$) is said to be secure if $\forall \vec{A}_n \in A_n$ and $\forall \vec{T}_n \in X_n$:

\[
\left( \frac{1}{N^n} \right)^{1+\frac{1}{\lambda}} \leq \Pr[\vec{T}_n | \vec{A}_n] \leq \left( \frac{1}{N^n} \right)^{1-\frac{1}{\lambda}}.
\]

(4.1)

Note that, if the client’s data access pattern is perfectly protected, $\Pr[\vec{T}_n | \vec{A}_n] = \frac{1}{N^n}$; i.e., no matter what location access sequence (that can accomplish a certain sequence with $n$ data accesses)
has been observed, it is impossible for the server to infer the client’s actual data access sequence hidden behind this observed pattern, because each of the $N^n$ data access sequences has the same probability $\frac{1}{N^n}$ to be the one. According to the above definition, when $\lambda \to \infty$, $Pr[\vec{T}_n|\vec{A}_n] \to \frac{1}{N^n}$ indeed.

Generally speaking, if an SE-ORAM($\lambda, N$) is secure, the advantage for the server to infer the client’s actual data access sequence $\vec{T}_n$ from a location access sequence $\vec{A}_n$ that has been observed, i.e., $|Pr[\vec{T}_n|\vec{A}_n] - \frac{1}{N^n}|$, is upper-bounded by $(\frac{1}{N^n})^{1-\frac{1}{\lambda}} - \frac{1}{N^n}$; the larger is $\lambda$, the smaller is the bound. Hence, parameter $\lambda$ quantifies the level of security that an SE-ORAM construction can attain.

4.3 The SE-ORAM Construction

This section elaborates the SE-ORAM construction in terms of storage organization, data query and data eviction algorithms.

4.3.1 Storage Organization and Initialization

4.3.1.1 Server-side Storage

In the server, the storage is initially organized as a complete binary tree. Each node on the tree can store up to $s$ data blocks, where $s$ is a system parameter and an even number. To simplify presentation, we denote the height of tree as $h$ and assume the total number of data blocks $N$ as $N = s \cdot \sum_{l=0}^{h} 2^l = s(2^{h+1} - 1)$. Hence, the number of level-$h$ nodes is $2^h$, which also is $\frac{N/s+1}{2} \approx \frac{N}{2s}$.

The content of each data block $B_i$ is encrypted probabilistically with a symmetric cipher (e.g., AES) before the blocks are randomly distributed to the nodes on the tree. Specifically, denoting the plain-text content of a block $B_i$ as $D_i$, we have $B_i = E(r|D_i)$, where $r$ is a nonce and $E$ is a symmetric encryption function.

In each node $n$, data blocks are randomly divided into two equal-size groups, called left group and right group and denoted as $G_L(n)$ and $G_R(n)$. Each block in the left group randomly picks a level-$h$ node $n'$ from the left branch of $n$, and the block is restricted to be evictable toward node $n'$ only; hence, we call the ID of node $n'$ as the path ID of the data block. Similarly, each block in
the right group also randomly selects a level-$h$ node from the right branch of $n$, whose ID becomes the block’s path ID.

As the data query and eviction processes go on, the tree may become incomplete and some nodes may become non-full (i.e., containing less than $s$ data blocks). Figure 4.1(a) shows an example of the server-side storage. Here, $h = 3$, two of the level-$h$ nodes (i.e., $n_{3,1}$ and $n_{3,6}$) are absent, and one level-$h$ node (i.e., $n_{3,2}$) is non-full. Also, the data blocks with path IDs of $n_{3,0}$, $n_{3,4}$ and $n_{3,7}$ cannot be completely contained in nodes between level 0 to level 3; hence, supplementary nodes have been introduced to provide additional storage, e.g., $n_{4,0}$ for $n_{3,0}$, $n_{4,4}$ and $n_{5,4}$ for $n_{3,4}$, and $n_{4,7}$ for $n_{3,7}$.

4.3.1.2 Client-side Storage

The client-side storage includes three parts: (i) an index table $I$ maintaining the mapping between data block IDs and their path IDs (therefore it has $N$ entries and each entry has $h$ bits); (ii) a data block cache $C$ used to cache data blocks; and (iii) a small secret storage storing the key for symmetric data encryption.

4.3.2 Data Query

When the client queries a data block of ID $t$ (denoted as $B_t$), it first checks whether $B_t$ is in $C$; if so, the block is accessed and retained in $C$. Otherwise, the client looks up the index table $I$ to obtain $B_t$’s path ID (i.e., the ID of a level-$h$ node, denoted as $n_t^h$ hereafter). Then, the client follows the steps below to obliviously retrieve $B_t$.

The client requests the server to return data blocks on the path from the root to the $n_t^h$. In response, the server first finds out all the nodes that should be returned to the client, based on the current topology of the tree: (i) Case I - if node $n_t^h$ is currently on the tree and has no supplementary nodes, all the nodes along the path from the root to $n_t^h$ should be returned. (ii) Case II - if $n_t^h$ is currently on the tree and has supplementary nodes, all the nodes along the path from the root to $n_t^h$ as well as all of $n_t^h$’s supplementary nodes should be returned. (iii) Case III - if node $n_t^h$ is
Figure 4.1 Query Examples. In (a), query target $B_t$ is at node $n_{2,2}$ and has path ID $n_{3,4}$. Node $n_{3,4}$ exists on the tree and has two supplementary nodes. The client requests the server to retrieve nodes from the root to $n_{5,4}$ which is the further supplementary node of $n_{3,4}$. Then, $B_L$ obliviously replaces $B_t$; finally, as node $n_{5,4}$ becomes empty after $B_L$ has moved, the node is removed from the tree. In (b), query target $B_t$ is at node $n_{1,0}$ and has path ID $n_{3,1}$. Node $n_{3,1}$ does not exist on the tree. The client requests the server to retrieve nodes on the path from the root to $n_{4,0}$, which is the longest path that has the largest overlap with the path from the root to $n_{3,1}$. Then, the client obliviously replaces $B_t$ with $B_L$.

absent, the server acts as follows. Let $n_t^{h_0}$ denote the node that is on the path from the root toward $n_t^{h_1}$ (as if $n_t^{h_1}$ were still there) and the furthest away from the root. Let $n_t^{h_1}$ denote the leaf node of the longest branch within the subtree rooted at $n_t^{h_0}$. Note that, the path from the root to $n_t^{h_1}$ is the longest path that has the largest overlap with the path from the root to $n_t^{h_1}$ (as if $n_t^{h_1}$ were still there). All the nodes along the path from the root to $n_t^{h_1}$ should be returned. Let us denote the nodes that should be returned as $n_t^0$, $n_t^1$, $\cdots$, $n_t^L$, where $n_t^0$ is the root and $n_t^L$ is the leaf node. Among them, suppose node $n_t^y$ on layer $y$ contains $B_t$.

The server returns only the blocks in $n_t^L$ in the first round. If $B_t$ is among the blocks, the client keeps $B_t$ locally, re-encrypts the rest of the blocks and uploads them back to the server; otherwise,
one arbitrary block denoted as $B_L$ is picked from the returned blocks, and the rest of the blocks are re-encrypted and uploaded back to the server.

Next, the server returns all the blocks in $n_{t}^{L-1}$. If $B_t$ is among the blocks, the block is kept locally, and the rest of the blocks in $n_{t}^{L-1}$ together with $B_L$ are re-encrypted and uploaded back to the server. Otherwise, all the blocks in $n_{t}^{L-1}$ are re-encrypted and uploaded to the server. This process continues until all the blocks on the selected path have been returned to the client, re-encryption and finally uploaded back to the server. Figure 4.1 shows two examples of data query.

4.3.3 Data Eviction

Data eviction should be conducted following the query process, to store the query target $B_t$ back to the server obliviously.

A path (i.e., a level-$h$ node) is selected uniformly at random for $B_t$, and then all the data blocks on the path are retrieved node-by-node. The eviction process should place $B_t$ into a node on the selected path before the blocks are all re-encrypted and uploaded back to the server. The ID of the path becomes the new path ID of $B_t$ and hence should be recorded in the client’s index table $I$. During the course of eviction, some other blocks may be moved; the movement should ensure that, a data block stays in a node on the path specified by its path ID or it stays in the local cache maintained by the client. The eviction steps are elaborated in the following, and an example containing evictions in four layers of the storage tree is given in Figure 4.2.

E1: Initial Step. Let $B_e$ denote the current block to evict (called the evicted block), and $n_e$ the current node (called the evicting node) to accommodate $B_e$’s eviction. Initially, $B_e = B_t$ and $n_e = \text{root}$. All the data blocks in $n_e$ are sent from the server to the client.

E2: Conditional Termination. If $n_e$ is non-full, the client writes $B_e$ into $n_e$. Then, $B_e$ is put into the left or right group of $n_e$ (i.e., $G_L(n_e)$ or $G_R(n_e)$) according to its path ID; note that, if $B_e$ is a dummy, it is randomly put into either $G_L(n_e)$ or $G_R(n_e)$.
Figure 4.2 Data eviction example. (a) Evicting data block from layer 0 to 1: Evicted block $B_e$ has path ID $n_{3,7}$ but the evicting node $n_{0,0}$ chooses to evict to left. Hence, $B_e$ is written obliviously to $n_{0,0}$, while a block with path ID $n_{3,0}$ (and therefore evictable to left) is selected from $n_{0,0}$ to the new evicted block. This is Case II in Section 4.3 E4. (b) Evicting data block from layer 1 to 2: Evicting node $n_{1,0}$ chooses to evict to right. No block (including $B_e$ and the blocks in $n_{1,0}$) is evictable to right. Hence, a dummy block is created to replace a randomly-selected real block in $n_{1,0}$, which is moved to the cache of the client. The dummy block then becomes the new evicted block. This is Case IV in Section 4.3 E4. (c) Evicting data blocks from layer 2 to 3: As the evicted block $B_e$ is a dummy, it remains as the evicted block no matter whether the evicting node chooses to evict to left or right. This is Case I in Section 4.3 E4. (d) Evicting data blocks from layer 3 to 4: The evicting node is non-full. So the evicted block is written to it obliviously and the eviction process terminates, as explained in Section 4.3 E4.
Another condition for the process to terminate is when $n_e$ is a level-$h$ node. $B_e$ should be written to the furthest supplementary node of $n_e$. If the supplementary node is full, an additional supplementary node is created to contain $B_e$.

For both cases, blocks in $n_e$ (and its supplementary nodes if applicable) should be re-encrypted and uploaded back to the server.

**E3: Selection of the Next Evicting Node.** Depending on the sizes of $G_L(n_e)$ and $G_R(n_e)$, the selection of the next evicting node (denoted as $n'_e$) works as follows:

- If $|G_L(n_e)| > |G_R(n_e)|$, the left child of $n_e$ is selected as $n'_e$ with probability $1 - p$ while the right child is selected as $n'_e$ with probability $p$, where $p = \frac{1}{2^\frac{1}{\lambda+1}}$ and $1 - p = \frac{2^\frac{1}{\lambda+1}}{2^\frac{1}{\lambda+1}}$.

- If $|G_L(n_e)| = |G_R(n_e)|$, the left and right children of $n_e$ have the same probability 0.5 to be selected as $n'_e$.

- If $|G_L(n_e)| < |G_R(n_e)|$, the left child of $n_e$ is selected to be $n'_e$ with probability $p$ while the right child is selected to be $n'_e$ with probability $1 - p$.

Note that, if $n'_e$ does not exist on the tree, it should be created: 1) If $n_e$ is a level-$h$ node or a supplementary node, a supplementary node $n'_e$ is created and linked to $n_e$; 2) otherwise, $n'_e$ is created as a left or right child node of $n_e$ accordingly.

**E4: Selection of the Next Evicted Block.** There are a few different cases. Case I - If $B_e$ is a dummy block, it remains to be the next evicted block denoted as $B'_e$. Case II - If $B_e$ is a real data block, and there is at least one block in $n_e \cup \{B_e\}$ (we also use $n_e$ to denote the set of all data blocks in $n_e$, for simplicity) that is evictable to $n'_e$, one such block is selected to be $B'_e$ and the selected block is replaced by $B_e$. Case III - If $B_e$ is a real data block, no data blocks in $n_e \cup \{B_e\}$ are evictable to $n'_e$, but $n_e$ contains dummy blocks, one dummy block is selected as $B'_e$ and the selected block is replaced by $B_e$. Case IV - If $B_e$ is a real data block, no data blocks in $n_e \cup \{B_e\}$ are evictable to $n'_e$, and $n_e$ does not contain any dummy blocks, a new dummy block is created to be $B'_e$, while the original $B_e$ is saved to the client’s local cache. Finally, all current blocks in $n_e$ are
re-encrypted and uploaded back to the server; then, after $B_e \leftarrow B_e'$ and $n_e \leftarrow n_e'$ are performed, the process continues to Step E2.

**4.3.4 Extra Query-Eviction Round**

With the above eviction algorithm, the dummy blocks at the storage server and the cached blocks at the client may keep increasing as more data blocks are queried. To bound the number of these blocks and hence the storage overhead, we propose to periodically remove dummy blocks and dump cached data blocks as follows.

Every time after an eviction process is completed, with probability $\rho$, the following extra round of query and eviction is conducted: The client randomly selects a path. Depending on the selected path, this step proceeds with one of the following two cases. Case I - the selected path contains dummy blocks. In this case, one dummy block is retrieved from the selected path following the above data query algorithm. Then, one real data block is randomly picked from the client’s cache, and evicted to the tree structure at the storage server following the above data eviction algorithm. Case II - the selected path does not contain any dummy blocks. In this case, one data block is randomly retrieved from the selected path following the above data query algorithm, and then evicted following the above data eviction algorithm.

**4.4 Security Analysis**

Recall that Section 4.2 defines the concepts of data access sequence and location access sequence, and introduces the notations of $X_n$, $A_n$, random variable $A(\vec{x}_n)$ for $\vec{x}_n \in X_n$, and conditional probability $Pr[\bar{T}_n|\bar{A}_n]$ for $\bar{T}_n \in X_n$ and $\bar{A}_n \in A_n$. To facilitate the security analysis in this section, we further introduce the following notations:

For any $\bar{A}_n \in A_n$, we expand it to $\bar{A}_n = q_1, e_1, \cdots, q_n, e_n$. Here, for each $i = 1, \cdots, n$, $q_i$ denotes the path accessed during the $i$-th query process and $e_i$ denotes the path accessed during the $i$-th eviction process.
Each $e_i$ in the above is further expanded to $e_i = e_{i,1}, e_{i,2}, \ldots, e_{i,h_i}$. Here, $e_{i,j} \in \{0, 1\}$ for $j \in \{0, 1, \ldots, h_i\}$. $e_{i,1}$ represents whether the root node (i.e., the first evicting node in the $i$-th eviction process) evicts data to its left (if $e_{i,1} = 0$) or right child (if $e_{i,1} = 1$), and $e_{i,j}$ represents whether the $(j - 1)$-th evicting node evicts data to its left (if $e_{i,j} = 0$) or right child (if $e_{i,j} = 1$).

Let $Pr[\vec{x}_n]$, where $\vec{x}_n \in X_n$, denote the probability that $\vec{x}_n$ is the client’s actual data access sequence.

Let $Pr[A_n | \vec{x}_n]$, where $A_n \in A_n$ and $\vec{x}_n \in X_n$, denote the conditional probability of $A(\vec{x}_n) = A_n$ given that $\vec{x}_n$ is the client’s actual data access sequence.

Let $Pr[q_i | \vec{x}_n; q_1, e_1, \ldots, q_{i-1}, e_{i-1}]$ denote the conditional probability of $q_i$ being selected to access during the $i$-th query process given that the client’s actual data access sequence is $\vec{x}_n$ and the location access sequence has been $q_1, e_1, \ldots, q_{i-1}, e_{i-1}$ before the $i$-th query is processed.

Let $Pr[e_{i,j} | \vec{x}_n; q_1, e_1, \ldots, q_i, e_i, \ldots, e_{i,j-1}]$ denote the conditional probability for the $i$-th evicting node to evict to left (if $e_{i,j}$ is 0) or right (if $e_{i,j}$ is 1), given that the client’s actual data access sequence is $\vec{x}_n$ and the location access sequence has been $q_1, e_1, \ldots, q_i, e_i, \ldots, e_{i,j-1}$ before this evicting node is accessed.

**Lemma 1.** In SE-ORAM($\lambda$, $N$), for $\forall \vec{x}_n \in X_n$ and $\forall i \in \{1, 2, \ldots, n\}$,

$$Pr[q_i | \vec{x}_n; q_1, e_1, \ldots, q_{i-1}, e_{i-1}] = \frac{2^s}{N}.$$

Proof. Initially and after being queried, data blocks are all distributed to the paths uniformly at random. Hence, every path has the same probability to be selected for each query. The probability is $\frac{2^s}{N}$ as the total number of paths is $\frac{N}{2^s}$. □

**Lemma 2.** In SE-ORAM($\lambda$, $N$), for $\forall \vec{x}_n \in X_n$, $\forall i \in \{1,2,\ldots,n\}$ and $\forall j \in \{1,2,\ldots,h_i\}$: $p \leq Pr[e_{i,j} | \vec{x}_n; q_1, e_1, \ldots, q_{i-1}, e_i, \ldots, e_{i,j-1}] \leq 1 - p$.

Proof. During an eviction process, the probability for an evicting node to evict a data block to its left (or right) child is between $p$ and $1 - p$. The lemma is therefore proved. □
**Theorem 1.** SE-ORAM($\lambda$, $N$) is secure under Definition 4.2. That is, for any $\vec{A}_n \in \mathcal{A}_n$ and $\vec{T}_n \in \mathcal{X}_n$,

$$(\frac{1}{N^n})^{1+\frac{1}{3\lambda}} \leq Pr[\vec{T}_n | \vec{A}_n] \leq (\frac{1}{N^n})^{1-\frac{1}{3\lambda}}.$$  

(4.2)

**Proof.** Since

$$Pr[\vec{T}_n | \vec{A}_n] = Pr[\vec{A}_n | \vec{T}_n] Pr[\vec{T}_n],$$

(4.3)

we need to compute $Pr[\vec{T}_n], Pr[\vec{A}_n | \vec{T}_n], Pr[\vec{x}_n]$ and $Pr[\vec{A}_n | \vec{x}_n]$.

First, as the server has no a priori knowledge of the client’s actual data access pattern, for $\vec{T}_n$ and any $\vec{x}_n \in \mathcal{X}_n$, it holds that

$$Pr[\vec{T}_n] = Pr[\vec{x}_n] = \frac{1}{N^n}.$$  

(4.4)

Second, due to Lemmas 1 and 2 and $Pr[\vec{A}_n | \vec{x}_n]$ being equal to $\prod_{i=1}^{n} \prod_{j=1}^{h_i} Pr[ q_i | \vec{x}_n; q_1, e_1, \cdots, q_{i-1}, e_{i-1} | \vec{x}_n; q_1, e_1, \cdots, q_i, e_i, \cdots, q_{i-1}, e_{i-1}]$, it follows that

$$\left(\frac{2s}{N}\right)^n \cdot \prod_{i=1}^{n} h_i \leq Pr[\vec{A}_n | \vec{x}_n] \leq \left(\frac{2s}{N}\right)^n \cdot (1 - p) \sum_{i=1}^{n} h_i.$$  

(4.5)

Hence,

$$\left(\frac{p}{1-p}\right) \sum_{i=1}^{n} h_i \leq \frac{Pr[\vec{A}_n | \vec{T}_n]}{Pr[\vec{A}_n | \vec{x}_n]} \leq \left(\frac{1-p}{p}\right) \sum_{i=1}^{n} h_i.$$  

(4.6)

Since $h_i \leq \log(N/s) < \log N$,

$$\left(\frac{p}{1-p}\right)^{n \log N} \leq \frac{Pr[\vec{A}_n | \vec{T}_n]}{Pr[\vec{A}_n | \vec{x}_n]} \leq \left(\frac{1-p}{p}\right)^{n \log N}.$$  

(4.7)

Based on Equations (4.3), (4.4) and (4.7), it holds that

$$\left(\frac{p}{2(1-p)}\right)^{n \log N} \leq Pr[\vec{T}_n | \vec{A}_n] \leq \left(\frac{1-p}{2p}\right)^{n \log N}.$$  

(4.8)

As $p = \frac{1}{2^\sqrt{\lambda+1}}$, Equation (4.8) becomes Equation (4.2), which completes the proof.

\[\square\]

### 4.5 Cost Analysis

#### 4.5.1 Storage Overhead

In SE-ORAM, the storage server initially stores only real data blocks exported by the client. As the system keeps running, dummy blocks are introduced or removed, and the server needs to store
some dummy blocks. However, when a dummy block is introduced, it always replaces a real data block which should be moved to the client’s cache; when a dummy block is removed, it is always replaced with a real data block previously cached by the client. Hence, the storage consumption at the server keeps unchanged. In this sense, there is no storage overhead at the server. However, extra storage overhead has been introduced to the client, who needs to cache real data blocks that have been replaced by dummies.

In the following, we analyze the number of dummy blocks in the storage, which is equal to the number of real data blocks that should be cached by the client. We first introduce the notation of node state and its transitions. Then, we analyze the probability to introduce a new dummy block during every data eviction process. Finally, we show that, with appropriate setting of system parameters (i.e., $\lambda \geq 2$ and $s = 2c\lambda \log N$ for $c \geq 1$), the number of dummy data blocks is bounded by $x \log N$ with a probability greater than $1 - \left(\frac{1}{N}\right)^{2x}$.

According to the eviction algorithm in SE-ORAM, a new dummy block may be introduced only when a data block is evicted from a node that is full and does not contain any dummy block. For any of such node, we use $(x, s - x)$ to represent its state, where $x$ is the number of data blocks in the left group and $s - x$ is the number of data blocks in the right group. Thus, the state transition probabilities are as follows:

$$Pr[(x+1, s-x-1)|(x, s-x)] =$$

\[
\begin{cases}
\frac{1-p}{2}, & \text{if } x < s - x, \\
\frac{1}{4}, & \text{if } x = s - x, \\
\frac{p}{2}, & \text{if } x > s - x,
\end{cases}
\]  

$$Pr[(x-1, s-x+1)|(x, s-x)] =$$

\[
\begin{cases}
\frac{p}{2}, & \text{if } x < s - x, \\
\frac{1}{4}, & \text{if } x = s - x, \\
\frac{1-p}{2}, & \text{if } x > s - x.
\end{cases}
\]
\[ Pr[(x - 1, s - x + 1)|(x, s - x)] \] can be computed similarly. Figure 4.3 shows the complete set of node state transitions. Based on the above analysis, we can get the following lemma.

**Lemma 3.** In SE-ORAM, any node on the storage tree has a probability less than \( 2^{-\frac{s}{2}} \) to stay in state \((0, s)\) (or state \((s, 0)\)); that is, \( Pr[(0, s)] = Pr[(s, 0)] < 2^{-\frac{s}{2}} \).

**Proof.** In the Markov chain of node state transition shown in Figure 4.3, the steady state distribution has the following property:

\[
\frac{1}{2} > Pr\left(\frac{s}{2} - 1, \frac{s}{2} + 1\right) = Pr[(0, s)] \cdot \left(\frac{1 - p}{p}\right)^{\frac{s}{2} - 1} = Pr[(0, s)] \cdot 2^{\frac{s-2}{2\lambda}}.
\]

Also because \( \lambda > 1 \), it follows that \( Pr[(0, s)] < 2^{-\frac{s}{2\lambda}} \cdot \frac{1}{2} < 2^{-\frac{s}{2\lambda}} \). Similarly, it can be proved that \( Pr[(s, 0)] < 2^{-\frac{s}{2\lambda}} \).

**Lemma 4.** In SE-ORAM, when \( s = 2c\lambda \log N \), the probability for an eviction process to introduce a new dummy block is less than \( \frac{\log N}{2N^c} \).

**Proof.** According to Lemma 3, for any node, \( Pr[(0, s)] = Pr[(s, 0)] < 2^{-\frac{s}{2\lambda}} \), which is less than \( \frac{1}{N^c} \) since \( s = 2c\lambda \log N \).

During an eviction process, at most one dummy data block may be introduced. And the introduction occurs only if: there is at least one evicting node that is in state \((0, s)\) (or \((s, 0)\)), block evicted to this node is evictable only to right (or left), and the node chooses to evict to left (or right). For this to occur, the probability is at most

\[
1 - \left(1 - (Pr[(0, s)] + Pr[(s, 0)]) \cdot \frac{p}{2}\right)^{b+1}.
\]
which is less than $1 - (1 - \frac{1}{2N^c})^{\log N}$ since $p = \frac{1}{1+2/\lambda} < \frac{1}{2}$ and $h+1 = \log(N/s) + 1 < \log N$. Expanding it, we obtain

$$1 - 1 + \log N \cdot \frac{1}{2N^c} - \sum_{i=1}^{\infty} \left[ (\log N)^2 (\frac{1}{2N^c})^{2i} - (\log N)^2 (\frac{1}{2N^c})^{2i+1} \right],$$

which is less than $\frac{\log N}{2N^c}$, since $(\log N)^2 (\frac{1}{2N^c})^{2i} > (\log N)^2 (\frac{1}{2N^c})^{2i+1}$ for every $i$. Hence, the Lemma is proved.

**Theorem 2.** In SE-ORAM, when $\rho \geq \frac{1}{2}$ and $s = 2c\lambda \log N$ for $c \geq 1$,

$$Pr[number of dummy blocks \leq x] > 1 - \left(\frac{1}{2c\lambda N^c}\right)^x \geq 1 - \left(\frac{1}{2\lambda}\right)^x.$$
\[ \log N \leq \frac{N}{2} \] for every \( N \geq 4 \); we do not consider \( N < 4 \) as it is trivial. Hence, \( p_{x+1,x} = \Pr[N_d = x | N_d = x + 1] \geq \frac{3s\rho}{2(1+\rho)N} \).

Since \( s = 2c\lambda \log N \) for \( c \geq 1 \), and \( \rho \geq \frac{1}{2} \) and thus \( (1+\rho)/\rho \leq 3 \), \( p_{x,x+1} p_{x+1,x} < \frac{2(1+\rho)N \log N}{6s\rho N^c} \leq \frac{1}{2c\lambda N^{c-1}} \).

So, \( \Pr[N_d > x] < \left( \frac{1}{2c\lambda N^{c-1}} \right)^x \leq \left( \frac{1}{2\lambda} \right)^x \); that is, \( \Pr[N_d \leq x] > 1 - \left( \frac{1}{2c\lambda N^{c-1}} \right)^x \geq 1 - \left( \frac{1}{2\lambda} \right)^x \). \qed

As the number of data blocks cached by the client is the same as the number of dummy blocks stored at the server side, we have the following corollary based on Theorem 2.

**Corollary 1.** In SE-ORAM, when \( \rho \geq \frac{1}{2} \) and \( s = 4 \log N \), the number of data blocks cached at the client side is bounded by \( x \log N \) with a probability of \( 1 - \left( \frac{1}{N} \right)^{2x} \).

### 4.5.2 Communication Overhead

Each query or eviction process needs to access a series of nodes along a path from the root to a leaf. The communication and computational costs for query and eviction are therefore affected by the height of the storage tree.

**Theorem 3.** In SE-ORAM, the height of the storage tree is upper-bounded by \( \log(N/s) + 3 \) with a probability of at least \( 1 - \left( \frac{1}{2} \right)^s \). When \( s = 2c\lambda \log N \), the probability is \( 1 - \left( \frac{1}{N} \right)^{2c\lambda} \).

**Proof.** In SE-ORAM, \( N \) data blocks are distributed to \( N/2^n \) paths uniformly at random. We first show that probability for a path to be assigned with more than \( 4s \) blocks is no greater than \( \frac{1}{N^{s}} \).

Assigning \( N \) blocks to \( N/2^n \) is a standard balls in bins game with \( N \) balls and \( N/2^n \) bins. The expected number of blocks assigned to each path is \( 2s \). According to Chernoff bound, the probability for any path to be assigned with more than \( 4s \) blocks is upper-bounded by \( e^{-2s/3} \leq 2^{-s} \). That is, the probability is at least \( 1 - \left( \frac{1}{2} \right)^s \) that every path is assigned with no more than \( 4s \) block.

A path has the longest length if all the blocks assigned to it have to be stored in its level-\( h \) node and supplementary nodes. In this extreme case, a path has a length of no longer than \( \log(N/s) + 3 \) if no more than \( 4s \) blocks are assigned to it.

So far, we have proved that the probability is at least \( 1 - \left( \frac{1}{2} \right)^s \) that the height of the tree is no larger than \( \log(N/s) + 3 \). When \( s = 2c\lambda \log N \), obviously the probability is \( 1 - \left( \frac{1}{N} \right)^{2c\lambda} \). \qed
For each query, all blocks on a path containing the target data block need to be downloaded and then uploaded; and in the following eviction process, all the data blocks on a randomly-selected path need to be downloaded and then uploaded. The number of nodes on each root-to-leaf path is $O(\log N)$ and each node stores $O(\log N)$ data blocks. Hence, the communication overhead is $O(\log^2 N)$ data blocks per query.

### 4.5.3 Performance Comparison

We instantiate the generic SE-ORAM by setting $\lambda = 2$, $c = 1$ and thus $s = 4\log N$, and compare the instantiated SE-ORAM with several state-of-the-art ORAMs including T-ORAM [49], G-ORAM [16], Path ORAM [54] and P-PIR [37], in terms of storage and communication overheads.

<table>
<thead>
<tr>
<th>ORAM</th>
<th>Client Storage Overhead</th>
<th>Server Storage Overhead</th>
<th>Communication Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>T-ORAM [49]</td>
<td>$O(B)$</td>
<td>$O(N \log N \cdot B)$</td>
<td>$(\log^2 N \cdot B)$</td>
</tr>
<tr>
<td>G-ORAM [16]</td>
<td>$O(\log^2 N \cdot B)$</td>
<td>$O(N \cdot B)$</td>
<td>$(\log^2 N \cdot B)$</td>
</tr>
<tr>
<td>Path ORAM [54]</td>
<td>$O(\log N \cdot B)$</td>
<td>$O(N \cdot B)$</td>
<td>$O(\log N \cdot B)$</td>
</tr>
<tr>
<td>P-PIR [37]</td>
<td>$O(B)$</td>
<td>$O(N \log N \cdot B)$</td>
<td>$O(\log N \cdot B)$</td>
</tr>
<tr>
<td>SE-ORAM</td>
<td>$O(\log N \cdot B)$</td>
<td>0</td>
<td>$O(\log^2 N \cdot B)$</td>
</tr>
</tbody>
</table>

Table 4.1 compares SE-ORAM with state-of-the-art ORAM constructions in terms of the client and server storage overheads as well as the communication overhead per query. As we can see, SE-ORAM does not consume any extra storage in the server other than $N \cdot B$ bits for the $N$ data blocks. On the contrary, the server storage overhead of each of the state-of-the-art ORAM constructions is $O(N \cdot B)$ or $(N \log N \cdot B)$ bits. Though the communication cost of SE-ORAM is on the same level as T-ORAM, as discussed in Section 1, it can be reduced to $O(\log N \cdot B)$ by adopting the additive Homomorphic encryption-based PIR primitives [37], similar to the way that P-PIR reduced the communication cost of T-ORAM from $O(\log^2 N \cdot B)$ to $O(\log N \cdot B)$. 
4.6 Summary

In this chapter, we introduce a generic security definition for ORAM constructions, which allows a client to configure a desired security level and manage the tradeoffs between security and performance. We also propose SE-ORAM, a generic storage-efficient ORAM construction with configurable security parameter $\lambda$. The results of extensive analysis show that, SE-ORAM achieves the configured level of security, introduces zero storage overhead to the storage server (i.e., the storage server only stores $N$ data blocks), and incurs $O(\log N)$ blocks storage overhead at the client, as long as $\lambda \geq 2$ and each node on the storage tree stores $4 \log N$ or more data blocks.
CHAPTER 5. OCTOPUS ORAM

5.1 Research Goal and Contributions

In chapter 4, we proposed SE-ORAM [33], which reduces the server side storage overhead to zero, but the limitation is that the client-server communication cost is \(O(\log^2 N \cdot B)\), which is higher than some state-of-the-art. In this chapter, we aim at reduce the client-server communication cost, while keeping the server storage overhead to be low.

We propose a new ORAM scheme [34], to accomplish the efficiency in both communication and server-side storage. More specifically, our proposed ORAM can significantly reduce the server storage overhead to around \(0.34N\) (i.e., the server only needs to allocate \(1.34N\) blocks when the client outsources \(N\) blocks) while maintaining a comparable level of communication cost: to server a query, the online communication cost is 3 blocks and eviction (maintenance) communication cost is no more than \(1.5 \log N\) blocks.

In the following, Section 5.2 defines the problem. Section 5.3 describes our proposed ORAM, which is followed by the security analysis in Section 5.4. Section 5.5 reports the performance comparisons with Partition ORAM. Finally, Section 5.6 summarises the chapter.

5.2 Problem Definition

Following the prior research on ORAM constructions, we also assume the server to be semi-honest (or honest but curious); that is, it stores data and serves the client’s requests according to the protocol that we deploy, but it may attempt to figure out the client’s access pattern.

Assume the client outsources \(N\) equal-size data blocks to the cloud storage server, and then needs to access the outsourced data every now and then. Each data access intended by the client, which should be kept private, is one of the following two types:
• Read a data block $D$ of unique ID $i$ from the storage, denoted as $D = (read, i)$ and formally a 3-tuple $(read, i, D)$; or

• Write a data block $D$ of unique ID $i$ to the storage, denoted as a 3-tuple $(write, i, D)$.

To accomplish a private data access, the client usually needs to access multiple locations of the storage. Each location access, which can be observed by the server, is one of the following types:

• Retrieve (i.e., read) a data block $D$ from location $l$ at the storage, denoted as $D = (read, l)$ and formally a 3-tuple $(read, l, D)$; or

• Upload (i.e., write) a data block $D$ to location $l$ at the storage, denoted as a 3-tuple $(write, l, D)$.

Similar to the security definition of ORAM in the prior research [19, 54, 53], we define the security of our proposed ORAM as follows.

**Definition** Let $\lambda$ be a security parameter, and $\vec{x} = \langle (op_1, i_1, D_1), (op_2, i_2, D_2), \cdots \rangle$ denote a private sequence of the client’s data accesses, where each $op$ is either a $read$ or $write$. Let $A(\vec{x}) = \langle (op'_1, l_1, D'_1), (op'_2, l_2, D'_2), \cdots \rangle$ denote the sequence of the client’s location accesses (observed by the server) in order to accomplish the data access sequence $\vec{x}$. An ORAM system is said to be secure if:

1. For any two equal-length private sequences $\vec{x}$ and $\vec{y}$ of data accesses, their corresponding location access sequences $A(\vec{x})$ and $A(\vec{y})$ are computationally indistinguishable; and

2. The ORAM system fails to operate with a probability of $O(2^{-\lambda})$.

### 5.3 The Proposed ORAM Scheme

To accomplish both server storage and communication efficiencies, we propose a new tree-based ORAM scheme.

Like the existing tree-based ORAM schemes [49, 54], we organize the cloud server storage into a tree structure. Whenever a data block is outsourced to the server, it is randomly assigned to a path,
and can only stay in the path until it is retrieved to the client. Hence, the leaf nodes are required to offer a space large enough to contain all the blocks that are randomly distributed to them; meanwhile, the non-leaf nodes need to offer an additional space to buffer data blocks when they are gradually evicted to the leaf nodes. Consequently, a high degree of storage space redundancy exists in most of the existing tree-based ORAM schemes. To reduce storage redundancy, we first adopt the following optimizations:

- To minimize the storage space of leaf nodes, we deliberately make the size of each leaf node to be very large and only a small fraction of redundant space is introduced to ensure a negligible probability of space overflow.

- To minimize the storage space of non-leaf nodes, we organize the tree as a $k$-ary ($k > 2$) instead of binary tree to make the non-leaf nodes to have much smaller storage than the leaf nodes.

The above optimizations, however, may significantly sacrifice the communication efficiency due to the resulting large node sizes. In particular, the size of non-leaf nodes increases rapidly with $k$. To address the conflicts, we further design new algorithms for data query and data eviction. The delicately designed data query algorithm makes the communication cost per query to be a constant independently of either the node size or the height of the tree. Based on a carefully selected value $k$ (i.e., eight) that balances the tradeoff between storage and communication efficiency, we design an eviction scheme that runs after a certain large number of queries to evict a large batch of data blocks at once; meanwhile, the cost of eviction can be spread over a long period of time through the widely-adopted de-amortization technique [49, 53].

In the following, we present the detail of the scheme in terms of storage organization, system initialization, data query, and data eviction. Here, $N$ denotes the number of real data blocks that the client wishes to export to the server. Adjustable system parameters, including $s > 1$, $0 < \alpha < 1$, $0 < \beta < 1$ and $0 < \gamma < 1$, are introduced for the tradeoffs between security and costs. $s$ is a parameter controls the size of each node, $\alpha$ and $\gamma$ controls how many dummy blocks introduced
in a non-leaf node and $\beta$ controls how many dummy blocks in a leaf node. The detail analysis of these parameters can be found in Section 5.4.

5.3.1 Server-side Storage

The server-side storage is organized as a balanced tree, called storage tree, in which each non-leaf node has up to eight child nodes. To hide the client’s access pattern to the server storage, the server stores two types of data blocks: real blocks that each stores the user’s data and has block ID in \{1, \cdots, N\}; dummy blocks that each is randomly-generated based on some seed and has block ID 0.

Specifically, let $L' = \lfloor \log_8 \frac{N}{3.5s} \rfloor$ and $Z' = \frac{N}{8^L}$. If $3.5s \leq Z' \leq 7s$, the storage tree is a complete 8-ary tree with height $L = L' + 1$ and each leaf node storing $Z_0 = (1 + \beta)Z'$ blocks. Otherwise, the storage tree is of height $L = L' + 2$, and the root has $\lfloor \frac{Z'}{3.5s} \rfloor$ child nodes while each child node is a root of a complete 8-ary tree with each leaf node storing $Z_0 = (1 + \beta)\frac{Z'}{\lfloor \frac{Z'}{3.5s} \rfloor}$ blocks. Each non-leaf node has a capacity of $Z_1 = 3.5(1 + \alpha)s + (1 + \gamma)s$ blocks. Each node $n_i$ is identified by a unique tuple $(l_i, id_i)$, where $l_i \in \{0, \cdots, L - 1\}$ is the ID of the layer that the node resides (note: the root node is at layer 0 while the leaf nodes are at layer $L - 1$), and $id_i \geq 0$ is the ID of the node on layer $l_i$ that indicates the order of the node on layer $l_i$ (from 0 at the leftmost to right).

For simplicity of presentation, we assume each non-leaf node has exactly eight child nodes hereafter. Fig 5.1 illustrates the storage tree.

5.3.2 Client-side Storage

The client maintains the following types of storage.

A position map is a table with $N$ entries, where each entry $i \in \{1, \cdots, N\}$ records the path ID of block $i$ (i.e., the ID of the leaf node on the path storing block $i$). Like other tree-based ORAMs [49, 53, 54, 51], we also guarantees the following invariant: each real block is stored only on the path identified by the path ID of the block.
For each node on the tree, the client keeps an index block which has one entry \((id, ah)\) for each block stored in the node. Here, \(id\) denotes the ID of the block and \(ah \in \{0, 1, 2\}\) indicates the access history of the block since the system initialization or the most recent data eviction process involving the node, whichever is more recent; specifically (i) \(ah = 0\) if the block has not been accessed; (ii) \(ah = 1\) if the block has been accessed as a query target; (iii) \(ah = 2\) if the block has been accessed, but never as a query target.

The client keeps a random seed \(seed\) and an eviction count \(c_e\) to track the number of evictions that have been conducted so far. These two pieces of information are used to generate dummy blocks. Specifically, when a dummy block needs to be generated to fill offset \(\phi\) at node \((l, id)\), the client first calculates the version number \(v\) of the node based on \(c_e\) and then the dummy block is generated as \(PRF(seed, l, id, \phi, v)\), where \(PRF\) is a pseudo-random generator that produces a block based on the given inputs. The version number \(v\) of each node is initialized as 0 and incremented after every eviction involving the node. For example, letting \(v_{(l, id)}\) denote the version number of node \((l, id)\), then \(v_{(2, 0)}\) is incremented when \(c_e = 1, 5, 9, \ldots\), \(v_{(2, 2)}\) is incremented when \(c_e = 2, 6, 10, \ldots\), \(v_{(2, 1)}\) is incremented when \(c_e = 3, 7, 11, \ldots\), and \(v_{(2, 3)}\) is incremented when \(c_e = \ldots\).
In addition, the client allocates temporary buffers to store blocks during query and eviction process, and some small storage to store secret keys used in encryption and decryption.

5.3.3 System Initialization

To initialize the system, the client encrypts all the $N$ real data blocks using a certain probabilistic encryption algorithm (e.g., AES with different initial vector for each encryption), randomly selects a path for each block, and stores the blocks to the leaf nodes such that each block is at the leaf node on the path selected for it. To hide the initial distribution of blocks to the leaf nodes, the leaf nodes are also filled with dummy blocks to make each of them to have exactly $Z_0$ real or dummy blocks. The client also initiates each non-leaf nodes by filling it with $Z_1$ dummy blocks.

5.3.4 Data Query

Suppose the client wishes to query block $D_t$ not in its local buffer, where $t \in \{0, \cdots, N - 1\}$ denotes the ID of the block. It looks up the position map to obtain $p_t$, which is the path ID of the node, and looks up the index blocks of path $p_t$ to identify the node that stores $D_t$. Then, the client selects path $p_t$ as the query path and launches the following query process.

Selecting Blocks to Access  For each non-leaf node $n'_i$ on path $p_t$ (where $i = 0, \cdots, L - 2$ represents the layer ID of the node), the client selects one block to access according to the following rules: if $n'_i$ contains $D_t$, $D_t$ is selected; otherwise, a non-accessed dummy block is randomly selected.

For the leaf node on $p_t$, the client selects one or two blocks to download. Specifically, let $S_0$, $S_1$ and $S_2$ denote the sets of blocks with $ah$ values 0, 1 and 2 respectively, and $s_0$, $s_1$ and $s_2$ denote the cardinalities of the sets. Then the client applies the following rules: If the leaf node contains $D_t$, depending on the access history of $D_t$, there are following cases.
• **Case I:** \( D_t \in S_0 \). The client picks the target to download. If \( s_1 + s_2 > 0 \), the client also randomly picks one block from \( S_1 \) with probability \( \rho \) or from \( S_2 \) otherwise (i.e., with probability \( 1 - \rho \)) to access, where

\[
\rho = \frac{s_1(s_0 + s_2)}{s_0(s_1 + s_2)}. \tag{5.1}
\]

• **Case II:** \( D_t \in S_1 \cup S_2 \). The client picks the target to download and meanwhile, picks another block randomly from \( S_0 \) to access.

If the leaf node does not contain the query target, the client randomly picks one block from \( S_0 \) and another one from \( S_1 \cup S_2 \) to access.

To summarize, let \( \phi_0, \cdots, \phi_{L-2} \) denote the offsets of the blocks selected from layers \( 0, \cdots, L-2 \) respectively, and \( \phi_{L-1,0} \) and \( \phi_{L-1,1} \) denote the offsets of the blocks selected from the leaf node.

**Downloading Blocks** The client sends request \( Q = (\phi_0, \cdots, \phi_{L-2}, \phi_{L-1,0}, \phi_{L-1,1}) \) to the server. Upon receiving the request, the server picks block \( D'_i \) from offset \( \phi_i \) in node \( n'_i \) of path \( p_t \), for each \( i \in \{0, \cdots, L-2\} \). Then, it conducts bit-wise XOR on each group of bits with the same offsets in these blocks, to obtain block \( D' = D'_0 \oplus \cdots \oplus D'_{L-2} \). After that, the server sends to the client \( D' \) together with blocks \( D'_{L-1,0} \) and \( D'_{L-1,1} \) located at offsets \( \phi_{L-1,0} \) and \( \phi_{L-1,1} \) in the leaf node.

**Recovering Target Block** Upon receiving the three blocks from the server, if the target block is one of \( D'_{L-1,0} \) and \( D'_{L-1,1} \), the client gets the target block immediately. Otherwise, the client re-generates target block from \( D' \) as follows. The client re-generates each dummy block \( D'_i \) selected from offset \( \phi \) at node \( (l, id) \), by \( D'_i = PRF(seed, l, id, \phi, v) \) where \( v \) is the version of node \( (l, id) \). Then, \( D_t = D' \oplus (\oplus_{(l, id)} D'_i \neq D_t) D'_i \).

**Updating Index Blocks** Finally the client updates the index blocks maintained by itself by: marking the copy of target block remaining on the storage tree as a dummy block that has been accessed as target; marking the other selected blocks as have been accessed.
Fig 5.2 shows examples of query process: In (a), the query target block is a non-accessed block located at a non-leaf node, while in (b), the query target block is located at a leaf node and has been accessed before. In both cases, one non-accessed block is always accessed from each non-leaf node on the query path while two blocks (one has been accessed and the other has not) are accessed from the leaf node. The pattern is independent of where the query target resides.

![Figure 5.2 Query Examples.](image)

### 5.3.5 Data Eviction

After every $s$ queries, the client has retained at its buffer $s$ blocks that are the targets of the most recent $s$ queries. We call these blocks as the current *evicting blocks*. The client randomly re-selects a path ID for each evicting block, and then launches a data eviction process. Note that, like in some existing ORAM constructions [53], the eviction (or shuffling) process can be carried out concurrently with data query processes and the process can spread over a long period of time. Our proposed eviction process can be de-amortized using the similar technology. To focus on the
basic ideas of this construction, we assume here that the eviction process is executed before any further data query is processed.

Each eviction process involves only one path, which we call eviction path, of the storage tree. The eviction path is selected in the reverse-lexicographic order, as used in [11]. The eviction process runs iteratively, one iteration for each node on the eviction path. We introduce variable $n'_e$ to denote the node currently involved in the eviction. Hence, $n'_e$ is initialized to $n_{0,0}$ (i.e., the root node).

To facilitate the explanation, we further divide the real blocks in $n'_e$, when $n'_e$ is not a leaf node, into up to eight groups denoted as $g_0, \cdots, g_{x-1}$ where $x$ represents the number of child nodes of $n'_e$. Here, $g_i$ for $i = 0, \cdots, x-1$ is the set of real blocks in $n'_e$ that can only be evicted to a child denoted as $n'_{e,i}$ of $n'_e$, because the paths associated with the blocks in $g_i$ all pass through $n'_e$ and $n'_{e,i}$. Note that, the grouping is just a temporary and logical grouping, which is known only by the client during eviction without requiring any data structure to keep the state. Next, we elaborate the operations of each iteration, which is also illustrated in Fig 5.3.

When $n'_e$ is a non-leaf node and its child $n'_{e,c}$ ($c \in \{0, \cdots, x-1\}$) is the next node on the eviction path, the eviction is conducted as follows. First, the client retrieves the $3.5(1+\alpha)s$ blocks, which must include the non-accessed real blocks, from $n'_e$; note that there is no need to retrieve the other $(1+\gamma)s$ blocks, which must include the already-accessed blocks, as it is public knowledge that at least $(1+\gamma)s$ blocks are dummy. The real blocks from $n'_e$ are divided into $x$ groups, and the current evicting blocks at the client’s buffer are distributed into the $x$ groups according to their path IDs.

Second, the client merges the real blocks in groups $c+1, \cdots, x-1, 0, \cdots, c-1$, re-encrypts them, randomly permutes them, inserts randomly-generated dummy blocks to them to make the total number of such blocks be $3.5(1+\alpha)s + (1+\gamma)s$, and uploads the blocks back to $n'_e$.

Third, the client retains the remaining real blocks, i.e., the blocks in group $c$, at the buffer. These blocks now become the new set of current evicting blocks, which will be evicted to child node $n'_{e,c}$ in the next iteration. Also, the client updates variable $n'_e$ to represent child node $n'_{e,c}$; that is, $n'_{e,c}$ now becomes the evicting node. Then, the next iteration of the eviction process starts.
When $n'_e$ is a leaf node, the client downloads all the blocks currently in $n'_e$, and merges the blocks with the current evicting blocks. Among these blocks, if the number of real blocks is more than $Z_0$, the client declares failure and aborts. Otherwise, the client adds or removes dummy blocks to the make the total number of blocks to be $Z_0$, and randomly permutes and re-encrypts them before uploading them back to $n'_e$. 

Figure 5.3  Eviction Overview.
5.4 Security Analysis

Now we study the failure probability and the obliviousness of the proposed ORAM.

5.4.1 Failure Probability Analysis

5.4.1.1 Failure Probability for A Query Process

According to Section 5.3, a query process fails at a node \( n_i \) on layer \( i \), for two cases:

- the \( \rho \) computed according to Equation (5.1) is greater than 1; i.e., \( s_{i,1} \cdot (s_{i,0} + s_{i,2}) > s_{i,0} \cdot (s_{i,1} + s_{i,2}) \), a.k.a., \( s_{i,1} > s_{i,0} \);

- when \( n_i \) is a non-leaf node that does not contain the target block, but it does not have non-accessed dummy blocks.

Lemma 5. The probability for a non-leaf node to use up \((1 + \gamma)s\) dummy blocks before it involves an eviction process is \(O(2^{-\lambda})\), as long as \( \gamma \geq 0.25 \) and \( s \geq 25\lambda \).

Proof. Between two consecutive evictions, the average time for a non-leaf node to be on a query path is \( s \). Hence, due to the Chernoff bound, the probability for the node to be on a query path for more than \((1 + \gamma)s\) times is no greater than

\[
\left( \frac{e^{\gamma}}{(1 + \gamma)^{1+\gamma}} \right)^s,
\]

which is less than \(2^{-\lambda}\) as long as \( s \geq 25\lambda \) and \( \gamma \geq 0.25 \).

As it is hard to directly compute \( Pr[s_{i,1} > s_{i,0}] \), we instead compute \( Pr[s_{i,1} + s_{i,2} > s_{i,0}] \). Since \( Pr[s_{i,1} + s_{i,2} > s_{i,0}] \geq Pr[s_{i,1} > s_{i,0}] \), \( Pr[s_{i,1} + s_{i,2} > s_{i,0}] \) is an upper bound of the failure probability of a query process.
Lemma 6. Let \( n_i \) denote an arbitrary node on layer \( i \) of the storage tree, and \( \xi_i \) denote the total number of nodes on the layer. If \( n_i \) is involved in at least one eviction process after every \( x \) queries launched by the client, then a query process fails at \( n_i \) with a probability no greater than

\[
\left( \frac{e^{\gamma}}{(1 + \gamma)^{1+\gamma}} \right)^x/\xi_i,
\]

where \( \gamma = \frac{\xi_i \cdot 3.5(1+\alpha)s}{2x} - 1 \).

Proof. For every \( x \) queries launched by the client, on average there are \( \tilde{q} = \frac{x}{\xi_i} \) query processes that have \( n_i \) on their selected query paths, because of the randomness in the path selection for blocks and that layer \( i \) has \( \xi_i \) blocks.

Therefore, the probability for \( \hat{q} = \frac{3.5(1+\alpha)s}{2} \) or more of these queries to select \( n_i \) on their query paths is less than

\[
\left( \frac{e^{\gamma}}{(1 + \gamma)^{1+\gamma}} \right)^{\hat{q}/\xi_i},
\]

where \( \gamma = \frac{\hat{q}}{\tilde{q}} - 1 = \frac{\xi_i \cdot 3.5(1+\alpha)s}{2x} - 1 \), according to the multiplicative Chernoff bound.

Note that, a query will not fail at \( n_i \) if \( n_i \) has not been on the query paths for \( \hat{q} \) times, because \( s_{i,0} > s_{i,1} + s_{i,2} \) if \( n_i \) has been on less than \( \hat{q} \) query paths.

Hence the lemma is proved.

Based on the above Lemmas, we have the following main theorem regarding the failure probability of a query process.

Theorem 4. (Failure Probability for A Query Process.) When an eviction process is always launched after every \( s \) queries and completed before any further query, a query process fails at a node with a probability of \( O(2^{-\lambda}) \) as long as \( \alpha \geq 0.34, \gamma \geq 0.25 \) and \( s \geq 25\lambda \).

Proof. On the storage tree, for every node \( n_i \) on layer \( i \) with totally \( \xi_i \) nodes, it is involved in an eviction after every \( x = \xi_i \cdot s \) queries. Then, applying Lemma 5, 6 with \( x = \xi_i \cdot s \), the theorem is proved.
5.4.1.2 Failure Probability for An Eviction Process

An eviction process can fail if and only if one of the following scenarios occurs. (i) **Failure Scenario I:** This scenario occurs during an eviction iteration (detailed in Section 5.3.5) with a non-leaf node as the current evicting node. After the current evicting blocks are merged with the existing blocks at the current evicting node, there are more than \( 3.5(1 + \alpha)s \) real blocks that can only be evicted to the child nodes other than the next evicting node. This would require more than \( 3.5(1 + \alpha)s \) real blocks (i.e., less than \( (1 + \gamma)s \) dummy blocks) to be uploaded to a non-leaf node and thus would lead to space overflow at the node. (ii) **Failure Scenario II:** This scenario occurs during the eviction iteration with a leaf node (with capacity \( Z_0 \) blocks) as the current evicting node. When the current evicting blocks are merged with the existing blocks at the current evicting node, the total number of real blocks become more than \( Z_0 \).

**Lemma 7.** As long as \( \alpha \geq 0.34 \) and \( s \geq 25\lambda \), the Failure Scenario I occurs with a probability of \( O(2^{-\lambda}) \).

**Proof.** Let us consider a non-leaf current evicting node \( n'_e \). Without loss of generality, we assume the leftmost child (i.e., child 0) of \( n'_e \) is the next evicting node. After the current evicting blocks have been merged with the blocks in \( n'_e \), all of these blocks are grouped into eight groups, where each group \( g_i \) for \( i = 0, \cdots, 7 \) includes the real blocks that can only be evicted to child \( i \).

Each \( g_i \) includes the real blocks evicted to \( n'_e \) in the last \( 8 - i \) eviction processes that involve \( n'_e \). Since, on average, each eviction process involving \( n'_e \) evicts \( s \) real blocks to \( n'_e \) and \( \frac{s}{8} \) real blocks can only be evicted to child \( i \) of \( n'_e \), the average size of \( g_i \), denoted as \( |g_i| \), is \( \frac{8-i}{8} \cdot s \) blocks. Thus, according to the multiplicative Chernoff bound, the probability for \( |g_i| > \frac{8-i}{8} \cdot s \cdot (1 + \alpha_i) \), i.e. \( Pr[|g_i| > \frac{8-i}{8} \cdot s \cdot (1 + \alpha_i)] \), is less than

\[
\left[ \frac{e^{\alpha_i}}{(1 + \alpha_i)(1 + \alpha_i)} \right]^{\frac{8-i}{8} \cdot s}.
\] (5.3)

According to inequality (5.3), when \( s \geq 25\lambda \), \( \alpha_1 \geq 0.265 \), \( \alpha_2 \geq 0.285 \), \( \alpha_3 \geq 0.31 \), \( \alpha_4 \geq 0.35 \), \( \alpha_5 \geq 0.41 \), \( \alpha_6 \geq 0.5 \), and \( \alpha_7 \geq 0.74 \), it holds that \( Pr[|g_i| > \frac{8-i}{8} \cdot s \cdot (1 + \alpha_i)] < 2^{-\lambda} \). That is, when
\[
\alpha = \frac{\sum_{i=1}^{7} [\frac{8i+1}{s} \cdot (1+\alpha_i)]}{\sum_{i=1}^{7} \frac{8i-1}{s}} \geq 0.339, \text{ the Failure Scenario I occurs with a probability less than } 7 \cdot 2^{-\lambda}, \text{ i.e., } O(2^{-\lambda}).
\]

**Lemma 8.** As long as \( \beta \geq 0.13 \) and \( s \geq 25\lambda \), the Failure Scenario II occurs with a probability of \( O(2^{-\lambda}) \).

**Proof.** According to the server storage organization, each leaf node has a capacity of \( Z' \) blocks where \( Z' \geq 3.5(1+\beta)s \). For each leaf node with ID \( i \), let us use random variable \( x_i \) to denote the number of real blocks that have \( i \) as their path IDs and \( \bar{x}_i \) to denote the mean of \( x_i \). Thus, it holds that \( \bar{x}_i = \frac{Z'}{1+\beta} \) due to the following reasoning: Each node has exactly eight child nodes, the total number of leaf nodes is \( 8^L' \), and thus \( \bar{x}_i = \frac{N}{8^L'} \). Meanwhile, \( Z' = (1+\beta) \frac{N}{8^L} \). Therefore, \( \bar{x}_i = \frac{Z'}{1+\beta} \).

Furthermore, due to the multiplicative Chernoff bound, for any leaf node with ID \( i \):

\[
Pr[x_i \geq Z'] = Pr[x_i \geq (1+\beta)\bar{x}_i] < \left( \frac{e^\beta}{(1+\beta)^{1+\beta}} \right)^{\bar{x}_i}.
\] (5.4)

Considering that \( \bar{x}_i \geq 3.5s \), \( Pr[x_i \geq Z'] < 2^{-\lambda} \) when \( \beta \geq 0.13 \) and \( s \geq 25\lambda \).

Hence, the Lemma is proved.

**Theorem 5.** (Failure Probability of an Eviction Process.) An eviction process fails at a node with a probability of \( O(2^{-\lambda}) \), as long as \( \alpha \geq 0.34 \), \( \beta \geq 0.13 \) and \( s \geq 25\lambda \).

**Proof.** (Proved based on Lemmas 7 and 8.)

Let consider an eviction step and introduce the following notations:

- \( n_i \): the current evicting node, which is a non-leaf node at layer \( i \).
- \( E_i \): the set of the current evicting blocks.
- \( S_i^m \): the set of target blocks for the most recent \( m \cdot s \) queries.
Lemma 9. $E_i$ is the subset of $S_i^{8^i}$ that must be evicted to the subtree rooted at $n_i$.

Proof. (By induction on layer $i$.) Base Case: When $i = 0$ (i.e., the root node is the current evicting node), $E_i$ is the set of the most recently queried $s = 8^i$ blocks, which must all be evicted to $n_0$.

Induction step: Suppose the proposition holds for all the nodes on layer $i = 0, \cdots, l - 1$. When $i = l$ (i.e., the current evicting node $n_l$ is at layer $l$), let $n_{l-1}$ be the parent of $n_l$. $E_l$ must be from the blocks that were evicted to $n_{l-1}$ during the most recent 8 eviction processes involving this parent, and only the blocks that must be evicted to the subtree rooted at $n_l$. Note that, when $n_{l-1}$ was the current evicting node, the current evicting blocks were from $S_{l-1}^{8^{l-1}}$, which was the most recent $8^{l-1} \cdots s$ queries at the time. Thus, $E_l$ must be from the most recent $8^l \cdot s$ queries, i.e., $S_l^{8^l}$.

Therefore, as $E_l$ is from 8 of such sets, it holds that $E_i$ is the subset of $S_l^{8^l}$ that must be evicted to the subtree rooted at $n_l$.

Hence, the lemma is proved based on the above steps. \hfill $\blacksquare$

Lemma 10. For each child node $n_{i,c}$ ($c \in \{0, \cdots, 7\}$) of $n_i$, with a probability of at least $1 - \left(\frac{e^\alpha}{(1+\alpha)(1+\alpha)}\right)^{s/8}$, no more than $(1 + \alpha)^{s/8}$ blocks among $E_i$ must be evicted to $n_{i,c}$.

Proof. Let $x_{i,c}$ denote the number of blocks in $E_i$ that must be evicted to the subtree rooted at $n_{i,c}$. The average value of $x_{i,c}$ is $\frac{s}{8}$ because: (i) there are $8^{i+1}$ distinct nodes on layer $i + 1$ (i.e., the layer $n_{i,c}$ resides); (ii) according to Lemma 9, $E_i$ is a subset of $S_i^{8^i}$ (with $8^i$ blocks) that must be evicted to the subtree rooted at $n_i$. According to the multiplicative Chernoff bound, $x_{i,c} > \frac{(1+\alpha)s}{8}$ with a probability less than $\left(\frac{e^\alpha}{(1+\alpha)(1+\alpha)}\right)^{s/8}$. That is,

$$Pr[x_{i,c} \leq \left(\frac{(1+\alpha)s}{8}\right)] > 1 - \left(\frac{e^\alpha}{(1+\alpha)(1+\alpha)}\right)^{s/8}.$$ 

Hence, the lemma is proved. \hfill $\blacksquare$

Lemma 11. Assume $n_{i,c}$, where $c \in \{0, \cdots, 7\}$, be the child of $n_i$ that is the next evicting node. After the blocks in $E_i$ have been evicted to node $n_i$, with a probability of at least

$$1 - 28\left(\frac{e^\alpha}{(1+\alpha)(1+\alpha)}\right)^{s/8},$$
there are less than \(3.5(1 + \alpha)s\) blocks in \(n_i\) that must be evicted to the child nodes of \(n_i\) other than \(n_{i,c}\).

**Proof.** After the blocks in \(E_i\) have been evicted to node \(n_i\), the blocks in \(n_i\) that must be evicted to child node \(n_{i,c+j} \mod 8\), which we denoted as \(S_{i,(c+j)\%8}\) where \(j = 1, \cdots, 7\), are from the blocks evicted to \(n_i\) in the last \(8 - j\) eviction processes that involve \(n_i\). According to Lemma 10, \(S_{i,(c+j)\%8}\) has no more than \(\frac{(8-j)(1+\alpha)s}{8}\) blocks with a probability of at least

\[
(1 - \left(\frac{e^\alpha}{(1 + \alpha)(1+\alpha)}\right)^{s/8})^{8-j}.
\]

So, \(\bigcup_{j=1}^{7} S_{i,(c+j)\%8}\) has no more than

\[
\sum_{j=1}^{7} \frac{(8-j)(1+\alpha)s}{8} = 28(1 + \alpha)s = 3.5(1 + \alpha)s
\]

blocks with a probability of at least

\[
(1 - \left(\frac{e^\alpha}{(1 + \alpha)(1+\alpha)}\right)^{s/8})^{28} > 1 - 28\left(\frac{e^\alpha}{(1 + \alpha)(1+\alpha)}\right)^{s/8}.
\]

Hence, the lemma is proved.

Second, we study Failure Scenario II, by stating and proving the following Lemma.

**Lemma 12.** At any time point, with a probability at least

\[
1 - \left(\frac{e^\beta}{(1 + \beta)(1+\beta)}\right)^{\frac{Z}{7s}},
\]

there are no more than \(Z\) real blocks in a leaf node.

**Proof.** The server-side storage tree has \(\frac{(1+\beta)N}{Z}\) leaf nodes, i.e., \(\frac{N}{7s}\) root-to-leaf paths; each of the \(N\) real blocks is randomly associated with one of these paths. Hence, for each leaf node with ID \((L - 1, j)\), where \(j \in \{0, \frac{N}{7s} - 1\}\), on average there are \(7s\) real blocks have path ID equal to \(j\). According to the multiplicative Chernoff Bound, the probability for no more than \(7(1 + \beta)s\) blocks to have path ID \(j\) is at least

\[
1 - \left(\frac{e^\beta}{(1 + \beta)(1+\beta)}\right)^{7s}.
\]

Hence, the lemma is proved.
5.4.2 Obliviousness Analysis

5.4.2.1 Obliviousness in Query Path Selection

When the system is initialized, the path ID of each block is selected randomly and independently of each other. After a block has been queried, its path ID is re-selected randomly and independently of the client’s data access pattern. Due to the randomness in the selection of path ID, the query path of each query process, which is determined by the path ID of the query target block, is random and independent of the client’s access pattern.

5.4.2.2 Obliviousness in Block Access from Query Path

According to the data query algorithm, the following access pattern has been followed: for a non-leaf node, if it contains the target block, the target block is accessed, otherwise, a randomly non-accessed dummy block is accessed. So the block accessed from a non-leaf node must have not been accessed before and is randomly selected. For a leaf node, the client must select one block that has already been accessed and one block that has not been accessed to access. Furthermore, the following Lemma 13 shows that, each of the blocks that have already been accessed has the same probability to be selected and each of the blocks that have not been accessed also has the same probability to be selected. Hence, each query process is also random and independent of the data access pattern.

Lemma 13. In every node on the query path selected for a query process, all the un-accessed blocks within the node have the same probability to be accessed; all the already-accessed blocks within the node have the same probability to be accessed.

Proof. Let us re-use the notations used in the data query algorithm (Section 5.3). For any node \( n'_i \neq n_t \) (i.e., the node does not contain the query target) on the query path, there are two cases:

- If the node has not been accessed since its most recent construction, one block is randomly selected to access; i.e., each block has the same probability \( \frac{1}{2^4s} \) to be accessed.
• If the node has been accessed before, one block is randomly selected from \( S_{i,0} \) to access (i.e., each block is \( S_{i,0} \) has the same probability of \( \frac{1}{s_{i,0}} \) to be accessed), and another block is randomly selected from \( S_{i,1} \cup S_{i,2} \) to access (i.e., each block in \( S_{i,1} \cup S_{i,2} \) has the same probability of \( \frac{1}{s_{i,1} + s_{i,2}} \) to be accessed).

For node \( n'_i = n_t \) (i.e., the node contains the query target) on the query path, if the node has not been accessed since its most recent eviction involving it (note: the system initialization is treated as an eviction process), \( D_t \) is any block on the node with the same probability \( \frac{1}{s_{i,0} + s_{i,1} + s_{i,2}} \). According to the query algorithm, \( D_t \) is selected to access; that is, each block has the same probability of \( \frac{1}{s_{i,0} + s_{i,1} + s_{i,2}} \) to be accessed.

If the node has been accessed since its most recent eviction, \( S_{i,0} \) is the set of blocks that have not been accessed, \( S_{i,1} \) is the set of blocks that have been accessed before as target, and \( S_{i,2} \) is the set of blocks that have been accessed before as non-target (i.e., dummy). \( D_t \) has the same probability to be any block belonging to \( S_{i,0} \cup S_{i,2} \); that is, it is in \( S_{i,0} \) with probability \( \frac{s_{i,0}}{s_{i,0} + s_{i,2}} \) or in \( S_{i,2} \) with probability \( \frac{s_{i,2}}{s_{i,0} + s_{i,2}} \). According to the query algorithm:

• If \( D_t \) is in \( S_{i,0} \) (occurring with probability \( \frac{s_{i,0}}{s_{i,0} + s_{i,2}} \)), \( D_t \) is selected to access (i.e., each block in \( S_{i,0} \) has the probability of \( \frac{1}{s_{i,0}} \) to be accessed); meanwhile, another block is randomly selected from \( S_{i,1} \) to access with probability \( \rho \) (i.e., each block in \( S_{i,1} \) has the probability of \( \frac{s_{i,0} + s_{i,2}}{s_{i,0}(s_{i,1} + s_{i,2})} \) to be accessed) or from \( S_{i,2} \) otherwise (i.e., each block in \( S_{i,2} \) has the probability of \( \frac{1-\rho}{s_{i,2}} = \frac{s_{i,0} - s_{i,1}}{s_{i,0}(s_{i,1} + s_{i,2})} \) to be accessed).

• If \( D_t \) is in \( S_{i,2} \) (occurring with probability \( \frac{s_{i,2}}{s_{i,0} + s_{i,2}} \)), \( D_t \) is selected to access (i.e., each block in \( S_{i,2} \) has the probability of \( \frac{1}{s_{i,2}} \) to be accessed); meanwhile, another block is randomly selected from \( S_{i,0} \) to access (i.e., each block in \( S_{i,0} \) has the probability of \( \frac{1}{s_{i,0}} \) to be accessed).

Summarizing the above two cases, each block in \( S_{i,0} \) always has the probability of \( \frac{1}{s_{i,0}} \) to be accessed; each block in \( S_{i,1} \) has the probability of

\[
\frac{s_{i,0}}{s_{i,0} + s_{i,2}} \cdot \frac{s_{i,0} + s_{i,2}}{s_{i,0}(s_{i,1} + s_{i,2})} = \frac{1}{s_{i,1} + s_{i,2}}
\]
to be accessed; each block in $S_{i,2}$ also has the probability of

$$\frac{s_{i,0}}{s_{i,0} + s_{i,2}} \cdot \frac{s_{i,0} - s_{i,1}}{s_{i,0}(s_{i,1} + s_{i,2})} + \frac{s_{i,2}}{s_{i,0} + s_{i,2}} \cdot \frac{1}{s_{i,1} + s_{i,2}} = \frac{1}{s_{i,1} + s_{i,2}}$$

to be accessed.

Hence, the lemma is proved. \hfill \Box

5.4.2.3 Obliviousness in eviction process

The eviction process is random and independent of the client’s data access pattern, due to the following reasons: (i) Each eviction process involves only one root-to-leaf path (called eviction path), and the order in which the paths are selected for as eviction paths is fixed and independent of data access pattern. (ii) During each eviction process, the processing for each node on the selected eviction path follows a fixed pattern which is independent of data access pattern: all the data blocks on the node are retrieved to the client; the blocks are all re-encrypted by the client; then, the same number of blocks (but may or may not be the same set of blocks) are uploaded back to the node.

Based on the above analysis, along with the fact that one eviction process always follows every $s$ query processes, we get the following theorem.

Theorem 6. As long as system parameters $\alpha \geq 0.34$, $\beta \geq 0.13$, $\gamma \geq 0.25$ and $s \geq 25\lambda$, the proposed ORAM is secure under Definition 5.2.

5.5 Performance Comparison

We have implemented our proposed Octopus ORAM, and conducted performance comparisons between Octopus ORAM and several state-of-the-art ORAM constructions that are the most related. Specifically, we conduct the following comparisons:

- We compare Octopus ORAM to Partition ORAM [53], which is one of the most communication-efficient ORAM that does not require intensive computation or multiple servers and shares
the same assumption with Octopus ORAM in that the client has a decent amount of local storage space (in particular, it assumes the client to have a local storage of $O(\sqrt{N})$ blocks).

- We compare Octopus ORAM with Path ORAM, which is one of the most communication-efficient ORAM that does not require intensive computation and only requires a small local storage (i.e., $O(\log N)$ blocks).

5.5.1 Comparison with Partition ORAM

In the evaluation, the system parameters $s$, $\alpha$, $\beta$ and $\gamma$ are adapted according to $N$ (i.e., the number of outsourced real blocks) to make the client-side storage size similar to that of Partition ORAM while assuring the failure probability of our constructions to be lower than $2^{-40}$. Note that, we have not implemented the Partition ORAM due to the lack of details on optimizations adopted by that design; so we use the performance results reported in [53] in this comparison. Also the block size is set to 64 KB, as used in [53].

<table>
<thead>
<tr>
<th>Capacity</th>
<th># Blocks</th>
<th>Client Storage Cost</th>
<th>Server Storage Overhead</th>
<th>Communication Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Partition ORAM</td>
<td>Octopus ORAM</td>
<td>Partition ORAM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>204 MB</td>
<td>289.2 MB</td>
<td>141 GB</td>
</tr>
<tr>
<td>64 GB</td>
<td>$2^{20}$</td>
<td>415 MB</td>
<td>292.7 MB</td>
<td>563 GB</td>
</tr>
<tr>
<td>256 GB</td>
<td>$2^{22}$</td>
<td>858 MB</td>
<td>307 MB</td>
<td>2.2 TB</td>
</tr>
<tr>
<td>1 TB</td>
<td>$2^{24}$</td>
<td>4.2 GB</td>
<td>592 MB</td>
<td>35 TB</td>
</tr>
<tr>
<td>16 TB</td>
<td>$2^{28}$</td>
<td>31 GB</td>
<td>5.15 GB</td>
<td>563 TB</td>
</tr>
<tr>
<td>256TB</td>
<td>$2^{32}$</td>
<td>101 GB</td>
<td>20.7 GB</td>
<td>2048 TB</td>
</tr>
<tr>
<td>1024TB</td>
<td>$2^{34}$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1 compares the ORAM constructions in terms of communication cost, client-side storage cost and server-side storage overhead. The client-side storage cost includes all the permanent or temporary storage space allocated at the client side, and the server-side storage overhead, which is computed as all the storage space allocated at the server side minus the storage space necessary to store the $N$ real data blocks. We have the following observations from the table.
• Octopus ORAM’s server storage overhead is only 12-16% of that incurred by Partition ORAM; note that, the storage overhead of Partition ORAM is more than twice of that necessary for storing the real blocks. This improvement in storage efficiency is important in practice, when the storage size is large. In the client’s perspective, as we discussed in Chapter 1, the monetary cost of renting large space is comparable to or even higher than the communication cost, and hence it is desired to reduce the server-side storage. In the server’s perspective, lower server storage overhead means lower monetary cost to maintain/upgrade storage devices and less labors to manage the space.

• Octopus ORAM incurs a client-server communication cost that is about 1.4-1.5 times of that by Partition ORAM. This demonstrates the tradeoff between the communication and the server-side storage costs.

5.5.2 Comparison with Path ORAM

To evaluate the performance of Octopus ORAM in a practical application scenario, we rented two AWS EC2 instances to run server and client software. The communication bandwidth between the two instance is around 700 Mbps (as measured using LANBench [48]), with a round trip delay of 50 ms added intentionally (as done in [61]), in order to simulate a practical scenario that the client has a high-speed Internet connection with the server. The instances are both of type AWS m4.xlarge, each has 4 vCPUs, 2.4 GHz, Intel Xeon E5-2686v4, and 16 GB memory.

For comparison purpose, we implemented Path ORAM (with index table stored at the client) and run the system on the same platform. The two constructions are compared in terms of the following metrics:

• *Communication Cost per Query*, which is measured as the average amount of data uploaded to or downloaded from the server, per query, in the unit of data block.

• *Query Delay*, which is measured as the average time elapse from when the client sends out a query request to when the client has received and decrypted the query target block. This
measures the query delay experienced by the client at the ideal scenario (i.e., a query request does not wait locally).

- **Processing Time per Query**, which is measured as the average time elapse from when the client sends out a query request to when the query and associated eviction operations have been completed and the client is able to send the next query. Note that, for simplicity, we do not process multiple queries concurrently.

We also compare their storage cost, and study the tradeoff in the above performance metrics. In the comparison, the parameters of Octopus ORAM are set as follows: \( \lambda = 40 \), \( s = 1024 \), \( \alpha = 0.34 \), \( \beta = 0.13 \) and \( \gamma = 0.25 \). For Path ORAM, the capacity of each node is set to 5 blocks. We choose \( N \) to range from \( 2^{20} \) to \( 2^{26} \), and block size \( B \) to range from 128 KB to 1 MB.

**Communication Cost per Query.** Figure 5.4 compares the communication cost per query between Octopus ORAM and Path ORAM. As we can see, the communication cost incurred by Octopus ORAM is 19-21\% of that by Path ORAM, as \( N \) and \( B \) vary.

**Query Delay.** Figure 5.5 compares the average query delay between Octopus ORAM and Path ORAM. As we can see, the average query delay incurred by Octopus ORAM is 8-33 \% of that by Path ORAM. This is mainly because: Octopus ORAM separates the query process from the eviction process, its query process only needs to download 2 or 3 blocks, and a query target block can be accessed immediately (if target block is in leaf node), or after generates and runs simple XOR operation (if target block is in non-leaf node) after the query process finishes. Path ORAM, on the other hand, combines the query and eviction processes. A query target block can be accessed, in the average case, only after the combined query and eviction process has download and decrypt half of the blocks that need to be processed, and the number of such blocks is much larger than that in Octopus ORAM.

Also note that, the average query delay is only about 20-200 ms with the above settings.

**Processing Time per Query.** Figure 5.6 compares the average processing time per query between Octopus ORAM and Path ORAM. As we can see from the figure, the average processing time incurred by Octopus ORAM is 10-30\% of that by Path ORAM. This is because: (i) Octopus
Figure 5.4 Comparing Communication Cost per Query Between Octopus ORAM and Path ORAM.

ORAM has smaller communication cost per query; (ii) Octopus ORAM separates query and eviction processes, which can be run in parallel and thus also reduce the processing time.

Storage Cost and Overhead. Table 5.2 compares Path ORAM and Octopus ORAM in terms of client-side storage cost and server-side storage overhead. As we can see, the server-side storage overhead of Octopus ORAM is only $\frac{1}{30}$ of that of Path ORAM. Meanwhile, Octopus ORAM has higher client-side storage cost; but, the cost is only a small fraction of the ORAM capacity.
5.6 Summary

We have proposed and evaluated a new ORAM construction called Octopus ORAM. Compared to state-of-the-art ORAM constructions, Octopus ORAM significantly improves the storage efficiency at the server and achieves the comparable level of communication efficiency, at the cost of increased client-side storage consumption. As we target at the application setting of hybrid cloud systems, the increased client-side storage consumption should be affordable to the clients who have local facility such as cloud storage gateway.
**Figure 5.6** Comparing the Processing Time per Query between Octopus ORAM and Path ORAM.

**Table 5.2** Comparing Storage Efficiency between Octopus ORAM and Path ORAM

<table>
<thead>
<tr>
<th>Capacity</th>
<th># Blocks</th>
<th>Client Storage Cost</th>
<th>Server Storage Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Path ORAM</td>
<td>Octopus ORAM</td>
</tr>
<tr>
<td>64 GB</td>
<td>$2^{20}$</td>
<td>0.25 MB</td>
<td>64.5 MB</td>
</tr>
<tr>
<td>256 GB</td>
<td>$2^{22}$</td>
<td>1 MB</td>
<td>66 MB</td>
</tr>
<tr>
<td>1 TB</td>
<td>$2^{24}$</td>
<td>4 MB</td>
<td>72 MB</td>
</tr>
<tr>
<td>4 TB</td>
<td>$2^{26}$</td>
<td>16 MB</td>
<td>96 MB</td>
</tr>
<tr>
<td>16 TB</td>
<td>$2^{28}$</td>
<td>64 MB</td>
<td>192 MB</td>
</tr>
<tr>
<td>64 TB</td>
<td>$2^{30}$</td>
<td>256 MB</td>
<td>576 MB</td>
</tr>
</tbody>
</table>
CHAPTER 6. THREE-SERVER OCTOPUS ORAM

6.1 Research Goal and Contributions

In chapter 5 we proposed Octopus ORAM [34] which reduces client-server communication cost to \( O(\log N \cdot B) \), some of the state-of-the-art achieve \( O(B) \) communication cost. In this chapter we aim at reduce client-server communication cost to \( O(B) \) while at the same time keep the server storage overhead low. Also, we remove the assumption in chapter 5 and most of related work, which is the server is semi-honest. In this chapter, we assume the servers will not collude with each other, but they could be malicious.

Similar to \( S^3 \) ORAM[26], our proposed three-server Octopus ORAM [35] also recruits three non-colluding cloud servers to act as the oblivious storage. But we significantly reduce the server storage overhead by storing only around \( 0.3N \) extra dummy blocks compared to \( 11N \) dummy blocks required by \( S^3 \) ORAM. Moreover, we design and employ several lightweight accountability mechanisms for the servers, such that each server can detect the misbehavior of other servers that interact with it. Compared to the state-of-the-art, our proposed scheme can simultaneously attain the following features:

- provable protection of clients’ access pattern privacy;
- low server storage overhead, which is around \( 0.3N \) blocks for every \( N \) real data blocks exported;
- low data query delay, which is only slightly longer than a communication round trip time between the client and server;
• accountability with multiple servers, which removes the less-realistic semi-honest assumption in a multi-server oblivious storage system;

• lower communication costs than S^3ORAM, the most related state-of-the-art scheme.

The above features are achieved by our novel designs of storage arrangement, data query algorithm and eviction algorithm. Note that, our scheme has made full use of the available moderate level of client-side storage, but the required storage capacity is still only as small as around 0.1% of the cloud server’s storage capacity.

In this chapter, we use the system model and security definition in chapter 2. We present the detailed design in terms of storage organization, query algorithm, eviction algorithm and accountability enhancements in section 6.2, 6.3, 6.4 and 6.5. Section 6.6 presents the security analysis. Performance evaluation and comparisons are reported in Section 6.7. Finally, Section 6.8 concludes the chapter.

6.2 System Architecture and Initialization

As shown in Figure 6.1, our proposed system is composed of one client and three servers, denoted as \( S_0, S_1 \) and \( S_2 \). Only one server (i.e., \( S_0 \)) needs to permanently store the data blocks exported by the client. The other two servers only store some meta-data and temporarily buffer some data blocks, to facilitate data query and eviction processes as well as to maintain system accountability, which are detailed in Sections 6.3, 6.4 and 6.5. The client also stores meta-data and a small subset of data blocks. In the following, we elaborate the storage organization at server \( S_0 \) and the client.

6.2.1 Storage Organization at Server \( S_0 \)

Let positive integers \( m^1 \) and \( q \), and positives fractions \( \alpha \) and \( \beta \) be system parameters. Let

\[
\xi(m, q) = \max\left(\frac{(m - 1) \cdot q}{2}, 2q\right).
\]  

\(^1m \) is a power of 2.
Each leaf node stores up $Z_0$ blocks.

Server $S_0$

Layer 0: Each non-leaf node stores $Z_1$ blocks.

Layer 1:
- (0,0)  (0,1)  (0,2)  (0,3)  (0,4)  (0,5)  (0,6)  (0,7)
- (1,0)  (1,1)  (1,2)  (1,3)  (1,4)  (1,5)  (1,6)  (1,7)
- (2,0)  (2,1)  (2,2)  (2,3)  (2,4)  (2,5)  (2,6)  (2,7)
- (3,0)  (3,1)  (3,2)  (3,3)  (3,4)  (3,5)  (3,6)  (3,7)

Layer $L-1$: (L-1,0) (L-1,1) (L-1,2) (L-1,3) (L-1,4) (L-1,5) (L-1,6) (L-1,7)

The data blocks are stored into an $m$-ary storage tree, in which each non-leaf node can have up to $m$ child nodes. When constructing the storage tree, we make the tree to be balanced and the number of data blocks at each leaf node to vary between $(1 + \beta) \cdot \xi(m, q)$ and $2(1 + \beta) \cdot \xi(m, q)$, for certain security purposes explained later in Section 6.6.

Specifically, the tree is constructed as follows:

1. Let $L' = \lceil \log_m \frac{N}{\xi(m, q)} \rceil$ and $Z' = \frac{N}{m^{Z'}}$. Obviously, $Z' \geq \xi(m, q)$.

2. If $Z' \leq 2\xi(m, q)$, the storage tree is organized as a complete $m$-ary tree with height $L = L' + 1$ where the capacity of each leaf node is $Z_0 = \lceil (1 + \beta) \cdot Z' \rceil$ blocks.

3. Otherwise (i.e., $Z' > 2\xi(m, q)$), the storage tree is organized as a tree of height $L = L' + 2$, and the root has $\lceil \frac{Z'}{\xi(m, q)} \rceil$ child nodes while each child node is a root of a complete $m$-ary tree with $L' + 1$ layers and $Z_0 = \lceil (1 + \beta) \cdot \frac{Z'}{\xi(m, q)} \rceil$ blocks at each leaf node.

4. Each non-leaf node has a capacity of $Z_1 = \lceil (1 + \alpha) \cdot \xi(m, q) \rceil$ blocks.

Each node $N_{l,i}$ is identified by a unique tuple $(l, i)$, where $l \in \{0, \ldots, L - 1\}$ is the ID of the layer that the node resides (note: the root node is at layer 0 while the leaf nodes are at layer $L - 1$), and
$i \geq 0$ is the ID of the node on layer $l$ that equals to the offset of the node on layer $l$ (from 0 at the leftmost towards right). Note that Figure 6.1 shows a storage tree when $m = 8$.

### 6.2.2 Storage Organization at the Client

The client maintains an index table for all of the $N$ real data blocks and an index block for each node on the storage tree. The index table has $N$ entries and each entry $i \in \{0, \ldots, N - 1\}$ has the following fields:

- **path ID** of block $i$, i.e., the ID of the leaf node on the path that block $i$ is assigned to;
- **secret key** $k_i$ which, as detailed in Section 6.2.3, randomly selected by the client to encrypt the block based on XOR operation;
- three **message authentication codes (MACs)** of the block, of which the computation and usage are explained in detail in Section 6.5.

Note that, following most of the tree-based ORAM constructions [49, 53, 16, 54, 51], our proposed scheme also enforces the policy that, a block is assigned to a path and the block must be stored on the path.

For each node on the tree, the index block has one entry $(id, ah)$ for each block it stores, where $id$ is the ID of the block, no matter whether the block is real or dummy, and $ah \in \{0, 1, 2\}$ indicates the access history of the block since the system initialization or the most recent data eviction process involving the node, whichever is more recent: (i) $ah = 0$ if the block has not been accessed; (ii) $ah = 1$ if the block has been accessed as a query target; and (iii) $ah = 2$ if the block has been accessed but never as a query target.

In addition, the client maintains a local buffer that stores the most recently accessed data blocks. The capacity of the buffer is at least $q$ blocks.
6.2.3 System Initialization

The client picks a pseudo random number generator $PRG_0(k)$, which takes a secret seed $k$ of $\lambda$ bits and outputs a pseudo-random sequence of $3\lambda$ bits. The client also picks and shares with the servers another pseudo random number generator function, denoted as $PRG_1(k)$, which takes a secret seed $k$ and outputs a pseudo-random sequence of bytes with the same length as a data block.

Before each real block (denoted as $\vec{D}_i$ which is a sequence of bits) of ID $i$ is exported to server $S_0$, the client encrypts the block as follows.

1. It randomly picks a secret seed $k_i$, and computes $PRG_0(k_i)$ whose output is denoted as $k_{i,0}|k_{i,1}|k_{i,2}$ where each $k_{i,j}$ has $\lambda$ bits and $|$ represents concatenation.

2. It computes $PRG_1(k_{i,0})$, $PRG_1(k_{i,1})$ and $PRG_1(k_{i,2})$ to generate three pseudo-random sequences of bytes, denoted as $\vec{R}_{i,0}$, $\vec{R}_{i,1}$ and $\vec{R}_{i,2}$, each of the same length as a data block.

3. It performs bit-wise XOR operations on each group of four bits with the same offset of the four bit-sequences $\vec{D}_i$, $\vec{R}_{i,0}$, $\vec{R}_{i,1}$ and $\vec{R}_{i,2}$, to encrypt $\vec{D}_i$ to

$$
\vec{D}'_i = \vec{D}_i \oplus \vec{R}_{i,0} \oplus \vec{R}_{i,1} \oplus \vec{R}_{i,2}.
$$

6.3 Data Query Algorithm

Assume the client wishes to query data block $\vec{D}_t$, where $t$ denotes the block ID, and the block is not in its local buffer. It looks up its index table to find path $p_t$ that contains $\vec{D}_t$, and looks up the index blocks of the path to locate the node containing $\vec{D}_t$. Then, it launches a query process in two phases: selecting some data blocks to access from $S_0$, based on the index table and index blocks that it stores, in order to hide the query target; interacting with the servers to retrieve query target $\vec{D}_t$. 
6.3.1 Phase I: Selecting Data Blocks to Access

For each node $N'_i$ on path $p_t$, where $i \in \{0, \cdots, L - 1\}$ represents the layer ID of the node, let $\Delta_{i,0}$, $\Delta_{i,1}$ and $\Delta_{i,2}$ denote the block sets with $ah$ being 0, 1 and 2, and $\delta_{i,0}$, $\delta_{i,1}$ and $\delta_{i,2}$ denote the sizes of these sets, respectively. The client selects data blocks from each node $N'_i$ to download, according to the rules presented in Algorithm 1, with the dual goals of hiding data access pattern and communication efficiency.

Algorithm 1 Rules for Selecting Blocks from $N'_i$ to Access (Output: $\Delta$ - a set of blocks selected to access)

1: $\Delta \leftarrow \emptyset$
2: if $N'_i$ contains query target $\vec{D}_t$ then
3:   add $\vec{D}_t$ to $\Delta$
4:   $\forall \vec{D} \in \Delta_{i,1}$, add $\vec{D}$ to $\Delta$ with probability $\frac{1}{\delta_{i,0}}$
5:   if $\vec{D}_t$ belongs to $\Delta_{i,0}$ then
6:     $\forall \vec{D} \in \Delta_{i,2}$, add $\vec{D}$ to $\Delta$ with probability $\frac{\delta_{i,2}}{\delta_{i,0}}$
7:     else //i.e., $\vec{D}_t$ belongs to $\Delta_{i,2}$
8:       randomly picks one $\vec{D}$ from $\Delta_{i,0}$; adds it to $\Delta$
9:     end if
10: else
11:    randomly picks one $\vec{D}$ from $\Delta_{i,0}$; adds it to $\Delta$
12:    $\forall \vec{D} \in \Delta_{i,1} \cup \Delta_{i,2}$, add $\vec{D}$ to $\Delta$ with probability $\frac{1}{\delta_{i,0}}$
13: end if

First, the algorithm hides data access pattern by making each block in $N'_i$ to be accessed with the same probability independently of where the query target resides, as stated in the following Lemma 14 with proof.

**Lemma 14.** During a query process with query path $p_t$, each block $\vec{D}$ in node $N'_i$ on $p_t$ is selected to access with the same probability of $\frac{1}{\delta_{i,0}}$, which is obviously independent of the location of the query target.

**Proof.** When the query target does not belong to $N'_i$, every block $\vec{D}$ is accessed with probability $\frac{1}{\delta_{i,0}}$ based on lines 10-11 of Algorithm 1. Otherwise, each block $\vec{D}$ must be in $\Delta_{i,0}$, $\Delta_{i,1}$ or $\Delta_{i,2}$. So we consider the three cases respectively.
Case I: \( \vec{D} \in \Delta_{i,0} \). Further there are two subcases: \( \Delta_{i,0} \) contains the query target or not.

- Subcase I-a: \( \Delta_{i,0} \) contains query target \( \vec{D}_t \). Only \( \vec{D}_t \) is accessed from \( \Delta_{i,0} \). Further due to the random distribution of blocks in \( \Delta_{i,0} \), every \( \vec{D} \) has the same probability of \( \frac{1}{\delta_{i,0}} \) to be accessed as query target.

- Subcase I-b: \( \Delta_{i,0} \) contains query target \( \vec{D}_t \). Based on line 8 of the algorithm, every \( \vec{D} \) has the probability of \( \frac{1}{\delta_{i,0}} \) to be accessed.

Case II: \( \vec{D} \in \Delta_{i,1} \). Every \( \vec{D} \) has the probability of \( \frac{1}{\delta_{i,0}} \) to be accessed, based on line 4 of the algorithm.

Case III: \( \vec{D} \in \Delta_{i,2} \). Further there are two subcases:

- Subcase III-a: \( \Delta_{i,0} \) contains query target \( \vec{D}_t \), which occurs with probability \( \frac{\delta_{i,0}}{\delta_{i,0} + \delta_{i,2}} \) in case III. In this subcase, every \( \vec{D} \) is accessed with probability \( \frac{\delta_{i,2}}{\delta_{i,0} + \delta_{i,2}} \).

- Subcase III-b: \( \Delta_{i,0} \) contains query target \( \vec{D}_t \), which occurs with probability \( \frac{\delta_{i,2}}{\delta_{i,0} + \delta_{i,2}} \) in case III. In this subcase, \( \vec{D} \) is accessed as the query target with probability \( \frac{1}{\delta_{i,2}} \).

To summarize, \( \vec{D} \) is accessed with the following probability in Case III:

\[
\frac{\delta_{i,0}}{\delta_{i,0} + \delta_{i,2}} \cdot \frac{\delta_{i,2}}{\delta_{i,0}^2 + \delta_{i,0}} + \frac{\delta_{i,2}}{\delta_{i,0} + \delta_{i,2}} \cdot \frac{1}{\delta_{i,2}} = \frac{1}{\delta_{i,0}}.
\]

Hence, the Lemma is proved.

Second, in terms of communication efficiency, the query algorithm requires only \( 1 + \frac{\delta_{i,1} + \delta_{i,2}}{\delta_{i,0}} \) blocks accessed from each node \( N'_i \). Further, as we study later in Section 6.6, \( \frac{\delta_{i,1} + \delta_{i,2}}{\delta_{i,0}} < 1 \) with an overwhelming probability of \( 1 - 2^{-\lambda} \). That is, no more than 2 blocks are accessed from each node \( N'_i \) on the query path with a probability at least \( 1 - 2^{-\lambda} \).
6.3.2 Phase II: Retrieving Query Target

The client sends a request to $S_0$, which contains:

- list $\vec{B} = \langle b_1, \cdots, b_x \rangle$ of $x$ block indices where, in each $b_i = (n_i, o_i)$, $n_i$ is the ID of a block on query path and $o_i$ is the offset of a block selected to access in Phase I;

- random permutation vector $\vec{V} = \langle v_1, \cdots, v_x \rangle$ of integers $\{1, \cdots, x\}$, which directs $S_0$ to put every block $b_i$ to offset $v_i$ after the permutation.

It also sends a request to $S_1$, which only contains one number in $\{1, \cdots, x\}$.

In response to the client’s request, $S_0$ makes a copy of the blocks indicated by $\vec{B}$, permutes the blocks as directed by $\vec{V}$, and then forwards the resulting block sequence to $S_1$.

Upon receiving the sequence, $S_1$ retains only the query target block, whose offset on the sequence is the index contained in the client’s request, and immediately returns the block to the client.

Having received the query target, the client updates its local meta-data to make the copy of the query target left on the storage tree as a dummy block. Then, it can start reading or writing to the query target locally.

6.4 Data Eviction Algorithm

After every $q$ queries, the client has retained at its buffer $q$ blocks that are the targets of the most recent $q$ queries. We call these blocks the current evicting blocks. The client randomly re-assigns a path for each evicting block, sends all these blocks in an ordered list to server $S_1$, and then launches a data eviction process to evict them into the storage tree at server $S_0$. Note that, as in existing ORAM constructions such as [53], the eviction process can be carried out concurrently with data query processes through some de-amortization mechanism. Due to page limit, we skip the de-amortization detail and focus on the main idea.

Every eviction process involves only one root-to-leaf path, which we call eviction path, on the storage tree at server $S_0$. The eviction path is selected in the reverse-lexicographic order, as illustrated by Figure 6.2.
An eviction process runs iteratively, one iteration for each node on the eviction path from the root to the leaf. We introduce variable $N'_e$ to denote the node currently involved in the eviction. Hence, $N'_e$ is initialized to $N_{0,0}$ (i.e., the root node). Also, when an eviction iteration begins, $S_0$ has an ordered list (denoted as $\vec{L}_0$) containing $Z_0$ or $Z_1$ blocks stored at node $N'_e$, depending on whether $N'_e$ is leaf or not; $S_1$ has an ordered list (denoted as $\vec{L}_1$) of $q$ blocks; $S_2$ has no data blocks. Then, the iteration, which involves the client and all the servers, runs as follows.

1. For each block $\vec{D}_i \in \vec{L}_0 \cup \vec{L}_1$, where $i$ represents the ID of the block, the client randomly picks a new key $k'_i$ and then generates a new set of keys $k'_{i,0}$, $k'_{i,1}$ and $k'_{i,2}$ where $PRG_0(k'_i) = k'_{i,0}|k'_{i,1}|k'_{i,2}$. Also, from the current version of key $k_i$ recorded in the index table, the client derives the current set of keys $k_{i,0}$, $k_{i,1}$ and $k_{i,2}$ where $PRG_0(k_i) = k_{i,0}|k_{i,1}|k_{i,2}$.

2. The client randomly constructs a permutation vector $\pi_0$ for $|\vec{L}_0|$ elements (i.e., a random permutation of numbers $0, \cdots, |\vec{L}_0| - 1$) where $|\vec{L}_0|$ denotes the length of $\vec{L}_0$, and sends the vector to $S_0$.

3. Upon receiving $\pi_0$, $S_0$ permutes $\vec{L}_0$ to $\vec{L}'_0 = \pi_0(\vec{L}_0)$, and sends $\vec{L}'_0$ to $S_1$. 

![Figure 6.2 Reverse-lexicographic Order: Every eviction process involves one root-to-leaf eviction path selected in the reverse-lexicographic order.](image)
4. Letting \( \mathcal{L}_0 \mid \mathcal{L}_1 = (\vec{D}_{i_0}, \ldots, \vec{D}_{i_{x-1}}) \) where \( x = |\mathcal{L}_0| + |\mathcal{L}_1| \), the client randomly constructs a permutation vector \( \pi_1 \) for \( x \) elements and the following ordered list (denoted as \( \vec{R}_1 \)):
\[
\vec{R}_1 = \langle (k_{i_0,0}, k'_{i_0,1}), \ldots, (k_{i_{x-1},0}, k'_{i_{x-1},1}) \rangle. \tag{6.3}
\]
Then, the client sends \( \pi_1 \) and \( \vec{R}_1 \) to \( S_1 \).

5. Upon receiving \( \mathcal{L}_0 \) from \( S_0 \) as well as \( \pi_1 \) and \( \vec{R}_1 \) from the client, \( S_1 \) first constructs \( \mathcal{L}_1 = \mathcal{L}_0 \mid \mathcal{L}_1 \), which we also denote as \( (\vec{D}_{i_0}, \ldots, \vec{D}_{i_{x-1}}) \). Next, it re-encrypts each block \( \vec{D}_{ij} \) (where \( j = 0, \ldots, x - 1 \)), based on key pair \( (k_{ij,0}, k'_{ij,1}) \) in \( \vec{R}_1 \), through the following steps:

- It computes pseudo-random blocks \( \vec{R}_{ij,0} = PRG_1(k_{ij,0}) \) and \( \vec{R}_{ij,1} = PRG_1(k'_{ij,1}) \).
- It updates \( \vec{D}_{ij} \) to \( \vec{D}'_{ij} = \vec{D}_{ij} \oplus \vec{R}_{ij,0} \oplus \vec{R}_{ij,1} \), where \( \oplus \) is the bit-wise XOR between two blocks (i.e., bit sequences).

Then, list \( (\vec{D}'_{i_0}, \ldots, \vec{D}'_{i_{x-1}}) \) is permuted according to \( \pi_1 \), and the resulting list (denoted as \( \mathcal{L}_2 \)) is sent to server \( S_2 \).

6. Letting \( \langle i'_0, \ldots, i'_{x-1} \rangle \) be the ordered list of IDs of the blocks in \( \mathcal{L}_2 \), the client sends to \( S_2 \) the following list of key pairs
\[
\vec{R}_2 = \langle (k_{i'_0,1}, k'_{i'_0,2}), \ldots, (k_{i'_{x-1},1}, k'_{i'_{x-1},2}) \rangle. \tag{6.4}
\]
The client also constructs a permutation \( \pi_2 \) for \( x \) elements, and sends \( \pi_2 \) to \( S_2 \).

7. Upon receiving \( \pi_2 \) and \( \vec{R}_2 \) from the client, as well as \( \mathcal{L}_2 = (\vec{D}'_{i_0}, \ldots, \vec{D}'_{i_{x-1}}) \) from \( S_1 \), server \( S_2 \) first re-encrypts each block in \( \mathcal{L}_2 \) based on the key pairs in \( \vec{R}_2 \), and then permutes the re-encrypted list according to \( \pi_2 \), as server \( S_1 \) does. The resulting list (denoted as \( \mathcal{L}_2' \)) is sent to server \( S_0 \).

8. Letting \( \langle i''_0, \ldots, i''_{x-1} \rangle \) be the ordered list of IDs of the blocks in \( \mathcal{L}_2' \), the client sends to \( S_0 \) the following list of key pairs
\[
\vec{R}_0 = \langle (k_{i''_0,2}, k'_{i''_0,0}), \ldots, (k_{i''_{x-1},2}, k'_{i''_{x-1},0}) \rangle. \tag{6.5}
\]
Besides, the client further constructs and sends to $S_0$ an ordered list $I$ with $q$ elements, which is a sub-stream of $\langle 0, \cdots, x - 1 \rangle$. The construction should meet the following requirements:

- **Case I:** $N_e'$ is a non-leaf node. For each $j \in \{0, \cdots, x - 1\}$, if $\vec{D}_{ij}'$ is a real block and it cannot be evicted to the next evicting node (i.e., the path that $\vec{D}_{ij}'$ is assigned to does not pass the next evicting node), then $j$ must not be in $I$.

- **Case II:** $N_e'$ is a leaf node. $I$ should contain only the IDs for dummy blocks.

9. Upon receiving $L_2'$ from $S_2$ as well as $\vec{R}_0$ and $I$ from the client, server $S_0$ re-encrypts each block in $L_2'$ based on the key pairs in $\vec{R}_0$, as $S_1$ and $S_2$ do. Then, from the resulting list of blocks, $S_0$ removes the list of blocks with offsets specified in $I$; these removed blocks are sent to server $S_1$ and become the new version of $\vec{L}_1$ if $N_e'$ is a non-leaf node, or discarded if $N_e'$ is a leaf node.

Fig 6.3 illustrates how the client and the servers cooperate during the eviction process, in a high level.

![Figure 6.3](image-url)  
**Figure 6.3** A High-level Illustration of Eviction Process.

### 6.5 Accountability Enhancements

In this section, we propose several accountability enhancements to the above data query and eviction algorithms, so that if a server maliciously changes a block, another server is able to detect.
The enhancements affect the storage organization, system initialization, data query algorithm and data eviction algorithm, in the following ways.

6.5.1 Enhancements to Storage and System Initialization

When the system is initialized, for each server \( S_i \) where \( i \in \{0, 1, 2\} \), the client randomly constructs \( \lambda \) blocks each with \( z \) bits, denoted as \( \vec{A}_{i,j} = \langle a_{i,j,0}, \cdots, a_{i,j,z-1} \rangle \) for \( j \in \{0, \cdots, \lambda - 1\} \), where each \( a_{i,j,y} \in \{0, 1\} \) for \( y \in \{0, \cdots, z-1\} \). Then, the client sends each \( \vec{A}_{i,j} \) to server \( S_i \), and the block should be kept secret only between server \( S_i \) and the client.

For each exported data block \( \vec{D} \), letting \( \langle d_0, \cdots, d_{z-1} \rangle \) denote its plain text, the client computes 3 message authentication codes (MACs) as follows.

- First, the client computes the following 3\( \lambda \) message authentication bits (MABs) for \( \vec{D} \):
  \[
  MAB_{i,j}(\vec{D}) = \oplus_{y \in \{0, \cdots, z-1\}} d_y \cdot a_{i,j,y},
  \]
  where \( i \in \{0, 1, 2\} \) and \( j \in \{0, \cdots, \lambda - 1\} \).

- Based on the MABs, the client computes the following 3 MACs for \( \vec{D} \):
  \[
  MAC_i(\vec{D}) = MAB_{i,0}| \cdots |MAB_{i,\lambda-1},
  \]
  where \( i \in \{0, 1, 2\} \) and \(|\) denotes concatenation.

Finally, the client stores \( MAC_0(\vec{D}) \), \( MAC_1(\vec{D}) \) and \( MAC_2(\vec{D}) \) to the entry of \( \vec{D} \) in the index table.

6.5.2 Enhancement to Data Query Algorithm

In the data query algorithm, we introduce an accountability enhancement to allow \( S_1 \) to check if \( S_0 \) has sent to it a correct sequence \( \vec{L} \). The detail is as follows.

During the query process, the client completely knows which blocks should be in \( \vec{L} \). Let \( \vec{I} = \langle i_0, i_1, \cdots \rangle \) denote the IDs of the blocks in the sequence. For each block with ID \( i_x \in \vec{I} \), the client computes an MAC of the block that can be checked by the \( S_1 \) as follows:
• From the index table, it retrieves $MAC_1(\vec{D}_{ix})$ (i.e., the MAC computed based on the block’s plain text and the secret block $\vec{A}_1$ known by $S_1$) and the current version of encryption key $k_{ix}$ for the block.

• It computes the two pseudo-random blocks that have been used to encrypt the block, i.e., $\vec{R}_{ix,0} = PRG_1(k_{ix,0})$, $\vec{R}_{ix,1} = PRG_1(k_{ix,1})$, and $\vec{R}_{ix,2} = PRG_1(k_{ix,2})$. Note that, the $x$-th block received by $S_1$ should be equal to $\vec{D}_{ix} \oplus \vec{R}_{ix,0} \oplus \vec{R}_{ix,1} \oplus \vec{R}_{ix,2}$ if it is correct.

• It computes $MAC_1'(\vec{D}_{ix})$ as
  \[
  MAC_1(\vec{D}_{ix}) \oplus MAC_1(\vec{R}_{ix,0}) \oplus MAC_1(\vec{R}_{ix,0}) \oplus MAC_1(\vec{R}_{ix,0}),
  \]
which should be equal to
  \[
  MAC_1(\vec{D}_{ix} \oplus \vec{R}_{ix,0} \oplus \vec{R}_{ix,1} \oplus \vec{R}_{ix,2})
  \]
according to the definition of $MAC_1(\cdot)$.

Then, $MAC_1'(\vec{D}_{ix})$ is sent to $S_1$ for checking.

Upon receiving $\vec{L}$ from $S_0$ and the ordered list of MACs from the client, $S_1$ applies $MAC_1(\cdot)$ to compute the MAC for each block in $\vec{L}$, and compares the resulting MAC with the MAC sent from the client. If a mismatch is found, $S_0$ will be identified to have modified some block.

### 6.5.3 Enhancements to Data Eviction Algorithm

The accountability enhancements to data eviction algorithm are similar to that applied for data query algorithm. That is, whenever a server $S_i$ ($i \in \{0,1,2\}$) receives a list of blocks from another server, $S_i$ needs to: (1) receive from the client an $MAC_i$ for each block on the list; (2) re-computes the $MAC_i$ for each block on the list; (3) find out if the above values match.

### 6.6 Security Analysis

According to the definition of security in Chapter 2, we first study the security of the proposed system in terms of obliviousness and failure probability. Then, we study the accountability of the system.
6.6.1 Obliviousness Analysis

In this subsection, we show the obliviousness of the query and eviction processes; i.e., these processes are random and independent of the client’s data access pattern. First of all, it is obvious that the interactions between servers and the client follow the same pattern, independent of the client’s access pattern. Hence, we focus to analyze the obliviousness of the processes inside server $S_0$.

6.6.1.1 Obliviousness in Query Path Selection

When the system is initialized, the path assigned to each block is selected randomly and independently of each other. After a block has been queried, its path is re-assigned randomly and independently of the client’s data access pattern. Due to the randomness in path assignment, the query path for each query process, which is determined by the path assigned to the query target block, is random and independent of the client’s access pattern.

6.6.1.2 Obliviousness in Block Access from Query Path

According to the data query algorithm, the following block access pattern has been enforced: from each node on the query path, the client must select one block that has not been accessed; meanwhile, every block that has already been accessed has the same probability to be accessed again according to Lemma 14.

6.6.1.3 Obliviousness in eviction process

The eviction process is random and independent of the client’s data access pattern, due to the following reasons: (i) Each eviction process involves only one root-to-leaf path (called eviction path), and the order in which the paths are selected for as eviction paths is fixed and independent of data access pattern. (ii) During each eviction process, the processing for each node on the selected eviction path follows a fixed pattern which is independent of data access pattern. Specifically, all
the data blocks on the node are re-encrypted and re-permuted by all the servers; then, the same number of blocks are stored back to the node.

6.6.2 Failure Analysis

In this subsection, we study the probabilities for a query process and an eviction process to fail.

6.6.2.1 Failure Probability for A Query Process

According to Algorithm 1, a query process fails only when the probability $\delta_{i,2}$ used in selecting a block (in Line 6) becomes greater than 1. Also, as discussed in Section 6.3, we aim to make $\frac{\delta_{i,1} + \delta_{i,2}}{\delta_{i,0}} \leq 1$ (which obviously makes $\frac{\delta_{i,1}}{\delta_{i,0}} \leq 1$) such that on average no more than 2 blocks are accessed from each layer of the storage tree during each query process. Hence, we here study the probability for $\frac{\delta_{i,1} + \delta_{i,2}}{\delta_{i,0}} > 1$, which is no less than the probability for a query process to fail. Our result is stated in the following Lemma 15.

**Lemma 15.** As long as $q \geq 25\lambda$ and $\alpha \geq 0.25$ and $\beta \geq 0.25$ when $m = 2, 4$, $Pr[\frac{\delta_{i,1} + \delta_{i,2}}{\delta_{i,0}} \leq 1] > 1 - 2^{-\lambda}$, i.e., any query process fails with a probability less than $2^{-\lambda}$.

**Proof.** Consider an arbitrary node $N_i$ on a $m$-ary storage tree, and let random variable $X$ denote the times that $N_i$ has been selected to be on a query path during two consecutive evictions involving the node. Obviously, $X \geq \delta_{i,1} + \delta_{i,2}$.

When $m = 2, 4$, according to the storage organization, the size of each node $N_i'$ on the storage tree is at least $2q \cdot \min(1 + \alpha, 1 + \beta) \geq 2.5q$; i.e., $\delta_{i,0} + \delta_{i,1} + \delta_{i,2} \geq 2.5q$. Since an eviction process is launched every $q$ queries, the mean of $X$ is $q$. Further according to the multiplicative Chernoff bound,

$$Pr[X \leq 1.25q] > 1 - \left(\frac{e^{0.25}}{1.25^{1.25}}\right)^q > 1 - 2^{-2\lambda}. \quad (6.10)$$

Hence,

$$Pr[\delta_{i,1} + \delta_{i,2} < \delta_{i,0}] \geq Pr[\delta_{i,1} + \delta_{i,2} \leq 1.25q] > 1 - 2^{-\lambda}. \quad (6.11)$$
When \( m \geq 8 \), the size of each node on the storage tree is at least \( \frac{m-1}{2} \cdot q \geq 3.5q \); i.e., \( \delta_{i,0} + \delta_{i,1} + \delta_{i,2} \geq 2.5q \). Due to Equation (6.10),

\[
Pr[\delta_{i,1} + \delta_{i,2} < \delta_{i,0}] > Pr[\delta_{i,1} + \delta_{i,2} \leq 1.25q] > 1 - 2^{-\lambda}.
\]  

\[ (6.12) \]

### 6.6.2.2 Failure Probability for An Eviction Process

An eviction process fails iff the following scenarios occur in Step 8) of the eviction algorithm.

(i) **Failure Scenario I**: The current evicting node (i.e., \( \mathcal{N}_e' \)) is a non-leaf node, and so \( q \) out of the \( x \) blocks in \( \mathcal{L}_2' \) need to be picked to send from \( S_0 \) to \( S_1 \). According to Case-I of the requirement, the \( q \) blocks should not contain any real block that cannot be evicted to the next evicting node, but failure will occur if there are more than \( x - q \) real blocks that cannot be evicted to the next evicting node. (ii) **Failure Scenario II**: The current evicting node \( \mathcal{N}_e' \) is a leaf node, and so \( q \) dummy blocks out of the \( x \) blocks in \( \mathcal{L}_2' \) need to be discarded. Failure will occur if there are less than \( q \) dummy blocks (i.e., more than \( x - q \) real blocks) in \( \mathcal{L}_2' \).

The results of our analysis are summarized as the following Lemmas 16 and 17. The proofs, which have to be skipped due to space limit, can be developed based on the analysis of the eviction process and the application of the multiplicative Chernoff bound.

**Lemma 16.** With \( q \geq 25\lambda \) and the following combinations of system parameters, i.e., \((m = 2, \alpha \geq 0.25), (m = 4, \alpha \geq 0.25), (m = 8, \alpha \geq 0.34)\) and \((m = 16, \alpha \geq 0.34)\), the Failure Scenario I occurs with a probability of \( O(2^{-\lambda}) \).

**Lemma 17.** With \( q \geq 25\lambda \) and the following combinations of system parameters, i.e., \((m = 2, \beta \geq 0.25), (m = 4, \beta \geq 0.25), (m = 8, \beta \geq 0.13)\) and \((m = 16, \beta \geq 0.09)\), the Failure Scenario II occurs with a probability of \( O(2^{-\lambda}) \).

Based on the above analysis on obliviousness and failure probabilities, we get the following theorem.
Theorem 7. The proposed system is secure under the security definition in Chapter 2 with \( q \geq 25 \lambda \) and the following combinations of system parameters: \( (m = 2, \alpha \geq 0.25, \beta \geq 0.25) \), \( (m = 4, \alpha \geq 0.25, \beta \geq 0.25) \), \( (m = 8, \alpha \geq 0.34, \beta \geq 0.13) \) and \( (m = 16, \alpha \geq 0.34, \beta \geq 0.09) \).

6.6.3 Accountability Analysis

The accountability of the proposed system relies on the security of the proposed MAC mechanism, which is formally stated and proved in the following.

Lemma 18. For \( \forall j \in \{0, 1, 2\} \) and distinct blocks \( \vec{D} \) and \( \vec{D}' \),

\[
Pr[MAB_{j,u}(\vec{D}) = MAB_{j,u}(\vec{D}')] = \frac{1}{2}, \quad u = 0, \ldots, \lambda - 1.
\] (6.13)

Proof. (By induction). Let \( \vec{D} \) and \( \vec{D}' \) differ by \( n \) bits on indices \( v_0, \ldots, v_{n-1} \); let \( j \in \{0, 1, 2\} \), \( u \in \{0, \ldots, \lambda - 1\} \) and \( v \in \{0, \ldots, z - 1\} \); let \( \vec{A}_{j,u}[v] \) denote the \( v \)-th bit on \( \vec{A}_{j,u} \) (recall that \( \vec{A}_{j,u} \) is a secret block shared only between the client and server \( S_j \)).

When \( n = 1 \), \( MAB_{j,u}(\vec{D}) = MAB_{j,u}(\vec{D}') \) iff \( \vec{A}_{j,u}[v_0] = 0 \). Because \( \vec{A}_{j,u} \) is randomly picked from \( \{0, 1\}^z \), \( Pr[\vec{A}_{j,u}[v_0] = 0] = \frac{1}{2} \). Hence, Equation (6.13) holds.

Assuming Equation (6.13) holds when \( \vec{D} \) and \( \vec{D}' \) differ by \( n \leq t \), we next prove the equation holds when \( n = t + 1 \). Without loss of generality, assume \( \vec{D}[v_t] = 0 \) and \( \vec{D}'[v_t] = 1 \). Let \( \vec{I}_0 \) be the \( z \)-bit block with 0 on every bit, \( \vec{I}_1 \) be the \( z \)-bit block with 1 on bit \( v_t \) but 0 on all other bits, and \( \vec{D}'' = \vec{D}' \oplus \vec{I}_1 \) (i.e., \( \vec{D}' = \vec{D}'' \oplus \vec{I}_1 \)). Hence, \( \vec{D} \) and \( \vec{D}'' \) differ in \( t \) bits \( v_0, \ldots, v_{t-1} \).

According to the induction assumption, \( Pr[MAB_{j,u}(\vec{D}) = MAB_{j,u}(\vec{D}'')] = \frac{1}{2} \). Also note that,
Pr[MAB\_j,u(\vec{I}_0) = MAB\_j,u(\vec{I}_1)] = \frac{1}{2} and \vec{D} = \vec{D} \oplus \vec{I}_0, \quad \vec{D'} = \vec{D}'' \oplus \vec{I}_1. Therefore, we have

\[
\begin{align*}
Pr[MAB\_j,u(\vec{D}) &= MAB\_j,u(\vec{D'})] = \frac{1}{2}\quad (6.14) \\
&= Pr[MAB\_j,u(\vec{D} \oplus \vec{I}_0) = MAB\_j,u(\vec{D}'' \oplus \vec{I}_1)] \quad (6.15) \\
&= Pr[MAB\_j,u(\vec{D}) = MAB\_j,u(\vec{D}'')] \times \\
&Pr[MAB\_j,u(\vec{I}_0) = MAB\_j,u(\vec{I}_1)] + \\
&Pr[MAB\_j,u(\vec{D}) \neq MAB\_j,u(\vec{D}'')] \times \\
&Pr[MAB\_j,u(\vec{I}_0) \neq MAB\_j,u(\vec{I}_1)] \quad (6.16) \\
&= \frac{1}{2} \times \frac{1}{2} + \frac{1}{2} \times \frac{1}{2} = \frac{1}{2}. \quad (6.17)
\end{align*}
\]

\textbf{Theorem 8.} If server \(S_i\) sends data block \(\vec{D}'\) instead of \(\vec{D}\) to server \(S_j\), where \(i \neq j\) and \(\vec{D} \neq \vec{D}'\), then:

\[
Pr[MAC\_j(\vec{D}') = MAC\_j(\vec{D})] = 2^{-\lambda}; \quad (6.18)
\]
i.e., the misbehavior of \(S_i\) is detected with a probability of \(1 - 2^{-\lambda}\).

\textit{Proof.} According to the MAC definition in Section 6.5 and Lemma 18,

\[
\begin{align*}
Pr[MAC\_j(\vec{D}') &= MAC\_j(\vec{D})] = \frac{1}{2} \quad (6.19) \\
&= \prod_{u=0}^{\lambda-1} Pr[MAB\_j,u(\vec{D}') = MAB\_j,u(\vec{D})] = 2^{-\lambda}. \quad (6.20)
\end{align*}
\]

\textbf{6.7 Performance Evaluation and Comparisons}

We have implemented the proposed system, and conducted performance comparisons with \(S^3\text{ORAM}\) [26], which is the newest and most-efficient ORAM construction that employs multiple non-colluding servers.
6.7.1 System Settings

We rent four AWS EC2 instances to run our implemented servers and client. As the communication latency between these instances are smaller than those between client and server and between the servers owned by different cloud owners, we conducted experiments to measure the communication latencies between AWS EC2 and Microsoft Compute Engine instances and add the measured average round trip delay 29 ms to the communication between our servers; we also measured the communication latencies between these cloud servers and a rented client located at the center of North America Continent, and add the measured average round trip delay 177.5 ms to the communication between our servers and client.

We set security parameter \( \lambda = 40 \), which makes the failure probability of each query and eviction process to be lower than \( 2^{-40} \). According to Theorem 7, we set \( q = 1024 \) which is greater than \( 25\lambda \); with different \( m \), we adopt the following combinations of system parameter by default: \((m = 2, \alpha = \beta = 0.25), (m = 4, \alpha = \beta = 0.25), (m = 8, \alpha = 0.34, \beta = 0.13) \) and \((m = 16, \alpha = 0.34, \beta = 0.09)\).

In each evaluation, we vary \( N \) (i.e., the number of real data blocks to export) between \( 2^{20} \) to \( 2^{26} \) and vary \( B \) (i.e., the size of each data block in bytes) between 16K to 1M.

We measure the following metrics: (1) client-server communication cost, which is measured as the average number of blocks sent between the client and the servers to serve each data query request; (2) inter-server communication cost, which is measured as the average number of blocks sent among the servers per data query; (3) query delay, which is measured as the average time elapsed from a query request is sent from the client till the requested data block arrives at the client; (4) server storage overhead, which is measured as the amount of storage consumed at the server other than that for storing \( N \) exported data blocks; and (5) client storage cost, which is measured as the amount of storage consumed at the client.

To optimize the system parameter selection, we also measure the system costs with varying \( m \) and results are shown in Table 6.1. Note that, the table only shows the results when \( N = 2^{20} \), as the trend is similar with different \( N \). As we can see from the table, when \( m \) increases, the client-server communication cost does not change; the inter-server communication cost decreases
and then increases; the server storage overhead decreases. Hence, in the following experiments, we set $m = 8$ to make our system to have low communication and storage overheads.

### 6.7.2 Comparison with $S^3$ORAM

#### 6.7.2.1 Communication Costs and Query Delay

As shown in Fig 6.4, our ORAM system incurs smaller inter-server communication cost, which is about 60-80% of that of $S^3$ORAM. Both schemes require a constant number of blocks to be transferred between the client and server for each query. Specifically, in our system, the communication cost ranges from 1 to 1.3 data blocks per query, which includes 1 target block downloaded from $S_1$ and some control messages for query and eviction. $S^3$ORAM needs to download 3 data blocks as well as a small size of meta-data.

In terms of query delay, as shown in Fig 6.5, our ORAM has similar but a slightly higher query delay. This is due to the fact that, the request data block needs to travel through the path of $S_0 \rightarrow S_1 \rightarrow$ client; the detour between the servers incurs some extra delay, but it is very small compared to the delay between client and server.

#### 6.7.2.2 Storage Overheads

Both schemes require 3 non-colluding servers. For $S^3$ORAM, all servers have the same structure, different in that each server stores a different secret-shared version of blocks. Our system stores data blocks on one server, i.e., $S_0$, while the other two servers only need to allocate small storage to facilitate query and eviction. Fig 6.6 shows the server-side storage overheads. Specifically, the

<table>
<thead>
<tr>
<th>$m$</th>
<th>Client-Server Comm. Cost</th>
<th>Inter-Server Comm. Cost</th>
<th>Server Storage Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1.3B</td>
<td>96B</td>
<td>1.5 N</td>
</tr>
<tr>
<td>4</td>
<td>1.3B</td>
<td>58B</td>
<td>1.1 N</td>
</tr>
<tr>
<td>8</td>
<td>1.3B</td>
<td>67B</td>
<td>0.3N</td>
</tr>
<tr>
<td>16</td>
<td>1.3B</td>
<td>90B</td>
<td>0.17N</td>
</tr>
</tbody>
</table>
server-side storage overhead of $S^3$ORAM is $11N$ data blocks, while the overhead of our system is $(\beta + \frac{1+\alpha}{7})N + \frac{(1+\alpha)s}{2}$, which is no more than $0.3N$ blocks.

As the cost of the increased server storage efficiency, our system requires a larger client-side storage space, which is around 0.1% of the server-side storage cost.

Also note that, both schemes require some computation at the server side. $S^3$ORAM requires its servers to execute addition and multiplication of Shamir Secret Sharing operations, while our ORAM requires server to run random number generator to produce pseudo random sequences and then perform XOR operations to decrypt or re-encrypt data blocks. Our ORAM also requires the server to conduct authentication, which is also XOR operations. Our evaluations show that, the delay caused by the computations is nearly negligible compared to the communication delay.
6.7.2.3 Summary

Compared to S$^3$ORAM, our ORAM achieves a same level of efficiency in client-server communication, a higher level of efficiency in server-server communication, and a significantly higher level of server-side storage efficiency, at the price of increased client-side storage requirement, which however is affordable for a client who maintains an on-premise facility such as a cloud storage gateway.

6.8 Summary

In this chapter, we have proposed a new oblivious cloud storage system to address the limitations of existing research efforts. Extensive analysis and evaluation have shown that, our proposed system
Figure 6.6 Server Storage Overhead

can simultaneously attain the features of provable protection of data access pattern, low data query delay, low server storage overhead; low communication costs, and accountability.
CHAPTER 7. CONCLUSIONS AND FUTURE WORKS

In this dissertation, we have presented three novel Oblivious RAM solutions to improve the state-of-the-art Oblivious RAM performance. 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:

First, we propose SE-ORAM: A Storage-Efficient Oblivious RAM. SE-ORAM is an ORAM construction with zero storage overhead at the server, while communication efficiency is similar or higher than state-of-the-art. SE-ORAM stores exactly $N$ number of data blocks on server side, which has zero server storage overhead. Note that, the server stores a small number of dummy blocks, and the real block replaced by dummy block is stored in user side storage. The number of introduced dummy blocks is no more than $x \log N$ with probability $1 - \frac{1}{N^2}$, as long as $\lambda \geq 2$ and each node on the storage tree can store $4 \log N$ or more data blocks. SE-ORAM incurs communication cost of $O(\log^2 N \cdot B)$ where $B$ is block size in unit of bits, which is higher than some state-of-the-art ORAM constructions.

Second, we propose Octopus ORAM: A Storage Efficient 8-ary Tree ORAM. Octopus ORAM aims at reducing the client-server communication cost in the cost of introducing a small amount of server storage overhead, which is much lower than existing ORAM constructions. Octopus ORAM can significantly reduce the server storage overhead to around $0.34N$ (i.e., the server only needs to allocate $1.34N$ blocks when the client outsources $N$ blocks) while maintaining a comparable level of communication cost: to server a query, the online communication cost is 3 blocks and eviction (maintenance) communication cost is no more than $1.5 \log N$ blocks.

Third, we propose Three Servers ORAM: An efficient and Accountable $k$-ary Tree Storage Efficient ORAM. The new three servers ORAM incurs low server storage overhead, which is around $0.3N$ blocks for every $N$ real data blocks exported, the client-server communication cost is $O(B)$
bits per query by average, and server-server communication cost is $O(\log N \cdot B)$, lower communication costs than $S^3$ORAM, the most related state-of-the-art scheme. By supporting accountability with multiple servers, the work removes the less-realistic semi-honest assumption in a multi-server oblivious storage system.

For the future work, there are multiple directions to work on. First of all, we would like to improve the performance of the system by further reducing the communication costs, especially the inter-server communication costs. Secondly, we plan to polish the implementation and eventually make the system more robust to deploy. We also plan to improve ORAM efficiency when the server is configured with trusted platform model (TPM). When the server has TPM enforcement, we will be able to migrate some computation work to server and then with re-designed algorithms, we can make ORAM more efficient.
BIBLIOGRAPHY


