Oblivious RAM: A Dissection and Experimental Evaluation

Zhao Chang, Dong Xie, Feifei Li

School of Computing, University of Utah, USA;

{zchang, dongx, lifeifei}@cs.utah.edu

ABSTRACT

Many companies choose the cloud as their data and IT infrastructure platform. The remote access of the data brings the issue of trust. Despite the use of strong encryption schemes, adversaries can still learn valuable information regarding encrypted data by observing the data access patterns. To that end, one can hide the access patterns, which may leak sensitive information, using Oblivious RAMs (ORAMs). Numerous works have proposed different ORAM constructions, but they have never been thoroughly compared against and tested on large databases. There are also no open source implementation of these schemes.

These limitations make it difficult for researchers and practitioners to choose and adopt a suitable ORAM for their applications. To address this issue, we provide a thorough study over several practical ORAM constructions, and implement them under the same library. We perform extensive experiments to provide insights into their performance characteristics with respect to efficiency, scalability, and communication cost.

1. INTRODUCTION

Increasingly, companies choose the cloud as their data and IT infrastructure platform. Many public cloud services are available, such as Amazon cloud and Microsoft Azure. These cloud platforms allow users to upload their data to the cloud and provide cloud computing services over the outsourced data. Many cloud providers also offer cloud-based database systems such as Amazon RDS and Redshift, Azure SQL, and Google Cloud SQL. While utilizing cloud services for building applications is a cost-effective solution, the remote access of the data inevitably brings the issue of trust, and the potential risk of compromising sensitive information is a serious challenge.

A necessary choice for keeping sensitive information private and secure on a cloud is to encrypt the data. To that end, encrypted databases such as Cipherbase [3, 2], CryptDB [29], TrustedDB [7], and Monomi [34], as well as various query execution techniques over encrypted databases [4, 19, 39] have been developed. But the access patterns of users' queries and operations can still leak data privacy and sensitive information, even if the data is encrypted before uploading to the cloud. Consider the following example [28]: if a sequence of queries q_1, q_2, q_3 is always followed by a stock-exchange action, the cloud can learn about the content of these queries, even if both the queries and the data are encrypted. The access patterns may allow the cloud to predict user actions when the similar sequence of queries appears again. Islam *et al.* [20] demonstrate that an attacker can identify as much as 80% of email search queries by observing the access pattern of an encrypted email repository alone. Furthermore, observing access patterns also relates to privacy issues in many applications, such as online social networks. For example, Persona social network [6] can perform operations on encrypted data. The objective is to protect users' sensitive information from the service provider. However, a curious observer can still use access patterns to correlate activities of different users: the observation based on the access patterns of users can contribute to correlating different users who are geographically close or have friendships [24].

To protect against this kind of sensitive information leakage, it is necessary to hide the access patterns of clients' operations in the cloud. One can completely seal a client's access patterns from the cloud by using Oblivious RAMs (ORAMs). ORAM is originally proposed by Goldreich [13] and Ostrovsky [26]. It allows a client to access encrypted data in the cloud while hiding her access patterns. Over around thirty years, various ORAMs have been proposed [13, 14, 28, 16, 18, 30, 21, 32, 33, 35, 23].

There are also efforts on designing oblivious query processing techniques for specific query operations. Li *et al.* [22] propose secure algorithms to compute theta-joins obliviously, i.e., keeping the sensitive access patterns private. Arasu *et al.* [5] present oblivious query processing algorithms for a rich class of database queries involving selections, joins, grouping and aggregation. In spatial databases, access patterns are often considered as sensitive information and need to be protected as part of privacy preserving requirements. Mouratidis *et al.* [25] propose solutions to perform privacy-preserving shortest path computation, while protecting access pattern information. These solutions are based on private information retrieval (PIR) [11, 36]. However, these PIR based solutions need excessive cloud storage and many costly PIR operations [38]. Recently, Xie *et al.* [38] propose ORAM based solutions to perform privacy preserving shortest path computation. In general, ORAM based solutions can provide much better performance and scalability than PIR based solutions [38].

Nevertheless, it is a challenging task for users to choose an appropriate ORAM construction for their applications owing to the following issues. *First*, many of these ORAMs are of only theoretical interests, i.e., the focus of these works is to improve worst-case theoretical bounds: they either hide large constant factors in their analyses or they are difficult to be implemented and used in practice. For example, the amortized computation overhead of

Proceedings of the VLDB Endowment, Vol. 9, No. 12
Copyright 2016 VLDB Endowment 2150-8097/16/08.

Basic-HR [14] is $O(\log^3 N)$ (see Table 1 for our notations), if the AKS sorting network [1] is chosen as the oblivious sort algorithm. But this amortized cost comes with a very large constant factor, since AKS sorting network needs about $6100 \cdot n \log n$ comparisons [28] for n items. In fact, the constant factor in Basic-HR is much larger than 6100, since this oblivious sort operation is performed for many times in the reshuffling step as originally presented in [26]. *Second*, these ORAMs have not been thoroughly compared with each other; and many have never been experimentally tested against large data. For example, Basic-SR and Basic-HR [13, 14] are only theoretically analyzed, and no evaluations are provided. For many ORAMs, there is a huge gap between the amortized computation overhead and the worst-case computation overhead. This makes the comparison among various ORAMs based solely on Big-O notations even more difficult and less meaningful. *Lastly*, high quality open source implementations of ORAM are lacking, and many ORAMs are rather contrived and not easy to be correctly implemented with efficiency and scalability. This presents a significant barrier for researchers and practitioners who want to employ an ORAM in their study.

There is a survey for ORAMs in the literature [10], but it covers only the Basic-SR and Basic-HR ORAMs. Furthermore, it does not provide any experimental evaluation or implementations. To address these issues, for the first time, we have implemented several practical ORAM constructions in the same framework, and evaluated them thoroughly under the same experimental environment. Our main contributions are summarized as follows:

- We provide a comprehensive survey on different ORAM constructions and perform a detailed time, space, and communication complexity analysis.
- We implement these ORAM constructions under the same library framework and optimize the implementations with respect to efficiency, scalability, and communication cost.
- We perform extensive experiments on large data to compare the performance of various ORAM constructions.
- We report insights gained from the comprehensive experimental results. Our study exposes the strength and weakness of different existing ORAMs, and provides guidelines on selecting a suitable construction under different scenarios.

2. PROBLEM DEFINITION

We first formally model the problem of hiding access patterns. The formulation includes a client and a cloud server. The client, who has a small and secured memory, wants to store and later retrieve her data using the large but untrusted cloud storage, while preserving the data privacy. Multiple clients may exist and retrieve the data as long as they are trusted by the client who is the original data owner and follow the same client side protocol.

In this paper, we consider the "honest-but-curious" cloud, which is consistent with existing ORAMs in the literature. To ensure confidentiality, the client needs to store the secret keys of a symmetric encryption scheme. The encryption should be done with a *semantically secure* encryption scheme, and therefore two encrypted copies of the same data block look different [28]. The client should re-encrypt a block before storing it back to the cloud and decrypt a block after retrieving it. Since these encryption/decryption operations are independent of the access patterns of any ORAM, we may omit them while describing different ORAMs.

Data is encrypted, retrieved, and stored in *atomic units* (i.e., blocks), same as in a database system. We must make all blocks of the same size; otherwise, the cloud can easily distinguish these blocks by observing the differences in size. We use N to denote the number of real data blocks in the database. Each block in the cloud or client storage contains B entries (note that the value of B may vary depending on the types of entries in a block, e.g., encrypted record versus stash index values at the client side). Table 1 lists some frequently-used notations in this paper.

N	Number of (real) data blocks in the database
B	Block size in terms of number of entries per block
C	The ratio of the total number of blocks to the number of real data blocks (used in TP-ORAM [32])
R	The background eviction rate, which is proportional to the data access rate (used in TP-ORAM [32])
H	The ratio of the number of blocks in each bucket to $\log N$ (used in BB-ORAM [30])
E	The background eviction rate, which is proportional to the data access rate (used in BB-ORAM [30])
Z	Number of blocks per bucket (used in Path-ORAM [33])

Table 1: Notations.

The objective of the client is to hide the access patterns of her operations from the observation of the cloud. Formally,

DEFINITION 2.1. **Security of ORAM**. *An input $\vec{y}$ of the client is an operation sequence of length M on her database. Suppose $\vec{y} = \{(op_1, id_1, block_1), \cdots, (op_M, id_M, block_M)\}$. Each operation op_i ($1 \le i \le L$) is either a read operation, denoted by read(id_i), which reads the block with the identifier id_i; or a write operation, denoted by write(id_i, $block_i$), which updates the block with the identifier id_i using the new block content $block_i$. The parameter $block_i$ is null for a read operation.*

Given the operation sequence as the input $\vec{y}$, the access pattern $A(\vec{y})$ is the (possibly randomized) sequence of accesses to the cloud storage. An oblivious RAM is considered secure if and only if for any two inputs $\vec{y}$ and $\vec{y'}$ of the client, of the same length, their access patterns $A(\vec{y})$ and $A(\vec{y'})$ are computationally indistinguishable for anyone but the client.

Definition 2.1 implies that we must make different access types (read and write operations) indistinguishable. A standard solution is to always perform read-and-then-write (potentially a dummy write) operations, which is commonly used in existing ORAMs.

Like most ORAMs, Definition 2.1 does not consider privacy leakage through a side-channel attack. For example, Definition 2.1 does not consider the time taken for each operation (timing attack).

3. PRELIMINARIES

Oblivious sort. *Oblivious Sort* is an oblivious algorithm used in Basic-SR [13] and Basic-HR [14] ORAMs. It sorts a set of items by accessing the items in a *fixed, predefined order*. Since such an access pattern is independent of the final order of the items, the corresponding sorting algorithm is an oblivious sort.

Existing works [13, 14] adopt the sorting network to perform oblivious sort. Both Batcher sorting network [8] and AKS sorting network [1] can be used. Assuming there are n items, the time complexity of Batcher sort network is $O(n \log^2 n)$. However, the constant factor in its Big-O notation is very small (approximately 0.5). The time complexity of AKS sorting network is $O(n \log n)$, but the hidden constant factor is as large as 6100 [28]. Thus, in most cases (i.e., $\log n < 6100$), Batcher sorting network performs much better than AKS sorting network. Recently, Goodrich proposes a new randomized Shell sort algorithm [15], which is also an oblivious sort algorithm with $O(n \log n)$ complexity. However, this randomized algorithm can *fail with a small probability*. Therefore, we perform oblivious sorting using Batcher sorting network, which is a deterministic, relatively efficient oblivious sort algorithm.

Oblivious hash. *Oblivious Hash* is an oblivious algorithm used in Basic-HR [14]. It hashes a set of items into an array of hash buckets, while hiding the information regarding the hash function (which bucket each item is hashed into) from the cloud. An oblivious hash is presented in [26], which is well-designed in theory. This oblivious hash algorithm [26] relies on an oblivious sort method. Suppose that the number of hash buckets is n, the size of each hash bucket is m (i.e., each bucket holds at most m items), and the total number of items, M, is not larger than nm. If each oblivious sort operation is performed in $O(n \log n)$, the complexity of the oblivious hash operation (total cost of hashing M items) is $O(mn(\log m + \log n))$. Since we choose the Batcher sorting network as the default oblivious sort method, thus, this complexity is $O(mn(\log^2 m + \log^2 n))$ for hashing $M \leq nm$ items.

Oblivious random permutation. *Oblivious Random Permutation* (in the context of ORAM) refers to a random ordering of T blocks *by the client*. It is the permutation selected at random with uniform probability from all possible permutations. Some ORAMs [13, 38] use random permutation in reshuffling operations. When the client has $\Omega(T)$ storage, we choose the Knuth shuffle [12] to generate the random permutation. The algorithm retrieves T blocks from the cloud server, generates a random permutation of T blocks in $O(T)$ time locally, and places them back to the cloud. When the client only has $o(T)$ storage, the cloud server has to be involved. The idea is to attach each block with a hash value and perform oblivious sort on these blocks according to the order of the hash values. The client retrieves two blocks at a time from the cloud according to the order in the oblivious sort method, compares them locally, and places them back (swaps them if necessary) to the cloud.

In TP-ORAM [32], for permuting blocks within a level in the client, it utilizes a keyed pseudo-random permutation (PRP) function, which plays a similar role to random permutation.

Oblivious storage. The conception of *Oblivious Storage* (OS) is first introduced in [9] as a practical ORAM implementation. Oblivious storage can be viewed as a collection of key-value pairs (i.e., items). The cloud server using an oblivious storage supports the following basic operations:

- *get*(k): return the value of the item with key k.
- *put*(k, v): if the cloud server stores an item with key k, then updates the value with v, else inserts a new item (k, v) into the oblivious storage.
- *getRange*(k_1, k_2): return all items with keys in range $[k_1, k_2]$.
- *delRange*(k_1, k_2): remove all items with keys in range $[k_1, k_2]$.

Oblivious storage relaxes the size of client memory from $O(1)$ in the original ORAM definition to a sub-linear scale (w.r.t. N) [38].

4. OBLIVIOUS RAMS

4.1 Basic Square Root ORAM

Basic Square Root ORAM (Basic-SR) is the first ORAM proposed in the literature [13]. It stores exactly $(N + 2\sqrt{N})$ blocks in the cloud storage. The first N blocks are encrypted database blocks. The extra storage in the cloud (i.e., the extra $2\sqrt{N}$ blocks) consists of two parts. The first $\sqrt{N}$ blocks are *dummy locations*, and the second $\sqrt{N}$ blocks are *shelter locations*. These extra locations are employed to hide the access patterns, in case the same block is read and/or written more than once by the client.

Basic-SR proceeds in rounds, and simulates $\sqrt{N}$ block operations of the client *in a round*. The initialization and rounds of processing in Basic-SR ORAM [13] are shown in Algorithm 1.

In the initialization phase, the first N blocks of the cloud storage store the encrypted data blocks, with identifiers 1 through N, from the client's database. The other $2\sqrt{N}$ blocks (i.e., blocks in dummy and shelter locations) are initialized with randomly generated blocks by the client with identifiers $(N+1)$ through $(N+2\sqrt{N})$ (Line 1). Right after initialization, the block with identifier i (i.e., id = i) locates at the ith location. *Note that block id is encrypted and hidden from the cloud server.*

Algorithm 1: Basic-SR ORAM

```
 1: Initialization: the first $N$ blocks in the cloud are encrypted data
    blocks with identifiers 1 through $N$ from client's database; set the
    other $2\sqrt{N}$ blocks (i.e., blocks in dummy and shelter locations) to
    randomly generated blocks with identifiers $(N + 1)$ to $(N + 2\sqrt{N})$.
 2: Initialize $p := 0$.
 3: while true do
 4:   $p := p + 1$;
 5:   Perform an oblivious random permutation $\pi$ of the blocks in the
      first $(N + \sqrt{N})$ locations (data + dummy blocks), and relocate
      the block in location $i$ $(1 \leq i \leq N + \sqrt{N})$ to location $\pi(i)$.
 6:   for $count := 1$ to $\sqrt{N}$ do
 7:     $j := (p - 1)\sqrt{N} + count$;
 8:     // Simulate the $j$th operation $(op_j, id_j, block_j)$ of the client.
 9:     Client retrieves the blocks in locations $(N + \sqrt{N} + 1)$
        through $(N + 2\sqrt{N})$ (i.e., shelter locations) from the cloud
        and looks for the block with the identifier $id_j$.
10:     if the block with $id_j$ is found in a shelter location $x$ then
11:       Retrieve a dummy block in location $\pi(N + count)$.
12:       Re-encrypt this block and write it back to same location.
13:     else
14:       Set $x := -1$. Retrieve the block $id_j$ in location $\pi(id_j)$.
15:       Encrypt a randomly generated block and write it back in
          location $\pi(id_j)$.
16:     // Client asks the cloud to scan the shelter locations again.
17:     for $i := 1$ to $\sqrt{N}$ do
18:       Read the block at shelter location $(N + \sqrt{N} + i)$.
19:       if $(N + \sqrt{N} + i) = x$ then
20:         if $op_j$ = write then
21:           Write encrypted $block_j$ to location $(N + \sqrt{N} + i)$.
22:         else if $op_j$ = read then
23:           Write the re-encrypted block at location
              $(N + \sqrt{N} + i)$ back to the same location.
24:       else if $i = count$ and $x = -1$ then
25:         if $op_j$ = write then
26:           Write encrypted $block_j$ to location $(N + \sqrt{N} + i)$.
27:         else if $op_j$ = read then
28:           Write the re-encrypted block with $id_j$ (retrieved in
              Line 14) into location $(N + \sqrt{N} + i)$.
29:       else
30:         Write the re-encrypted block at location $(N + \sqrt{N} + i)$
            back to location $(N + \sqrt{N} + i)$.
31:   Perform an oblivious sort on blocks in all $(N + 2\sqrt{N})$ locations,
      where the sorting order is based on the block identifiers.
```

The pth round in Basic-SR is described in the while loop (Line 4-Line 31). First, at the beginning of each round, client performs an *oblivious random permutation* with the cloud as described in Section 3. The blocks in locations 1 through $(N + \sqrt{N})$ are randomly permuted according to a permutation π (Line 5). The Basic-SR ORAM assumes that the client has only $O(1)$ storage, hence, this permutation is performed by interactive oblivious sorting with the cloud. In this process, the client uses a hash function $h : id \rightarrow \mathbb{Z}^+$ to map a block identifier to a random hash value (to be attached to a block and used for oblivious sorting to achieve oblivious random permutation). Note that using a standard cryptographic hash function such as SHA-256, h uses only constant space and client can remember h locally. Since block id is encrypted inside a block, after the oblivious random permutation, cloud loses track of where these $(N + \sqrt{N})$ blocks (data + dummy blocks) locate.

Next, $\sqrt{N}$ operations of the client are processed in this round (Line 6-Line 30). The jth operation $(op_j, id_j, block_j)$ is simulated as follows. The client retrieves the shelter locations (from location $(N + \sqrt{N} + 1)$ to location $(N + 2\sqrt{N})$) and checks whether the block with identifier id_j is in one of these locations (Line 9).

If the block with id_j is found in location x where $x \in [N + \sqrt{N} + 1, N + 2\sqrt{N}]$, the client accesses a dummy block in location $\pi(N + count)$ (Line 10-Line 12, read and write the re-encrypted, same block back), where *count* is an index to keep track of the number of operations performed so far in the current round. Otherwise, the client retrieves the block with identifier id_j in location $\pi(id_j)$ (Line 13-Line 15, read it and write a random block back). Since the permutation function π is not stored in the client (which would require $O(\frac{N+\sqrt{N}}{B})$ space), to access location $\pi(N + count)$ or $\pi(id_j)$, an *interactive binary search* is performed on blocks in locations 1 through $(N + \sqrt{N})$ between client and cloud, according to the hash values produced by h.

Lastly, the client scans through the $\sqrt{N}$ shelter locations again, reads and writes each shelter block in this process (Line 17-Line 30). In particular, if the block id_j has been found in a shelter location and the operation is write, the client writes $block_j$ into the same location. If the block id_j has not been found in any shelter location, and has been retrieved among $(N+\sqrt{N})$ data and dummy locations, the client writes either that block (re-encrypted) or $block_j$ into the shelter location $(N + \sqrt{N} + count)$ for read and write respectively. Otherwise, the client simply writes the same block (re-encrypted) back to the same shelter location.

In a nutshell, for each operation, from the perspective of the cloud, the client needs to scan all shelter locations, reads a seemingly random location from the first $(N + \sqrt{N})$ data and dummy locations, and finally reads and writes every shelter location again. Clearly, the access to any block id_j is oblivious to the cloud.

Finally, at the end of a round (after $\sqrt{N}$ operations), we relocate all $(N + 2\sqrt{N})$ blocks back to their locations in the initialization phase, using an oblivious sort by their identifiers (Line 31).

It is straightforward to see that the cloud storage is $O(N)$ and the client storage is $O(1)$. The computation overhead depends on the choice of the oblivious sort algorithm. The key observation is that an oblivious sort is performed every $\sqrt{N}$ operations, which dominates other costs associated with an operation (such as scanning the shelter locations, whose cost is only $O(\sqrt{N})$).

When an $O(n \log n)$ oblivious sort is chosen, the amortized cost per operation is $O(\sqrt{N} \log N)$ and the worst-case cost per operation (i.e., the last operation in a round that calls the oblivious sort) is $O(N \log N)$. When an $O(n \log^2 n)$ oblivious sort is used, the amortized cost per operation is $O(\sqrt{N} \log^2 N)$ and the worst-case cost per operation is $O(N \log^2 N)$. Since client storage is only $O(1)$ in this case, the number of communication rounds per operation, and the total number of blocks communicated per operation are the same as the number of blocks accessed in the cloud storage per operation. In other words, they are given by the same Big-O expression as that for the time complexity shown above.

Remarks. For simplicity, for all subsequent ORAMs we focus on only *read* operations. As seen in Basic-SR, the only difference for a *write* operation is to write $block_j$ instead of writing back a re-encrypted block that has been read. Furthermore, for each ORAM, since for every operation, each block accessed by the cloud needs to be retrieved and re-encrypted by the client, thus, the query overhead for the cloud in terms of number of blocks accessed, the communication overhead in bytes, and the client's computation cost has the same Big-O complexity, and we refer to them as the *Computation Overhead per operation* of an ORAM.

4.2 Interleave Buffer Shuffle SR-ORAM

Interleave Buffer Shuffle Square Root ORAM (IBS-SR) is proposed in [38]. The oblivious storage scheme is used for building the ORAM. It relaxes the client memory size from $O(1)$ in Basic-SR to $O(\sqrt{N})$. IBS-SR stores $(N + \sqrt{N})$ blocks in the cloud storage. The extra storage in the cloud is similar to dummy locations in Basic-SR. The client buffer can hold $\sqrt{N + \sqrt{N}}$ blocks, which serves similar purpose to the shelter locations on the cloud in Basic-SR. One contribution of IBS-SR is to propose a new oblivious shuffle algorithm, *Interleave Buffer Shuffle* (IBS) (Line 6-Line 17 in Algorithm 2), to optimize the oblivious storage scheme in [17]. During the IBS operation, the Knuth shuffle [12] is performed to generate an oblivious random permutation.

Algorithm 2: IBS-SR ORAM

```
 1: Initialization: choose a hash function $h$; the first $N$ blocks in the
    cloud storage are real data blocks attached with hash values $h(1)$
    through $h(N)$; the other $\sqrt{N}$ blocks are dummy random blocks
    attached with the hash values $h(N + 1)$ through $h(N + \sqrt{N})$.
 2: Initialize $p := 0$.
 3: Let $T := \sqrt{N + \sqrt{N}}$.
 4: while true do
 5:     $p := p + 1$;
 6:     Choose a new hash function $h'$.
 7:     for $i := 1$ to $T$ do
 8:         Retrieve and delete the blocks with the hash values
            $h((i - 1)T + 1)$ through $h(iT)$.
 9:         Update each block's hash value using $h'$.
10:         Perform a random permutation on these $T$ blocks using
            the new hash values.
11:         Insert these $T$ blocks back into the cloud storage.
12:     Choose a new hash function $h''$.
13:     for $i := 1$ to $T$ do
14:         Retrieve and delete the blocks with the hash values
            $h'(0 \cdot T + i), h'(1 \cdot T + i), \cdots, h'((T - 1)T + i)$.
15:         Update each block's hash value using new hash function $h''$.
16:         Perform a random permutation on these $T$ blocks with the
            new hash values.
17:         Insert these $T$ blocks back into the cloud storage.
18:     for $count := 1$ to $\sqrt{N}$ do
19:         $j := (p - 1)\sqrt{N} + count$;
20:         // Simulate the $j$th operation $(op_j, id_j, block_j)$ of the client.
21:         Look for the block with the identifier $id_j$ in the client buffer
            using the hash value $h''(id_j)$.
22:         if the block with identifier $id_j$ is found in client buffer then
23:             Retrieve a dummy block with hash value $h''(N + count)$.
24:         else
25:             Retrieve the block with the hash value $h''(id_j)$ (i.e., with
                identifier $id_j$).
26:     Write all re-encrypted blocks in the client buffer to the cloud
        storage. Clear the client buffer.
```

IBS-SR ORAM [38] is presented in Algorithm 2. Initially, the client chooses a hash function h like SHA-256. The first N blocks of the cloud storage are real data blocks attached with the hash values $h(1)$ through $h(N)$. The other $\sqrt{N}$ blocks are initialized to randomly generated dummy blocks attached with the hash values $h(N + 1)$ through $h(N + \sqrt{N})$ (Line 1). Like Basic-SR, IBS-SR proceeds in passes, and for each *pass*, IBS-SR [38] simulates $\sqrt{N}$ operations of the client. Let T be $\sqrt{N + \sqrt{N}}$ (Line 3). The pth pass is described in the while loop (Line 5-Line 26).

At the beginning of each pass, all $(N + \sqrt{N})$ blocks on the cloud are randomly permuted with the help of the client buffer (Line 6-Line 17). This oblivious random permutation works in two rounds, since the client buffer can only hold T blocks. In the first round, the client retrieves T blocks at a time, randomly permutes each T blocks locally, and maps $(N + \sqrt{N})$ blocks to T different sets over T iterations (Line 6-Line 11). After the first round, each set will have T blocks. In the second round, the client permutes the blocks in each set once more and generates the final permutation on all $(N + \sqrt{N})$ blocks (Line 12-Line 17). To make the blocks before and after re-shuffling indistinguishable for the cloud, a new hash function should be chosen at the beginning of each of the two rounds (Line 6 and Line 12).

Next, $\sqrt{N}$ operations of the client are simulated in the current pass (Line 18-Line 25). The jth ($j = (p-1)\sqrt{N} + count$) operation $(op_j, id_j, block_j)$ is simulated as follows. The client checks whether the block with identifier id_j is in the client buffer using the hash value $h''(id_j)$ (Line 21). If it is found there, the client retrieves a dummy block with the hash value $h''(N + count)$ (Line 22-Line 23). Otherwise, the client retrieves the block with the hash value $h''(id_j)$ (i.e., with identifier id_j) (Line 24-Line 25). Finally, at the end of the current pass, the client writes all re-encrypted blocks in the client buffer to the cloud storage and clears the buffer (Line 26).

For each operation, from the perspective of the cloud, the client reads a seemingly random location. Furthermore, the Interleave Buffer Shuffle (the shuffling performed at the beginning of each pass) is an oblivious operation in the eyes of the cloud. Hence, the access to block id_j for any j is oblivious to the cloud.

Obviously, the cloud storage is $O(N)$ and the client storage is $O(\sqrt{N})$. The amortized cost per operation is $O(\sqrt{N})$, and the worst-case cost per operation is $O(N)$. Since the client storage is $O(\sqrt{N})$ rather than $O(1)$, the amortized number of communication rounds per operation is $O(1)$, and the worst-case number of communication rounds per operation is $O(\sqrt{N})$.

4.3 Basic Hierarchical ORAM

Basic Hierarchical ORAM (Basic-HR) [14] asks the cloud to organize the blocks into a hierarchical structure. For each level, the blocks are stored according to a randomly selected hash function. From top to bottom, each level contains an increasing number of hash buckets. Each hash bucket contains $\log_2 N$ blocks. Suppose there are L levels in the cloud storage. The last level (i.e. level L) has to contain at least N hash buckets.

In the original Basic-HR [14], each level ℓ ($1 \le \ell \le L$) contains 4^ℓ hash buckets. In our implementation, each level ℓ contains 2^ℓ hash buckets. Thus, the storage in level $\ell + 1$ is exactly twice than that in level ℓ ($1 \le \ell < L$). This setting is consistent with all newer hierarchical ORAMs and binary-tree ORAMs (eg. [30, 32, 33]). This change does not affect the complexity analysis. In our implementation, the number of levels L is $\lceil \log_2 N \rceil$. Each level ℓ ($1 \le \ell \le L$) will be (obliviously) reshuffled after every 2^ℓ client operations.

Basic-HR ORAM [14] is presented in Algorithm 3. During initialization, client chooses ℓ different hash functions, one for each level ($2 \le \ell \le L$). N data blocks are hashed into the corresponding hash buckets in level L (i.e., the last and largest level), based on their block identifiers. All other blocks in any hash buckets from all levels are initialized to random dummy blocks.

The jth client operation $(op_j, id_j, block_j)$ is simulated in the while loop (Line 4-Line 19). First, the client retrieves both buckets in level 1 and looks for the block with the identifier id_j (Line 6). If found, the client keeps a copy and writes a dummy block back; otherwise, the client writes back the re-encrypted block. Next, the client checks levels 2 through L (Line 7-Line 11). If the block with the identifier id_j has already been found, the client reads a random bucket in level ℓ (Line 8-Line 9). Otherwise, the client scans the hash bucket that the block id_j might have been hashed into in level ℓ (Line 10-Line 11), based on the hash function of this level and the identifier id_j. If the block is found in that bucket, the client retrieves it and writes a dummy block back into the same location in that bucket; otherwise, every retrieved block will be written back to the same location with a re-encryption of the same block.

After going through all levels, client writes the re-encrypted block

Algorithm 3: Basic-HR ORAM

```
1: **Initialization:** choose a different hash function for each level $\ell$
   ($2 \le \ell \le L$); $N$ data blocks are hashed into the corresponding hash
   buckets in level $L$; set all other blocks in any hash bucket from any
   level to random dummy blocks.
2: Initialize $j := 0$.
3: **while true do**
4:    $j := j + 1$;
5:    // Simulate the $j$th operation $(op_j, id_j, block_j)$ of the client.
6:    Scan both buckets in level 1 and look for the block $id_j$. If
      found, write a dummy block back; otherwise, write the
      re-encrypted block back.
7:    **for** $\ell := 2$ to $L$ **do**
8:       **if** the block $id_j$ has already been found **then**
9:          Scan a random bucket (retrieve all blocks in the bucket)
            in level $\ell$ and write the re-encrypted block back.
10:      **else**
11:         Scan all blocks in the hash bucket that block $id_j$ might be
            hashed into. If it is found in the bucket, write a dummy
            block back; otherwise, write the re-encrypted block back.
12:   **if** $j$ is an odd number **then**
13:      Write the re-encrypted block $id_j$ into 1st bucket in level 1.
14:   **else**
15:      Write the re-encrypted block $id_j$ into 2nd bucket in level 1.
16:   Let $d := \max\{1 \le x < L \mid j \mod 2^x = 0\}$.
17:   **for** $\ell := 1$ to $d$ **do**
18:      Pick a new hash function for level $\ell + 1$.
19:      Shuffle *data blocks*, obliviously to cloud, in level $\ell$ and level
         $\ell + 1$ together into level $\ell + 1$ using the new hash function.
```

id_j back to level 1. If j is an odd number, it is written into the first bucket; otherwise, the second bucket is used. All *data blocks* in level ℓ should be reshuffled after every 2^ℓ client operations, and client can find the levels that need to perform reshuffling based on this constraint after each operation (denoted by d in Line 16). For each such level ℓ, client shuffles the data blocks in level ℓ and level $\ell + 1$ together into level $\ell + 1$ using *oblivious hash* with a new hash function for level $\ell + 1$ (Line 17-Line 19). Since in the default Basic-HR, client only has $O(1)$ storage, these reshuffling operations should be done via the oblivious hash algorithm.

In summary, for each operation, from the perspective of the cloud, the client scans and writes a seemingly random bucket from each level, and writes the re-encrypted block back to level 1. The data blocks in level 1 are obliviously reshuffled into level 2 every 2 operations, and data blocks in level 2 are obliviously reshuffled into level 3 every 4 operations, and so far so forth for each level. The oblivious hash hides the access patterns during reshuffling. Hence, the access to any block id_j is oblivious to the cloud.

The cloud storage is $O(N \log N)$ and the client storage is $O(1)$. The computation overhead depends on the choice of the oblivious sort algorithm (a subroutine called by the oblivious hash algorithm proposed in [26]). When an $O(n \log n)$ oblivious sort is chosen, the amortized computation overhead per operation is $O(\log^3 N)$ and the worst-case overhead per operation is $O(N \log^2 N)$. When an $O(n \log^2 n)$ oblivious sort is chosen, the amortized computation overhead per operation is $O(\log^4 N)$ and the worst-case overhead per operation is $O(N \log^3 N)$. Since the client storage is only $O(1)$, the number of communication rounds is the same with the total number of blocks accessed for an operation.

4.4 TP-ORAM

TP-ORAM [32] is an improved hierarchical ORAM by leveraging *partitioning*. Each partition of TP-ORAM is viewed as an ORAM blackbox, providing a read and write interface, while hiding the access patterns within that partition [32]. Thus, a new partition based ORAM can be constructed by using the framework in [32] and supplying an ORAM construction for each partition.

The original TP-ORAM [32] chooses $P = \sqrt{N}$ partitions. In the cloud storage, each partition contains $L = (\log_2 \sqrt{N}) + 1 = \frac{1}{2}\log_2 N + 1$ levels. Except for the largest level (i.e. level L), each level ℓ contains 2^ℓ blocks. Level L needs to store $(2^L + \epsilon) = (2\sqrt{N} + \epsilon)$ blocks, where ϵ is to accommodate the fact that some partitions may have more blocks than others when the blocks are assigned randomly to the partitions [32]. Thus, each partition holds $(4\sqrt{N} - 2 + \epsilon)$ blocks in total. In [32], the total number of blocks in each partition is set to $4.6\sqrt{N}$. Therefore, the total cloud storage is $4.6N$ blocks (i.e. $C = 4.6$ in Table 1). For each level ℓ, at most half of the blocks are data blocks, and the others are dummy blocks.

The client storage consists of the following components. The first component is the shuffling buffer (or job queue). The shuffling buffer is used for temporarily storing two or more levels in a partition for the reshuffling operation. The size of the shuffling buffer is $O(\sqrt{N})$. The second component is the stash [1]. Stash is a client cache that stores *data blocks* retrieved from the cloud temporarily. In TP-ORAM [32], the stash consists of P groups, and each group corresponds to one partition of the cloud storage. The total stash size is also $O(\sqrt{N})$, which is theoretically proved and empirically demonstrated in [32]. The third component is the position map. For a data block with identifier id_j, the position map keeps track of which partition the block currently resides in (denoted by $position[id_j]$). In TP-ORAM, for each data block, the position map is extended to also record the level number and the location in the level. The total size of the position map is $O(\frac{N}{B})$.

TP-ORAM [32] is presented in Algorithm 4. Initially, the client stash is empty. Each of N data blocks is assigned to an independently selected random partition. Index s is used to identify the next group to be "evicted" in the stash.

The jth client operation $(op_j, id_j, block_j)$ is simulated in the while loop (Line 4-Line 26). First, the client randomly generates a partition number r (Line 6). By checking the position map, the client finds partition p that block id_j currently resides in on the cloud (Line 7). Client sets this block to partition r in his position map (Line 8). Next, client searches for block id_j in the pth group of his stash (i.e., $stash[p]$). If found, the block is read and deleted from $stash[p]$, and a dummy block is retrieved from partition p from the cloud. (Line 9-Line 11). Otherwise, the block is read directly from partition p in the cloud (Line 12-Line 13). The client then adds block id_j into $stash[r]$ (Line 14).

Client performs a piggy-backed eviction to group p in the stash after each operation (Line 16-Line 19); and periodically (based on eviction rate R, Line 21), client performs an eviction to group s in the stash (Line 22-Line 25) and sets s to the next group (Line 26). This is to prevent the stash from building up.

The client evicts blocks in the stash to the cloud. In a piggy-backed eviction, if the client performs an operation on the block that currently resides in partition p, it can piggy-back a write-back to partition p just after the preceding operation. The background evictions take place at a rate (i.e., the background eviction rate, denoted by R in Table 1) proportional to the number of client operations. Background evictions are performed by going through each group of the stash at a fixed rate R (i.e., sequential evictions) in a round-robin fashion. For each group, a data block in the stash is selected at random, deleted from the stash, and written back (re-encrypted) to the corresponding partition in the cloud.

The client also performs shuffling data blocks in a recently accessed partition after every few operations. The idea behind this shuffling operation is similar to that in the hierarchical ORAM.

[1]The term "stash" is adopted in Path-ORAM [33]. The stash plays the same role as the data cache does in TP-ORAM [32].

Algorithm 4: TP-ORAM

```
1:  Initialization: the client stash is empty; each of $N$ data blocks is
    assigned to an independently selected random partition.
2:  Initialize $j := 0$. $s := 1$.
3:  while true do
4:      $j := j + 1$;
5:      // Simulate the $j$th operation $(op_j, id_j, block_j)$ of the client.
6:      Let $r$ be a random integer in the range $[1, P]$.
7:      $p := position[id_j]$;
8:      $position[id_j] := r$;
9:      if block $id_j$ is found in $stash[p]$ then
10:         Read and delete block $id_j$ from $stash[p]$.
11:         Read a dummy block from partition $p$ in the cloud storage.
12:     else
13:         Read block $id_j$ from partition $p$ in the cloud storage.
14:     Add block $id_j$ into $stash[r]$.
15:     // Piggy-backed eviction
16:     if $stash[p]$ is empty then
17:         Write a dummy block to partition $p$ in the cloud storage.
18:     else
19:         Write a block from $stash[p]$ to partition $p$ in the cloud
            storage. Remove the block from $stash[p]$.
20:     // Sequential eviction
21:     if $\lfloor j \cdot R \rfloor - \lfloor (j-1) \cdot R \rfloor = 1$ then
22:         if $stash[s]$ is empty then
23:             Write a dummy block to partition $s$ in the cloud storage.
24:         else
25:             Write a block from $stash[s]$ to partition $s$ in the cloud
                storage. Remove the block from $stash[s]$.
26:         $s := (s \mod P) + 1$;
```

To reduce client storage, we can recursively apply an ORAM to build the position map. That is to say, we can store the position map in another smaller ORAM in the cloud rather than store it in the client. This can be recursively applied until the space cost of the position map in the client becomes $O(1)$. However, this clearly leads to higher computation and communication overheads.

Since each partition of TP-ORAM can be viewed as an ORAM blackbox, Stefanov *et al.* [32] employ a Cuckoo Hashing ORAM [16] (rather than the original partition ORAM) as the blackbox partition ORAM to build their recursive TP-ORAM *for theoretical interest*. Since ensuring security of Cuckoo Hashing ORAMs in practice is non-trivial (refer to Section 5.1), we adopt the original partition ORAM as the blackbox partition ORAM to implement the recursive construction in practice. In our construction, for each block that the client uploads to the cloud in the eviction process, the client needs to update its position map by recursively accessing corresponding blocks in a series of smaller ORAMs. The amortized number of such blocks for each eviction is $O(\log N)$.

TP-ORAM also supports concurrent read and write operations [32], as a way to optimize the worst-case shuffling cost. Note that "concurrent" read and write operations in this context refer to the fact that shuffling is integrated while processing an operation, rather than after every few operations. It does NOT refer to processing "concurrent" client operations. Since it does not improve the amortized cost, and complicates the implementation significantly, we do not adopt this approach in our study. Therefore, we perform extensive experiments and make detailed evaluation on non-concurrent TP-ORAM constructions (both non-recursive and recursive) and omit "concurrent" ones.

In summary, for each operation, the cloud sees that client reads and then writes the block from/to a seemingly random partition. The access pattern within a partition is protected by the ORAM of the partition. The sequential background eviction follows a fixed, predefined order and scans through partitions one by one. Hence, the access patterns are oblivious to the cloud.

For the non-recursive, non-concurrent TP-ORAM, cloud storage is $O(N)$ and client storage is $O(\sqrt{N} + \frac{N}{B})$. The amortized computation overhead per operation is $O(\log N)$ and the worst-case computation overhead per operation is $O(\sqrt{N})$. Since batch accesses to blocks from the same partition in the cloud can be used (see details in [32]), both the amortized and worst-cost number of communication rounds are $O(1)$ per operation. If "concurrent" read and write are used, the worst-case computation overhead per operation can be reduced to $O(\log N)$.

For recursive, non-concurrent TP-ORAM, using Cuckoo Hashing ORAM [16] as the partition ORAM of theoretic interest, client storage is reduced to $O(\sqrt{N})$, but the amortized computation overhead per operation will be as large as $O(\log^2 N/\log B)$ and the number of communication rounds per operation will be $O(\log N/\log B)$ rather than $O(1)$. If "concurrent" read/write is used, the worst-case computation overhead per operation is reduced to $O(\log^2 N/\log B)$.

For the recursive, non-concurrent TP-ORAM, in our construction (i.e., using the original partition ORAM as the blackbox partition ORAM), client storage is reduced to $O(\sqrt{N})$, but the amortized computation overhead per operation will become $O(N^{\log\log N/\log B})$ and the worst-case overhead will be $O(N^{\log N/4\log B + O(1)})$. The number of communication rounds per operation will be $O(N^{\log\log N/\log B})$. If "concurrent" read/write is used, the worst-case computation overhead per operation can be reduced to $O(N^{\log\log N/\log B})$.

4.5 Basic Binary-Tree ORAM

Basic Binary-Tree ORAM (BB-ORAM) is proposed in [30]. BB-ORAM is a binary-tree ORAM that achieves poly-logarithmic amortized and worst-case cost. It requires no oblivious sorting or reshuffling. The cloud storage is treated as a binary tree and each data block is mapped to a leaf node (but not necessarily stored there), selected uniformly at random, in the binary tree of the cloud storage. A block is always placed in some node along the path from the root to the leaf node that it is mapped to in the binary tree. The client uses a position map to keep track of the index of the leaf node that any data block b is currently mapped to.

Note that BB-ORAM has been improved by Path-ORAM [33] (see details in Section 4.6), which uses a similar construction with new optimizations. Thus, we investigate and examine Path-ORAM in details, as the state-of-the-art binary-tree ORAM, rather than presenting the details of BB-ORAM.

The non-recursive ORAM construction requires $O(\frac{N}{B})$ blocks of storage in the client. To reduce this storage cost, one may recursively apply the same ORAM over the index structure in the client, just like the recursive TP-ORAM.

4.6 Path-ORAM

Path-ORAM [33] is an optimized binary-tree ORAM. The cloud storage is treated as a binary tree and each data block is mapped to a leaf node (but not necessarily stored there), selected uniformly at random, in the binary tree of the cloud storage. A block is always placed in some node along the path from the root to the leaf node that it is mapped to in the binary tree.

The cloud stores a binary tree, which is similar to that in BB-ORAM. The only difference is each bucket contains Z blocks (as highlighted in Table 1) rather than $H \log N$ blocks. It suffices to choose a small constant for the value of Z such as $Z = 4$ [33]. The cloud server stores a binary tree of $L = \lceil \log_2 N \rceil$ levels (root is at level 0) with 2^L leaves. Each node of the tree is called a bucket. Each bucket contains Z blocks (as highlighted in Table 1). The bucket will be padded with dummy blocks, if it contains less than Z data blocks. It suffices to choose a small constant for the value of Z such as $Z = 4$ [33]. Let $x \in [0, 2^L - 1]$ denote the xth leaf node in the tree. The xth leaf node u defines a unique path from the root node to u, denoted by $P(x)$. Let $P(x, \ell)$ denote the bucket in level ℓ $(0 \le \ell \le L)$ of path $P(x)$ in the tree.

Algorithm 5: Path-ORAM

```
1: Initialization: client stash $S$ is empty; buckets in cloud contain
   dummy random blocks (except locations with data blocks which
   are oblivious to cloud); for each data block $b$, its position map
   $position[b]$ is initialized to the leaf node index that it is mapped to.
2: Initialize $j := 0$.
3: while true do
4:     $j := j + 1$;
5:     // Simulate the $j$th operation $(op_j, id_j, block_j)$ of the client.
6:     Let $r$ be a random integer in the range $[0, 2^L - 1]$.
7:     $x := position[id_j]$;
8:     $position[id_j] := r$;
9:     Retrieve path $P(x)$; find block $id_j$ in a bucket from $P(x)$; insert
       all data blocks from $P(x)$ to stash $S$.
10:    for $\ell := L$ to 0 do
11:        Let $S'$ be a subset of $S$, where for each block with an
           identifier $s$ in $S'$, $P(x, \ell) = P(position[s], \ell)$.
12:        if $|S'| > Z$ then
13:            Delete some blocks from $S'$ and make $|S'| = Z$.
14:        else
15:            Append dummy random blocks to $S'$ and make $|S'| = Z$.
16:        $S := S - S'$;
17:        Writes the blocks in $S'$ back to the bucket location $P(x, \ell)$.
```

The client storage consists of two components. The first one is a stash (i.e., a buffer to cache accessed data blocks at the client side) with size $O(\log N) \cdot \omega(1)$. It is shown that the stash size can only exceed $O(\log N) \cdot \omega(1)$ with probability at most $N^{-\omega(1)}$ (i.e., negligible in N) [33]. Our experimental results show that after each ORAM read/write operation, the stash always holds no more than 30 blocks when $Z \ge 4$. Client also maintains a position map, which is similar to that in BB-ORAM. For any unstashed data block b, the position map keeps track of the index of the leaf node that b is currently mapped to. The size of the position map is $O(\frac{N}{B})$.

Path-ORAM [33] is presented in Algorithm 5. Initially, the client stash S is empty. The buckets in cloud are initialized to dummy blocks with random bytes (except the locations that data blocks are stored). For each data block b, its position map $position[b]$ is initialized to the leaf node index that it is mapped to.

The jth client operation $(op_j, id_j, block_j)$ is simulated in the while loop (Line 4-Line 17). First, the client generates a random leaf node index r (Line 6). By checking the position map, client finds the leaf node index x, which defines the path $P(x)$ that block id_j currently resides in (Line 7). Next, client assigns block id_j to path $P(r)$ (i.e., to another random path, Line 8). Client retrieves the path $P(x)$ from the cloud (which guarantees to hold block id_j), checks each bucket in every level of $P(x)$, and finds block id_j. In this process, all data blocks found in $P(x)$ are inserted into stash S (Line 9).

Finally, client writes the path $P(x)$ back to the cloud from the bottom level bucket to the top level bucket, which ensures that blocks in the stash can be pushed as deep down into the tree as possible (Line 10-Line 17). For each level ℓ in the path, let S' be a subset of the stash S, where for each data block with an identifier s in S', $P(x, \ell) = P(position[s], \ell)$, i.e., it can be written back to the bucket location $P(x, \ell)$ (Line 11). Client sets the size of stash S' to Z, by deleting some blocks from S' or appending random blocks to S' (Line 12-Line 15). Client then removes all data blocks in S' from the stash S (Line 16) and writes the data blocks in S' back to the bucket location $P(x, \ell)$ (Line 17).

One may reduce client storage by building the position map using a recursive ORAM in a similar fashion as that in TP-ORAM.

In summary, for each operation, from the perspective of the cloud, the client reads and then writes a seemingly random path. Clearly, the access to any block id_j is oblivious to the cloud.

For the non-recursive Path-ORAM, the cloud storage is $O(N)$ and the client storage is $O(\log N) \cdot \omega(1) + O(\frac{N}{B})$. Since no reshuffling operations are performed in Path-ORAM, both the amortized computation overhead and the worst-case computation overhead are $O(\log N)$ per operation. As the path can be retrieved or written back in one round, both the amortized and the worst-cost number of communication rounds are $O(1)$ per operation. For the recursive Path-ORAM, the client storage can be reduced to $O(\log N) \cdot \omega(1)$, but the computation overhead per operation will be as large as $O(\frac{\log^2 N}{\log B})$ and the number of communication rounds per operation will be $O(\frac{\log N}{\log B})$ rather than $O(1)$.

4.7 A Comparison of ORAMs

Table 2 compares these ORAMs in terms of cloud storage, client storage, number of communication rounds per operation, and computation overhead per operation. Recall that for all ORAMs, per operation, number of blocks accessed by the cloud, client's computation cost, and number of blocks communicated have the same Big-O expression for their complexity; thus we use the computation overhead to denote the Big-O complexity of these 3 metrics.

5. OTHER ORAMS

5.1 Cuckoo Hashing ORAM

Cuckoo Hashing is proposed in [27]. Its objective is to ensure constant time in the worst case for each lookup operation and amortized constant time for each insertion operation. Cuckoo Hashing ORAMs are investigated in [28, 16, 18]. However, it has been shown that cuckoo hashing may potentially lead to a severe reduction in security [21]. Kushilevitz *et al.* [21] explicitly construct an adversary that breaks the security of the scheme of [28]. Since ensuring security of such ORAMs in practice is non-trivial, we do not evaluate this ORAM in our study.

5.2 Balanced ORAM

Balanced ORAM (B-ORAM) [21] proposes a novel theoretical ORAM. It uses only $O(1)$ memory in the client, and its amortized computation overhead per operation is $O(\frac{\log^2 N}{\log\log N})$. It can be viewed as a combination of Bucket Hashing ORAM (a variant of Basic-HR [14]) and Cuckoo Hashing ORAM. However, according to [21], the last level in its Bucket Hashing ORAM (i.e., the $(K-1)$th level) needs to store $2^{K-k}\log_2 N$ hash buckets. Based on the settings in [21], $K = 7\log_2\log_2 N$ and $k = \log_2\log_2 N + 1$. Thus, the last level stores $2^{6\log_2\log_2 N - 1}\log_2 N = \frac{1}{2}\log_2^7 N$ hash buckets. Note that common Bucket Hashing ORAM requires that the last level should store no less than N hash buckets. If this is the case, the Cuckoo Hashing ORAM will not be used in B-ORAM. B-ORAM [21] theoretically uses $O(\log^7 N) + O(N) = O(N)$ in its deduction and assumes $\frac{1}{2}\log_2^7 N < N$. However, in practice, the number of data blocks, N, is typically far less than 2^{34}. For example, when the size of each block is 4 KB, $N = 2^{34}$ means 64 TB of data, which is very rare in typical applications. When $N < 2^{34}$, we actually have $\frac{1}{2}\log_2^7 N > N$. For example, when $N = 2^{20}$, $\frac{1}{2}\log_2^7 N$ is 640 million but N is only a little more than 1 million. That is to say, in practice, B-ORAM is essentially equal to Bucket Hashing ORAM (i.e., a variant of Basic-HR). Thus, we do not evaluate the performance of B-ORAM in our experiments.

5.3 Secure Multi-Party Computation

Some recent works explore building an ORAM for secure multi-party computation (SMC) [35, 23]. SMC is a powerful cryptographic primitive that allows multiple parties to perform rich data analytics over their private data, while preserving each party's data privacy [23]. That is to say, all parties can obtain the computation result, but no party can learn the data from another party. However, the cloud database model is clearly different from that of SMC. Therefore, these ORAMs [35, 23] are designed for a different context and we do not evaluate them in our study.

5.4 Additional ORAMs

PrivateFS [37] is an oblivious file system based on a new *parallel* Oblivious RAM mechanism. The objective is to enable access to remote storage and keep both the file content and client access patterns secret. Its major contribution is to *support multiple ORAM clients*. Shroud [24] is a general storage system and functions as a virtual disk, which can hide the access patterns from the cloud servers running it. It achieves this objective by adapting oblivious algorithms to enable parallelization. It focuses on using *many inexpensive coprocessors acting in parallel* to improve request latency. ObliviStore [31] is a high performance, distributed ORAM based cloud data store. It uses an ORAM construction that is similar to TP-ORAM [32]. The major contribution of ObliviStore is to achieve high throughput by making I/O operations asynchronous. It can prevent information leakage not only through access patterns but also through timing of I/O events.

Since our focus is to evaluate classical ORAMs with one cloud server and one client, we omit the details of these ORAMs.

6. EXPERIMENTAL EVALUATION

All experiments were conducted between two machines via 1 Gbps LAN. We use a Ubuntu 14.04 PC with Intel Core i7 CPU (8 cores, 3.20 GHz) and 6 GB main memory as the client. The cloud server is a Ubuntu 14.04 machine with Intel Xeon E5645 CPU (24 cores, 2.40 GHz), 95 GB main memory and 1 TB hard disk.

In our experiments, the cloud server hosts a MongoDB instance as the outsourced cloud database and storage. The cloud server needs to support storing and retrieving data blocks. Therefore, we implemented a MongoDB connector class, which supports the following basic operations on *data blocks inside the MongoDB engine*: insertion, deletion, and update. By utilizing these basic operations on data blocks, the cloud server can support read and write operations from a client using different ORAMs.

All ORAMs are implemented in C++. SHA-256 and AES/CFB from Crypto++ library are adopted as our hash function and encryption/decryption function respectively in all ORAMs. The key length of AES encryption is 128 bits. We set the size of each encrypted block to 4 KB (the same setting is used in [31, 38]).

We perform $3N$ read/write operations for each parameterization of an ORAM. By default, we use uniform random access patterns to perform these reads/writes, since the operation cost of an ORAM is independent of the distribution of access patterns [32].

Default parameter values. The default values for some key parameters are as follows. We set the block size to 4 KB (which is the same as earlier studies in [31, 38]). We set C, the ratio of the total number of blocks to the number of real data blocks in TP-ORAM to 4.6 (the same with the setting in [32]). We set the background eviction rate, R, in TP-ORAM to 0.9. This makes the worst-case stash size in TP-ORAM to be about $\sqrt{N}$ blocks (shown both in our experiments and in Figure 8 of [32]). We set H, the ratio of the number of blocks in each bucket to $\log N$ in BB-ORAM [30], to 2. We set E, the background eviction rate in BB-ORAM [30], to 2 as well according to the asymptotic analysis in [30]. We set Z, the number of blocks in each bucket in Path-ORAM [33], to 4.

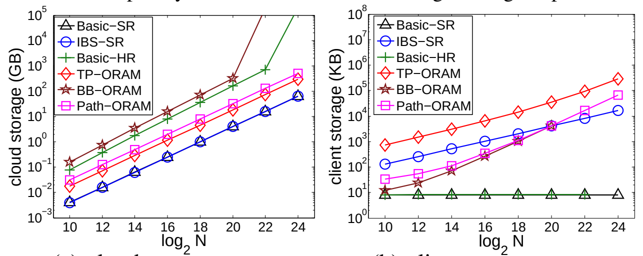
6.1 Cloud and Client Storage Costs

Figure 1(a) shows the cloud storage cost of the six ORAMs, as database size N increases from 2^{10} blocks to 2^{24} blocks (i.e., from 4 MB to 64 GB). BB-ORAM and Basic-HR has much larger cloud

ORAM Construction		Computation Overhead [a]		Cloud Storage	Communication Round		Client Storage
		Amortized	Worst-Case		Amortized	Worst-Case	
Basic-SR [13]	$O(n \log n)$ Oblivious Sort	$O(\sqrt{N} \log N)$	$O(N \log N)$	$O(N)$	$O(\sqrt{N} \log N)$	$O(N \log N)$	$O(1)$
	$O(n \log^2 n)$ Oblivious Sort	$O(\sqrt{N} \log^2 N)$	$O(N \log^2 N)$	$O(N)$	$O(\sqrt{N} \log^2 N)$	$O(N \log^2 N)$	$O(1)$
IBS-SR [38]		$O(\sqrt{N})$	$O(N)$	$O(N)$	$O(1)$	$O(\sqrt{N})$	$O(\sqrt{N})$
Basic-HR [14]	$O(n \log n)$ Oblivious Sort	$O(\log^3 N)$	$O(N \log^2 N)$	$O(N \log N)$	$O(\log^3 N)$	$O(N \log^2 N)$	$O(1)$ [b]
	$O(n \log^2 n)$ Oblivious Sort	$O(\log^4 N)$	$O(N \log^3 N)$	$O(N \log N)$	$O(\log^4 N)$	$O(N \log^3 N)$	$O(1)$
BB-ORAM [30]	Non-Recursive	$O(\log^2 N)$	$O(\log^2 N)$	$O(N \log N)$	$O(\log^2 N)$	$O(\log^2 N)$	$O(\frac{N}{B})$
	Recursive	$O(\log^3 N)$	$O(\log^3 N)$	$O(N \log N)$	$O(\log^3 N)$	$O(\log^3 N)$	$O(1)$
TP-ORAM [32]	Non-Recursive, Non-Concurrent	$O(\log N)$	$O(\sqrt{N})$	$O(N)$	$O(1)$	$O(1)$	$O(\sqrt{N} + \frac{N}{B})$
	Non-Recursive, Concurrent	$O(\log N)$	$O(\log N)$	$O(N)$	$O(1)$	$O(1)$	$O(\sqrt{N} + \frac{N}{B})$
	Recursive, Non-Concurrent [c]	$O(\frac{\log^2 N}{\log B})$	$O(\sqrt{N})$	$O(N)$	$O(\frac{\log N}{\log B})$	$O(\frac{\log N}{\log B})$	$O(\sqrt{N})$
	Recursive, Concurrent [c]	$O(\frac{\log^2 N}{\log B})$	$O(\frac{\log^2 N}{\log B})$	$O(N)$	$O(\frac{\log N}{\log B})$	$O(\frac{\log N}{\log B})$	$O(\sqrt{N})$
	Recursive, Non-Concurrent [d]	$O(N^{\frac{\log \log N}{\log B}})$	$O(N^{\frac{\log N}{4 \log B}+O(1)})$	$O(N)$	$O(N^{\frac{\log \log N}{\log B}})$	$O(N^{\frac{\log \log N}{\log B}})$	$O(\sqrt{N})$
	Recursive, Concurrent [d]	$O(N^{\frac{\log \log N}{\log B}})$	$O(N^{\frac{\log \log N}{\log B}})$	$O(N)$	$O(N^{\frac{\log \log N}{\log B}})$	$O(N^{\frac{\log \log N}{\log B}})$	$O(\sqrt{N})$
Path-ORAM [33]	Non-Recursive	$O(\log N)$	$O(\log N)$	$O(N)$	$O(1)$	$O(1)$	$O(\log N) \cdot \omega(1) + O(\frac{N}{B})$
	Recursive	$O(\frac{\log^2 N}{\log B})$	$O(\frac{\log^2 N}{\log B})$	$O(N)$	$O(\frac{\log N}{\log B})$	$O(\frac{\log N}{\log B})$	$O(\log N) \cdot \omega(1)$

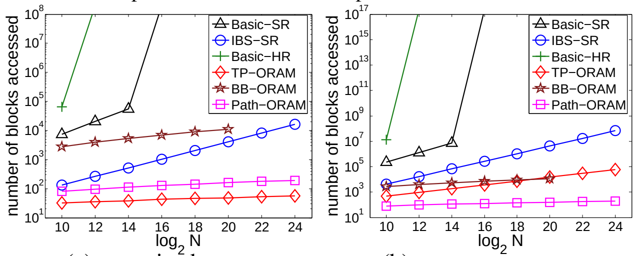
Table 2: Comparison of different ORAMs' performance.

[a]For each operation of the client, the number of blocks retrieved/stored in the cloud, the total communication overhead in bytes, the cost of encryption/decryption in the client, and the total running time in the client have the same Big-O complexity. All of them are shown as the Computation Overhead.
[b]In fact, for each of $\log_2 N$ levels in Basic-HR [14], the client needs to store a hash function. Thus, it needs extra client storage to save $O(\log N)$ hash functions. However, in the practical setting, this cost is much less than the size of a constant number of blocks.
[c] The complexity of recursive TP-ORAM using Cuckoo Hashing ORAM [16] as the partition ORAM of theoretical interest.
[d] The complexity of recursive TP-ORAM using the original partition ORAM as the blackbox partition ORAM in our implementation.



(a) cloud storage cost. (b) client storage cost.

Figure 1: Cloud and client storage costs.



(a) amortized cost. (b) worst-case cost.

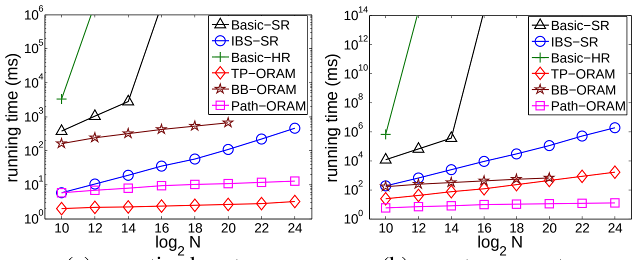
Figure 2: Number of blocks accessed per operation in cloud.

storage than other constructions because of $O(N \log N)$ cloud storage overhead. The cloud storage of Basic-HR reaches 704 GB for a 16 GB database. Although the cloud storage cost of the other ORAMs is the same in theory as $O(N)$, Path-ORAM and TP-ORAM have larger constant factors (8X and 4.6X when N is sufficiently large) than Basic-SR and IBS-SR, as reflected in Figure 1(a). Basic-SR and IBS-SR show the smallest cloud storage overhead, using just a little more storage than the database size.

Figure 1(b) shows the client storage cost. For TP-ORAM and Path-ORAM, since the stash size may vary, we always report their worst-case client storage in our studies. But the impact of the stash size is rather limited: after each ORAM read/write operation, the stash size is only a small fraction of the client storage. Basic-SR and Basic-HR use around 10 KB (a little more than two blocks) client memory, since they need only $O(1)$ client storage. The client storage of IBS-SR is around $\sqrt{N}$ blocks, confirming its theoretical bound. Path-ORAM shows a little less client storage cost than IBS-SR when N is small and a little larger client storage cost when N becomes larger. When N is small, the $O(\log N)$ *data blocks* in an accessed path dominate the client storage of Path-ORAM. When N is large, the $O(\frac{N}{B})$ *blocks* in the position map dominate the client storage of Path-ORAM. BB-ORAM employs a little less client storage than Path-ORAM, since client communicates with the cloud block by block and does not need the stash. Lastly, TP-ORAM has the largest client storage, which uses roughly 282 MB client storage when the database size reaches 64 GB.

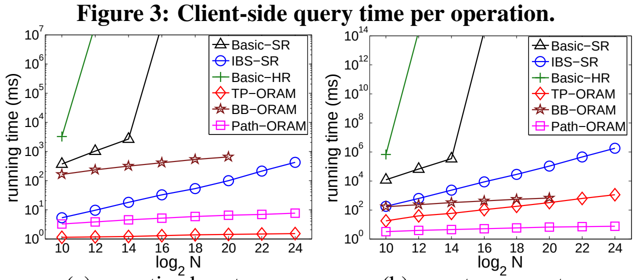
6.2 Query Performance in the Cloud

Since in all ORAMs, there are hardly any expensive computation involved on the cloud and the query cost in the cloud is dominated by the number of blocks the cloud server needs to access. Thus, to study the query performance in the cloud, we report the number of blocks accessed in each client operation for different ORAMs in Figure 2. We show both the amortized cost per operation and the worst-case cost per operation over $3N$ read/write operations (as described earlier in our experimental setup), when N varies from 2^{10} to 2^{24}. Clearly, TP-ORAM has achieved the best amortized cost and Path-ORAM has demonstrated good amortized cost and the best worst-case cost.



(a) amortized cost. (b) worst-case cost.

Figure 3: Client-side query time per operation.



(a) amortized cost. (b) worst-case cost.

Figure 4: Cost of encryption/decryption per operation.

Both TP-ORAM and Path-ORAM have excellent scalability, in terms of both amortized and worst-case costs, as database size increases. When database size increases from 2^{10} to 2^{24} blocks, the average number of blocks accessed per operation only increases from 32 to 56 blocks for TP-ORAM, and increases from 88 to 200 blocks for Path-ORAM. Path-ORAM does have better worst-case cost compared with TP-ORAM. Its worst-case cost is the same with its amortized cost, while TP-ORAM needs to access almost 60,000 blocks per operation in the worst case when $N = 2^{24}$. TP-ORAM accesses more blocks than Path-ORAM in the worst case due to its costly shuffling operation.

In contrast, IBS-SR shows a sharp increase in cost. For a database with 2^{24} blocks, it needs to access more than 16,000 blocks per operation on average, and in the worst case it has to retrieve nearly 6.7×10^7 blocks for an operation. The results confirm the theoretical analysis earlier that IBS-SR accesses more blocks in the cloud, since its amortized cost is $O(\sqrt{N})$ compared with $O(\log N)$ in the case of TP-ORAM and Path-ORAM. BB-ORAM accesses more blocks per operation than IBS-SR when N is small. But BB-ORAM performs much better than IBS-SR in the worst case, since its worst-case cost is the same with its amortized cost.

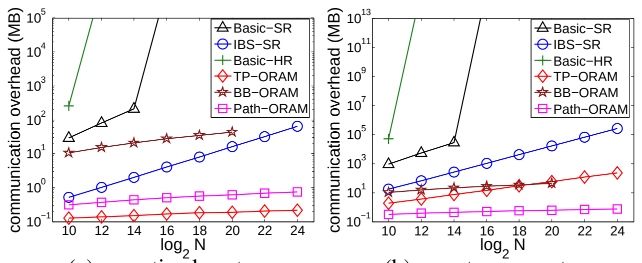
Lastly, both Basic-SR and Basic-HR ORAMs perform poorly and lead to much more expensive cloud overhead.

6.3 Query Cost for the Client

We next study the query cost at the client side. We report the *running time* per operation for each ORAM, in terms of both amortized and worst-case costs in Figure 3. Since each block accessed in the cloud storage needs to be decrypted and re-encrypted by the client, the running time at the client side is roughly linear to the number of blocks an ORAM has to access in the cloud.

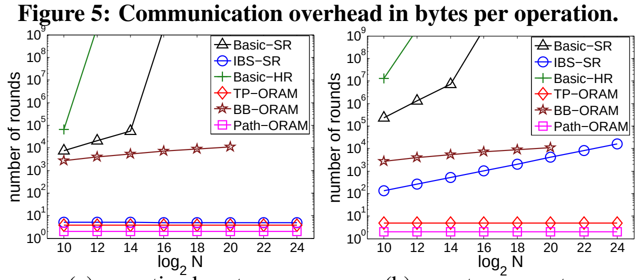
The cost of encryption/decryption operations paid by the client for each operation is shown in Figure 4. Comparing the results in Figure 3 and those in Figure 4, they confirm that encryption/decryption operations dominate the computation cost in the client.

That said, TP-ORAM shows the smallest computation cost for the client on average, which takes only a few milliseconds per operation on average and increases very slowly as database size increases. Path-ORAM also has small overhead for the client and takes about 10 milliseconds per operation on average. It does have smaller worst-case overhead compared with TP-ORAM (which can reach more than 1,000 milliseconds per operation).



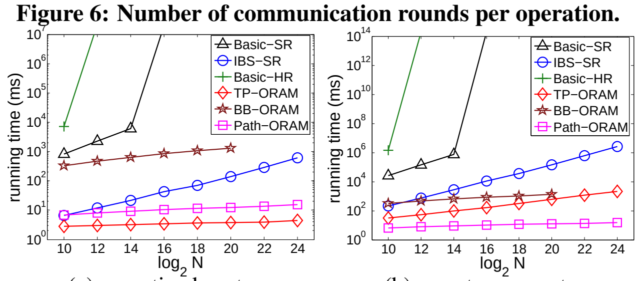
(a) amortized cost. (b) worst-case cost.

Figure 5: Communication overhead in bytes per operation.



(a) amortized cost. (b) worst-case cost.

Figure 6: Number of communication rounds per operation.



(a) amortized cost. (b) worst-case cost.

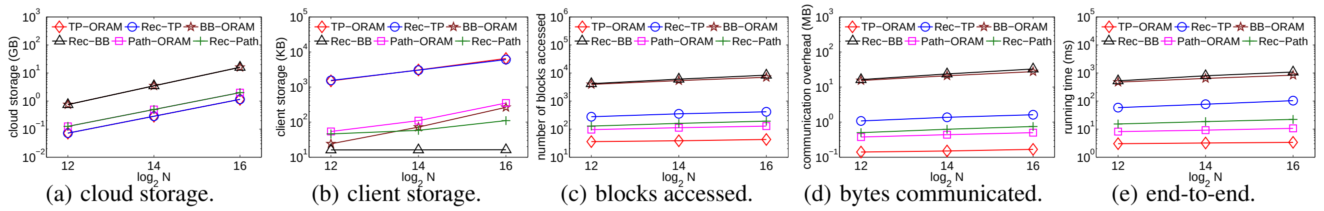
Figure 7: End-to-end running time per operation.

6.4 Communication and End-to-End Cost

We measure the communication overhead in terms of both total bytes transmitted and the rounds of communication needed.

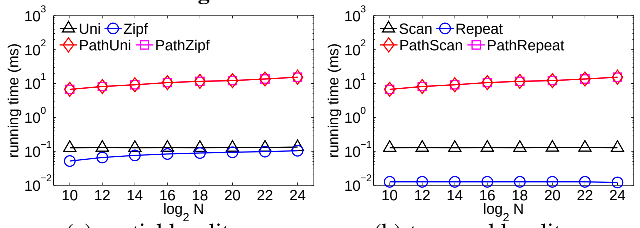
The total bytes communicated between client and cloud, per operation, is shown in Figure 5. Since all accessed blocks in the cloud need to be retrieved by the client, this result is consistent with that in Figure 2. The number of communication rounds per operation is shown in Figure 6. Path-ORAM only needs two communication rounds (i.e. retrieving and storing the blocks along the accessed path) for each operation. TP-ORAM only needs a constant number of communication rounds, since batch read/write operations can be performed while retrieving/storing levels of blocks. IBS-SR needs $O(\sqrt{N})$ rounds of communication during the shuffling phase. However, in the amortized case, it only needs a constant number of communication rounds, since the communication rounds in the shuffling operation are amortized into $\sqrt{N}$ operations. BB-ORAM needs many more communication rounds than IBS-SR in the amortized case, since its client communicates with the cloud block by block. Basic-SR and Basic-HR need many more communication rounds, due to their $O(1)$ client storage requirements.

We also measure the end-to-end running time for each operation of these ORAMs, where the results are shown in Figure 7. The measurement starts when client submits a query operation and stops when client gets the final query response. Given our findings so far, it is not surprising to note that TP-ORAM shows the best amortized cost per operation, and Path-ORAM has the best worst-case performance. Note that in our setting, the bandwidth between the client and the cloud is 1 Gbps, and Crypto++ 5.6.0 Benchmarks show that the rate of encryption/decryption is 108 MB/Second using AES/CFB (128-bit key). Hence, the bottleneck of the end-to-end running time lies in client computation.



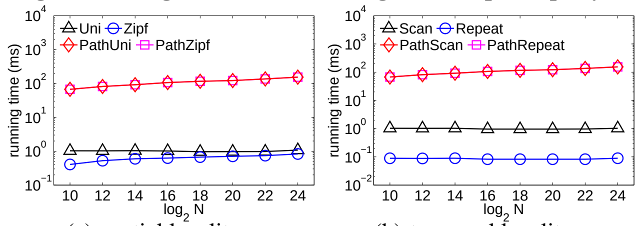
(a) cloud storage. (b) client storage. (c) blocks accessed. (d) bytes communicated. (e) end-to-end.

Figure 8: Recursive ORAMs vs. non-recursive ORAMs: amortized cost used for (c), (d), and (e).



(a) spatial locality. (b) temporal locality.

Figure 9: Using ORAM vs. not using ORAM (point query).



(a) spatial locality. (b) temporal locality.

Figure 10: Using ORAM vs. not using ORAM (range query).

6.5 Recursive ORAMs

For TP-ORAM, BB-ORAM and Path-ORAM, to reduce client storage, we can recursively apply them to build their position map, i.e., we can store the position map in another smaller ORAM in the cloud rather than storing it in the client. It can be recursively applied until the position map size in the client becomes $O(1)$.

Figure 8 shows the comparison between recursive ORAMs and non-recursive ORAMs. As Figure 8(a) shows, the cloud storage of these recursive ORAMs is nearly the same with that of their corresponding non-recursive ones. The client storage has reduced dramatically for recursive BB-ORAM compared to non-recursive BB-ORAM, due to $O(1)$ client storage for the recursive BB-ORAM. As for TP-ORAM and Path-ORAM, such reduction is limited, since the position map does not dominate the client storage when N is not large enough. But recursive constructions lead to the increase of all other costs including number of blocks accessed by the cloud, communication overhead, and the overall end-to-end running time. This overhead is significant for TP-ORAM. In the amortized case, for every $O(\log N)$ blocks that the client uploads to the cloud in each eviction, the client has to update its position map by recursively accessing corresponding blocks in a series of smaller ORAMs.

6.6 Using ORAM vs. Not Using ORAM

We made a comparison between queries using ORAM and that of not using ORAM. For queries without ORAM, the client re-encrypts a block before directly storing it back to the cloud, and decrypts a block after directly retrieving it, but does not hide any access patterns. We investigated both point and range queries. A point query is to read/write block id_j. A range query with length ℓ is to read/write blocks id_j through $id_j + \ell - 1$. By default, $\ell = 10$.

We vary both *query distribution* and *query permutation*. In other words, we vary the distribution and the permutation of block identifiers id_j in a query workload. We perform $10N$ read/write operations for each case. To study spatial locality, we consider uniform distribution (Uni) and the distribution obeying Zipf's law with skewness parameter $s = 1$ (Zipf). To study temporal locality, we consider scanning all blocks pass by pass (Scan) and repeatedly accessing each block for ten times (Repeat) under uniform distribution. We report the performance comparison using Path-ORAM (i.e., PathUni, PathZipf, PathScan, and PathRepeat), which has best worst-case cost per operation among all ORAMs, and that of not using ORAM (i.e., Uni, Zipf, Scan, and Repeat).

Figure 9 shows the average end-to-end query time per operation regarding point queries. As expected, query distribution and query permutation have no influence on the performance of Path-ORAM. However, for queries without using ORAM, the query workload with skewed query distribution (i.e., Zipf) clearly leads to better performance (than Uni). Furthermore, "Repeat" can perform nearly 10X faster than "Scan" in the case of not using ORAM, due to much higher cache hit rates. But they show no difference for queries with ORAM. Figure 10 shows similar trends for range queries.

6.7 Remarks

Our results indicate that Path-ORAM and TP-ORAM have the best overall performance with respect to all metrics. TP-ORAM typically has the best amortized cost per operation, while Path-ORAM has much smaller variance in terms of cost per operation, and hence, has the best worst-case cost per operation. IBS-SR has acceptable performance, and achieves smaller cloud storage overhead than Path-ORAM and TP-ORAM (by almost one order of magnitude), hence, can be used when reducing cloud storage overhead is a significant concern. Basic-SR and Basic-HR have poor performance, but they require only $O(1)$ client storage. BB-ORAM shows acceptable performance and recursive BB-ORAM only needs $O(1)$ client storage, and can be used for relatively small databases and when client device or computation environment is extremely tight on memory space (e.g., a sensor).

We expect IBS-SR, TP-ORAM, and Path-ORAM will be used for most applications. Figures 11, 12, and 13 give further insights into their performance by tuning their key parameters.

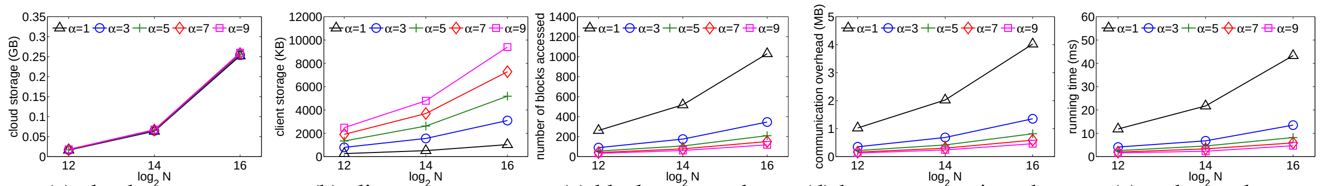
7. CONCLUSION AND FUTURE WORK

This paper provides a comprehensive and detailed analysis on a wide spectrum of existing ORAMs. Despite the strong privacy guarantee, there are still some major limitations while using ORAM to build an encrypted database. First, using an ORAM harms the performance of the database, since it destroys any locality of references and also adds operation overheads due to shuffling and other procedures needed to hide access patterns. Second, ORAM only supports read and write operations. Thus, using ORAM to answer more complex queries, such as joins, may lead to large overhead, which needs new techniques to be developed to optimize such operations. Last but not least, how to support concurrency in an efficient and scalable manner using ORAM is still a major challenge.

Lastly, note that our ORAM library is released as an open source library, named SEAL-ORAM, on GitHub.

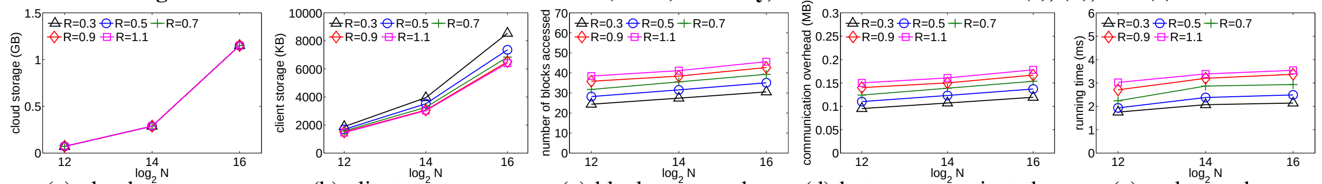
8. ACKNOWLEDGMENT

We appreciate the comments from the anonymous reviewers. Zhao Chang, Dong Xie and Feifei Li were supported in part by NSF grants 1514520 and 1200792.



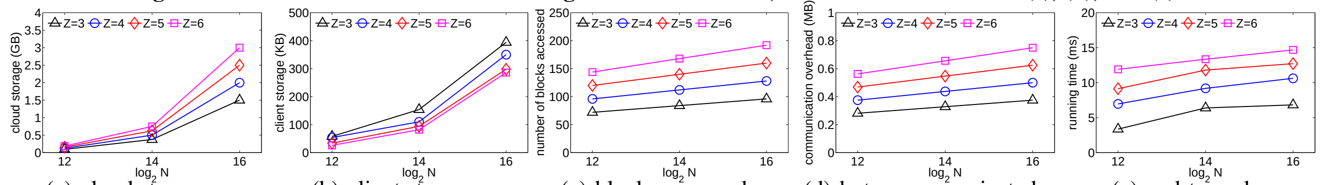
(a) cloud storage. (b) client storage. (c) blocks accessed. (d) bytes communicated. (e) end-to-end.

Figure 11: Effect of α in IBS-SR: client uses $O(\alpha\sqrt{N})$ memory, amortized cost used for (c), (d), and (e).



(a) cloud storage. (b) client storage. (c) blocks accessed. (d) bytes communicated. (e) end-to-end.

Figure 12: Effect of R in TP-ORAM: background eviction rate, amortized cost used for (c), (d), and (e).



(a) cloud storage. (b) client storage. (c) blocks accessed. (d) bytes communicated. (e) end-to-end.

Figure 13: Effect of Z in Path-ORAM: number of blocks in each bucket, amortized cost used for (c), (d), and (e).

9. REFERENCES

[1] M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n \log n)$ sorting network. In *STOC*, pages 1–9, 1983.

[2] A. Arasu, S. Blanas, K. Eguro, M. Joglekar, R. Kaushik, D. Kossmann, R. Ramamurthy, P. Upadhyaya, and R. Venkatesan. Secure database-as-a-service with cipherbase. In *SIGMOD*, 2013.

[3] A. Arasu, K. Eguro, M. Joglekar, R. Kaushik, D. Kossmann, and R. Ramamurthy. Transaction processing on confidential data using cipherbase. In *ICDE*, pages 435–446, 2015.

[4] A. Arasu, K. Eguro, R. Kaushik, and R. Ramamurthy. Querying encrypted data. In *SIGMOD*, pages 1259–1261, 2014.

[5] A. Arasu and R. Kaushik. Oblivious query processing. In *ICDT*, pages 26–37, 2014.

[6] R. Baden, A. Bender, N. Spring, B. Bhattacharjee, and D. Starin. Persona: An online social network with user-defined privacy. In *SIGCOMM*, pages 135–146, 2009.

[7] S. Bajaj and R. Sion. Trusteddb: A trusted hardware-based database with privacy and data confidentiality. *TKDE*, 26(3):752–765, 2014.

[8] K. E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*, pages 370–314, 1968.

[9] D. Boneh, D. Mazieres, and R. A. Popa. Remote oblivious storage: Making oblivious RAM practical. In *MIT CS Technical Report*, 2011.

[10] E. E. Chapman. A survey and analysis of solutions to the oblivious memory access problem. In *Tech Report, Portland State*, 2012.

[11] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan. Private information retrieval. *J. ACM*, 45(6):965–981, 1998.

[12] R. Durstenfeld. Algorithm 235: Random permutation. *Communications of the ACM*, 7(7):420, 1964.

[13] O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *STOC*, pages 182–194, 1987.

[14] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3), 1996.

[15] M. T. Goodrich. Randomized Shellsort: A simple oblivious sorting algorithm. In *SODA*, pages 1262–1277, 2010.

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

[17] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Practical oblivious storage. In *CODASPY*, 2012.

[18] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *SODA*, pages 157–167, 2012.

[19] H. Hacigümüs, B. R. Iyer, C. Li, and S. Mehrotra. Executing SQL over encrypted data in the database-service-provider model. In *SIGMOD*, pages 216–227, 2002.

[20] M. S. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS*, 2012.

[21] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *SODA*, pages 143–156, 2012.

[22] Y. Li and M. Chen. Privacy preserving joins. In *ICDE*, 2008.

[23] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi. ObliVM: A programming framework for secure computation. In *S&P*, 2015.

[24] J. R. Lorch, B. Parno, J. W. Mickens, M. Raykova, and J. Schiffman. Shroud: Ensuring private access to large-scale data in the data center. In *FAST*, pages 199–214, 2013.

[25] K. Mouratidis and M. L. Yiu. Shortest path computation with no information leakage. *PVLDB*, 5(8):692–703, 2012.

[26] R. Ostrovsky. Efficient computation on oblivious RAMs. In *STOC*, pages 514–523, 1990.

[27] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[28] B. Pinkas and T. Reinman. Oblivious RAM revisited. In *CRYPTO*, pages 502–519, 2010.

[29] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *SOSP*, pages 85–100, 2011.

[30] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *ASIACRYPT*, pages 197–214, 2011.

[31] E. Stefanov and E. Shi. ObliviStore: High performance oblivious cloud storage. In *S&P*, pages 253–267, 2013.

[32] E. Stefanov, E. Shi, and D. X. Song. Towards practical oblivious RAM. In *NDSS*, 2012.

[33] E. Stefanov, M. van Dijk, E. Shi, C. W. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: An extremely simple oblivious RAM protocol. In *CCS*, pages 299–310, 2013.

[34] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing analytical queries over encrypted data. *PVLDB*, 6(5):289–300, 2013.

[35] X. S. Wang, Y. Huang, T.-H. H. Chan, A. Shelat, and E. Shi. SCORAM: Oblivious RAM for secure computation. In *CCS*, 2014.

[36] P. Williams and R. Sion. Usable PIR. In *NDSS*, 2008.

[37] P. Williams, R. Sion, and A. Tomescu. PrivateFS: A parallel oblivious file system. In *CCS*, pages 977–988, 2012.

[38] D. Xie, G. Li, B. Yao, X. Wei, X. Xiao, Y. Gao, and M. Guo. Practical private shortest path computation based on oblivious storage. In *ICDE*, 2016.

[39] B. Yao, F. Li, and X. Xiao. Secure nearest neighbor revisited. In *ICDE*, pages 733–744, 2013.