EAI Endorsed Transactions
on Security and Safety

Research Article EAI.EU

Octopus ORAM: An Oblivious RAM with Communication and Server Storage Efficiency

Qiumao Ma, Wensheng Zhang*

Iowa State University, Ames, IA 50011

Abstract

Most of existing ORAM constructions have communication efficiency as the major optimization priority; the server storage efficiency, however, has not received much attention. Motivated by the observation that, the server storage efficiency is as important as communication efficiency when the storage capacity is very large and/or the outsourced data are not frequently accessed, we propose in this paper a new ORAM construction called Octopus ORAM. Through extensive security analysis and performance comparison, we demonstrate that, Octopus ORAM is secure; also, it significantly improves the server storage efficiency, achieves a comparable level of communication efficiency as state-of-the-art ORAM constructions, at the cost of increased client-side storage, and the increased client-side storage should be affordable to the clients who adopt local facilities such as cloud storage gateways.

Received on 04 March 2019; accepted on 26 April 2019; published on 29 April 2019

Keywords: Cloud Storage, Data Access Pattern Privacy, Oblivious RAM

doi:10.4108/XX.X.X.XX

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 performance as well as security and privacy concerns, it is popular for organizational clients of cloud storage to deploy on-premise storage gateways. Figure 1 shows an example architecture for a hybrid cloud storage system, where an organizational client of cloud storage runs an on-premise, small-scale, private facilitate called storage gateway. The storage gateway has moderate storage and computing resources; for scalability and cost-efficiency, it outsources the majority of its data to one or multiple off-premise, public, larger and more scalable cloud storage servers. Meanwhile, the storage gateway stores the meta-data and a moderate set of the frequently-accessed or recently-accessed data, to manage data as well as reduce the frequency and latency of accessing data directly from the remote server.

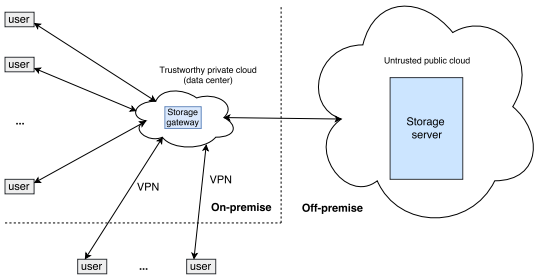


Figure 1. Hybrid Cloud Storage System

Research and Markets [32] projects the cloud storage market to have an annual growth rate of 29.73% to reach a total market size of 92.488 billion by 2022, and identifies the fast-increasing adoption of storage gateway as one major factor contributing to the growth.

When the client of a cloud storage cares the security of its data, it can encrypt the data before outsourcing. However, encryption can only protect data confidentiality. The client's access pattern to the outsourced data, if not properly protected, can be

*Corresponding author. Email: wzhang@iastate.edu

easily leaked to the cloud server or attackers who are interested in such information. Even worse, it has been found that, the leaked data access pattern can sometimes be used to infer the content of encrypted data [17].

The oblivious RAM (ORAM) model [11], which was originally proposed for software protection, has been regarded as a provable approach to protect the client's access pattern to outsourced data. In recent years, various ORAM constructions [12–16, 18, 21, 25, 28, 30, 31, 37–39, 39, 44, 45] have been developed.

As we survey in Section 7, the efficiency of client-server communication and client-side storage has been the main priority for optimization in developing ORAM constructions, but the efficiency in server storage has not received much attention. Particulary, to outsource N real data blocks to the cloud storage, Partition ORAM [37] incurs a communication cost of about $1.25 \log N$ blocks per data query, requires a client-side storage of $O(cN)$ (where c is a small number) blocks, and needs to store about $4N$ real and dummy blocks at the server. Path ORAM [38] has the similar level of communication efficiency, while the client-side storage is reduced to only $O(\log N)$ blocks; but it requires the server to store about $10N$ real and dummy blocks. Recently, S^3ORAM [16] was developed based on the deployment of multiple non-colluding servers; for each data query, this construction incurs only a constant number of blocks for client-server communication, but it requires at least $12N$ blocks to be stored at the servers.

The prioritization on communication efficiency over server storage efficiency is based on the assumption that, the monetary cost for communication is much higher than that for storage. However, this is not always true. Considering the Amazon S3 service in North America, 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, when the cloud server needs to store a large amount of data, the momentary cost for storage could be comparable to or even higher than that for communication. For example, consider a client of Amazon S3 uses high-speed Internet to connect to the cloud server, and the network bandwidth is 1 Gbps. If the client keeps accessing data from the server, and the bandwidth is fully utilized for the accesses, 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. Hence, as long as the client has data with several hundreds of TB or more to store at the server, the monetary cost for storage is comparable to or even higher than that for communication. This is even more evident in practice, considering that the client may not access the outsourced data too frequently, as frequently-accessed data could be cached locally.

Our Contributions. In this paper, we propose a new ORAM construction, called *Octopus ORAM* due to the adoption of 8-ary tree (i.e., each non-leaf node on the tree can have up to eight child nodes) as the server-side data structure for storing outsourced data blocks, to accomplish the efficiency in both communication and server storage, at the cost of increased but affordable client-side storage.

Specifically, we first propose a basic Octopus ORAM with the following features (assuming N data blocks are outsourced to the server; s, α and β are system parameters):

- Only one server is required.
- The amortized client-server communication cost per query is $(9 + 7\alpha) \log_8 \frac{N+3.5s}{3.5s}$ blocks.
- The storage overhead at the server is only $(\beta + \frac{1+\alpha}{7})N + \frac{(1+\alpha)s}{2} \approx 0.3N$ blocks.
- It is proved that, as long as $s \geq 25\lambda$, $\alpha \geq 0.34$ and $\beta \geq 0.13$, the Octopus ORAM fails with a probability of $O(2^{-\lambda})$.
- It is proved that, the Octopus ORAM does not disclose the client's data access pattern.

Then, we propose several optimizations to the basic Octopus ORAM:

- We propose a de-amortized eviction algorithm to allows query and eviction processes to run concurrently.
- We propose a piece-by-piece eviction algorithm to reduce the client-side storage overhead of the Octopus ORAM.
- We extends the Octopus ORAM to employ three non-colluding cloud servers, which reduces the client-server communication cost to about 2 blocks per query, at the cost of introducing a server-server communication cost of about $6 \log N$ blocks per query.

We compare Octopus ORAM with several state-of-the-art ORAM constructions through analytical or implementation-based study, which is reported in detail in Section 6. The major results are as follows.

- Compared to Partition ORAM [37]: The Octopus ORAM with a single server reduces the server storage overhead to be less than $\frac{1}{6}$ of that of Partition ORAM, while its client-server communication cost becomes 0.5 times larger than that of Partition ORAM. The Octopus ORAM with three servers achieves the same level of server storage efficiency as achieved by the Octopus ORAM with a single server, but it reduces the

client-server communication cost to only 2 blocks; as a tradeoff, it introduces background communication between the servers, of which the cost is less than 3 times of the client-server communication cost incurred by Partition ORAM. Note that, large client-server communication results in a data access delay that is directly experienced by the client, but the delay caused by server-server communication in the background can be hidden from the client.

- Compared to the Path ORAM [38], Octopus ORAM decreases the server storage overhead by 30 times, and has a lower communication cost which is about 23-30% of that of Path ORAM. As a tradeoff, it increases the client-side storage, which is affordable to a client that deploys an on-premise facility such as cloud storage gateway.
- Compared to the S^{3}ORAM [16], the Octopus ORAM with three servers decreases the server storage overhead by 33 times, and halves the client-side communication cost. As a tradeoff, it increases the client-side storage, which is also affordable to a client deploying an on-premise facility such as cloud storage gateway.

Organization. The rest of the paper is organized as follows. Section 2 defines the problem. Section 3 describes the basic Octopus ORAM, which is followed by the security and cost analysis in Section 4. Section 5 proposes several optimizations to the basic Octopus ORAM. Section 6 reports the performance comparisons against Partition ORAM, Path ORAM and S^{3}ORAM. Section 7 briefly surveys the related work. Finally, Section 8 concludes the paper.

2. Problem Definition

2.1. System Model

We consider a system consisting of a client and one or multiple non-colluding cloud servers. Note that, the proposed basic Octopus ORAM needs only a single cloud storage server, while an extended version assumes the existence of three non-colluding servers (as assumed in related works such as [16]) and at least one of them being cloud storage server.

Based on the hybrid cloud architecture discussed in Section 1, we assume the client has a local storage with a decent capacity, which however is much smaller than the capacity of the cloud storage server.

Following the prior research on ORAM constructions, we also assume the server(s) 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.

2.2. Security Definitions

Assume the client outsources N equal-size data blocks to the cloud storage server, and then accesses the data every now and then.

Each *data access* intended by 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 $D = (read, i)$ and formally a 3-tuple $(read, i, D)$; or (ii) write a data block D of unique ID i to the storage, denoted as a 3-tuple $(write, i, D)$.

To accomplish each 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: (i) 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 (ii) 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 [11, 37, 38], we define the security of our proposed ORAM as follows.

Definition Let λ 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 (i) 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 (ii) the ORAM system fails to operate with a probability of $O(2^{-\lambda})$.

3. The Basic Octopus ORAM

In this section, we present the detailed design of the basic Octopus ORAM, in terms of storage organization, system initialization, data query, and data eviction. The scheme uses integer $s > 1$, and fractions $0 < \alpha < 1$ and $0 < \beta < 1$, as system parameters to adjust the tradeoffs between security and costs.

3.1. Storage Organization and Initialization

Server-side Storage. The server-side storage is organized as a balanced tree, called *storage tree*, in which each non-leaf node can have up to eight child nodes.

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 $Z = (1 + \beta)Z'$ 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 $Z = (1+\beta)\frac{Z'}{\lfloor \frac{Z'}{3.5s} \rfloor}$ blocks. Each non-leaf node has a capacity of $3.5(1+\alpha)s$ 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).

Figure 2 illustrates the storage tree, which is a balanced 8-ary tree.

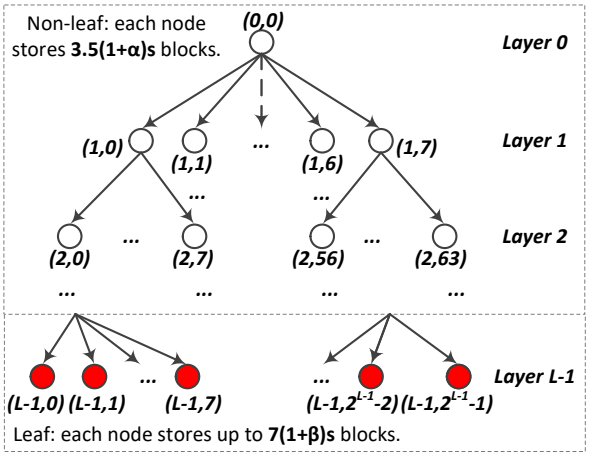


Figure 2. Server Storage Organization.

Client-side Storage. The client maintains an index table for all the N real data blocks and an index block for each node on the tree. The index table has N entries and each entry $i \in \{0, \cdots, N-1\}$ records the *path ID* of block i (i.e., the ID of the leaf node on the path storing block i). Note that, Octopus ORAM follows most of the tree-based ORAM constructions [9, 34, 35, 37, 38] and enforces the following policy: a block with a path ID must be stored on the path identified by the ID. 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 ($id \in \{1, \cdots, N\}$ if the block is real and $id = 0$ if the block is 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:

- $ah = 0$ if the block has not been accessed;
- $ah = 1$ if the block has been accessed as a query target;
- $ah = 2$ if the block has been accessed, but never as a query target.

System Initialization. To initialize the system, the client encrypts all the N 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 real or dummy blocks.

3.2. Data Query

Suppose the client wishes to query data block D_t that is not in its local buffer, where t denotes the ID of the block. It looks up its index table to obtain p_t, which is the path ID of the node, and looks up the index blocks of the path to identify the node that stores D_t. Then, the client selects path p_t as the *query path* and launches a query process as follows.

For each node n'_i on path p_t, where $i \in \{0, \cdots, L-1\}$ represents the layer ID of the node, let $S_{i,0}$, $S_{i,1}$ and $S_{i,2}$ denote the sets of blocks with ah values 0, 1 and 2 respectively, and $s_{i,0}$, $s_{i,1}$ and $s_{i,2}$ denote the cardinalities of the sets. The client selects one or two data blocks from each n'_i to download, according to the following rules, with the purpose of making each data block in n'_i to be downloaded with the same probability.

- *Case I: node n'_i contains the query target.* Depending on the access history of the target block, there are following subcases.
  - *Case I-a: the target belongs to set $S_{i,0}$.* The client picks the target to download. If $s_{i,1} + s_{i,2} > 0$, the client also randomly picks one block from $S_{i,1}$ with probability ρ or from $S_{i,2}$ otherwise (i.e., with probability $1-\rho$) to download, where
  $$\rho = \frac{s_{i,1}(s_{i,0} + s_{i,2})}{s_{i,0}(s_{i,1} + s_{i,2})}. \quad (1)$$
  - *Case I-b: the target belongs to $S_{i,1}$ or $S_{i,2}$.* The client picks the target to download and meanwhile, picks another block randomly from $S_{i,0}$ to download.
- *Case II: node n'_i does not contain the query target.* The client randomly picks one block from $S_{i,0}$ and another block from $S_{i,1} \cup S_{i,2}$ to download.

Note that, when a block is selected by the client to download, the server simply sends a copy of the block to the client and still keeps the block on the storage tree.

Upon receiving the selected blocks from the server, the client keeps only the query target block while discards the other blocks. Also, the client updates the index blocks maintained by itself through: 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.

Figure 3 illustrates the query process: In (a), the query target block has not been accessed before and In (b), the target block has been accessed before.

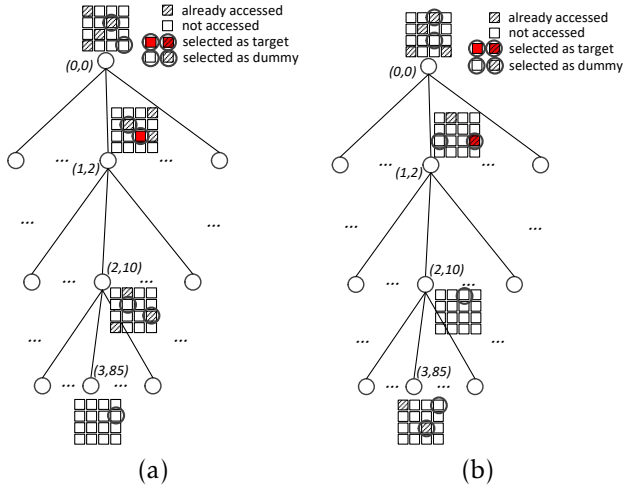


Figure 3. Query Examples.

The above query algorithm ensures that the client can always retrieve the query target, with the cost of retrieving up to two blocks from each layer of the storage tree. Meanwhile, the query process is random and independent of the client's data access pattern and thus can hide the data access pattern, due to the following reasons:

- The query path is the path associated with the query target, which has been selected for the query target randomly and independently. Hence, the selection of query path is independent of the client's data access pattern.

- On the query path, up to two blocks are selected from each node n'_i to download, one block from the set of blocks (i.e., $S_{i,0}$) that haven't been accessed yet and another from the set of blocks (i.e., $S_{i,1} \cup S_{i,2}$) that have been accessed already. As we formally show in Section 4 (Lemma 4), the above rule ensures that, each block in $S_{i,0}$ has the same probability to be selected and each block in $S_{i,1} \cup S_{i,2}$ also has the same probability to be selected, no matter whether the node contains the target or not. Hence, the selection of blocks to download is also random and independent of the client's data access pattern.

Also note that, the above query process could fail when enforcing the Case I of the rule, if probability ρ computed from Equation (1) is greater than 1. We defer the study of the failure probability to Section 4.

3.3. 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. Like in some existing ORAM constructions [37], 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. 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. In Section 5, we will discuss in detail on how to parallelize the eviction and query processes.

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 illustrated by Figure 4.

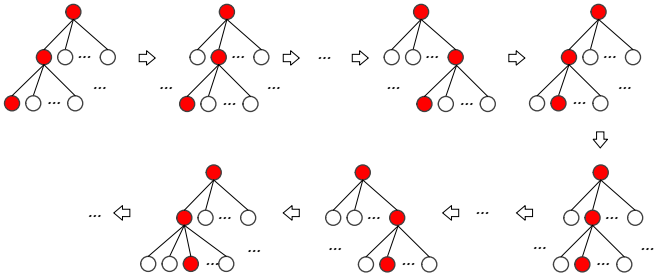


Figure 4. Reverse-lexicographic Order.

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 based on the following different case scenarios, which is also illustrated in Figure 5.

Case I: 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. First, the client retrieves all the $3.5(1+\alpha)s$ blocks from n'_e. 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 the paths associated with them.

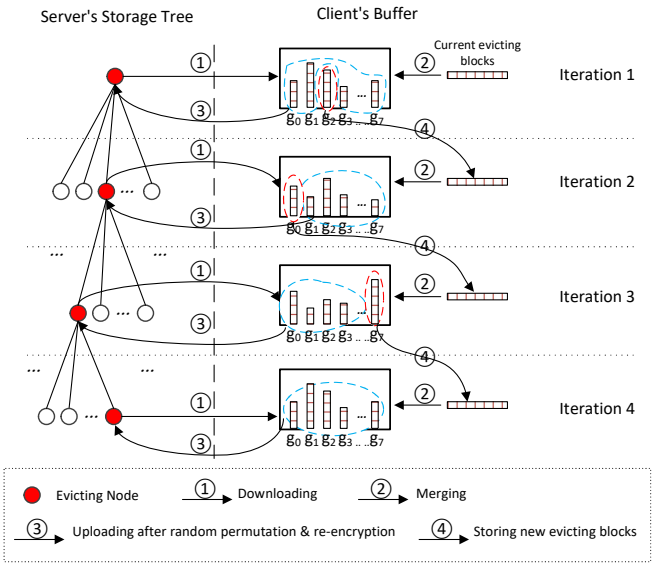


Figure 5. Eviction Overview.

Second, the client merges the real blocks in groups $c+1, \cdots, x-1, 0, \cdots, c-1$, randomly permutes and re-encrypts them, adds randomly-composed dummy blocks to them to make the total number of such blocks be $3.5(1+\alpha)s$, 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 block 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.

Note that, in the second step, it is possible that

$$|\bigcup_{i\in\{c+1,\cdots,x-1,0,\cdots,c-1\}} g_i| > 3.5(1+\alpha)s, \tag{2}$$

i.e., there are more than $3.5(1+\alpha)s$ real blocks that can only be evicted to the child nodes other than $n'_{e,c}$. When this occurs, more than $3.5(1+\alpha)s$ would be uploaded to n'_e, which would cause the storage space of the node to overflow. Hence, this is a failure scenario, and the client should declare failure and abort the process. In Section 4, we will analyze the probability for such failure to occur.

Case II: 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, the client declares failure and aborts; in Section 4, we will analyze the probability for such failure to occur. Otherwise, the client adds or removes dummy blocks to the make the total number of blocks to be Z, and randomly permutes and re-encrypts them before uploading them back to n'_e.

4. Security and Cost Analysis

In this section, we first study the failure probability and the obliviousness of the basic Octopus ORAM. Then, we analyze the costs of the construction.

4.1. Failure Probability Analysis

Octopus ORAM could fail at a query or eviction process.

Failure Probability for A Query Process. According to Section 3.2, a query process fails at a node n'_i which is on layer i, if and only if ρ computed according to Equation (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., $s_{i,1} > s_{i,0}$. 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 1. Let n_i denote an arbitrary node on layer i of the storage tree, and ξ_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

$$(\frac{e^\gamma}{(1+\gamma)^{1+\gamma}})^{x/\xi_i}, \tag{3}$$

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 $\tilde{q} = \frac{x}{\xi_i}$ queries that have n'_i on the query paths, because of the randomness in the path selection for blocks and that layer i has ξ_i nodes.

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

$$(\frac{e^\gamma}{(1+\gamma)^{1+\gamma}})^{x/\xi_i},$$

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 $s_{i,0} > s_{i,1}+s_{i,2}$ as long as n'_i has been on less than $\hat{q}$ query paths.

Hence the lemma is proved. □

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

Theorem 1. (Failure Probability for A Query Process in the Basic Octopus ORAM.) When an eviction process is always launched after every s queries and completed before any further query, as the Basic Octopus ORAM does, a query process fails at a node with a probability

no greater than

$$(\frac{e^{1.75\alpha+0.75}}{(1.75\alpha+1.75)^{1.75\alpha+1.75}})^{s}.$$

In particular, the probability is no greater than $2^{-\lambda}$ when $\alpha \geq 0.34$ and $s \geq 1.1\lambda$.

Proof. On the storage tree, for every node n'_i on layer i with totally ξ_i nodes, it is involved in an eviction after every $x = \xi_i \cdot s$ queries. Then, applying Lemma 1 with $x = \xi_i \cdot s$, the theorem is proved. □

Failure Probability for An Eviction Process. An eviction process can fail if and only if one of the following scenarios occurs.

- *Failure Scenario I:* This scenario occurs during an eviction iteration (detailed in Section 3.3) 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 $3.5(1+\alpha)s$ blocks to be uploaded to a non-leaf node and thus would lead to space overflow at the node.

- *Failure Scenario II:* This scenario occurs during the eviction iteration with a leaf node (with capacity Z 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.

Lemma 2. 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. Firstly, let us consider a current evicting node n'_e that is a non-leaf node with 8 children. Without loss of generality, we assume the leftmost child of n'_e, i.e., child 0, 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 $i = 0, \cdots, 7$ contains the real blocks that can only be evicted to child i. Note that the real blocks in each g_i were evicted to n'_e in the last $8-i$ eviction processes that involve n'_e.

On average, each eviction process involving n'_e evicts s real blocks to n'_e, and among them $\frac{s}{8}$ real blocks can only be evicted to child i of n'_e; hence, the average size of g_i, denoted as $|g_i|$, is $\frac{8-i}{8} \cdot s$ blocks. Due 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

$$[\frac{e^{\alpha_i}}{(1+\alpha_i)^{(1+\alpha_i)}}]^{\frac{8-i}{8}\cdot s}. \quad (4)$$

Following inequality (4), 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 $\alpha = \frac{\sum_{i=1}^{7}[\frac{8-i}{8}\cdot s\cdot(1+\alpha_i)]}{\sum_{i=1}^{7}[\frac{8-i}{8}\cdot s]} \geq 0.339$, the Failure Scenario I occurs with a probability less than $7 * 2^{-\lambda}$, i.e., $O(2^{-\lambda})$.

Next, we consider a general scenario that the current evicting node n'_e has $x \leq 8$ children. The above proof can be simply changed to that the number of groups is x instead of 8. After group 0 is evicted to the leftmost child, the rest groups (i.e., groups $1, \cdots, x-1$) should not have more blocks than the groups $1, \cdots, 8$ in the above proof. Hence, the probability for the number of blocks in groups $1, \cdots, x-1$ to exceed $3.5(1+\alpha)s$ blocks is also $O(2^{-\lambda})$. □

Lemma 3. 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. On the server-side storage tree, 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 $\bar{x}_i = \frac{Z}{1+\beta} \geq 3.5s$ due to the following reasoning:

- When the root of the storage has exactly 8 children, the total number of leaf nodes is $8^{L'}$, and thus $\bar{x}_i = \frac{N}{8^{L'}}$. Due to $Z = (1+\beta)\frac{N}{8^{L'}}$, it holds that $\bar{x}_i = \frac{Z}{1+\beta}$.

- When the root of the storage has less than 8 children, the total number of leaf nodes is $8^{L'} \cdot \lfloor \frac{Z'}{3.5s} \rfloor$, and thus $\bar{x}_i = \frac{N}{8^{L'}\cdot\lfloor\frac{Z'}{3.5s}\rfloor}$. Due to $Z = (1+\beta)\frac{Z'}{\lfloor\frac{Z'}{3.5s}\rfloor}$, it holds that $\bar{x}_i = \frac{Z}{1+\beta}$.

Further due to the multiplicative Chernoff bound, for any leaf node with ID i:

$$Pr[x_i \geq Z] = Pr[x_i \geq (1+\beta)\bar{x}_i] < (\frac{e^{\beta}}{(1+\beta)^{(1+\beta)}})^{\bar{x}_i}. \quad (5)$$

Because $\bar{x}_i \geq 3.5s$, it holds that $Pr[x_i \geq Z] < 2^{-\lambda}$ when $\beta \geq 0.13$ and $s \geq 25\lambda$. □

Following Lemmas 2 and 3, we have the following main theorem regarding the failure probability for Octopus ORAM.

Theorem 2. (Failure Probability of an Eviction Process in the Basic Octopus ORAM.) 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 $s \geq 25\lambda$.

4.2. Obliviousness Analysis

In this subsection, we analyze the obliviousness of the query and eviction processes. That is, we show that these processes are random and independent of the client's data access pattern.

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 the determined by the path ID of the query target block, is random and independent of the client's access pattern.

Obliviousness in Block Access from Query Path. From each node on the query path selected for a query process, the client must select one block that has already been accessed and one block that has not been accessed to access. As stated and proved in Lemma 4, the algorithm for block selection ensures that, each of the blocks that have already been accessed has the same probability to be selected; 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 4. 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 in the data query algorithm (Section 3.1). 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 as follows:

- If the node has not been accessed since its most recent construction, one block is randomly selected to access; hence, each block has the same probability to be selected.
- 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 construction, D_t is any block on the node with the same probability. According to the query algorithm, D_t is selected to access; hence, each block has the same probability to be selected.

If the node has been accessed since its most recent construction, $S_{i,0}$ is the set of blocks that have not been accessed before, $S_{i,1}$ 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-targets (i.e., dummies). 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 ρ (i.e., each block in $S_{i,1}$ has the probability of $\frac{\rho}{s_{i,1}} = \frac{s_{i,0}+s_{i,2}}{s_{i,0}(s_{i,1}+s_{i,2})}$ 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

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

to be accessed. □

Obliviousness in eviction process. The eviction process is random and independent of the client's data access pattern, due to the following reasons:

- 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.
- 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 about failure probability and obliviousness, we get the following theorem.

Theorem 3. (Security of the Basic Octopus ORAM.) As long as system parameters $\alpha \geq 0.34$, $\beta \geq 0.13$ and $s \geq 25\lambda$, the basic Octopus ORAM is secure under Definition 2.2.

4.3. Cost Analysis

Communication Cost. For each query process, up to two blocks are downloaded from each node on the query path.

After every s query processes, an eviction process is launched. During an eviction process, one root-to-leaf path is selected to access. All the data blocks on the eviction path are downloaded and then replaced with the same number of blocks uploaded. Hence, in total $7(1+\alpha)(L-1)s + 2Z$ blocks are transferred between the client and the server.

To summarize, the average communication cost per query is

$$2L + [7(1+\alpha)\cdot L + 2Z]/s \approx (9+7\alpha)\log_8 \frac{N+3.5s}{3.5s} \quad (6)$$

blocks.

Server-side Storage Cost. The number of leaf nodes on the server-side storage tree is at most $8^{L-1} = \frac{N+3.5s}{3.5s}$, and the number of non-leaf node is no more than $1 + \cdots + 8^{L-2} < 8^{L-1}/7 = \frac{N+3.5s}{24.5s}$. Considering that each leaf nodes store $(1+\beta)N$ data blocks and each non-leaf node stores $3.5(1+\alpha)s$ blocks, the total number of blocks stored by the server is no more than

$$(1+\beta+\frac{1+\alpha}{7})N + \frac{(1+\alpha)s}{2} \quad (7)$$

blocks. In particular, when $\alpha = 0.34$ and $\beta = 0.13$, the server-side storage overhead is about $0.3N$ blocks.

Client-side Storage Cost. The client stores an index table, which takes at most $N \cdot \log 8^{L-1} = 3(L-1)N$ bits. It also stores an index block for each node, where every dummy block takes 3 bits (one for ID 0 and two for *ah*) and every real block takes $\log N + 2$ bits ($\log N$ for ID and two for *ah*); hence, the index blocks consumes about $N \cdot (\log N + 3)$ bits. Besides the above permanent storage consumption, the client also needs to store the recently queried s blocks, and up to Z blocks at the time of eviction. Hence, the overall storage consumption is about

$$N \cdot (\log N + 3) + [s + 7(1+\beta)]B \quad (8)$$

bits, where B denotes the size of a block.

5. Optimizations

In this section, we discuss several optimizations that can improve the performance of the basic Octotupus ORAM.

5.1. De-amortized Eviction

In the basic Octopus ORAM, no query can be processed when an eviction process is on going, and the eviction process could take a long time when system parameter s and the number of outsourced blocks are large. Hence, we propose to de-amortize this process and thus allow the query and eviction processes to run concurrently.

The De-amortization Algorithm. To de-amortize an eviction process, we first divide the process evenly into s sub-processes called *eviction steps*, each having the same amount of communication workload. Specifically, letting $s \cdot w$ denote the total number of block transferred between the client and the server during an eviction process, Then, the Octopus ORAM system with de-amortized eviction works as follows.

1. Each query process is uniquely identified by a non-positive integer, from 0 and onwards consecutively. The query process is conducted in the same way as in the basic Octopus ORAM.
2. After every $i \cdot s$ (where $i \geq 1$) queries have been processed, a new eviction process is launched. As the eviction process is divided into s eviction steps, each step is identified by $(i-1)\cdot s + j$ where $j = 0, \cdots, s-1$.
3. For each query identified by $k \geq s$, the client starts processing it only if $k - s$ or more eviction steps have already completed.
4. When processing query $k \geq s$, like in the basic Octopus ORAM, the path that contains the query target is selected as the query path. Then, one or two blocks are selected from each node to retrieve according to the same query algorithm in Section 3.2. Note that, for a node that is currently involved in an eviction step, its blocks can be all at the server, all at the client, or partly at the server and partly at the client; the block selection is not affected by the distribution.

Security Analysis. Applying the de-amortized eviction algorithm does not change the obliviousness property of the Octopus ORAM or the failure probability of an eviction process. However, it makes a node to have more blocks accessed before an eviction process arrives at the node. Lemma 5 states how this change impacts the failure probability of a query process, the proof of which is presented in the Appendix.

Lemma 5. In an Octopus ORAM with de-amortized eviction, as long as $s \geq 8\lambda$ and the storage tree has at least 4 leaf nodes, a query process fails with a probability of $O(2^{-\lambda})$.

Proof. Let ξ_l denote the number of nodes at layer $0 \leq l \leq L-1$, where $\xi_{L-1} \geq 4$. According to the de-amortized eviction algorithm, between two consecutive

eviction processes (note: the system initialization is treated at the first eviction process) completed at a node on layer $l \in \{0, \cdots, L-1\}$, there should be at most $q_l = s \cdot \xi_l + s \cdot \frac{l+1}{L}$ queries processed.

For the root node of the storage tree, it is obvious that $q_0 < 1.5s < 1.75(1+\alpha)s$ as long as $L > 1$ (which is implied by the existence of 4 leaf nodes); hence, a query process will never fail at the root.

For a non-root node at layer l (where $l \geq 1$), it is obvious that $q_l < (\xi_l + 1)s \leq 1.25\xi_l \cdot s$; hence, among these queries, the average number of queries that involve this node is $\hat{q}_l = \frac{q_l}{\xi_l} < 1.25s$. Due to the multiplicative Chernoff bound, the probability for more than $1.75s$ queries to involve this node is

$$(\frac{e^{0.4}}{1.4^{1.4}})^{1.25s} < (0.915)^s,$$

where $1.4 = 1.75/1.25$. The probability is less than $2^{-\lambda}$ as long as $s \geq 8\lambda$. □

Cost Analysis. Applying the de-amortized eviction algorithm only changes the temporal distribution of communication workload, but not the overall cost of communication. The computation and server-side storage costs are not changed either. However, it increases the client-side storage cost, because the client now needs to have two dedicated buffers, one for recently queried target blocks and one for eviction. Note that, with the basic Octopus ORAM, only one buffer is needed, which is used to support eviction at the eviction time and used to store the recently queried target blocks during other time. The increase in storage consumption is bounded by s blocks.

5.2. Piece-by-piece Eviction

During an eviction process, the client needs to store the s *evicting blocks* as well as all the blocks in *evicting node* n'_e; recall that a non-leaf evicting node has $3.5(1+\alpha)s$ blocks and a leaf evicting node has Z blocks. To reduce the client-side storage cost, we further propose piece-by-piece eviction in the following.

Dividing Blocks into Pieces. To facilitate piece-by-piece eviction, we first divide the plain text of each block into $\theta - 1$ *pieces* each of τ bits. That is, each block of ID i is split into pieces $d_{i,1}, d_{i,2}, \cdots, d_{i,\theta-1}$. Before being exported to the remote storage server, the plain-text data block is encrypted piece by piece with a secret key k, as shown in Figure 6:

$$\begin{aligned}
&c_{i,0} = E_k(r_i), \text{ where } r_i \text{ is a random number used as IV;}\\
&c_{i,1} = E_k(c_{i,0} \oplus d_{i,1});\\
&c_{i,2} = E_k(c_{i,1} \oplus d_{i,2});\\
&\cdots,\\
&c_{i,\theta-1} = E_k(c_{i,\theta-2} \oplus d_{i,\theta-1}).
\end{aligned}$$

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

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

It contains θ pieces and has $\tau \cdot \theta$ bits. Note that, a block in cipher text is longer than the block in plain text due to the addition of IV. This communication and storage overhead however is not brought by our proposed piece-by-piece eviction; as long as a block is encrypted probabilistically, IV is needed.

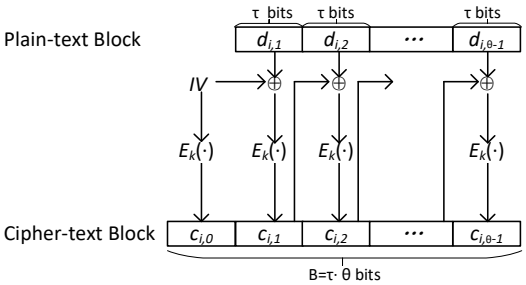


Figure 6. Format of a data block.

Piece-by-piece Eviction Algorithm. When an eviction process is working on evicting node n'_e with a capacity of c blocks, the eviction buffer consists of the following segments:

- *Segment 0*, which can store $c+s$ pieces that are used as IV pieces for re-encryption;
- *Segment 1*, which can store $c+s$ pieces that are used as IV pieces for decryption.
- *Segment 2*, which can store $c+s$ pieces that are used to store pieces waiting to be decrypted and then re-encrypted.
- *Segment 3*, which can store the s current evicting blocks.

The piece-by-piece eviction process at n'_e is as follows.

The client first devices a permutation function π that can randomly permute $s+c$ pieces. Specifically, π should permute an input block sequence, which includes the s current evicting blocks followed by the c blocks currently at n'_e, to an output block sequence, which includes the s new evicting blocks followed by the c blocks that should be evicted to n'_e; that is, π should work according to the eviction algorithm in Section 3.3.

Then, the client randomly picks $s+c$ new IV pieces and stores them to Segment 0 of the eviction buffer. It also retrieves the first pieces of the s evicting blocks and the first pieces of the c blocks currently at n'_e, and saves these $c+s$ pieces, which are the old IV pieces, to Segment 1 of the eviction buffer.

Next, the client iteratively re-encrypts, permutes and uploads pieces $1, \cdots, \theta - 1$ of the current s evicting blocks and c blocks at n'_e. Specifically, each iteration $i = 1, \cdots, \theta - 1$ runs as follows.

1. The client retrieves the pieces with offset i from the s current evicting blocks stored at Segment 3 and the c blocks stored at n'_e, and stores them to Segment 2 of the eviction buffer.

2. The client permutes all the pieces in Segment 0 using π, saves the first s pieces of the permuted sequence to Segment 3, and uploads the rest c pieces to n'_e.

3. A temporary variable iv is used by the client for re-encryption. For each $j = 0, \cdots, c + s - 1$, the client lets iv be piece j of Segment 1, replaces the piece j of Segment 1 with the piece j of Segment 2, and updates the piece j of Segment 2 by decrypting it with its secret key the iv as the IV. Hence, after this step finishes, Segment 1 stores the pieces with offset i from the s current evicting blocks and the s blocks stored at n'_e, which will be used as the IV pieces for decryption in the next iteration; Segment 2 stores the plain texts which will be re-encrypted in the next step.

4. The client re-encrypts pieces in Segment 2 with its secret key and the pieces in Segment 0 as the IV pieces. Then, the pieces in cipher text are copied to Segment 0 as IV pieces for re-encryption in the next iteration.

As we can see, the piece-by-piece eviction algorithm reduces the size of eviction buffer from $Z + s$ blocks to s blocks plus $3Z\tau$ bits.

5.3. Extension to Multiple Servers

As analyzed in subsection 4.3, the client-server communication cost is $O(\log_8 N)$ blocks per query in the basic Octopus ORAM. To reduce the cost, we propose an optimization based on employing three non-colluding cloud servers, denoted as $\mathcal{S}_0$, $\mathcal{S}_1$ and $\mathcal{S}_2$. Among them, Server $\mathcal{S}_0$ hosts the storage tree, $\mathcal{S}_1$ stores the most recently accessed s data blocks, and $\mathcal{S}_2$ only helps in re-encrypting data blocks.

System Initialization. The client picks a pseudo random number generator $PRG_0(k)$, which takes a secret seed k of l bits (where l is a security parameter) and outputs a pseudo-random sequence of $3l$ bits. The client also 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 data block.

In the index table maintained by the client, the entry for each block with ID i now stores two fields: the path ID of the block and a secret key k_i which is randomly selected by the client to *encrypt* the block.

Initially, before each real block D_i with ID i is outsourced, the block is encrypted as follows. The client first randomly picks a secret seed k_i. Then, it computes $PRG_0(k_i)$ to obtain $k_{i,0} \| k_{i,1} \| k_{i,2}$, a concatenation of three secret seeds of the same length. Next, it computes $PRG_1(k_{i,0})$, $PRG_1(k_{i,1})$ and $PRG_1(k_{i,2})$ to generate three pseudo-random blocks denoted as $R_{i,0}$, $R_{i,1}$ and $R_{i,2}$. Finally, it performs bit-wise XOR operations on the four blocks D_i, $R_{i,0}$, $R_{i,1}$ and $R_{i,2}$, to encrypt D_i to $D'_i = D_i \oplus R_{i,0} \oplus R_{i,1} \oplus R_{i,2}$.

Data Query Process. During a query process, up to two blocks need to be accessed at each layer of the storage tree, just like in basic Octopus ORAM. For simplicity, suppose the client needs to retrieve exactly two blocks from each layer of the storage tree at S_0, and the IDs of these blocks are $i_0, i_1, \cdots, i_{2L-1}$. The client sends a request to $\mathcal{S}_0$, which contains (i) the offsets of these selected blocks on the layers where they reside and (ii) a random permutation function; the client also sends to $\mathcal{S}_1$ a request which contains a number between 0 and $2L - 1$.

In response to the client's request, $\mathcal{S}_0$ makes a copy of the selected blocks (i.e., the ones with IDs $i_0, i_1, \cdots, i_{2L-1}$), permutes the copies using the permutation function provided by the client, and forwards the resulting block sequence to $\mathcal{S}_1$. Upon receiving the sequence, $\mathcal{S}_1$ retains only the query target block whose offset on the sequence is the number contained in the client's request, and immediately returns a copy of the block to the client. After the client has accessed the block, the block may be updated, and then re-encrypted and sent back to $\mathcal{S}_1$.

Data Eviction Process. Next, we present the revised eviction algorithm for the three servers to carry out. For simplicity, we consider only the scenario that an eviction process is launched immediately after s queries have been processed and completed before any further query can be processed; based on this algorithm, a de-amortized eviction algorithm can be developed accordingly to allow concurrent execution of eviction and query.

When the eviction process begins, $\mathcal{S}_1$ stores the s blocks queried most recently. It appends $\alpha \cdot s$ dummy blocks to the end of the s blocks. Then, the eviction process runs iteratively, one iteration for each node on the eviction path. Like in Section 3.3, we use 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). Next, we elaborate the operations of each iteration using the following notations.

Let $\mathcal{L}_0$ denote a sorted list of blocks currently stored in node n'_e (on the storage tree hosted by $\mathcal{S}_0$); $\mathcal{L}_1$ denote the list of blocks currently stored at server $\mathcal{S}_1$ and sorted

according to the order they were stored; $\mathcal{L}'_0$ denote the ordered list of blocks that should be stored to n'_e after the eviction; $\mathcal{L}'_1$ denote the ordered list of blocks that should be stored to $\mathcal{S}_1$ after the eviction. Hence, $\mathcal{L}_0 \cup \mathcal{L}_1 = \mathcal{L}'_0 \cup \mathcal{L}'_1$. Also, for every block of ID i belonging to $\mathcal{L}_0 \cup \mathcal{L}_1$, let k_i denote its current secret seed and k'_i denote the new secret seed randomly picked for the block; furthermore, let $PRG_0(k_i) = k_{i,0}||k_{i,1}||k_{i,2}$ and $PRG_0(k'_i) = k'_{i,0}||k'_{i,1}||k'_{i,2}$.

1. The client sends to $\mathcal{S}_0$ a permutation function π_0, and asks $\mathcal{S}_0$ to permute blocks in n'_e with π_0 and send the resulting block sequence $\mathcal{L}_0$ to $\mathcal{S}_1$. $\mathcal{S}_1$ appends $\mathcal{L}_1$ to the end of $\mathcal{L}_0$, and thus gets a new list $\mathcal{L}_0||\mathcal{L}_1$.

2. The client sends to $\mathcal{S}_1$ a permutation function π_1 and the following list of secret seeds

$$\langle (k_{i_0,0}, k'_{i_0,1}), \cdots, (k_{i_{n-1},0}, k'_{i_{n-1},1}) \rangle,$$

where $n = |\mathcal{L}_0| + |\mathcal{L}_1|$, and $\langle i_0, i_1, \cdots, i_{n-1} \rangle$ is the list of IDs of blocks in $\mathcal{L}_0||\mathcal{L}_1$. Upon receiving π_1 and the list, $\mathcal{S}_1$ updates each block $D_{i_j} \in \mathcal{L}_0||\mathcal{L}_1$ to $D_{i_j} \oplus R_{i_j,0} \oplus R'_{i_j,1}$ where $R_{i_j,0} = PRG_1(k_{i_j,0})$ and $R'_{i_j,1} = PRG_1(k_{i_j,1})$. Then, $\mathcal{S}_1$ permutes the blocks with π_1 and sends the resulting list, denoted as $\mathcal{L}'$, to $\mathcal{S}_2$.

3. The client sends to $\mathcal{S}_2$ the following list of secret seeds

$$\langle (k_{i'_0,1}, k'_{i'_0,2}), \cdots, (k_{i'_{n-1},1}, k'_{i'_{n-1},2}) \rangle,$$

where $n = |\mathcal{L}'|$, and $\langle i'_0, i'_1, \cdots, i'_{n-1} \rangle$ is the list of IDs of blocks in $\mathcal{L}''$. Upon receiving the list, $\mathcal{S}_2$ updates each block $D_{i'_j} \in \mathcal{L}''$ to $D_{i'_j} \oplus R_{i'_j,1} \oplus R'_{i'_j,2}$ where $R_{i'_j,1} = PRG_1(k_{i'_j,1})$ and $R'_{i'_j,2} = PRG_1(k'_{i'_j,2})$. Then, $\mathcal{S}_2$ sends the updated list of blocks, denoted as $\mathcal{L}''$, to $\mathcal{S}_0$.

4. The client sends to $\mathcal{S}_0$: (i) a list of offsets each between 0 ,and (ii) $n-1$ and (ii) the following list of secret seeds

$$\langle (k_{i''_0,2}, k'_{i''_0,0}), \cdots, (k_{i''_{n-1},2}, k'_{i''_{n-1},0}) \rangle,$$

where $n = |\mathcal{L}''|$, and $\langle i''_0, i''_1, \cdots, i''_{n-1} \rangle$ is the list of IDs of blocks in $\mathcal{L}''$. Upon receiving the list of secret seeds, $\mathcal{S}_0$ updates each block $D_{i''_j} \in \mathcal{L}''$ to $D_{i''_j} \oplus R_{i''_j,2} \oplus R'_{i''_j,0}$ where $R_{i''_j,2} = PRG_1(k_{i''_k,2})$ and $R'_{i''_j,0} = PRG_1(k'_{i''_j,0})$. Then, $\mathcal{S}_0$ retains the blocks with the offsets specified in the offset list received from the client. If n'_e is not a leaf node, $\mathcal{S}_0$ sends the rest blocks to $\mathcal{S}_1$; otherwise, it simply discards the rest blocks.

Cost Analysis. In the following, we analyze the communication and storage costs of the Octopus ORAM with three servers.

- *Client-Server Communication Cost*: During a query process, the client sends $2L$ offsets and a permutation vector for these offsets to $\mathcal{S}_0$; meanwhile, it receives the query target block from $\mathcal{S}_1$ and needs to write it back after re-encryption.

  During an eviction, which occurs after every s query processes, the client does not exchange data blocks with the server; however, it needs to send secret seeds and permutations to the servers. The number of seeds sent to the server is no more than $6 \cdot [3.5(1+\alpha)+1] \cdot (L-1) + 6 \cdot [7(1+\beta)+1]$ per query. In particular, when $l = 128$ bits, $\alpha = 0.34$ and $\beta = 0.13$, the number of bytes sent as seeds is about $192 \log \frac{N}{3.5s}$; furthermore, if $N < 2^{40}$, this is less than 8 K bytes.

  As offsets and the permutation vector are small, the client-server communication cost is about $2B + 8K$ bytes per query, where B is the block size in bytes.

- *Server-Server Communication Cost*: During a query process, at most $2L$ blocks are sent from $\mathcal{S}_0$ to $\mathcal{S}_1$. When an eviction process works on an evicting node n'_e, the s current evicting blocks and all the blocks in n'_e are sent for three times between the servers. Hence, considering that an eviction process occurs after every s queries, the total server-server communication cost per query is up to $[3.5(1+\alpha)+1]s \cdot (L-1) + [7(1+\beta)+1]s$ blocks. In particular, when $\alpha = 0.34$ and $\beta = 0.13$, the cost is no more than $6 \log \frac{N+3.5s}{3.5s}$ blocks.

- *Server-side Storage Cost*: The storage cost of $\mathcal{S}_0$ is the same as the server in the basic Octopus ORAM. $\mathcal{S}_1$ needs to store the s most recently queried data blocks. Besides, each of $\mathcal{S}_1$ and $\mathcal{S}_2$ also needs a temporary buffer that can store up to $Z+s$ blocks to support re-encryption and permutation operations during an eviction process. Thus, compared to the basic Octopus ORAM, the Octopus ORAM with three servers introduces an extra storage cost of $2s + Z < (7\beta + 9)s$ blocks.

- *Client-side Storage Cost*: Compared to the basic Octopus ORAM, the Octopus ORAM with three servers increases the client-side storage cost sightly by $N \cdot l$ bits, where l is the size of a secret seed in the unit of bit.

6. Performance Comparisons

We implement the proposed schemes, and conduct performance comparisons between Octopus ORAM and

several state-of-the-art ORAM constructions that are the most related. Specifically, we conduct the following three comparisons:

- We compare Octopus ORAM to Partition ORAM [37], 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 the Octopus ORAM with a single server to Path ORAM, which is one of the most communication-efficient ORAM that does not require intensive computation or multiple servers and only requires a small local storage (i.e., $O(logN)$ blocks).
- We compare the Octopus ORAM with 3 servers to the S^{3}ORAM [16] with also 3 servers, which is a state-of-the-art ORAM construction requiring multiple non-colluding servers.

6.1. Comparison with Partition ORAM

We implement two versions of Octopus ORAM, i.e., the Octopus ORAM with a single server and the Octpus ORAM with 3 servers. For both versions, the optimizations of de-amortized eviction and piece-by-piece eviction have been applied. The system parameters s, α and β 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 [37] in this comparison. Also the block size is set to 64 KB, as used in [37].

Table 1 compares the ORAM constructions in terms of the client-side storage cost, which 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.

- When Octopus ORAM uses a similar size of client-side storage as Partitiona ORAM, 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 Section 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 with 3 servers does not increase the server-side storage noticeably; but it incurs around twice client-side storage cost, compared to the other two constructions. The increase is mainly due to the storage of secret seeds for encrypting and decrypting data blocks.

Table 2 compares the communication costs between the ORAM constructions. Particularly, we measure and compare the client-server and server-server communication costs per query, in the unit of data block. The observations from the table are as follows.

- With similar client-side storage cost, Octopus ORAM with single server 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.
- Octopus ORAM with single server can be conveniently extended to Octopus ORAM with 3 servers. With the extension, the client-server communication cost can be sharply reduced to about 2 blocks per query, at the price of introducing server-server communication cost that is less than 3 times of the client-server communication cost of Partition ORAM. Besides the storage-communication tradeoff discussed above, this comparison demonstrates another tradeoff between the data query latency experienced by the client and the overall communication cost: our proposed ORAM significantly reduces the query latency at the cost of introducing more server-server communication occurring in the background.

6.2. Comparison with Path ORAM

To evaluate the performance of the Octopus ORAM with a single server 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 [33]), with a round trip delay of 50 ms added intentionally (as done in [44]), in order to simulate a practical scenario that the client has a high-speed Internet connection with the server.

Table 1. Comparing Storage Efficiency Between Partition ORAM and Octopus ORAMs

Capacity	# Blocks	Client Storage Cost			Server Storage Overhead		
		Partition ORAM	Octopus (1 Server)	Octopus (3 Servers)	Partition ORAM	Octopus (1 Server)	Octopus (3 Servers)
64 GB	2^{20}	204 MB	203.5 MB	16.5 MB	141 GB	16.8 GB	18.8 GB
256 GB	2^{22}	415 MB	415 MB	66 MB	563 GB	67.7 GB	71.9 GB
1 TB	2^{24}	858 MB	858 MB	264 MB	2.2 TB	0.3 TB	0.3 TB
16 TB	2^{28}	4.2 GB	4.3 GB	4.1 GB	35 TB	4.7 TB	4.8 TB
256TB	2^{32}	31 GB	31.7 GB	66 GB	563 TB	80.1 TB	80.4 TB
1024TB	2^{34}	101 GB	103.42 GB	264 GB	2018 TB	314.1 TB	315 TB

Table 2. Comparing Communication Efficiency Between Partition ORAM and Octopus ORAMs

Capacity	# Blocks	Client-Server			Server-Server		
		Partition ORAM	Octopus (1 Server)	Octopus (3 Servers)	Partition ORAM	Octopus (1 Server)	Octopus (3 Servers)
64 GB	2^{20}	22.5 X	33.6 X	2 X	-	-	56 X
256 GB	2^{22}	24.1 X	36 X	2 X	-	-	73 X
1 TB	2^{24}	25.9 X	41 X	2 X	-	-	70 X
16 TB	2^{28}	29.5 X	43.7 X	2 X	-	-	87 X
256TB	2^{32}	32.7 X	50.3 X	2 X	-	-	86 X
1024TB	2^{34}	34.4 X	50.6 X	2 X	-	-	87 X

The instances are both of type AWS m4.xlarge, each has 4 vCPUs, 2.4 GHz, Intel Xeon E5-2686v4, and 16 GB memory. Note that, Octopus ORAM has embedded the optimizations of de-amortized and piece-by-piece eviction.

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:

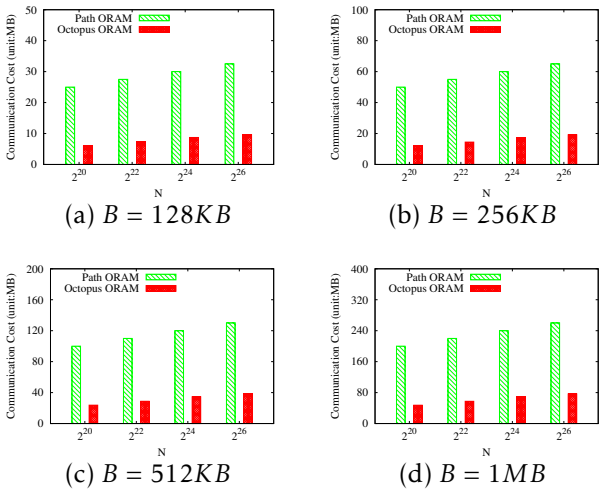
- *Communication Cost per Query*, which is measured as the average amount of data uploaded to or downloaded from the server, per query, in the unit of data block.
- *Query Delay*, which is measured as the average time elapse from when the client sends out a query request to when the client has received 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: $\lambda = 40$, $s = 1024$, $\alpha = 0.34$ and $\beta = 0.13$. 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 7 compares the communication cost per query between Octopus ORAM and Path ORAM. As we can see, the communication cost incurred by Octopus ORAM is 23-30 % of that by Path ORAM, as N and B vary.

Query Delay. Figure 8 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 and decrypt a small number of blocks (i.e., up to 2 blocks from each layer of the storage tree which has a much smaller height than the tree used in Path ORAM), and a query target block can be accessed immediately after the query process finishes.

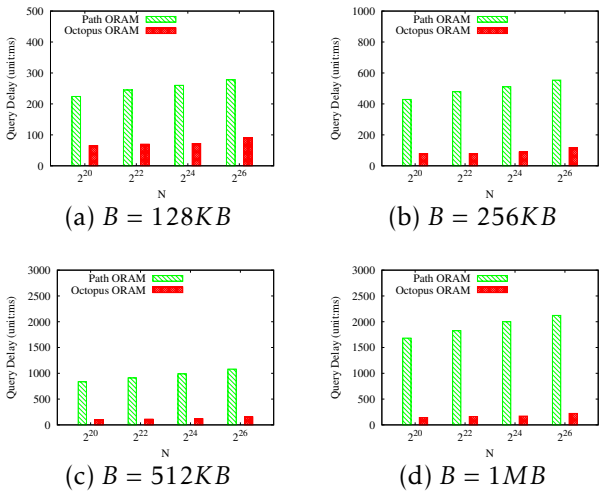


(a) $B = 128KB$ (b) $B = 256KB$

(c) $B = 512KB$ (d) $B = 1MB$

Figure 7. Comparing Communication Cost per Query Between Octopus ORAM and Path ORAM.

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 downloaded and decrypted half of the blocks that need to be processed, and the number of such blocks is much larger than that in Octopus ORAM.



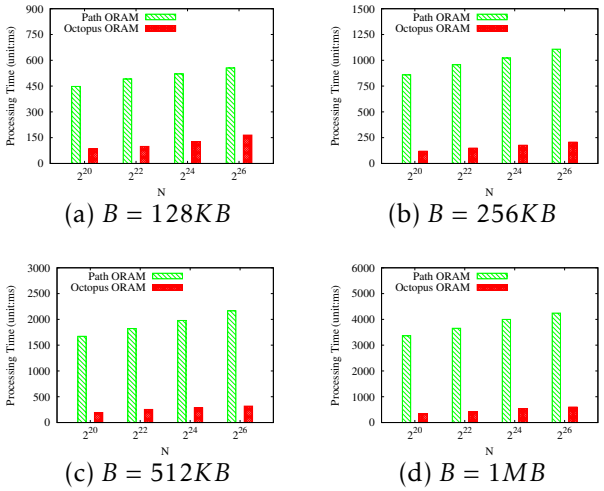
(a) $B = 128KB$ (b) $B = 256KB$

(c) $B = 512KB$ (d) $B = 1MB$

Figure 8. Comparing Query Delay Between Octopus ORAM and Path ORAM.

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

Processing Time per Query. Figure 9 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 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.



(a) $B = 128KB$ (b) $B = 256KB$

(c) $B = 512KB$ (d) $B = 1MB$

Figure 9. Comparing the Processing Time per Query between Octopus ORAM and Path ORAM.

Storage Cost and Overhead. Table 3 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.

Table 3. Comparing Storage Efficiency between Octopus ORAM (with single server) and Path ORAM

Capacity	# Blocks	Client Storage Cost		Server Storage Overhead	
		Path ORAM	Octopus ORAM	Path ORAM	Octopus ORAM
64 GB	2^{20}	0.25 MB	64.5 MB	576 GB	18.8 GB
256 GB	2^{22}	1 MB	66 MB	2.2 TB	76 GB
1 TB	2^{24}	4 MB	72 MB	9 TB	304 GB
4TB	2^{26}	16 MB	96 MB	36 TB	1.2 TB
16 TB	2^{28}	64 MB	192 MB	144 TB	4.7 TB
64 TB	2^{30}	256 MB	576 MB	576 TB	19 TB

6.3. Comparison with S^{3}ORAM

The results of comparing Octopus ORAM with S^{3}ORAM are as follows.

- *Client-Server Communication Cost*. Both constructions require a constant number of blocks to be transferred between the client and server for each query. Specifically, S^{3}ORAM needs to transfer 6 data blocks, while Octopus ORAM needs to transfer 2 data blocks plus about 8KB data.

- *Server-Server Communication Cost*. Both constructions have a similar level of server-server communication cost, which is around $6 \log N$ data blocks per query.

- *Server-side Storage Cost*. Both constructions 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. Octopus ORAM stores blocks on one server, i.e., $\mathcal{S}_0$, while the other two servers only need to allocate small storage to facilitate query and eviction. Specifically, the server-side storage overhead of S^{3}ORAM is $11N$ data blocks, while the overhead of Octopus ORAM is $(\beta + \frac{1+\alpha}{7})N + \frac{(1+\alpha)s}{2})$, which is no more than $0.3N$ blocks.

- *Client-side Storage Cost*. Octopus ORAM requires larger client-side storage space than S^{3}ORAM, which is similar to the comparison between Path ORAM and Octopus ORAM.

- *Server Computational Cost*. Both constructions require moderate level of computation at the server side. Specifically, S^{3}ORAM requires its servers to execute addition and multiplication of Shamir Secret Sharing operations, while Octopus 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.

7. Related Works

Since the first introduction of oblivious RAM simulator by Goldreich and Ostrovsky [10, 11, 27] for software protection, the idea of ORAM has been extensively explored for protecting a user's access pattern to outsourced data.

Hash-based and Index-based ORAMs. ORAM constructions can be roughly categorized into two classes, hash-based ORAMs and index-based ORAMs, based on the techniques used for look up data blocks.

Hash-based ORAMs [7, 11–14, 18, 19, 28, 41–43, 45] usually organize the server storage as a hierarchy of layers. Each layer contains either a series of buckets [11, 41, 44, 45], or a pair of Cuckoo Hash tables with stash [12–15, 18, 28]. In a bucket ORAM proposed in [11], the server needs to additionally store $(2 \log N - 1)N$ dummy blocks in order to host its client's N real data blocks; its communication cost is $O(\log^3 N)$ blocks per query, with a constant client-side storage. In a bucket ORAM proposed in [41, 44, 45], the server additionally stores at least N dummy blocks and cN bits $(0 < c < 1)$ of Bloom Filters for each layer; its communication cost is $O(\log^2 N \log \log N)$ blocks per query, with a client-side storage of $O(\log^2 N)$ blocks. In a Cuckoo Hash ORAM [12–15, 18, 28], the server stores at least $7N$ dummy data blocks; its communication cost is $O(\log^2 N)$ blocks per query with a constant client-side storage, or $O(\log N)$ blocks per query with a client-side storage of $O(N^c)$ blocks $(0 < c < 1)$.

Index-based ORAMs [3–6, 8, 9, 20–26, 29–31, 34–39, 46] use index table for data lookup. They require the client to either store the index table locally, or outsource it to the server recursively in a way similar to storing their data, at the expense of increased communication cost. Representative index-based ORAMs include Partition ORAM [37], binary tree ORAM (T-ORAM) [34], Path ORAM [38], Gentry's ORAM [9], P-PIR [21] and SCORAM [40]. Partition ORAM organizes its server-side storage as a number of partitions, where each partition is a fully-functional Oblivious RAM. In Partition ORAM, the server side storage needs to store about $3N$ dummy blocks, and incurs a communication cost of $O(\log N)$ blocks per query, with a client-side storage of $O(cN)$ blocks. The other index-based ORAMs organize their server-side storage as a tree, where each node is a bucket storing a certain number of data blocks. For T-ORAM and P-PIR, the server needs to store $(2 \log N - 1)N$ dummy blocks, and incurs a communication cost of $O(\log^2 N)$ blocks per query, with a constant client-side storage. Path ORAM and SCORAM each stores at least $5N$ dummy blocks at the server, and incurs a communication cost of $O(\log N \cdot B)$ blocks per query, with a client-side storage of $O(\log N) \cdot \omega(1)$ blocks, where $\omega(1)$ is a security parameter. At last, Gentry's ORAM requires the server to store at least N dummy blocks, and it achieves a communication cost of $O(\log^2 N \log \log N)$ blocks per query, with a client-side storage of $O(\log^2 N)$ blocks.

ORAMs with Constant Client-Server Bandwidth-blowup. In most of the research works on ORAM design, communication efficiency has been the top priority for optimization. Recently, several ORAMs with constant client-server bandwidth-blowup have been developed. Devadas et al. proposed Onion-ORAM [6], which achieves $O(1)$ bandwidth-blowup by requiring the client and server to interactively run partially Homomorphic Encryption operations; this work however has been considered not practical due to the

high computational cost and the requirement on large data block size. Moataz et al. proposed C-ORAM [25], which claims to also achieve $O(1)$ bandwidth-blowup and meanwhile significantly reduces the computational cost of Onion-ORAM and lifts the requirement of large block size. Moataz et al. also proposed CHF-ORAM [23], which claims to further improve the computational efficiency of C-ORAM by deploying multiple non-colluding servers. Unfortunately, C-ORAM and CHF-ORAM were found flawed [1]. Most recently, Hoang et al. [16] proposed S^3ORAM also based on the deployment of multiple (at least three) non-colluding servers, which achieves $O(1)$ bandwidth-blowup for client-server communication but requires $O(\log N)$ bandwidth-blowup for communication between the servers. Note that, this means the design achieves $O(1)$ data access delay for the client, but the overall communication cost per data request is $O(\log N)$ blocks. Also, to outsource N data blocks, each server in S^3ORAM needs to store $4N$ blocks (and so $12N$ blocks for the 3 servers). Both Onion-ORAM and S^3ORAM only require $O(1)$ local storage at the client.

Uniqueness of Our Work. As summarized in Section 1, our proposed work is unique in that, besides being efficient in client-server communication, it also significantly improves the efficiency of server-side storage.

8. Conclusion and Future Work

We designed and evaluated 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. In the future, we plan to develop further optimizations to reduce the communication cost without sacrificing the server storage efficiency.

References

[1] I. Abraham, C. Fletcher, K. Nayak, B. Pinkas, and L. Ren. Asymptotically Tight Bounds for Composing ORAM with PIR. In *ICAR International Workshop on Public Key Cryptography*, 2017.

[2] V. Bindschaedler, M. Naveed, X. Pan, X. Wang, and Y. Huan. Practicing Oblivious Access on Cloud Straoge: athe Gap, the Fallacy, and the New Way Forward. In *Proc. CCS*, 2015.

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

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

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

[6] S. Devadas, M. van Dijk, C. Fletcher, L. Ren, E. Shi, and D. Wichs. Onion ORAM: A Constant Bandwidth Blowup Oblivious RAM. In *Proc. Theory of Cryptography Conference*, 2015.

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

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

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

[10] O. Goldreich. Towards a theory of software protection and simulationon oblivious RAMs. In *Proc. SIGACT STOC*, 1987.

[11] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAM. *Journal of the ACM*, 43(3):431–473, May 1996.

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

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

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

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

[16] T. Hoang, C. Ozkaptan, A. Yavuz, J. Guajardo, and T. Nguyen. S3ORAM: A Computation-Efficient and Constant Client Bandwidth Blowup ORAM with Shamir Secret Sharing. In *ACM CCS*, 2017.

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

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

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

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

[21] T. Mayberry, E.-O. Blass, and A. H. Chan. Efficient private file retrieval by combining ORAM and PIR. In *Proc. NDSS*, 2014.
[22] T. Mayberry, E.-O. Blass, and G. Noubir. Multi-Client Oblivious RAM secure against Malicious Servers. In *IACR Cryptology ePrint Archive*. International Association for Cryptologic Research, 2015.
[23] S. Moataz, E.-O. Blass, and T. Mayberry. CHf-ORAM: A Constant Communication ORAM without Homomorphic Encryption. In *Proc. IACR Cryptology ePrint Archive*, 2015.
[24] T. Moataz, E.-O. Blass, and G. Noubir. Recursive trees for practical ORAM. In *Proc. FC*, 2015.
[25] T. Moataz, T. Mayberry, and E.-O. Blass. Constant Communication ORAM with Small Blocksize. In *Proc. CCS*, 2015.
[26] T. Moataz, T. Mayberry, E.-O. Blass, and A. H. Chan. Resizable tree-based Oblivious RAM. In *IACR Cryptology ePrint Archive*. International Association for Cryptologic Research, 2014.
[27] R. Ostrovsky. Efficient computation on oblivious RAMs. In *Proc. SIGACT STOC*, 1990.
[28] B. Pinkas and T. Reinman. Oblivious RAM revisited. In *Proc. CRYPTO*, 2010.
[29] L. Ren, C. W. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, and S. Devadas. Ring ORAM: Closing the gap between small and large client storage Oblivious RAM. In *IACR Cryptology ePrint Archive*. International Association for Cryptologic Research, 2014.
[30] L. Ren, C. W. Fletcher, X. Yu, A. Kwon, M. van Dijk, and S. Devadas. Unified Oblivious-RAM: Improving recursive ORAM with locality and pseudorandomness. In *IACR Cryptology ePrint Archive*. International Association for Cryptologic Research, 2014.
[31] L. Ren, C. W. Fletcher, X. Yu, M. van Dijk, and S. Devadas. Integrity verification for path Oblivious-RAM. In *Proc. HPEC*, 2013.
[32] Research and Markets. Cloud storage market - forecasts from 2017 to 2022. In *https://www.researchandmarkets.com/research/lf8wbx/cloud_storage*, 2017.
[33] Z. Saw. LANBench,A Simple LAN / TCP Network Benchmark Utility. In *http://www.zachsaw.com/*, 2017.
[34] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *Proc. ASIACRYPT*, 2011.
[35] E. Stefanov and E. Shi. Multi-Cloud Oblivious Storage. In *Proc. CCS*, 2013.
[36] E. Stefanov and E. Shi. ObliviStore: high performance oblivious cloud storage. In *Proc. S&P*, 2013.
[37] E. Stefanov, E. Shi, and D. Song. Towards practical oblivious RAM. In *Proc. NDSS*, 2011.
[38] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *Proc. CCS*, 2013.
[39] X. Wang, T.-H. H. Chan, and E. Shi. Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. In *Proc. CCS*, 2015.
[40] X. S. Wang, Y. Huang, T.-H. H. Chan, A. Shelat, and E. Shi. SCORAM: oblivious RAM for secure computation. In *Proc. CCS*, 2014.
[41] P. Williams and R. Sion. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *Proc. CCS*, 2008.
[42] P. Williams and R. Sion. Usable PIR. In *Proc. NDSS*, 2008.
[43] P. Williams and R. Sion. Access privacy and correctness on untrusted storage. In *Proc. TISSEC*, 2013.
[44] P. Williams, R. Sion, and A. Tomescu. PrivateFS: a parallel oblivious file system. In *Proc. CCS*, 2012.
[45] P. Williams, R. Sion, and A. Tomescu. Single round access privacy on outsourced storage. In *Proc. CCS*, 2012.
[46] X. Yu, L. Ren, C. W. Fletcher, A. Kwon, M. van Dijk, and S. Devadas. Enhancing Oblivious RAM performance using dynamic prefetching. In *IACR Cryptology ePrint Archive*. International Association for Cryptologic Research, 2014.