ThinORAM: Towards Practical Oblivious Data Access in Fog Computing Environment

Yanyu Huang, Bo Li, Zheli Liu*, Jin Li, Siu-Ming Yiu, Thar Baker, Brij B. Gupta

Abstract—Oblivious RAM (ORAM) is important to applications that require hiding of access patterns. However, most of the existing implementations of ORAM are very expensive, which are infeasible to be deployed in lightweight devices, like the gateway devices as fog nodes. In this paper, we focus on how to apply the expensive ORAM to protect the access pattern of lightweight devices and propose an ORAM scheme supporting thin-client, called "ThinORAM", under non-colluding clouds. Our proposed scheme removes complicated computations in the client side and requires only $O(1)$ communication cost with reasonable response time. We further show how to securely deploy ThinORAM in the fog computing environment to achieve oblivious data access to minimum client cost. Experiments show that our scheme can eliminate most of the client storage and reduce the cloud-cloud bandwidth by 2×, with 2× faster response time, when compared to the best scheme that aims at reducing client side overheads.

Index Terms—Data privacy, oblivious data access, Internet of Things, fog computing.

1 Introduction

Fog computing is designed as an extension of cloud computing. It deploys the fog nodes (micro data centers or high-performance data analytic machines) in network's edge in order to gain real-time insights from the data collected or to promote data thinning at the edge, by dramatically reducing the amount of data that needs to be transmitted to the cloud center. One typical use case is the smart traffic light system, which can change its signals based on surveillance of incoming traffic to prevent accidents or reduce congestion. Without having to transmit unnecessary data to the cloud, analytics at the edge can cut cost and achieve real-time analysis.

In general, fog computing framework [1] has three layers: terminal layer, fog layer and cloud layer. Figure 1 illustrates a typical cloud-based service architecture [2] supporting fog computing. IoT devices (terminals) [3], [4], [5], [8], [9], [10] as weak computing equipment continuously generate real-time data. The fog nodes are responsible for storing, processing and analyzing these data. They usually make some real-time analysis immediately based on the collected data. But in many cases, they will also upload some necessary data or intermediate results to the cloud for further analysis or collaborative computing. When the fog nodes require these historical data or intermediate results to participate in the computations or decisions, the fog nodes access them in the cloud server that provides certain services, such as query services.

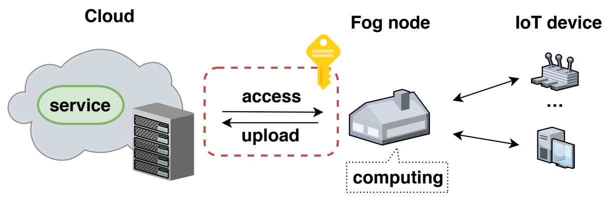


Fig. 1. Cloud-based service architecture supporting fog computing.

1.1 The necessity of oblivious data access

Data privacy remains a major issue in the fog computing environment because the IoT devices always collect personal information like locations, the number of heartbeats, readings of smart meters, and so on. Some studies [6], [7] indicate that these sensitive data should be encrypted during uploading and downloading to the cloud. Considering that IoT devices are resource-constrained and lack the ability to perform encryption operations, Alrawais et al. [12] pointed out that fog nodes can act as cryptographic computation proxies for IoT devices. In other words, fog computing is beneficial for solving many security and privacy issues of IoT, and massive schemes [6], [7] have been proposed.

Unfortunately, more and more researchers realize that just encryption alone is not enough to protect data privacy [16], [17]. Fog devices as the edge of the Internet obviously also face many security and privacy threats [13], [14], [15]. Some fatal attributes of access patterns, e.g., *where*, *when* and *how often* fog devices accessing the encrypted data, will leak sensitive information. In smart traffic system [11], the privacy data such as location information of the vehicle must be protected. Although the location data is encrypted and stored in the cloud, if the storage location is fixed, the cloud server can identify whether it is the operation of the same vehicle by whether or not access to the same

- *Yanyu Huang, Bo Li and Zheli Liu are with the College of Cyber Science, College of Computer Science, Tianjin Key Laboratory of Network and Data Security Technology, Nankai University, China. E-mail: onlyerir@163.com, nankailibo@163.com, liuzheli@nankai.edu.cn. Corresponding author: liuzheli@nankai.edu.cn.*
- *Jin Li is with the School of Computer Science, Guangzhou University, China. E-mail: jinli71@gmail.com.*
- *Siu-Ming Yiu is with the University of Hong Kong, China. Email: smyiu@cs.hku.hk.*
- *Thar Baker is with the Liverpool John Moores University, United Kingdom. Email: t.baker@ljmu.ac.uk.*
- *Brij B. Gupta is with the National Institute of Technology Kurukshetra, Kurukshetra, India. Email: bbgupta.nitkkr@gmail.com.*

data block. This leakage is *where*. Furthermore, when the fog node rewrites or reads the same data block, the cloud server may guess whether the vehicle is in a driving state. This leakage comes from the access time of the encrypted data block, which is *when*. Similarly, the frequency at which the fog node reads and writes vehicle data also leaks the frequency of use of the vehicle, which is *how often*.

To solve this problem, oblivious RAM (ORAM) [18], an access pattern hiding mechanism is proposed. It is useful for users to hide the data access patterns from untrusted outsourcing storage. Although quite a number of ORAM schemes have been proposed in the literature, most of them are impractical due to high computational cost. Actually this may be due to its nature: in order to hide the access pattern, each ORAM access operation is accompanied by accessing some dummy blocks to hide which block is interested, which results in a large amount of data being transmitted, shuffled and re-encrypted. These operations also lead to huge storage for additional dummy blocks in the server side (e.g., the cloud), which require a substantial network bandwidth between the cloud and the client, and large storage for cached data in the client side. These large implementation overhead make ORAM schemes infeasible for some emerging important areas, such as fog nodes or limited bandwidth-constrained environment.

Motivation. Many existing ORAMs [19], [20], [21], [22] in the literature were designed to make ORAMs efficient and practical. However, they still cannot meet the requirements of real-time analysis and rapid response in the fog environment. It is very necessary to propose a lightweight and "thin" ORAM that achieves: 1) $O(1)$ network bandwidth; 2) small block size; 3) constant client storage; 4) constant client computational overhead.

1.2 ORAMs with small client cost

We first review some related ORAMs with small client cost, which are shown in Table 1. Oblivistore [22] is one of the fastest ORAM implementations known to date, but it still requires a large client storage which consists of storage cache ($O(\sqrt{N})$), evict cache ($O(\sqrt{N})$) and shuffle buffer ($O(\sqrt{N})$), where N is the number of real blocks. Ring ORAM [21] is a simple and low latency ORAM construction by leveraging the XOR technique to reduce the bandwidth required by the server and the client. However, the client still requires $O(\sqrt{N}) + cN$ storage space, and frequent encryption and access operations in the eviction procedure. Onion ORAM [20] reduces the required block size by using homomorphic encryption. However, homomorphic encryption leads to heavy client side computational cost and low response time. By using a different approach, multi-cloud oblivious storage (MCOS) [19], which is constructed based on multiple clouds and partition of ORAM [29], can achieve $O(1)$ communication cost and as little as possible client overhead.

Among these ORAMs, MCOS [19] is the most suitable ORAM with small client cost. Its basic idea is to leverage the *computational ability* among non-colluding clouds. More specifically, it moves the frequent shuffle operation in the client to the clouds. When reading a block, instead of downloading some dummy blocks and the interested block to the client, it selects one cloud to shuffle these blocks, adds an onion layer of encryption and sends them to the other cloud. Then, it fetches the interested block from the other cloud. When writing a block, instead of downloading the blocks in filled consecutive levels and shuffling them in the client, it allows the cloud to directly shuffle them, add an onion layer of encryption and store them to the other cloud. Using this approach, one cloud observes the permutation while the other observes the accesses to these blocks. However, they cannot observe both the permutation and data access, thus hiding the access patterns.

TABLE 2
ORAM parameters and notations

Notation	Meaning
N	Number of real data blocks
K	Number of clouds
L	Number of levels in ORAM
C_i	The i-th cloud
$position[u] = (cloud, p, l, offset, r)$	u: The identifier of a block; *cloud*: The cloud of u; p: The partition of u; l: The level of u; *offset*: The offset of u r: The random value
Q	Hybrid-shuffle condition
B	Block size

1.3 Contributions

In this paper, we consider the problems and challenges of achieving the oblivious data access in lightweight devices, and propose an ORAM scheme supporting thin-clients, named "ThinORAM". ThinORAM transfers most computation from the client to the server. It has fixed client storage, $O(1)$ communication cost and the removal of the client complicated computations in reasonable response time. We further discuss how to securely apply ThinORAM in the fog computing environment (named "ThinORAM-$\mathcal{F}$") to achieve oblivious data access with minimum client cost (storage, computation, and communication).

Our experiments show that the proposed ThinORAM has a satisfactory performance, which improves about 2× in response time, reduces most of client storage size and about 2× in network bandwidth between clouds, compared to the best scheme (i.e., MCOS [19]) that aims at reducing the client side overhead.

2 Related Work

Some ORAM schemes had been proposed in past years. In general there are three kinds of construction which are tree-based ORAM [20], [21], partition-based ORAM [19], [22], [29], [32] and Square root ORAM [27], [28]. Recent works pay attention to tree-based ORAM but it either has a high computational cost or has an intolerable communication bandwidth. However, partition-based ORAM can achieve practical application requirements by dividing the original ORAM architecture into multiple shares.

All notations used throughout the rest of the paper are summarized in Table 2.

TABLE 1
Comparison with ORAMs with small client cost. Denote N as number of blocks in ORAM and X as the fixed number of blocks in the client cache.

Schemes	Client storage			Network bandwidth	Computational overhead	Encryption
	Eviction cache	Shuffling buffer	Others			
ObliviStore [22]	$O(\sqrt{N})$	$O(\sqrt{N})$	$O(\sqrt{N})$	$O(\log N)$	$O(\log N)$	SE
Ring ORAM [21]	none	none	$O(\sqrt{N}) + cN$	$O(\log N)$	$O(\log N)$	SE
OnionORAM [20]	none	none	$O(\log N)$	$O(1)$	$\widetilde{\omega}(B \log^4 N)$	AHE
MCOS [19]	$O(\sqrt{N})$	none	$O(1)$	$O(1)$	$O(\log N)$	SE
ThinORAM	$O(X)$	none	none	$O(1)$	$O(\log N)$	SE
ThinORAM-$\mathcal{F}$	$O(1)$	none	none	$O(1)$	$O(\log N)$	SE

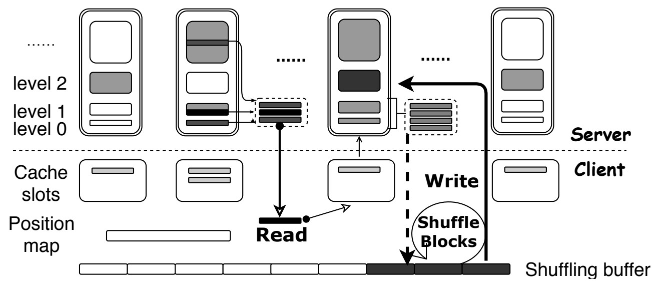


Fig. 2. Partition-based ORAM. Reading a block requires a real block and some dummy blocks to be downloaded to the client. To defend the linkability attack, the fetched block will be stored into a cache slot of a fresh partition. Writing a block requires to download the blocks in consecutive levels and shuffle them in the client.

2.1 Partition-based ORAM

The main idea of a partition-based ORAM (e.g., SSS [29] and ObliviStore [22]) is dividing an entire ORAM into several partitions evenly. Each partition represents a fully functional ORAM which supports standard ORAM operations. In these schemes, an ORAM with capacity of N blocks is usually divided into $O(\sqrt{N})$ partitions, each of which has $O(\sqrt{N})$ blocks. There are $L := \log \sqrt{N} + 1$ levels in a partition, where level i can store 2^i real blocks and 2^i or more dummy blocks, $i \in \{0, 1, ..., L-1\}$. In client storage, there are cache slots with capacity of $O(\sqrt{N})$ blocks and a shuffle buffer with capacity of $O(\sqrt{N})$ blocks. The framework of a partition-based ORAM is shown in Figure 2.

Take the SSS [29] as an example, the client can read or write a block, which is taken as date access. Regular data access consists of two steps: reading from a partition and writing to a partition. The client reads the real block from the specific level of the target partition and reads the dummy blocks from other filled levels of this partition (one dummy block from each of these levels). After that, the client will write it into a random cache slot to defend the linkability attack. If the purpose of this access is to write a block, the fetched block will be changed to data need to be written. The evict operation will randomly select a block from cache slots and call the write function. When writing a data block u back to a partition, it may cause a shuffle operation to disrupt the order and positions of all the blocks.

2.2 Multi-cloud oblivious storage scheme

The MCOS [19] scheme has been proposed as a special application of partition structure. It still divides the entire ORAM into $O(\sqrt{N})$ partitions, but uses two non-collusion clouds to store one partition. The access operation in MCOS [19] is similar to that in SSS [29]. The main difference is that MCOS achieves $O(1)$ client-cloud bandwidth by using two-cloud shuffle operation for both reading a block and writing a block. When reading a block, one cloud will shuffle all the blocks selected from the filled levels of a partition, and send to the other cloud, where the client will retrieve the interested from. When writing a block, one cloud will permute, onion-encrypt all blocks in consecutive filled levels, and send them to the other cloud, where these blocks will be stored into the next level.

Limitations. Although MCOS [19] can be regarded as the most suitable solution for the ORAM with small client cost, there still exists two limitations:

1) *Large client cache storage*. Like other partition-based ORAMs, it requires a large client cache for defending the linkability attack and to ensure access sequence of partitions cannot leak the users' access pattern. In general, the client cache is less than 0.15% of the ORAM capacity (i.e., less than 1.5GB of data for an ORAM of 1TB in capacity). But this amount of storage is still too big for thin-clients (e.g., sensors in the Internet of Things).
2) *Low response time in each data access*. To hide whether access is read or write, the whole data access is made up of reading a block and writing a block. When reading a block, MCOS [19] utilizes two clouds to return a single data block from several blocks. When writing a block, the MOCS still requires a cloud to shuffle blocks in consecutive levels, onion-encrypts them and writes them to another cloud. Both the cloud-cloud communication cost in the read operations and the frequent shuffle operations in the write operations lead to the slow response time.

2.3 XOR technique

The XOR technique [32] can be used to retrieve a single block in one cloud server obliviously. The main idea is XORing several blocks into one result block. All blocks are specific and can be recovered easily except the target one. When recovering the target block, the receiver just needs to XOR the result block with the specific blocks.

Let *Enc*(u_0, r_0), *Enc*(u_1, r_1),..., *Enc*(u_x, r_x) denote the ciphertexts of x dummy block, where *Enc* is a randomized symmetric cipher (e.g., AES) and *Enc*(u_r, r_r) a real block which need to be received. The server can return a single block by computing *Enc*(u_0, r_0) $\oplus$ *Enc*(u_1, r_1) $\oplus$ ... $\oplus$ *Enc*(u_x, r_x) $\oplus$ *Enc*(u_r, r_r). Then the real block can be recovered by the client when XORing the returned block with *Enc*(u_0, r_0) $\oplus$ *Enc*(u_1, r_1) $\oplus$ ... $\oplus$ *Enc*(u_x, r_x). For convenience, the client can set the plaintext of dummy block as 0.

This technique can maintain the size of data blocks and benefit to achieve constant communication. It has been applied to distributed systems [30] and other ORAM implementations such as SR-ORAM [31].

2.4 Security definition

We improve the standard security definition for a multi-cloud ORAM. For a complete access behavior, a server or a cloud gains no information in addition to the access pattern that occurs on its own cloud. That is to say, a server will not know: 1) which data is being accessed; 2) how old it is (when it was last accessed); 3) whether the same data is being accessed (linkability); 4) access pattern (sequential, random, etc); or 5) whether the access is a read or a write.

Definition 1. [***Security definition***]: *Let $\vec{y}$:= ((op_1, u_1, $data_1$), (op_2, u_2, $data_2$),..., (op_M, u_M, $data_M$)) denote a data request sequence of length M, where each op_i denotes a read(u_i) or a write(u_i, data) operation. Specifically, u_i denotes the identifier of the block being read or written, and $data_i$ denotes the data being written. Let $A(\vec{y}, C)$ denote the sequence of accesses to the cloud C given the sequence of data requests $\vec{y}$. An ORAM construction is said to be secure if for any two data request sequences $\vec{y}$ and $\vec{z}$ of the same length, there exists a security parameter λ and a negligible function $negl$ satisfies*

$$Pr[A(\vec{y}, C) = 1] - Pr[A(\vec{z}, C) = 1] < negl(\lambda).$$

where Pr denotes probability.

3 ThinORAM Scheme

In this section, we show the details of our proposed "ThinORAM" (constructed based on the idea of a partition-based ORAM), which requires *very little client storage*, *light client computation* and *constant network bandwidth*.

3.1 Storage structure

Server storage. ThinORAM divides the N blocks into $O(\sqrt{N})$ partitions. Like most ORAMs, we set the block size as 4KB. Different from MCOS [19], all the $O(\sqrt{N})$ partitions will be uniformly distributed in K ($K \geq 3$) non-colluding clouds. That is, ThinORAM adopts the "distributed storage" structure, which is shown in Figure 3.

Client storage. In ThinORAM, we remove the shuffle buffer and set the maximum size of evict cache $Cache$ to be X. The position map can be defined as tuples of (*u*, *cloud*, *p*, *l*, *offset*, *r*), where *r* is the random value. For convenience, we denote *position*[*u*] as the tuple of data block u in the position map, denote C_i as the i-th cloud, and denote $Cache[i]$ as the i-th block cached in the $Cache$.

Initial data value. A dummy block has the default value as the ciphertext of zero. Each data is encrypted by taking a random value as initial vector (IV). The top level is initialized as dummy blocks.

```
Access(op, u, data*):
1: $C_i \leftarrow position[u].cloud$, $p \leftarrow position[u].p$
2: if $p == -1$ then
3:     $C_i \leftarrow UniformRandom(C_1...C_K)$
4:     $data \leftarrow ReadCloud(C_i, \perp)$   // read a dummy block
6:     $data \leftarrow ReadCache(u)$   // read from cache
7: else
8:     $data \leftarrow ReadCloud(C_i, u)$   // read the real block
9: if $op == write$ then
10:     $data \leftarrow data^*$
11: $UnlinkEvict(u, data)$
12: $Return\ data$

UnlinkEvict(u, data):
1: $C_n \leftarrow UniformRandom(C_1...C_K)$ except $C_i$
2: $position[u].cloud \leftarrow C_n$, $position[u].p \leftarrow -1$
3: $WriteCache(u, data)$
4: if $|Cache| == X$ then
5:     $r \leftarrow UniformRandom(0...X-1))$
6:     $(u, data) \leftarrow Cache[r]$, $C_n \leftarrow position[u].cloud$
7:     $WriteCloud(C_n, u, data)$
```

Fig. 4. Algorithms for data access and evict operations.

3.2 Data access

There are three operations in a data access (as shown in Figure 4):

- *ReadCloud*(C_i, *u*): Read a block *u* from the *i*-th cloud.
- *WriteCloud*(C_i, *u*, *data*): Write a chosen block with identifier of *u* and value *data* to the *i*-th cloud.
- *UnlinkEvict*(*u*, *data*): Randomly select a block from the client evict cache and evict it into its pre-specified cloud. For each fetched block *u*, its related cloud will be changed to a random other cloud, and then it will be stored into the client evict cache. There are at least three clouds for the consideration of security.

Single data access is made up of two steps: *ReadCloud* and *UnlinkEvict*. *ReadCloud* will read an interested block from a cloud. *UnlinkEvict* will re-specify the fetched block to another random cloud, store it into the evict cache, randomly select a block from evict cache and call *WriteCloud* to write it to its pre-specified cloud.

3.2.1 *Reading from a cloud*

Instead of shuffle operation between clouds to read a single block in MCOS [19], ThinORAM makes use of XOR technique to generate a single block and returns it back to the client. It maintains $O(1)$ communication, meanwhile, it reduces the response time and the cloud-cloud bandwidth by avoiding the shuffling between the clouds.

The left part of Figure 3 shows an example of *Readcloud*, i.e., read a data u from Cloud C_1. The detailed algorithm of *Readcloud* is shown in Figure 5. Firstly, the client looks up the location of block u and sends the offset of block u and other dummy blocks to the cloud. Then, the cloud simply XORs the blocks according to the received offset, then returns the resulting single block to the client. Finally, the client encrypts each "000...000" with its corresponding

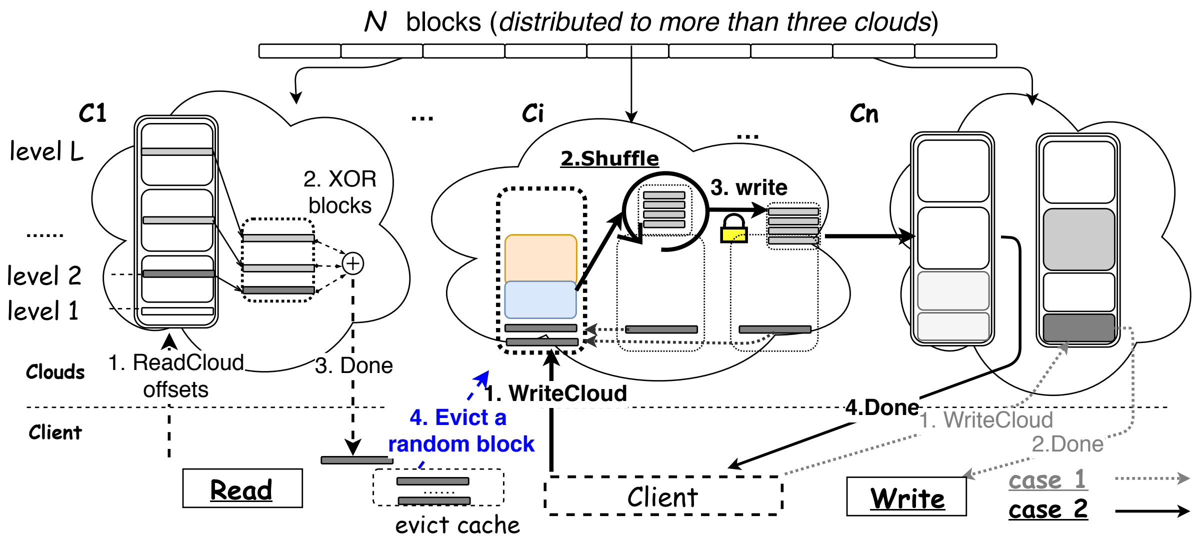


Fig. 3. Illustration of ThinORAM. To read a block from Cloud C_1, the cloud will be required to do XOR operation over all specified blocks of the filled levels, and return the resulting single block to the client. The client will evict the read block to another random cloud immediately. To write a block to C_n (case 1), because the hybrid-shuffle condition is not satisfied, the client will directly write it into an unfilled first level. Otherwise (case 2), when writing to C_i and trigger the hybrid-shuffle, the client will hybrid-shuffle both blocks in consecutive filled levels of the selected partition and blocks from several first levels of other partitions, onion encrypt and write them to another random cloud (e.g., C_n).

```
ReadCloud($C_i$, $u$):
Client:
 1:  $(p^*, l^*, offset^*, r^*) \leftarrow position[u]$
 2:  if $p^* == -1 \;||\; u == \perp$ then
 3:      $p^* \leftarrow UniformRandom(0...\frac{\sqrt{N}}{K})$
 4:  $OffB \leftarrow \varnothing$
 5:  for $l = 0, 1, ..., L-1$ do
 6:      if $l \neq l^*$ then
 7:          $OffB \leftarrow OffB \bigcup GetNextDummy(p^*, l)$
 8:      else
 9:          $OffB \leftarrow OffB \bigcup offset^*$
 10:  send $OffB$ to the server
Server:
 11:  $block \leftarrow blockAt(OffB[0])$
 12:  for $j = 1, ..., |OffB| - 1$ do
 13:      $block \leftarrow block \oplus blockAt(OffB[j])$
 14:  send $block$ to the client
Client:
 15:  for $l = 0, 1, ..., |OffB| - 1$ do
 16:      if $offset^* \neq OffB[j]$ then
 17:          $r \leftarrow getRndFromOffset(OffB[j])$
 18:          $block \leftarrow block \oplus Enc_k(\text{“000...000”}, r)$
 19:  $block \leftarrow Dec_k(block, r^*)$
 20:  Return $blocks$
```

Fig. 5. Reading a block from a cloud.

```
WriteCloud($C_n$, $u$, $data$):
 1: if $NeedHybridShuffle(C_n)$ then
 2:     $blks \leftarrow HybridShuffle(u, data)$
 3:     $C_w \leftarrow randomOtherCloud(C_n)$
 4:     $StoreToCloud(blks, C_w)$
 5: else
 6:     $p \leftarrow randomPartition(C_n)$;
        //select random partition p with unfilled first level in $C_n$
 7:     $StoreToLevel(data, 0, p)$    //store data to first level

NeedHybridShuffle($C_n$):
 1: Let $count$ as the number of all filled first levels
 2: if $count \geq Q$ then
 3:     Return $true$
 4: else
 5:     for $p = 0, 1, ..., \frac{\sqrt{N}}{K}$ do
 6:         $l \leftarrow firstFilledLevel(p)$
 7:         if $AcessNumber(l, p) \geq 2^l$ then
 8:             Return $true$
 9:     Return $false$
```

Fig. 6. Writing a block to cloud. If the hybrid-shuffle condition is satisfied, ThinORAM will shuffle blocks from a partition and first levels in other partitions, then store them to another cloud. Otherwise, it will store the data directly to the first level of a random partition.

random value, XORs it to retrieve the interested block, and finally decrypts to get the real value.

3.2.2 Writing to a cloud

Instead of frequent shuffle operation between clouds to write a block in MOCS, ThinORAM develops "**hybrid-shuffle**" operation (details in Section 3.3) to reduce the frequency of shuffle operations. The details are shown in Figure 6.

When writing a block to C_n, ThinORAM will first judge whether the hybrid-shuffle condition is satisfied. As shown in Figure 3, if the condition is not satisfied (case 1), ThinORAM will store the data directly to the first level of a random partition. Otherwise (case 2), ThinORAM will execute a hybrid-shuffle operation to shuffle the blocks in both consecutive filled levels of this partition and filled first levels of other partitions and store these shuffled blocks $blks$ to a random other clouds.

Assume we want to store the shuffled blocks to Cloud C_n (shown in Figure 7), ThinORAM will first compute the suitable level l as $\lceil \log |blks| \rceil + 1$, select a partition p whose level l is unfilled, and store them into this level. If there is no such a partition, ThinORAM will select a partition whose

```
StoreToCloud($blks$, $C_n$):
1: $l \leftarrow \lceil \log |blks| \rceil + 1$
2: for $p^* = 0, 1, ..., \frac{\sqrt{N}}{K}$ do
3:     if $l$ $is$ $unfilled$ then
4:         $StoreToLevel(blks,\ l,\ p^*)$
5:         Return
6: $p \leftarrow MostAccessedTopLevel()$
7: $StoreToLevel(blks,\ L-1,\ p)$
8: //store to top level (replace the visited or dummy blocks)

MostAccessedTopLevel():
1: $p \leftarrow 0,\ \ l \leftarrow L-1$
2: for $partition\ p^* = 1, ..., \frac{\sqrt{N}}{K}$ do
5:     if $AcessNumber(l,\ p^*) > AcessNumber(l,\ p)$ then
6:         $p \leftarrow p^*$
7: Return $p$
```

Fig. 7. Store shuffled blocks to a cloud.

top level has been accessed the most, and merge the shuffled blocks to the visited blocks or dummy blocks in its top level.

3.3 Hybrid Shuffle

Hybrid-shuffle condition. Denote Q as the number of filled first levels allowed in a cloud. A hybrid-shuffle operation would be executed, when the number of filled first levels is up to Q. Another condition would also lead to a hybrid-shuffle operation is that a partition has been accessed for many times. In partition-based ORAMs, if more than half of blocks are accessed in a level, the shuffle operation must be executed to ensure enough dummy blocks can be used. The details are shown in algorithm *NeedHybridShuffle* of Figure 6.

Hybrid-shuffle operation. The motivation of hybrid-shuffle operation is to shuffle blocks in the first levels of other partitions to ensure that most of the first levels are empty. One hybrid-shuffle operation enables many subsequent write operations to be done directly on the first level. Its security (analyzed in Lemma 2) is guaranteed by the "unlinked ORAM operations". As shown in Figure 11, there are three steps.

Step 1: *Select the partition to be shuffled.*

The algorithm of *MostAccessed* (shown in Figure 11) describes how to select a partition p to be shuffled. The chosen partition must be accessed the most. It will compute the shuffle-rate of each partition as $\frac{AcessNumber(l,\ p)}{2^l}$, that is, the percentage of data blocks being accessed in the first filled level l. ThinORAM will select the partition p with the maximum shuffle-rate as the result.

Step 2: *Collect blocks to be shuffled.*

Three kinds of blocks will be collected into the set $blks$: 1) the block u to be written; 2) the unread blocks in the consecutive filled levels of the selected partition p; 3) the blocks in the filled first level of other partitions.

Step 3: *Shuffle blocks and onion encrypts them.*

For all the blocks in $blks$, ThinORAM will first permute their positions randomly. After shuffling, ThinORAM will onion encrypt them to protect the data privacy, like that in MCOS [19]. Finally, ThinORAM saves these blocks as a new block set $blks'$ and returns to *WriteCloud* function to be stored to another random cloud.

```
HybridShuffle($u$, $data$):
1: $(p,\ l) \leftarrow MostAccessed()$
2: $blks \leftarrow \varnothing \cup data$
3: for $l^* = l$ to $L-1$ do
4:     if $level\ l^*\ is\ filled$ then
5:         $blks \leftarrow blks \cup \{unread\ blocks\ in\ level\ l^*\}$
6:     else
7:         $break$
8: for $partition\ p^* = 0$ to $\frac{\sqrt{N}}{K}$ do
9:     if $level\ 0\ is\ filled$ && $p^* \neq p$ then
10:        $blks \leftarrow blks \cup \{unread\ blocks\ in\ level\ 0\ of\ p^*\}$
11: $blks' \leftarrow Shuffle(blks)$
12: Return $blks'$

MostAccessed($C_n$):
1: $p^* \leftarrow 0;\ l^* \leftarrow 0;\ rate^* = 0.0$
2: for $partition\ p = 0, 1, ..., \frac{\sqrt{N}}{K}$ do
3:         $l \leftarrow firstFilledLevel(p)$
4:         $rate \leftarrow \frac{AcessNumber(l,\ p)}{2^l}$
5:         if $rate > rate^*$ then
6:             $p^* \leftarrow p;\ l^* \leftarrow l$
7: Return $(p^*,\ l^*)$
```

Fig. 8. Hybrid-shuffle operation.

4 Analysis

4.1 Shuffle frequency analysis

In ThinORAM, shuffle operation is necessary to hide the positions of blocks when these blocks are transferred between clouds. As mentioned before, single data access consists of two important steps, that is, reading a block and writing a block.

Theorem 1. *In ThinORAM, large shuffle will happen after $O(Q)$ accesses, where Q is the hybrid-shuffle condition.*

Proof. Let l denote the level number of the first filled level. In ThinORAM, there are two cases leading to the shuffle operation, we analyze the bounds respectively and evaluate the low bound.

Case 1: The total number of blocks stored in Level 0 of all partition is larger than Q. In this case, after every Q accesses, regular shuffle will happen. Therefore, shuffle frequency of ThinORAM would be $F_T = \frac{1}{Q}$.

Case 2: Similar to traditional ORAM like SSS [29], if there exist continuous levels, all blocks in these levels including the latest block will be shuffled to the next level. As mentioned in [29], the large shuffles happen very infrequently, i.e., with exponentially decreasing frequency.

Based on the condition of Case 1 and Case 2, the frequency in Case 2 is lower than Case 1 obviously. So after $O(Q)$ accesses, large shuffle will happen. □

4.2 Cloud-cloud bandwidth analysis

In the following analysis, the bandwidth we mention refers to the bandwidth between the clouds.

Read Bandwidth. Let $\overrightarrow{Q}_s$ denote the access sequence of reading a block. In MCOS, $\overrightarrow{Q}_s$ consists the access operations of two clouds in a "non-distributed" storage structure. For each read operation, MCOS requires the blocks in each filled level of a partition to be transferred to a cloud for shuffling and sends the resulted blocks to other clouds for fetching.

Therefore, cloud-cloud bandwidth must be more than the blocks read in a partition, which is the $O(\log N)$ blocks. On the contrary, ThinORAM has none of cloud-cloud read bandwidth for the XOR technique.

Therefore, we can conclude that read bandwidth of ThinORAM is much smaller compared to MCOS [19].

Write Bandwidth. In MCOS, the bandwidth at i-th access in an initially empty partition proceeds as follows:

$$A_i = \begin{cases} 0 & i \text{ is odd} \\ (2^j - 1) \cdot B & i \text{ is even, } i/2^j \mod 2 = 1 \end{cases}$$

Therefore, the total download bandwidth after T accesses is: $A = \sum_{i=1}^{T} A_i$. The amortized bandwidth cost is $O(\log N) \cdot B$. In the worst case, the client has to download about $2.3\sqrt{N}$ blocks to shuffle and return these padding fresh dummy blocks, it is up to $O(\sqrt{N}) \cdot B$.

Claim 1. *In ThinORAM, the cloud-cloud bandwidth for writing is saved about* $(2^{\lfloor logQ \rfloor} - 2) \cdot B$*, for applying the hybrid-shuffle operation.*

Proof. At $i - th$ access in an initially empty partition, the bandwidth is as follows:

$$A_i = \begin{cases} 0, & \text{i} < \text{Q} \\ 0, & \text{(i-Q)} \mod (Q - 2^{\lfloor logQ \rfloor}) \neq 0 \\ 2^{\lfloor logQ \rfloor} \cdot B, & \text{(i-Q)} \mod (Q - 2^{\lfloor logQ \rfloor}) = 0 \end{cases}$$

In a typical partition structure, the total number of blocks between level 1 and level $l_B - 1$ is $2^{\lfloor logQ \rfloor} - 2$. Compared with the typical structure, in ThinORAM, the bandwidth $(2^{\lfloor logQ \rfloor} - 2) \cdot B$ of writing these blocks is saved. □

Similar to MCOS, the total download bandwidth after T accesses is: $A = \sum_{i=1}^{T} A_i$. The amortized bandwidth cost is lower than $O(\log N) \cdot B$. The frequency of shuffle will be reduced even under the worse case scenario of the bandwidth.

In terms of storage capacity, ThinORAM keeps the top level structure to deal with the worst case. The storage capacity will be reduced to $\sqrt{N} - 2^{\lfloor logQ \rfloor} + 2$ blocks.

4.3 Security analysis

Let s be the security parameter, where ϵ is a function negligible in s.

Lemma 1. *Reading from a cloud leaks no information.*

Proof. In the beginning, the client looks up the position map to get a block from partition p of C_i. During this process, the client fetches a dummy block from each filled level except the level which contains the interested block, then applies XOR operation to these fetched blocks. After that, C_i sends the result to the client. In order to make sure there are enough dummy blocks to support the normal read operation, *hybrid-shuffle* operation in *writeCloud* is completely necessary. When reading a block from partition p of C_i, C_i only knows the behavior of reading a block from partition p but not the location of this block or any other information.

Let $read(B_i)$ denote a reading block B_i, the sequence of reading a block can be denoted as $Q_s = \{read(B_1)\ , \ldots, read(B_n)\}$. Even $read(B_i) = read(B_j)$, it will still leak no information, because the random of the *WriteCache* makes adversary know nothing about where the block is between these two reading operations. When $read(B_i) \neq read(B_j)$, the probability of recognizing these two independent operations is:

$$\left| P_r[\bigcap_{i=1}^{n} read(B_i)] \right| = \prod_{i=1}^{n} |P_r[read(B_i)]| \leq \prod_{i=1}^{n} (\tfrac{1}{2}) = \tfrac{1}{2^n}.$$

□

Lemma 2. *Writing to a cloud leaks no information.*

Proof. Write operation selects one or more filled levels from Partition p which has been accessed for the most times than other partitions on C_i and a number of blocks which come from the first level of every partition on C_i, then execute hybrid-shuffle and sends the result to another cloud (i.e., C_j).

If C_i is malicious, after several accesses, it can get some accessed blocks. Let B_x be the original block, which exists on C_i before sending to C_j for hybrid-shuffle and onion encryption, B_y is the block which exists in C_i now,

$$\forall i, j, s.t., |Pr[(B_x \in C_i) = (B_y \in C_i)]| < \epsilon(s)$$

If C_j is malicious, under the non-colluding condition, it cannot get Hybrid-shuffle key which is stored in C_i. So C_j cannot link the latter blocks with the former blocks. □

Lemma 3. *Hybrid-shuffle leaks no information.*

Proof. Let blk denote blocks need to be shuffled and let C_1 denote the cloud which performs shuffle operation. The shuffled blocks blk' will be stored in cloud C_2. Let $pos[blk]$ denote the position of blocks blk and let Π denote shuffle permutation according to keyed pseudorandom permutation (PRP) function.

Suppose C_1 is malicious. All blocks blk and position information $pos[blk]$ are exposed to cloud C_1. Cloud C_1 generates a pseudo-random one-time shuffling key k to ensure that after the shuffle procedure the $pos[blk]$ and $pos[blk']$ are completely unrelated. That is, there exists a negligible function $negl$ and a security parameter λ satisfies

$$Pr[\Pi_k(pos[blk]) = 1] - Pr[\Pi_k(pos[blk']) = 1] < negl(\lambda).$$

The shuffled blocks blk' will be stored in cloud C_2 which will not collude with cloud C_1, which ensure the following operation about blk' is independent of cloud C_1.

Suppose C_2 is malicious. All the information acquired by the cloud C_2 is only the blk' and position information $pos[blk']$ after being shuffled. Cloud C_2 can gain no information about shuffle permutation Π without shuffling key k. Subsequent data access can be considered random and independent of data access prior to shuffle. □

Lemma 4. *Unlinked Evict can be applied securely to defend linkability attack.*

Proof. In partition-based ORAM, there exists linkability attack. When writing a fetched block, it will reveal the partition in the process. In the next access, it may read a block from the same partition. In that case, the cloud can deduce with a certain probability that the two consecutive accesses are for the same block. Since it cannot be ignored, there may exist a weak linkability attack inside the cloud.

In ThinORAM, the probability that a cloud can distinguish whether the same block is accessed when the same partition is accessed is

$$Pr[partition] = \frac{1}{\log N}$$

When the size of client cache is X, and in each access, the client will choose a block in cache randomly for writing. The probability that chooses the same block is

$$Pr[cache] = \frac{1}{X}$$

Notice that in "HybridShuffle", Q blocks will be shuffled to another cloud including the latest one. If the written block is shuffled immediately, the cloud will no longer get the information of consecutive reads and writes. The probability that does not execute "HybridShuffle" is

$$Pr[HybridShuffle] = \frac{1}{Q}$$

Therefore, the probability that a cloud can successfully identify that two adjacent accesses are on the same block is

$$\begin{aligned} Pr[same] &= Pr[partition] \cdot Pr[cache] \cdot (1 - Pr[HybridShuffle]) \\ &= \frac{1}{\log N} \cdot \frac{1}{X} \cdot (1 - \frac{1}{Q}) \\ &= \frac{Q-1}{\log N \cdot X \cdot Q} \end{aligned}$$

In summary, as X increases, the probability of identifying the same block in adjacent accesses decreases. In order to ensure less leakage, we will maximize the capacity of X. However, it will bring heavy cost to the client. Therefore, we have the upper bound of cache size. To maintain the same degree of security with SSS [29], we define the upper bound of X as $O(\frac{\sqrt{N}}{\log N})$. In practice, we can reduce X appropriately to ensure that the client can operate normally, though this will reduce the level of security. □

5 Application of ThinORAM in Fog Computing

5.1 Challenges of applying ThinORAM

In any kind of ORAMs, position map acts as the most important role for storing the location information of each block. It always occupies a large amount of client storage. To reduce client storage, most ORAMs outsource the position map to the clouds and look up it in a recursive way. However, this approach results in heavy offline bandwidth. Even ThinORAM can achieve minimal client overhead, it is still unsuitable for lightweight fog nodes or IoT devices. *How to configure position map* is the first challenge for applying ThinORAM in practical IoT applications.

From Claim 4, there exists a trade-off between evict cache size and security in ThinORAM. The bigger the evict cache size is, the higher the security is. We can adopt a big evict cache size if we want to achieve the ideal security. But it brings a heavy storage overhead (the upper bound is $\frac{\sqrt{N}}{\log N}$) for the client. *How to balance the relationship between evict cache size and security* is our second challenge.

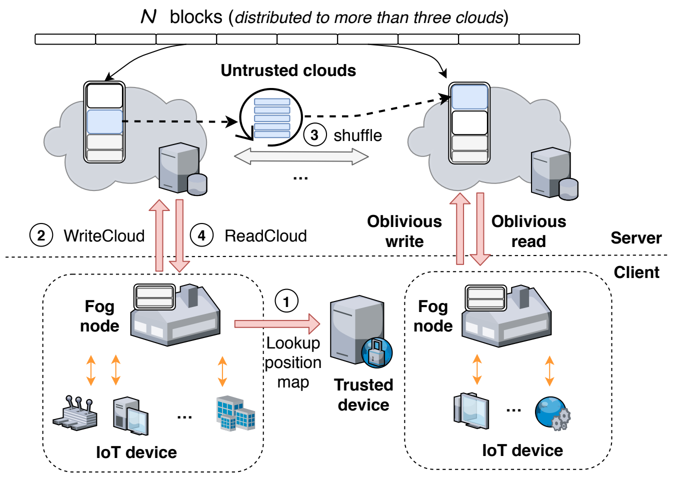


Fig. 9. Illustration of ThinORAM-$\mathcal{F}$

5.2 ThinORAM-$\mathcal{F}$ in fog computing

For the practical requirement, we inevitably need to deploy a trusted device in the original fog environment which is described in Section 1. The trusted device is the hardware using trusted processors or acting as the trusted center (e.g., trusted gateway). It can securely store some sensitive data.

ThinORAM-$\mathcal{F}$. Considering efficiency and security, we move the position map in the cloud server to the trusted center. The interaction between fog nodes and IoT devices is considered as an internal operation of the client. We call ThinORAM under this deployment as "ThinORAM-$\mathcal{F}$". The framework of "ThinORAM-$\mathcal{F}$" is shown in Figure 9, which is similar to ThinORAM except that the fog nodes play the role of the client.

Single data access in ThinORAM-$\mathcal{F}$ can be described as follows: firstly, the client (fog nodes) obtains the position of a target data block from position map in the trusted device. If the target data block is in the cloud server rather than the fog nodes, the fog nodes perform *ReadCloud* to fetch the block in the cloud. Next, the client stores the fetched data block into the trusted device and gets a new random data block from the trusted device. Finally, it calls *WriteCloud* to write the new data block to the cloud. If the shuffle operation is triggered, the cloud shuffles data blocks into another cloud following the rule which is the same as ThinORAM.

By storing the position map and evict cache into the trusted device, ThinORAM-$\mathcal{F}$ can achieve the minimal client storage and allow the fog nodes or IoT devices to quickly look up the block's position. The size of evict cache is adjusted by the fog node configuration.

5.3 Performance Evaluation

5.3.1 *Implementation details*

We implemented both MCOS and ThinORAM-$\mathcal{F}$ in Java. Like SEAL-ORAM [26], we use MongoDB to store and access data blocks. In the server side, we initialize a partition by using a *collection* and each block is stored as a *document* in the corresponding *collection*. About cryptographic algorithm, we adopt AES-CBC to encrypt all blocks. We provide server and client programs to implement the actual ORAM. Especially, we build the communication mechanism

TABLE 3
Comparison of network bandwidth.

Data access number		100	1000	5000	10000
ThinORAM-$\mathcal{F}$	write cloud	26.3k	348.7k	2139.7k	5357.8k
	cloud to cloud	44.5k	700.9k	3787.4k	7344.4k
	read cloud	35.3k	357k	1874.5k	3519.4k
MCOS [19]	write cloud	33.3k	341.5k	1635.3k	3164.6k
	cloud to cloud	175.5k	1245.4k	8558.1k	14566.9k
	read cloud	28.3k	286.2k	1411.3k	2895.2k

between the server and the client based on Java socket API. The source code and test cases can be found in GitHup: *https://github.com/emigrantMuse/ThinORAM.git*.

Deployment. Based on the three-layer architecture of the fog computing mentioned in Section 1, we implemented the configuration and simulation of the fog node and cloud server. We deploy the server in three cloud providers and rent one server per cloud. Each server has 1TB capacity, 4Mbps bandwidth, 4GB memory and 2 core CPU. And we deploy the client in a gateway device with 1.8 GHz dual-core processor, 256MB memory and 512MB flash installed with Linux OS, to simulate fog nodes. Since the client of ThinORAM is deployed on the fog nodes, the IoT device is replaced by the data generation algorithm simulated by the fog nodes in our experiments.

Test cases. In our experiments, we set N=40,000 and Q=12. In data access, a block will be accessed randomly. After reading a block, we change its data or leave it alternatively. Then, we write it back to other cloud and shuffle in a specified way described in Section 3.3.

5.4 Performance Evaluation

According to the results of experiments, we can conclude that when compared to MCOS: 1) The shuffle frequency in ThinORAM-$\mathcal{F}$ has been reduced by about 45×; 2) The bandwidth between cloud and cloud in ThinORAM-$\mathcal{F}$ is about 2-3× smaller than that in MCOS; 3) The response time in ThinORAM-$\mathcal{F}$ has been improved about 2×.

Data access distribution. The distribution of data access is shown in Figure 10, where the number of reading from the cloud in ThinORAM-$\mathcal{F}$ is approximately equal to the number of writing to cloud. Especially, the number of interactions between the clouds is about twice that of read and write operations. We note that communication between cloud to cloud occupies the majority of access operations.

Shuffle frequency. Hybrid-Shuffle sets the parameter Q to control the frequency of shuffling. The frequency of shuffle operation decreases as Q increases shown in Figure 11. As shown in Figure 12, we know that the number of shuffle operations in ThinORAM-$\mathcal{F}$ is far fewer than that in MCOS. The shuffle frequency in ThinORAM-$\mathcal{F}$ is about 1/45× lower than the MCOS. By reducing the number of shuffle operations, we can appropriately reduce the network bandwidth and response time.

Response time. Figure 13 shows the comparison of response time between ThinORAM-$\mathcal{F}$ and MCOS. We note that ThinORAM-$\mathcal{F}$ has better performance than MCOS. Under different visits, MCOS is 2× slower than ThinORAM-$\mathcal{F}$. The improvement comes from the followings: 1) XOR operations take little time in read operation; 2) the frequency of shuffle operations is reduced, and accordingly the communication between the clouds is reduced.

Network bandwidth. Table 3 shows the detailed network bandwidth in different number of data operations in ThinORAM-$\mathcal{F}$. The bandwidth is divided into three parts: read cloud, write cloud and cloud to cloud. For ThinORAM-$\mathcal{F}$, the bandwidth for cloud to cloud accounts for the largest proportion, which accounts for 60% of the total bandwidth. The bandwidth for read cloud is approximately equal to write cloud. As the data access number increases, the bandwidth of read cloud is gradually greater than that of write cloud. Compared with MCOS, read and write bandwidth in ThinORAM-$\mathcal{F}$ is slightly less than the read and write bandwidth in MCOS. The bandwidth for cloud to cloud in ThinORAM-$\mathcal{F}$ is much smaller than the corresponding bandwidth in MCOS, which is reduced by about 2-3×.

Based on our experiments, we can conclude that the operations between clouds occupy the majority of the bandwidth. In terms of the shuffle frequency and response time, ThinORAM-$\mathcal{F}$ is much lower than those of MCOS. ThinORAM-$\mathcal{F}$ requires lower bandwidth than MCOS and its performance can meet the needs of practical applications.

6 Conclusions

To enable ORAM to be feasible in emerging areas such as Internet of Things, there is a strong demand for a "thin" client ORAM construction for storage-limited and computation-limited devices. Based on this, we propose our "ThinORAM" scheme under non-colluding clouds, which has small client storage and low computation requirement in the client side. Compared to other ORAMs, ThinORAM can achieve the best (minimum) client-side overheads (storage and computation), with the best response time, as verified by our experiments. We believe that the advantages of "non-colluding clouds" have not yet been fully explored. We expect to see better schemes that can rely on this valid assumption.

Acknowledgment

This work was supported by the National Natural Science Foundation of China (No.61672300), National Natural Science Foundation of Tianjin (No. 18ZXZNGX00140) and National Natural Science Foundation for Outstanding Youth Foundation (No. 61722203).

References

[1] H. Pengfei, D. Sahraoui, N. Huansheng, Q. Tie, "Survey on fog computing: architecture, key technologies, applications and open issues", in *Journal of network and computer applications*, 2017, 98, pp. 27-42.
[2] L. Jiang, L. Da Xu, H. Cai, Z. Jiang, F. Bu, B. Xu, "An IoT-oriented data storage framework in cloud computing platform", in *IEEE Transactions on Industrial Informatics*, 2014, 10(2), pp. 1443-1451.
[3] R. Swati Sucharita, P. Deepak, S. Suraj, M. Saraju P, Z. Albert Y, "Building a sustainable Internet of Things: Energy-efficient routing using low-power sensors will meet the need", in *IEEE Consumer Electronics Magazine*, 2018, 7(2), pp. 42-49.
[4] L. Rakesh Kumar, R. Amiya Kumar, T. Zhiyuan, S. Suraj, P. Deepak, S. NVR, P. Mukesh, R. Rohit, T. Shankar Sharan, "Building Scalable Cyber-Physical-Social Networking Infrastructure using IoT and Low Power Sensors" in *IEEE Access*, 2018, 6, pp. 30162-30173.

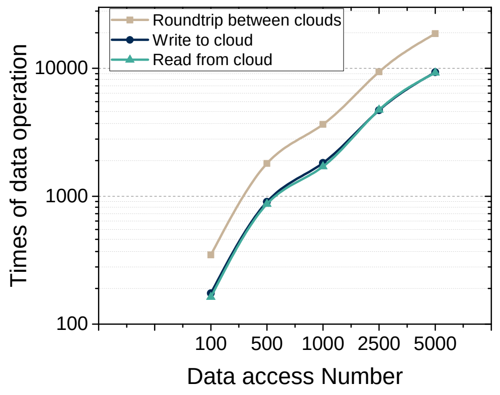


Fig. 10. The distribution of data access. *N=40000, Q=12.* Each data access will cause a reading operation and a writing operation.

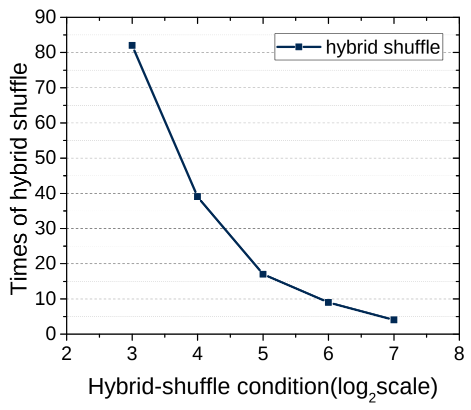


Fig. 11. The frequency of shuffle operation. N=40000, data access number=300. The number of shuffles decreases as the size increases.

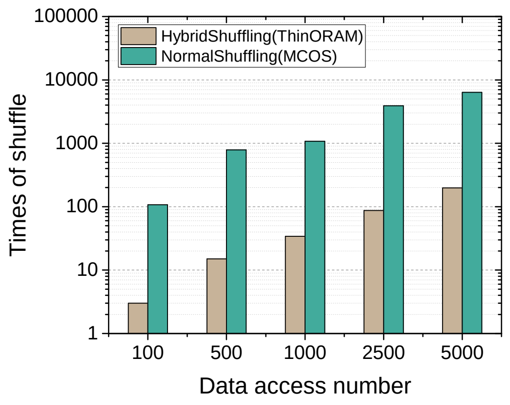


Fig. 12. Comparison in number of shuffle operation. The number of shuffle operation in ThinORAM-$\mathcal{F}$ is about 1/45 times that of MCOS.

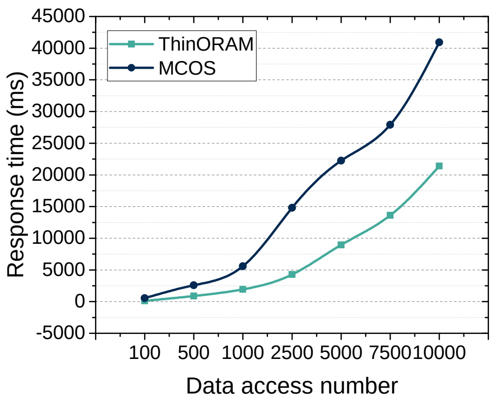


Fig. 13. Comparison in response time. The ThinORAM-$\mathcal{F}$ is about 2$\times$ faster than MCOS. Because each operation of MCOS requires cloud-cloud communication.

[5] S. Rathin Chandra, S. Suraj, P. Deepak, Z. Albert Y, "Location of Things (LoT): A review and taxonomy of sensors localization in IoT infrastructure", in *IEEE Communications Surveys & Tutorials*, 2018, 20(3), pp. 2028-2061.

[6] F. Tao, Y. Cheng, L. Da Xu, L. Zhang, B. H. Li, "CCIoT-CMfg: cloud computing and internet of things-based cloud manufacturing service system", in *IEEE Transactions on Industrial Informatics*, 2014, 10(2), 1435-1442.

[7] Y. Yang, X. Liu, R. H. Deng, "Lightweight Break-glass Access Control System for Healthcare Internet-of-Things", in *IEEE Transactions on Industrial Informatics*, 2017, DOI:10.1109/TII.2017.2751640.

[8] L. Chang, R. Rajiv, Z. Xuyun, Y. Chi, G. Dimitrios, C. Jinjun, "Public auditing for big data storage in cloud computing–a survey", in *2013 IEEE 16th International Conference on Computational Science and Engineering*. IEEE, 2013, pp. 1128-1135.

[9] R. Rajiv, R. Omer, N. Surya, Y. Mazin, J. Philip, W. Zhenya, B. Stuart, W. Paul, J. Prem Prakash, G. Dimitrios, others, "The next grand challenges: Integrating the internet of things and data science", in *IEEE Cloud Computing*, 2018, 5(3), pp. 12-26.

[10] K. Alireza, K. Alireza, R. Rajiv, "Elasticity management of streaming data analytics flows on clouds", in *Journal of Computer and System Sciences*, 2017, 89, pp. 24-40.

[11] W. WenQiang, Z. Xiaoming, Z. Jiangwei, L. Hock Beng, "Smart traffic cloud: An infrastructure for traffic applications"", in *2012 IEEE 18th International Conference on Parallel and Distributed Systems*. IEEE, 2012, pp. 822-827.

[12] A. Arwa, A. Abdulrahman, H. Chunqiang, C. Xiuzhen, "Fog computing for the internet of things: Security and privacy issues", in *IEEE Internet Computing*, 2017, 21(2), pp. 34-42.

[13] B. Flavio, M. Rodolfo, Z. Jiang, Addepalli, "Sateesh Fog computing and its role in the internet of things", in *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. ACM, 2012, pp. 13-16.

[14] S. Ivan, W. Sheng, "The fog computing paradigm: Scenarios and security issues", in *2014 Federated Conference on Computer Science and Information Systems*. IEEE, 2014, pp. 1-8.

[15] M. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In Network and Distributed System Security Symposium (NDSS), 2012.

[16] M. S. Islam, M. Kuzu, M. Kantarcioglu, "Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation", in *NDSS*, 2012.

[17] D. Cash, P. Grubbs, J. Perry, T. Ristenpart, "Leakage-abuse attacks against searchable encryption," in *CCS*, 2015, pp. 668-679.

[18] O. Goldreich, R. Ostrovsky, "Software protection and simulation on oblivious RAMs," in *JACM*, 1996, 43(3), pp. 431-473.

[19] E. Stefanov, E. Shi, "Multi-cloud oblivious storage," in *CCS*, 2013, pp. 247-258.

[20] S. Devadas, M. van Dijk, C. W. Fletcher, L. Ren, E. Shi, D. Wichs, "Onion ORAM: A constant bandwidth blowup oblivious RAM," in *TCC*, 2016.

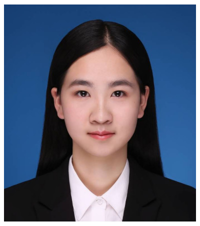
[21] L. Ren, C. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, S. Devadas, "Constants Count: Practical Improvements to Oblivious RAM," in *USENIX Security*, 2015.

[22] E. Stefanov, E. Shi, "Oblivistore: High performance oblivious distributed cloud data store," in *NDSS*, 2013.

[23] D. S. Roche, A. J. Aviv, S. G. Choi, "A Practical Oblivious Map Data Structure with Secure Deletion and History Independence," in *S&P*, 2016.

[24] Y. Jia, T. Moataz, S. Tople, P. Saxena, "OblivP2P: An Oblivious Peer-to-Peer Content Sharing System," in *USENIX Security*, 2016.
[25] V. Bindschaedler, M. Naveed, X. Pan, X. Wang, Y. Huang, "Practicing oblivious access on cloud storage: the gap, the fallacy, and the new way forward," in *CCS*, 2015, pp. 837-849.
[26] Z. Chang, D. Xie, F. Li, "Oblivious RAM: a dissection and experimental evaluation," in *VLDB*, 2016, 9(12), pp. 1113-1124.
[27] G. Oded, O. Rafail, "Software protection and simulation on oblivious RAMs", in *Journal of the ACM (JACM)*, 1996, 43(3), pp. 431-473.
[28] Z. Samee, W. Xiao, R. Mariana, G. Adrià, D. Jack, E. David, K. Jonathan, "Revisiting square-root ORAM: efficient random access in multi-party computation", in *2016 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2016, pp. 218-234.
[29] E. Stefanov, E. Shi, D. Song, "Towards practical oblivious RAM," in *NDSS*, 2012.
[30] S. Cetin, Z. Victor, E. Amr, L. Huijia, T. Stefano, "Taostore: Overcoming asynchronicity in oblivious data storage", in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 198-217.
[31] W. Peter, S. Radu, "Single round access privacy on outsourced storage", in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012. pp. 293-304.
[32] J. Dautrich, E. Stefanov, E. Shi, "Burst ORAM: Minimizing ORAM response times for bursty access patterns", in *USENIX Security*, 2014, pp. 749-764.

Yanyu Huang received BS Degree of Information Security from China University of Geosciences, Wuhan, China, in 2016. Currently, she studies for the master degree in computer science at Nankai University. Her research interests include applied cryptography, data privacy protection.



Bo Li graduated from the College of Computer and Control Engineering at Nankai University and received BS Degree of Engineering in 2017. Currently, she studies for the master degree in computer science at Nankai University. Her research interests include applied cryptography, data privacy protection.

Zheli Liu received the BSc and MSc degrees in computer science from Jilin University, China, in 2002 and 2005, respectively. He received the PhD degree in computer application from Jilin University in 2009. After a postdoctoral fellowship in Nankai University, he joined the College of Computer and Control Engineering of Nankai University in 2011. Currently, he works at Nankai University as an Associate Professor. His current research interests include applied cryptography and data privacy protection.

Jin Li received the BS degree inmathematics from Southwest, University, in 2002 and the PhD degree in information security from Sun Yatsen University, in 2007. Currently, he works at Guangzhou University as a Professor. He has been selected as one of science and technology new star in Guangdong province. His research interests include applied cryptography and security in cloud computing. He has published over 50 research papers in refereed international conferences and journals and has served as the program chair or program committee member in many international conferences.

Siu-Ming Yiu received a BSc in Computer Science from the Chinese University of Hong Kong, a MS in Computer and Information Science from Temple University, and a PhD in Computer Science from The University of Hong Kong. He received two research output prizes, one from the department in 2013 and one from the faculty in 2006. He was selected for Outstanding Teaching Award by the University in 2009, the Teaching Excellence Award in the Department in 2001, 2003, 2004, 2005, 2007, 2009, and 2010. He also received the Best Teacher Award of the Faculty of Engineering twice (2005 and 2009). Before he joined the Department as a faculty member, he has worked as an Analyst Programmer for a couple of years. Besides basic research, he has been involving in various industrial projectsinvolved in quite a number of industrial projects.

Thar Baker is a Senior Lecturer in Software Systems in the Department of Computer Science at the Faculty of Engineering and Technology. He has received his PhD in Autonomic Cloud Applications from LJMU in 2010. Dr Baker has published numerous refereed research papers in multidisciplinary research areas including: Cloud Computing, Distributed Software Systems, Big Data, Algorithm Design, Green and Sustainable Computing, and Autonomic Web Science. He has been actively involved as member of editorial board and review committee for a number peer reviewed international journals, and is on programme committee for a number of international conferences. Dr. Baker was appointed as Expert Evaluater in the European FP7 Connected Communities CONFINE project (2012-2015). He worked as Lecturer in the Department of Computer Science at Manchester Metropolitan University (MMU) in 2011.

Brij B. Gupta received the Ph.D. degree from IIT Roorkee, India. He was a Post-Doctoral Research Fellow in UNB, Canada. He is currently working as an Assistant Professor with the Department of Computer Engineering, National Institute of Technology Kurukshetra, India. He spent over six months with the University of Saskatchewan, Canada, to complete a portion of his research. He has visited several countries to present his research. He has published over 45 research papers in international journals and conferences of high repute. His research interest includes information security, cyber security, cloud computing, Web security, intrusion detection, computer networks, and phishing. He is member of ACM, SIGCOMM, The Society of Digital Information and Wireless Communications (SDIWC), Internet Society, and the Institute of Nanotechnology, and a Life Member of the International Association of Engineers and the International Association of Computer Science and Information Technology. worldwide.