MiniORAM: Minimizing Storage and Bandwidth for Layer-based Partition ORAMs

Anonymous Author(s)

ABSTRACT

Oblivious RAM (ORAM) is no longer a theoretical topic, application to real scenarios is feasible now. From a practical point of view, further reducing the storage (for both server and client) and bandwidth is important. In this paper, we focus on layer-based partition ORAMs and propose MiniORAM that can (i) reduce the server storage to $2.8N$ (N is the number of real blocks), which is 1.64× less than existing ones; (ii) reduce the bandwidth from $O(B \log N)$ to $O(B)$, where B is the size of block which is set to be $\widetilde{\Omega}(\log^4 N)$; and (iii) eliminate the necessity of a shuffle buffer in the client side. Technically, we combine the Private Information Retrieval (PIR) into the design of ORAM operations and exploit the innovative usage of PIR (named "hidden-shuffle") to achieve our improvement. Although additively homomorphic encryption is used, we parallelize the encryption operations using C codes with OpenMP based on GMP library. Experiments show that a single data access requires only milliseconds (the speed up is at least 600× when compared to a Java implementation without parallelization). It is feasible for practical applications.

KEYWORDS

Oblivious RAM, partition ORAM, access pattern, private information retrieval

1 INTRODUCTION

There is an increasing trend for both individuals and organizations to use clouds for storing massive data and cloud-based applications. For sensitive data, users may encrypt them before uploading. However, encryption alone may not be secure enough because a malicious server can still gain sensitive information from user's access patterns [20], e.g., *where*, *when* and *how frequent* users access their data. Prior works have shown that a cloud server can learn some valuable information about its clients' stored data, such as keyword search queries or encryption keys [4, 19, 32], through monitoring the queries made by the clients and recording the access frequencies of each data block.

Oblivious RAM (ORAM). ORAM, initially proposed by Goldreich and Ostrovsky [20], is a general cryptographic primitive to allow oblivious accesses of sensitive data (i.e., hiding access patterns), through shuffling and re-encrypting the data when it is read or written each time to unlink accesses to the same piece of data. Roughly speaking, there are two major ORAM structures: layer-based structure [1, 10–13, 18, 28] in which the storage is divided into levels; and tree-based structure ORAM [3, 8, 9, 22]) where the data is stored in a binary tree. There are many applications for ORAM such as cloud outsourced storage [23, 24], secure processors [2], oblivious peer-to-peer content sharing system [33], oblivious databases [5, 30], and secure multi-party computation [7, 8, 12, 14, 33].

In ORAM, the server storage is always bigger than the capacity of real blocks. Generally, dummy blocks are needed in each access and play an important role on access pattern. When reading data, lots of dummy blocks and an interested block must be downloaded together from the server to hide which block is read. When writing data, dummy blocks are needed to hide whether dummy one or not is written back to the server. As a result, dummy blocks not only occupy a large amount of server storage space, but also increase the network bandwidth. Although many ORAM schemes [6–8, 14, 16, 21, 23, 24, 26, 31] exist, very few focus on how to reduce both server storage and network bandwidth. Cloud providers (e,g. AWS, Alibaba) usually charge clients based on the size of storage used and the amount of network traffic.

1.1 Partition-based ORAMs and Challenges

The overheads (e.g. storage, bandwidth, efficiency) induced by hiding access patterns are significant. A break-through came from a novel partitioning technique proposed by Stefanov et.al [7]. They proposed to partition a single ORAM of N blocks into $\sqrt{N}$ sub ORAMs of size roughly $\sqrt{N}$ blocks each. Partition can be applied to both layer-based (e.g. SSS [7], Oblivistore [24], burstORAM [14] and MCOS [23]) and tree-based ORAMs (e.g. CURIOUS [26]). Partition-based ORAMs were shown to achieve the best performance and become possible to be applied in practical situation [26], especially for devices with sufficient storage.

Issues and challenges. We mainly focus on the storage and bandwidth problems of the layer-based partition ORAMs, in which each sub ORAM is a layer-based ORAM with $\sqrt{N}$ blocks. Except on the server storage with $4.6\times N$ capacity, there are two other aspects should be minimized.

First, the client cache should be minimized to make it practical for devices with small storage size. The existing schemes (take SSS [7] as an example) rely on two kinds of client cache. One is cache of evict operation, for defending linkability attack and ensuring that access sequence of partitions cannot leak the users' access patterns. In general, the evict cache is about 0.15% of the ORAM capacity (i.e., 1.5GB for an ORAM of 1TB in capacity); the other is cache of shuffle operation, for shuffling data in the client. In the worst case, the shuffle buffer size is about $2.3\times\sqrt{N}$ (i.e., 2.3GB for an ORAM of 1TB in capacity). It's obvious that the big client cache would limit the scope of partition-based ORAMs.

Second, the network bandwidth should be minimized, especially in the worst case. When writing a block to a partition, the partition ORAM always requires downloading all the blocks in the consecutive filled levels, reshuffling them in the client side and then writing them back into the next unfilled level. With the increase of the number of data access, the consecutive filled levels will be more and more, leading to the one-time download blocks will be huge. Use the scheme of SSS [7] as an example, in the worst case, the client would download 2.3GB data (for an ORAM of 1TB in capacity) and rewrite them to the server, huge network bandwidth and low response time bring a great challenge when applying the partition-based ORAMs into real applications.

Technically, minimizing the storage and bandwidth of partition-based ORAMs, there exist two challenges. Because dummy blocks play a critical role and incurs big overheads for server storage and network bandwidth. Our first challenge is "**how to reduce the number of dummy blocks**"; Recall that the shuffle operation needs to download and reshuffle all the data when consecutive levels are filled, leading to the bandwidth blowup and the cost of both client computation and client storage. Our second challenge is "**how to eliminate the shuffle operation**".

1.2 Related Works

Some ORAMs have implicitly or explicitly leveraged server computation to minimize the network bandwidth [14, 16, 21, 25, 31], or reduce the number of online roundtrips [27].

XOR technique. BurstORAM [14] and RingORAM [16] used XOR technique to integrate real and dummy data into one single block, when reading a block. The client can recover the real data using also XOR operations. However, XOR requires that the dummy blocks can be used only once, leading to no help for reducing the number of dummy blocks.

PIR-based ORAMs. Recently, Private Information Retrieval (PIR) [21, 25, 31] has been widely applied to simplify the read operation and minimize the bandwidth costs. In 2014, Path-PIR [25] leverages PIR to improve ORAM online bandwidth, but its overall bandwidth blowup is still polylogarithmic. In 2016, OnionORAM [21] further leverages PIR to achieve constant bandwidth blowup with the block size of $\widetilde{\Omega}(\log^5 N)$.

In most PIR-based ORAMs, PIR is mainly used to read a block from specified blocks whatever real or dummy, summarized as "**PIR-reading**". It sends some helper values (called PIR vector, with the length greatly shorter than block size) to the server; the server runs a computation to product a single block and returns it back. PIR can be implemented by homomorphic encryption, for consideration of efficiency, mainly by the additively homomorphic encryption (AHE). However, the ciphertext expansion caused by AHE makes PIR to be impractical in most real applications. Aiming at this problem, OnionORAM [21] technically develops a solution for ciphertext expansion issue.

Path-PIR [25] introduces another usage of PIR, that is, how to hiddenly write a block by PIR, named as "**PIR-writing**". It first runs PIR-reading to fetch the target data block; then, it computes a change value using the new data value and the fetched old value; finally, it sends the PIR vector with the change value to the server, and requires the server to run a homomorphic computation for hiddenly changing the target block.

Notice that all the PIR-based ORAMs are all over tree-based ORAMs. How to combine the PIR with partition-based ORAMs is also an open problem. Actually, PIR is a powerful tool which can hide the position of interested block, reuse the dummy blocks and *even can change the block's content, i.e., rewrite it.* Recall that each dummy block can only be accessed once before it is overwritten or reshuffled in layer-based ORAMs. This trigger us to think about *how to extend the usage of PIR to reduce the number of dummy blocks and integrate the reshuffling process into the write operations.*

1.3 Overview of our Techniques

Upper moving operation. To minimize both the network bandwidth and client storage (i.e., shuffle buffer) of layer-based partition ORAMs, most effective way is to remove shuffle operation. Instead of downloading and reshuffling blocks in the consecutively filled levels, we can simply move these blocks up to next unfilled level. This operation is named as *upper moving.* However, this operation must result in some leakages, including: (i) position relation leakage of a block, if the blocks are simply moved; (ii) block attribute leakage in the worst case. In this case, the blocks in consecutive levels will be moved to the top level and overlay some existing blocks in the top level whose attribute is dummy or noisy, that is to say, their attributes will be leaked.

All ORAM operations implemented by PIR. To prevent the leakages and ensure the security, we design all the ORAM operations by PIR whatever read or write, to protect the privacy of each data access. Besides utilizing PIR-reading to hiddenly read a block, we apply the PIR-writing to achieve hiddenly write a block back to a partition. Most importantly, when writing a dummy block to a partition, we develop a novel usage of PIR, i.e., *hidden-shuffle operation*, to randomly move a real block in the filled levels into the other levels obliviously. This is most technically challenging, which will be discussed later in more detail.

Hidden-shuffle operation. A PIR-writing operation has three steps: prepare-reading, computing change value and writing. The former is to get the old value of the target block. We set the dummy blocks as the ciphertexts of 0 encrypted by additively homomorphic encryption scheme used in PIR, providing the ability to directly write a block into the dummy block without need of fetching the old value by pre-reading. Thus, when writing a dummy block, we can use the prepare-reading step to do something useful. When writing a real block, the PIR-writing operation is the same as other schemes. But when writing a dummy block, we utilize the prepare-reading step to fetch a real block, then write it to another position where a dummy block is stored resulting in shuffling

Table 1: Comparison with other layer-based ORAMs and PIR-based ORAMs. (N: the number of real blocks, B: the size of a block.)

Scheme	Server storage	Client storage		bandwidth	**Scheme**	block size	bandwidth
		evict cache	shuffle buffer				
SSS [7]	4.6N	$O(\sqrt{N})$	$O(\sqrt{N})$	$O(B\log N)$ (amortized cost) $O(B\sqrt{N})$ (the worst case)	**Path-PIR** [25]	$\omega(\log^5 N)$	$O(B\log N)$
ObliviStore [24]	4.6N	$O(\sqrt{N})$	$O(\sqrt{N})$	$O(B\log N)$	**AHE Onion ORAM** [21]	$\widetilde{\Omega}(log^5 N)$	$O(B)$
MiniORAM	2.8N	$O(\sqrt{N})$	–	$O(B)$	**MiniORAM**	$\widetilde{\Omega}(log^4 N)$	$O(B)$

one data block during a write operation. There is no need for an explicit reshuffling.

Improved layered storage structure. To enable hidden-shuffle operation, we need to re-design the layered storage structure. For each partition with $\sqrt{N}$ real blocks and $L := \llcorner\log\sqrt{N}\lrcorner$ levels, the top level is designed to contain $2\times\sqrt{N}$ blocks, providing a high probability for writing a block into the top level which never needs upper moving, but other level is designed to contain total 2^l blocks whatever real or dummy; moreover, a special level (denote as level #) with one block is below the level 0, for adding a dummy block before writing a block.

1.4 Our Contributions

We propose MiniORAM, which can minimize storage size (both server and client) and reduce the worst-case bandwidth. Both the storage size and the bandwidth are the smallest among all known partition-based ORAMs. Table 1 summarizes our results: (i) server storage is reduced from $4.6N$ to $2.8N$; (ii) network bandwidth is reduced from $O(B\log N)$ to $O(B)$; and (iii) we eliminate the shuffle buffer in the client side.

Technically, MiniORAM adopts the upper moving to replace shuffle operation, combines the PIR to design the whole ORAM operations and exploits the hidden-shuffle operation (an innovative usage of PIR technique) to shuffle a real block in a normal data access (when writing a dummy block). The security analysis shows that leakages caused by upper moving operation have no influence on oblivious data access. Through setting the block size to $\widetilde{\Omega}(log^4 N)$, the bandwidth can be bounded to $O(B)$. That is to say, MiniORAM achieves constant bandwidth.

From the view point of implementation, we parallelize the PIR implementation to improve its performance, and provide two versions of implementation based on Paillier cryptosystem. Experiments show that our C version with the support of OpenMP has a much better performance, which improves the speed of PIR-reading or PIR-writing by at least 600×. The source codes are available in GitHub: *https://github.com/emigrantMuse/MiniORAM.*

2 PRELIMINARIES

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

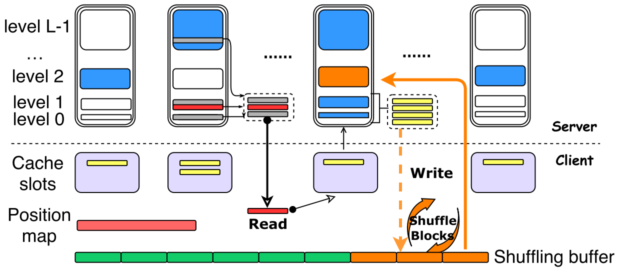

Figure 1: the framework of partition-based ORAM.

2.1 Partition-based ORAM

2.1.1 Storage structure. Figure 1 shows the framework of a partition-based ORAM (SSS [7] is a typical example). It can support concurrent reads or writes on different partitions.

Server Storage. Let N be the number of real blocks stored in an ORAM. We divide the blocks into $\sqrt{N}$ partitions, each has about $\sqrt{N}$ real blocks. In each partition, there are $L := \log\sqrt{N}+1$ levels with 2^{i+1} blocks, including at most 2^i real blocks and at least 2^i dummy ones $i \in \{0, 1, ..., L-1\}$. In practice, blocks are randomly distributed to the partitions. a partition may contain slightly more than $\sqrt{N}$ real blocks, so the top level usually is allocated more space (in our case, we allocate enough storage for $2\times\sqrt{N}$ blocks) to make the storage failure possibility negligible [7].

Client Storage. There exist evict cache, shuffling buffer and position map. In order to defend linkability attack, the evict cache temporarily stores fetched real blocks from the server. It consists of $\sqrt{N}$ cache slots in total, equal to the number of partitions on the server. The shuffling buffer is to rearrange the data blocks when consecutive filled levels inside a partition ORAM need to be merged into the next level. It can be up to capacity of $O(\sqrt{N})$. The position map is to store the location of data block as a tuple of $(u,\ p,\ l,\ o)$, u is the block identifier, p is the block corresponding partition, l and o is the corresponding level and offset within the level l in the partition.

2.1.2 Access operations. Data access in a partition-based ORAM involves three operations, i.e., *read*, *write* and *evict*. Note that *shuffle* is done inside the write operation. Take SSS as an example which are shown in Figure 1.

Read. Assume that the interested data block is block u in partition p, the client will fetch an unread dummy block from each filled level except the level with block u. If the

Table 2: ORAM parameters and notations

Notation	Meaning	Notation	Meaning
N	Number of real blocks	B	Size of data blocks in ORAM
L	Number of levels	$\mathsf{E}(x)$	The encryption of x by AHE
S	Evict cache in the client	$\mathsf{D}(x)$	The decryption of x by AHE
S_i	The i-th cache slot	d_x	The data in position x
u	The identifier of a block	$\oplus$	The homomorphic addition operation
R_v	Set of locations of requested blocks and vectors in PIR-reading	$*$	The repeated $\oplus$ operations
W_v	Set of locations of requested blocks and vectors in PIR-writing	$E(x,r)$	The symmetric encryption of x
$position[u] = (p, l, o)$	p: the partition of u; l: the level of u; o: the offset of u	$D(x,r)$	The symmetric decryption of x

block u is found in the cache slots, one dummy block is read from each level.

Write. Assume that we need to write a data block u back to a partition. Let 0,1, $\cdots$,l denote consecutively filled levels, and thus Level l+1 is empty. In the case, the client will first download all data blocks from Level 0 to Level l into shuffle buffer in the client and execute reshuffling operation with re-encryption and permutation. After this, the client writes them back to level l+1 in the server. If there exists no consecutive filled level, block u will be written to Level 0.

Evict. The client puts the fetched real blocks in evict cache without writing it back immediately to defend the linkability attack. Like in SSS, the block will be stored in a random cache slot corresponding to a partition until the eviction process evicts it back to the server.

2.2 Security Definition

We adopt standard ORAM security definition. Informally, the server should not learn anything about: 1)which data the client is accessing; 2) how old it is (when it was last accessed); 3) whether the same data is being accessed (linkability); or 4) whether the access is a read or a write. Like previous work, we do not consider information leakage through the timing channel, such as when or how frequently the client makes data requests.

Definition 2.1. [**Data Access**]: Let $A(\vec{y})$ be the possibly randomized accesses pattern to the remote storage which is depend on sequence of data requests $\vec{y}$.

Let $\vec{y}$ denote the set of $(op_i, u_i, data_i)$, assume that i = $1, ..., n$, that means the data request sequence of length n, where op_i denotes a read(u_i) or a write(u_i, $data_i$) operation, u_i denotes block identifier and $data_i$ denotes data to be written.

Definition 2.2. [**Security Definition**]: An ORAM construction is said to be secure if (1) For any two data request $\mathtt{y}$ and $\mathtt{z}$ of the same length, their access pattern $A(\mathtt{y})$ and $A(\mathtt{z})$ are computationally indistinguishable by anyone but the client, (2) the ORAM construction is correct in the sense that it returns correct results for any input $\mathtt{y}$ with probability $\leq 1 - negl(|\mathtt{y}|)$.

3 SPECIAL USAGE OF LINEAR-PIR

In this section, we develop some new usages of PIR, which can be applied in the design of MiniORAM. First, we review the definition of Linear-PIR in Kushilevitz and Ostrovsky [15].

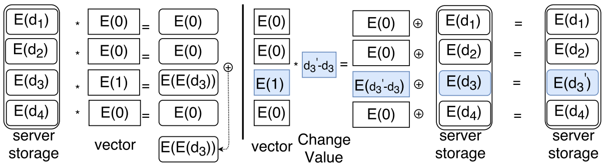

Figure 2: The specific process of PIR operations. (The left one shows how to read the required block d_3; the right one writes a new value d_3' to replace the old d_3 value.)

3.1 Additively Homomorphic Encryption

We present the basic definitions related to additively homomorphic encryption in this section.

Let $M(resp., C)$ be the set of plaintexts $(resp., ciphertexts)$. An encryption scheme is said to be additive homomorphic if for any given encryption key k, the encryption function E and corresponding decryption function D satisfy that $\forall m_1, m_2 \in M, \exists \oplus, s.t.$

$$\mathsf{D}(\mathsf{E}(m_1) \oplus \mathsf{E}(m_2)) = m_1 + m_2 \tag{1}$$

where $\oplus$ is an efficiently computable function.

Also, based on the above property, $\forall m_i \in M, \exists *, s.t.$

$$\mathsf{E}(\mathsf{E}(m_1)) = \mathsf{E}(m_1) * \mathsf{E}(1)$$

$$\mathsf{E}(0) = \mathsf{E}(m_1) * \mathsf{E}(0)$$

Note that, $*$ is repeated operations of $\oplus$.

3.2 Linear-PIR and its Operations

Path-PIR [25] introduces Linear-PIR, which can be constructed by using an IND-CPA *additively* homomorphic encryption scheme. In Linear-PIR, there are two main operations: PIR-reading and PIR-writing.

PIR-reading. To fetch a block whose position is x in the server, it can hide the fetched block by following steps:

(1) *PrepareRead(x)*: The client generates a vector $Q = \{q_1, q_2, \dots, q_n\}$, where $q_x = \mathsf{E}(1)$, and $q_i = \mathsf{E}(0), \forall i \neq x$.

(2) *ExecuteRead(Q)*: After receiving the vector Q, the server first computes: $sum = \sum_{i=1}^{n} q_i * \mathsf{E}(d_i)$, where $\mathsf{E}(d_i)$ is the data block stored in the selected block for the read operation. Next, the server returns sum to the client. The whole process is shown in the left diagram of Figure 2. Notice that the block we want is twice encrypted and stored in sum.

(3) *DecodeResponse(sum)*: The client decrypts *sum* twice to get the target block, that is, $d_x = \mathsf{D}(\mathsf{D}(sum))$.

PIR-writing. To write a block into the position x in the server, it can hide the target position by following steps:

Step 1: Read the old value in position x;

The client performs a *PIR-reading* operation to fetch the stored data d_x from position x. Because the position x of fetched block will be written a new block, we denote the fetched d_x as *OldValue*.

Step 2: Generate the change value and PIR vector;

To update the value in the position x to a new value y, the client first computes the *ChangeValue* as $y' = y - d_x$. Then, the client generates a vector $Q = \{q_1, q_2, ..., q_n\}$, where $q_x = \mathsf{E}(1)$, and $q_i = \mathsf{E}(0), \forall i \neq x$. Finally, the client sends the vector Q and *ChangeValue* (i.e., y') to the server.

Step 3: Compute the ciphertexts and update the block.

After receiving the vector Q and y', the server first generates a new vector $Q' = \{q'_1, q'_2, ..., q'_n\}$, where $q'_i = q_i * y', \forall i \in \{1, 2, ..., n\}$. Then, the server perform $\oplus$ operation by q'_i and the data stored in the corresponding position. Finally, the server updates the results. The whole process is shown shown in the right diagram of Figure 2.

3.3 Hidden-shuffle Operation

A hidden-shuffle operation is embedded in the operation of writing a dummy block. We fetch a *real* block in *PIR-reading* step. This real block is written to another position where a dummy block is stored, resulting in reshuffling this real block to somewhere else.

Initial setup. Hidden-shuffle operation requires that all dummy blocks are encryption of a constant value c chosen by the client, in our case $c = 0$. However, after setting c as 0, the change value will be real data when writing to a position of dummy block. To prevent this leakage, it further requires the real block to be encrypted by a symmetric encryption (like AES) with a random value before uploading. For convenience, denote $E(x, r)$ as the symmetric encryption of x with a random value r.

Operation details. Hidden-shuffle operation is a special usage of PIR-writing, so that it also contains three steps:

Step 1. The client randomly selects a set of positions $pos = \{p_1, p_2, ..., p_n\}$, and fetches the data of a real block in position $p_r \in pos$ by performing a *PIR-reading* operation. In fact, the client will get the symmetric encryption of d_r with a random value r_0, that is, $E(d_r, r_0)$.

Step 2. The client chooses a position $p_d \in pos$ containing a dummy block $\mathsf{E}(c)$. Next, it decrypts the fetched value to get the plaintext d_r, encrypts it with a new random value r_1 to get $E(d_r, r_1)$, and computes the *ChangeValue* as $y' = E(d_r, r_1) - c$, and. Then, the client generates a vector $Q = \{q_1, q_2, ..., q_n\}$, where $q_r = \mathsf{E}(1)$, and $q_i = \mathsf{E}(0), \forall i \neq r$. Finally, the client sends the vector Q and *ChangeValue* (i.e., y') to the server.

Step 3. After retrieving Q and y', the server first generates a new vector $Q' = \{q'_1, q'_2, ..., q'_n\}$, where $q'_i = q_i * y', \forall i \in \{1, 2, ..., n\}$. Then, it computes $\mathsf{E}(d_i) \leftarrow q'_i \oplus \mathsf{E}(d_i)$, where $q'_i \in Q'$ and $p_i \in pos$. The data in position p_r and p_d will be changed as following:

$$\begin{cases} p_r : \mathsf{E}(E(d_r, r_0)) \to \underset{\substack{\downarrow \\ \text{OldValue}}}{\mathsf{E}(E(d_r, r_0))} \oplus \mathsf{E}(0) = \underset{\substack{\downarrow \\ \text{NewValue}}}{\mathsf{E}(E(d_r, r_0))} \\ p_d : \mathsf{E}(c) \to \quad \underset{\substack{\downarrow \\ \text{OldValue}}}{\mathsf{E}(c)} \quad \oplus \mathsf{E}(E(d_r, r_1) - c) = \underset{\substack{\downarrow \\ \text{NewValue}}}{\mathsf{E}(E(d_r, r_1))} \end{cases}$$

Through the above steps, the real block will be permuted to another location containing a dummy block. The data in position p_r will be deleted logically.

4 MINIORAM SCHEME

In this section, we present the details of our proposed layer-based partition, MiniORAM. We first describe its server and client storage structures, and then its data access operations illustrated by Figure 3.

4.1 Storage structure

Three kinds of data block are used in MiniORAM: real block, noisy block (the logically deleted blocks) and dummy block. Notice that every dummy block will be initialized as a ciphertext of zero encrypted by additively homomorphic encryption scheme, which is the basis of our special PIR operations described in Section 3.

Server storage. MiniORAM also divides the whole O-RAM into $\sqrt{N}$ partitions. However, it redesigns the storage structure of each partition to reduce the number of dummy blocks, from the following aspects: 1) set the number of levels to $L := \llcorner \log \sqrt{N} \lrcorner$; 2) set the capacity of level l ($0 \leq l \leq L-1$) to 2^l, containing all the dummy, noisy or real blocks; 3) set the capacity of top level to $2.3\sqrt{N}$ (although each partition is supposed to hold $\sqrt{N}$ data block, in reality, since the distribution is probability, according to [7], the number of real blocks can be up to $1.15\sqrt{N}$, we follow [7] to allocate double for the top level). The top level will be filled up as dummy blocks initially.

MiniORAM also has a special level # with only one dummy block under the level 0. This level is used to add a dummy block when writing a block.

Client storage. It maintains an evict cache, position map, but removes the shuffle buffer which saves about $2.3\sqrt{N}$ (in the worst case) in the client storage. The evict cache and position map are the same as before.

For convenience, we denote *position*$[u]$ as the tuple of (u, p, l, o), which denotes the position of block u in the position map. When block u is in the evict cache, the *position*$[u]$ would be like (u, -1, -1, o), in which o is the real position in the S. In order to identify the dummy blocks or real ones, a list of blockIDs for every partition is defined in MiniORAM. The blockIDs can simply implement the function of choosing the dummy, real or noisy blocks of a level.

```
Access(op, u, data*):
$p \leftarrow position[u].p$
$r \leftarrow UniformRandom(1, ..., \sqrt{N})$
 //randomly select empty place to store fetched block
if block $u$ is in $S_p$, then
    $data \leftarrow S_p.ReadThenDelete(u)$
    $ReadPartition(p, \perp)$
else
    $ReadPartition(p, u)$ //the block u is logically deleted
if $op = write$, then
    $data \leftarrow data*$
$WriteCache(S_r)$
$evict(v)$ //the same as other partition ORAMs
Return data
```

Figure 3: Algorithms for Access.

```
ReadPartition(p, u):
Client : if u=⊥, then
             $(Null, Null, Null) \leftarrow position[\perp]$
         else
             $(p, l, o) \leftarrow position[u]$
         $R_v \leftarrow PrepareRead(p, l, o)$
Server : $R \leftarrow executeRead(R_v)$
Client : if (p, l, o) exists in the server-side, then
             $DecodeResponse(R)$
         else
             Return ⊥
```

Figure 4: Algorithms for ReadPartition.

Initialization. The top level in a partition will be set filled initially. It is no need to write actual blocks to the server because all the blocks in the top level can be zeroed. Given the block size $\widetilde{\Omega}(\log^4 N)$, the bandwidth between the client and server is $O(B)$, the scheme achieves constant bandwidth.

4.2 Access Operations

Similar with the traditional ones, MiniORAM preserves the main steps including *ReadPartition*, *WriteCache* and *Evict* (shown in Figure 3). In MiniORAM, these operations are designed based on PIR, their details will be shown in the rest of this section. For the consideration of security, access operations in MiniORAM have two invariants:

(1) The block positions selected to be written on the server should be at least two unreal locations when writing a real block to the server.

(2) The blocks selected to be read had better at least two real blocks during reading real interested block from the server.

4.3 Reading from a Partition

Comparing with original schemes that downloads several blocks to hide an interested block, MiniORAM makes use of *PIR-reading* to produce only one block (shown in Figure 4), reducing the bandwidth from $O(\log N)$ blocks to $O(1)$ blocks.

```
PrepareRead(p,l*,o*):
$L \leftarrow$ the total number of levels
$m \leftarrow 0$
for level $l$=0 to $L$-1 in partition $p$ do
    if level $l$ is unfilled, then
        continue
    if (p, l*, o*) exists in server-side, then
        $R_v[m++] := (p, l*, o*, E(1))$
    else
        $o \leftarrow random()$//select a random offset in p
        $R_v[m++] := (p, l, o, E(0))$
```

Figure 5: Algorithms for PrepareRead.

First, the client runs the *PrepareRead* to generate a PIR vector R_v. As shown in Figure 5, it generates an element $\mathsf{E}(0)$ for a random block of a filled level in partition p. But for the real block u in given level, it generates an element $\mathsf{E}(1)$. As a result, when reading a dummy block from p, R_v only contains $\mathsf{E}(0)$.

Then, the server performs $executeRead(R_v)$ described in section 3.2, products a single block R and sends it back to the client.

Finally, The client decrypts R twice to get the stored value. For a real block, the client must run a symmetric decryption to retrieve the plain data. If the interested block is in evict cache, R is just a dummy block result that doesn't need to be decrypted.

4.4 Writing to a Partition

Comparing with original schemes that downloads and reshuffles blocks in consecutive filled levels, MiniORAM combines PIR-writing into write operation, and embeds a hidden shuffle operation for a real block in a normal write operation (when writing a dummy block). However, from the point of view of the server, it can not distinguish what type of block is written.

As shown in Figure 6, we define *WriteParition* function as *AddDummy*, *WriteReal* or *WriteDummy*, and *Upper_moving*. To better present its work flow, we use the following steps for presentation. We also illustrate these five steps in Figure 7.

Step 1: Add a dummy block;

The *AddDummy()* is used to produce a dummy block in the level #, to ensure that there is at least one location we can write.

Step 2: Read a block from the partition;

Recall that a PIR-Writing operation must first read a block u by *PIR-reading* from the partition. MiniORAM deals with the dummy block and real block writes differently, so the choices of u are different as follow:

- *WriteReal*. The location for u is selected as the position of a dummy or noisy block.
- *WriteDummy*. The location for u is selected as the position of a random real block.

```
WritePartition(p ,u*, data*):
AddDummy();
if u* is real block, then
    WriteReal: write real block u* to p
else
    WriteDummy: write dummy block u* to p
Client : if there exist consecutive levels in p,
         send the last consecutive level $l_0$ to server, then
Server : Upper_moving
WriteReal:
Client :  Read the list of blockIDs;
          $(p, l', o') \leftarrow RandomDummyorNoisy()$
          // randomly choose place of dummy or noisy block
          $R_v \leftarrow PrepareRead(p, l', o')$//level # is considered.
Server : $R \leftarrow ExecuteRead(R_v)$;
Client : $OldValue \leftarrow DecodeResponse(R)$.
          $ChangeValue \leftarrow data* - OldValue$;
          $W_v \leftarrow PrepareWrite(p, l', o', ChangeValue)$;
Server : $ExecuteWrite(W_v)$;
```

```
WriteDummy:
Client :  Read the list of blockIDs;
          // randomly choose locatition of real data block
          $(p, l', o') \leftarrow RandomReal()$
          $R_v \leftarrow PrepareRead(p, l', o')$ //level # is considered.
Server : $R \leftarrow ExecuteRead(R_v)$;
Client : $OldValue \leftarrow DecodeResponse(R)$.
Client :  Read the list of blockIDs
          $(p, l'', o'') \leftarrow RandomDummy()$
          $ChangeValue \leftarrow OldValue - data*$
          $W_v \leftarrow PrepareWrite(p, l'', o'', ChangeValue)$
Server : $ExecuteWrite(W_v)$;
Upper_moving:
o*=0;
for (l=$l^*$ , 0 to $l_0$)
    //contain consecutive levels including level $l*$
    for (o=0 to $2^i$, the $o$ in level # has only a value 0. )
        $BlockAt(p, l, o) \rightarrow BlockAt(p, l_0 + 1, o*++)$
    //all blocks in consecutive levels are moved to level $l_0 + 1$.
```

Figure 6: Algorithms for WritePartition.

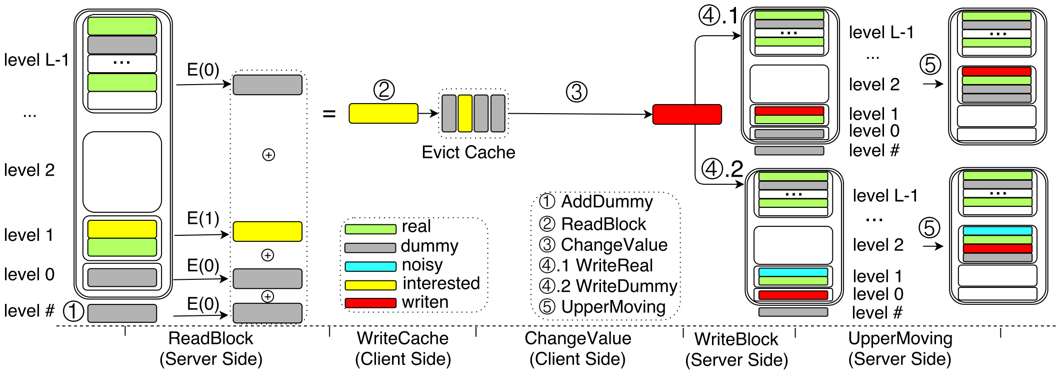

Figure 7: WritePartition detailed operations

After reading the block u in position x, if it is real, it will be decrypted to get stored data *OldValue* (i.e., $E(d_x, r_0)$, a symmetric encryption of d_x with a random value r_0). Otherwise, it will not be decrypted.

Step 3: Compute the *ChangeValue*;

- *WriteReal.* The client computes *ChangeValue* by $NewValue - OldValue$, where $NewValue = E(d'_x, r_1)$, d'_x is the new value of block u, r_1 is a random value.
- *WriteDummy.* The client decrypts the original real data d_x and computes *ChangeValue* by $NewValue - 0$, where $NewValue = E(d_x, r_1)$. In this case, the hidden shuffle operation wants to write d_x into a position of dummy block whose value d_{dummy} is 0.

Step 4: Write the data back;

- *WriteReal.* The server writes *NewValue* to the position x by the addition of *ChangeValue* and *OldValue*.
- *WriteDummy.* The server writes the block u to a fresh chosen dummy location by the addition of *ChangeValue* and d_{dummy}. The original location of x will be identified to own a noisy block.

Step 5: Move blocks up to the next level.

If there exist consecutive filled levels in the target partition, the client will inform the server to execute *Upper_moving*, to move blocks in consecutive levels to the next unfilled level.

However, in the worst case, *Upper_moving* operation cannot be done directly because of non-empty blocks in the top level. MiniORAM adopts a *overlay* mechanism to solve this problem. As we know, there are total $\frac{\sqrt{N}}{2}$ blocks in the levels under top level. In this case, the client will select $\frac{\sqrt{N}}{2}$ positions of noisy or dummy blocks in the top level, send these positions to the server, then notice the server to overlay blocks in these positions as the blocks moved from the bottom levels.

5 LEAKAGE AND SECURITY ANALYSIS

The upper moving operation is very simple and mainly dependent on the server operation, it will leak two leakages including 1) position relation leakage of a block; and 2) block attribute leakage. With the aspect of both leakages, MiniORAM makes use of PIR technique and special design of data access pattern to guarantee oblivious data access.

5.1 Position Relation Leakage

Leakage. For *upper moving* operation, the before and after block position can be related to each other on the server side. The location of special block cannot be exposed for long time including position relation of it, because it can be recorded by its access frequency. According to Theorem 1, the position relation can be broken naturally by reads and writes in MiniORAM.

Theorem 1. *Given data access sequence* $\overrightarrow{b} = (b_1, ..., b_m)$*, position access sequence* P=\{*position*$[b_1]$, ... , *position*$[b_m]$\}, $\overrightarrow{b}$ *is produced by operation op, where* b_i *is data block for* $op \in \{Read, WriteDummy, WriteReal\}$ $(\forall i \in \{1, ..., m\})$*. The probability that the server can recover* $\overrightarrow{b}$ *is negligible.*

Proof. Remark that x denotes the number of filled levels for accessed partition ($x \geq 2$), $\Pi(s)$ denotes ORAM protocol designed in MiniORAM, s is security parameter. $Pr(position[b_i])$ denotes the possibility of recognizing position for b_i. $Pr[A(b_i)]$ denotes the possibility of recognizing access operation for which block, whether is b_i or not by PPT adversary.

During the upper moving, supposed that x' denotes the number of consecutive filled levels for the accessed partition. P'=\{*position*$[a_1]$, ... , *position*$[a_n]$\} denotes location access sequence, $\overrightarrow{a} = (a_1, ..., a_n)$ denotes data access sequence for upper moving ($n = 2^{x'}$). The position change can be gotten easily for PPT adversary.

$$\sum_{i=1}^{n} |\Pr(position[a_i])| = n$$

where one upper moving can constructs $O(\log N)$ location relations after $O(\log N)$ data accesses.

Security guarantee. Reads and writes can protect which block is interested. Even if one location is always accessed, the adversary can get no useful information such as any interested block b_i access frequency. Because each time *WriteDummy* obliviously shifts the location for a real block, *WriteReal* and *Read* can be combined to obliviously re-encrypt the value of fetched real block and re-distribute matching partition randomly. So after $O(\log N)$ accesses, writes and reads have been executed at least $O(\log N)$ times. The block relation can be efficiently broken.

When $b_i = b_j$ $(\forall i, j \in \{1, ..., m\}, i \leqslant j)$, that is to say, the client accesses block a twice in the sequence.

1) If $op = Read$, after the client accesses a at time i from partition p,

$$\Pr(position[b_i] | b_i.op = Read) = \frac{1}{x} \leqslant \frac{1}{2}$$

a will be written into the client cache and evicted to a random partition p'. The p' is independent of p. So the server cannot recognize that it can access the same block twice.

$$|Pr[A(b_i = b_j)]| \leqslant negl(s)$$

For writes, *WriteDummy* can be set to happen with the same possibility of *WriteReal*. Both the operations cannot be distinguished by the adversary.

2) If $op = WriteReal$, position$[b_i]$ can be hidden naturally by PIR-reading,

$$\Pr(position[b_i] | b_i.op = WriteReal) = \frac{1}{x+1} < \frac{1}{2}$$

The b_i and b_j cannot be also recognized by PPT adversary because of the random mapping to independent location in client cache slot which is described in [7], and the eviction sequence cannot leak any information for the written block, because it looks random or sequential. So,

$$|Pr[A(b_i = b_j)]| \leqslant negl(s)$$

3) If $op = WriteDummy$, the hidden shuffle operation can be executed in the process. The aim of the operation is to shift the real block position hiddenly. At the time i, the b_i is shuffled with the change to the value of position$[b_i]$. At the time j, the b_j is shuffled to change the value of position$[b_j]$. But it can not be ensured that the two accesses are for the same block.

$$\Pr(position[b_i] | b_i.op = WriteDummy) \leqslant \frac{1}{2}$$

When $b_i \neq b_j$, it is easy to see that the access patterns of b_i and b_j are independent since they are mapped into two partitions randomly and independently.

So,

$$\begin{aligned} \Pr(P) &= \Pr(\bigcap_{i=1}^{m} position[b_i]) \\ &= \prod_{i=1}^{m} \Pr(position[b_i]) \\ &\leqslant \prod_{i=1}^{m} (\frac{1}{2}) \leqslant \frac{1}{2^m} \end{aligned}$$

Thus, a PPT server cannot guess the request sequence from the access pattern. □

5.2 Block Attribute Leakage

Leakage. In the worst case that all the levels are filled, upper moving cannot be normally executed. It needs overlaying $\frac{\sqrt{N}}{2}$ dummy and noisy positions by the blocks in bottom levels. The bottom levels will be empty again to keep the partition perform normally. The $\frac{\sqrt{N}}{2}$ dummy and noisy positions are leaked but it can have no influence on the oblivious data access. According to Theorem 2, the invariants of writes and reads as well as read or write operation can prove no leakage of real block information.

Theorem 2. *Let* $\Pr([{b_i}^{op}_{pos}])$ *be the probability that the adversary guesses it right whether the positions accessed by executing op related to block* b_i *is real or not;* $\Pr[\overrightarrow{b}]$ *be the probability that an adversary is able to guess which block is real and interested by the client (data access sequence* $\overrightarrow{b} = (b_1, ..., b_m)$*).*

$$\Pr[\overrightarrow{b}] \leq \frac{1}{2^m} + negl(m)$$

Proof. All the positions in all filled levels are chosen to look random and independent no matter reads or writes. The top level can contain at most $\sqrt{N}$ real blocks and at least $\sqrt{N}$ dummy blocks in theory. Remark that x denotes the number of filled levels for accessed partition ($x \geq 2$).

Because of overlaying the dummy and noisy position on the top level, some blockIDs will be leaked and changed. But after this, the top level still keep the same contain limitation to at most $\sqrt{N}$ real blocks in theory. So,

$$\Pr([{b_i}^{op}_{pos}]) \leq \frac{1}{2}$$

1) If $op = WriteReal$. Because of **Invariants.(1)** in Section 4.2, the written real block has at least two unreal position choices to evict.

$$\Pr([{b_i}^{op}_{pos}]) \leq \frac{1}{2}, i \in \{1, ..., m\}$$

$$\Pr([{b_1}^{op}_{pos}], ..., [{b_m}^{op}_{pos}]) = \prod_{i=1}^{m} \Pr([{b_i}^{op}_{pos}]) \leq \frac{1}{2^m}$$

2) If $op = Read$, assume that $(x_1, ..., x_m)$ are produced by *PIR-Writing* in *WriteReal* and *WriteDummy*, obviously,

$$\Pr[(x_1, ..., x_m)] = \prod_{i=1}^{m} \Pr([{x_i}^{op}_{pos}]) \leq \frac{1}{2^m}$$

For *PIR-Reading*, because of **Invariants.(2)** in Section 4.2, even in the overlaying operation with the leakage of the dummy and noisy location, it can not get the historical interested block, because of at least two real blocks accessed at the same time. The location of data access sequence in the PIR-Reading process $(y_1, ..., y_m)$ is the same as *PIR-Writing*, so

$$\Pr[(y_1, ..., y_m)] = \Pr[(x_1, ..., x_m)] = \prod_{i=1}^{m} \Pr([{y_i}^{op}_{pos}]) \leq \frac{1}{2^m}$$

Therefore, for any access sequence $(b_1, ..., b_m)$, the server cannot know whether b_i is a real block or not. The overlaying of dummy and noisy blocks will not leak any information. □

5.3 Security Analysis

The scheme security relies on three aspects: (1) data read can hide data information; (2) data write can hide data information; (3) Evict operation can defend the linkability attack, so the security can be guaranteed.

Theorem 3. *MiniORAM cannot leak any useful information to keep data access oblivious.*

Proof. The eviction is the same as SSS [7], which can guarantee the defence to the linkability attack. Based on Theorem 1, shuffling can be executed hiddenly. The position relation can be broken in MiniORAM. The adversary can not recognize any useful data blocks in MiniORAM. Based on Theorem 2, the accessed blocks can not be recognized which is real. Even some dummy and noisy blocks are exposed after several accesses, it can not help adversary to get the position of interested blocks for historical accesses.

Therefore, the useful information including the position, access time and frequency of interested block can not be leaked so that MiniORAM can keep the security. □

6 ANALYSIS AND COMPARISON

In order to highlight the advantages of MiniORAM, we mainly compare MiniORAM with SSS [7] and OnionORAM [21], which are taken as the examples of partition-based ORAM and tree-based ORAM.

6.1 Server Storage

The construction of MiniORAM is consist of the top level and bottom levels. The top level has $2.3\sqrt{N}$ data blocks. Since our PIR operation will rewrite dummy blocks on each visit, these blocks can be reused. We only require $0.5\sqrt{N}$ data block storage for bottom levels. Thus, our server storage for each partition is $2.8\sqrt{N}$, that is the overall storage is $2.8N$.

In SSS [7], the maximum size of each partition is $4.6\sqrt{N}$ to guarantee that the failure probability is negligible. Therefore, the overall storage is $4.6N$, which is about twice that of MiniORAM.

OnionORAM [21] has bounded feedback ORAM's server storage, which is $O(2^{L+1} \cdot Z \cdot B) = O(BN)$, where Z is bucket size, L is the length of tree path. The bucket is nodes of a binary tree. Both MiniORAM and OnionORAM [21] maintain server storage of $O(BN)$.

6.2 Block Size

In following analysis of block size in MiniORAM, we define γ as the length of the ciphertext of the additively homomorphic encryption. The whole communication complexity is including *reading from a Partition* and *writing to a Partition.* In *reading from a Partition*, the client sends PIR-vector with size $O(\gamma \cdot L)$ bits to the server and downloads a single block B. The reading operation induces an overhead of $O(\gamma \cdot L + B)$, where L is the number of filled levels as $\Theta(\log N)$. In *writing to a Partition*, the client send PIR-vector with size $O(\gamma \cdot L)$ and a block B with *ChangeValue* to the server. Then server generates the ciphertext of vector can meet the requirement of addition of homomorphic encryption by . Due to acquiescent reading operation before writing operation, the overhead in writing is $O(\gamma \cdot L + B + \gamma \cdot L + B)$.

To have constant communication complexity in B, the block size should be $B \in \Omega(\gamma \cdot L) \in \Omega(\gamma \cdot \log N)$, which can absorbed the cost of PIR-vectors asymptotically.

In terms of SSS [7], it do not adopt PIR technique so that its block size is unlimited. In practice, the block size is $B > \frac{c}{8} \log N$, where $c \leq 1.1$. Even MiniORAM has larger block size than SSS [7], other attributes are more important than this in the overall performance comparison.

In OnionORAM [21], due to the block size must be asymptotically larger than the select vectors, it is set to $B = \Omega(\gamma \log^2 \lambda \log^2 N)$ bits. Based on current attacks [17], $\lambda = \omega(\log N)$ and $\gamma = O(\lambda^3)$. Therefore, MiniORAM has smaller block size $\widetilde{\Omega}(\log^4 N)$ than $\widetilde{\Omega}(\log^5 N)$ in OnionORAM [21].

6.3 Network Bandwidth

MiniORAM achieves constant network bandwidth. A mentioned before, the network bandwidth of reading operation is $O(\gamma \cdot L + B)$ and writing operation is $O(\gamma \cdot L + B + \gamma \cdot L + B)$. Considering *Upper_moving* operation, there is no extra bandwidth between client and server except some position information about where the blocks should be move to. Normally, it can be ignored. Since we set $B \in \Omega(\gamma \cdot \log N)$, we achieve constant communication overhead of $O(B)$.

In SSS [7], the bandwidth required is $O(\log N \cdot B)$ for a read operation. For a write operation, the bandwidth depends on whether reshuffling is needed or not. When there exist consecutive filled levels (reshuffling), the client downloads all blocks of those levels. The bandwidth at i-th access in an initial empty partition is as follows:

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

So the total 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 buffer and returns these padding fresh dummy blocks, it is up to $O(\sqrt{N} \cdot B)$. The SSS [7] requires larger network bandwidth than MiniORAM needs.

As in OnionORAM [7], the cost of homomorphic select vectors could be asymptotically absorbed only if OnionORAM [7] sets large block size in $\widetilde{\Omega}(\log^5 N)$. For that, OnionORAM achieves constant bandwidth blowup in $O(B)$. Even OnionORAM [7] has constant bandwidth in $O(B)$ as same as MiniORAM, MiniORAM has lower network bandwidth according to smaller block size.

7 EXPERIMENTS

Until now, there are some ORAM schemes based on PIR. However, few of them have provided their implementation details. To our knowledge, the PIR implementation is always based on the additively homomorphic encryption. Moreover, the block size in ORAM is always larger than the key size of homomorphic encryption. Therefore, ORAM based on PIR seems impractical.

ORAM implementations. We implemented both MiniORAM and SSS [7] in Java. We utilized MongDB to store and access data blocks, like in SEALORAM library [29]. We created a collection for each partition, initialized each block as AHE encryption of 0 by Paillier Cryptosystem, so that ORAM data accesses will be *updated* operations in MongDB. To encrypt a data block with a long length, we divide it into some sub blocks, the length of which is less than the key size of Pailier. We further set all operations to be executed in a certain partition. The source codes can be found in GitHub: *https://github.com/emigrantMuse/MiniORAM*.

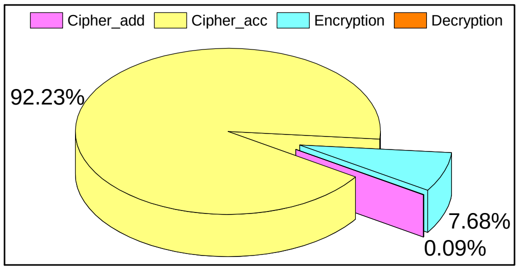

Figure 8: The percentage of execution time of four cryptographic operations of the overall execution time.

Basic Test Unit. To provide the same execution sequence and ensure the cycling occurrence of the worst case, we write $\sqrt{N}$ blocks with ID from 0 to $\sqrt{N}$ into a chosen partition. This is defined as a basic test unit in our experiments.

Experiment environment. We focus on the performance of ORAM operations, while not the bandwidth, so that the following parameters are chosen: $N = 102400$, block size=2048 byte and key size=128 byte. All our experiments are performed on Computer of Intel(R) Core(TM) i7-7500U CPU @ 2.70GHZ with Ubuntu OS, 16GB memory and 8 threads for each CPU.

7.1 Cryptographic Operations in MiniORAM

In MiniORAM, there are four cryptographic operations (encryption, decryption, cipher_add ($\oplus$) and cipher_accumulation ($*$) simplified as cipher_acc) and one database operation (upper_moving). In comparison of cryptographic operations, the execution time of *upper_moving* operation can be ignored. We focus on the execution times of the cryptographic operations.

The results are shown in Figure 8, 9, 10 and Table 3. Figure 8 shows the overall percentage of execution time of the four operations in a data access. Figure 9 and 10 show the execution time of different number of data accesses in the server side or the client side. Table 3 shows the execution frequencies of different operations.

From Figure 10, for the *client* side, encryption is the most time consuming step, contributing about 99% of the overall execution time of all cryptographic operations (note that y-axis is in log scale). All the operations except encryption can be ignored. While in the server side (see Figure 9), the cipher_acc operation is the most time consuming step, contributing about 99% of all cryptographic operations. If we consider *all* operations, for the server side, cipher_acc is still the most time consuming operation, contributing about 92% of the total execution time, which is 13× more than the second time-consuming operation (encryption).

Table 3 shows the frequencies of different cryptographic operations in the scenario with 5,000 data accesses. In the

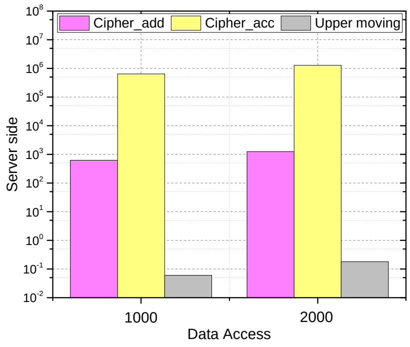

Figure 9: Comparison of execution time in server side.

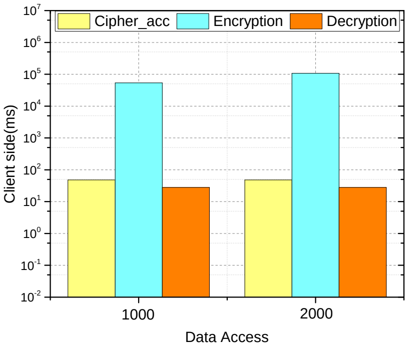

Figure 10: Comparison of execution time in client side.

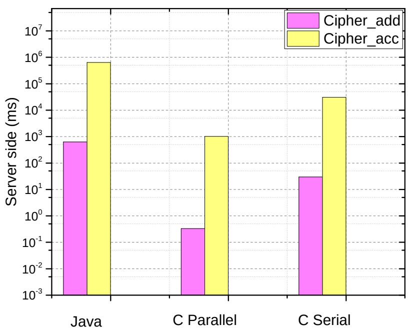

Figure 11: Comparison of execution time of C and Java.

Table 3: Frequencies of crypto operations.

Data access	Server side		Client side		
	Cipher_add	Cipher_acc	Cipher_acc	Encrypt	Decrypt
5000	2×10^{6}	2×10^{6}	1×10^{0}	4×10^{4}	2×10^{0}

server side, the number of operations of cipher_add and cipher_acc are similar, however recall that we cannot compute cipher_acc operation in one step, we call cipher_add many times to compute cipher_acc (since we only use additively homomorphic encryption, but not fully homomorphic encryption), thus it becomes the bottleneck of all operations. For the client side, it is clear that the number of encryption operations is more than the others, which supports our results. Since the cipher_acc operation is the most time-consuming operation in the server, how to improve cipher_acc is the key to make MiniORAM more efficient.

7.2 Improvement of PIR Implementation

Table 4: The executing time of different cryptographic operations in different settings (Java vs C; sequential vs parallel; and two different key sizes). Basic unit: ms per operation.

key size \ operation			Encrypt	Decrypt	Cipher_acc	Cipher_add
1024	Java		12	14	48	0.047
	C	Sequential	0.694	0.724	2.28	0.0002
		Parallel	0.694	0.724	0.076	0.000074
2048	Java		110	125	410	0.57
	C	Sequential	3.466	3.536	14514.176	0.739
		Parallel	3.455	3.536	451.584	0.221

Because the Java version of PIR implementation relies on Java internal implementation of BigInteger, we develop a new version of C code based on the open source library (GMP library), to reduce the execution time of PIR on the server. We imported OpenMP module to parallelize PIR operations. Note that the Java version is sequential.

We test the average execution time of four cryptographic operations in two versions of PIR implementation. From the Table 4, we can see that the key size 1024 has a better performance, which is about 4× that of key size 2048. It is obvious that the cipher_acc is the most time consuming operation. We focus on this operation. The execution time in Java version is about 21× slower than that of the C sequential version and about 631× slower than the C parallel version (network latency is ignored which is time consuming). Note that in the C sequential version, we use an algorithm called *cipher accumulation* (shown in Appendix A) to improve the efficiency of *cipher_acc*. In fact, we parallelize the execution of the *cipher_add* operations to realize *cipher_acc* since *cipher_acc* requires many times of *cipher_add*. The absolute execution time for this operation can be reduced to only 0.076ms in our C parallel version.

7.3 MiniORAM Evaluation

The comparison of execution time of the two implementations (C vs Java) is shown in Figure 11. We can see that in the server side, the Java version is 20× slower than the C sequential version, and 600× slower than the C parallel version and on the client side, the java version is about 17× slower than the C version.

As a result, we can conclude that *cipher_accumulation* costs most of time and in terms of the response time, the Java version is much slower than the C version. The most important point is that every single data access in the C parallel version only costs about 700 ms, which means that it is feasible for MiniORAM to be used in practical applications.

8 CONCLUSIONS

We exploited the power of PIR and created two interesting operations (hidden-shuffle operation and level compression) to be used in our MiniORAM design. With these operations, we are able to reduce the number of dummy blocks used and eliminate the explicit shuffling step resulting in a reduction in the storage (both server and client) as well as bandwidth. From our knowledge, MiniORAM requires the smallest storage and bandwidth among all known layer-based partition ORAMs. With respect to practical performance, we speed up PIR operations using GMP library and OpenMP support. The access time for a real block is only in the range of milliseconds. It is feasible to apply MiniORAM in real applications. We think that our ideas of extending PIR may be useful in solving other problems. For further work, we will try to see if it is possible to further reduce storage and bandwidth.

REFERENCES

[1] Dan Boneh, David Mazieres, and Raluca Ada Popa. 2011. Remote oblivious storage: Making oblivious RAM practical. (2011).

[2] S. Devadas C. W. Fletcher, M. V. Dijk. A secure processor architecture for encrypted computation on untrusted programs. *STC*.

[3] Kai-Min Chung, Zhenming Liu, and Rafael Pass. 2014. Statistically-secure ORAM with\ tilde {O}(\ logˆ 2 n) Overhead. In *Advances in Cryptology–ASIACRYPT 2014*. Springer, 62–81.

[4] J. Perry D. Cash, P. Grubbs and T. Ristenpart. Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation. *CCS*.

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

[6] E. Stefanov E. Shi, T.-H. Chan and M. Li. Oblivious RAM with O(log3(N)) Worst-Case Cost. *ASIACRYPT*.

[7] D. Song E. Stefanov, E. Shi. Towards practical oblivious RAM. *NDSS*.

[8] E. Shi C. Fletcher L. Ren X. Yu E. Stefanov, M. van Dijk and S. Devadas. Path ORAM: an extremely simple oblivious RAM protocol. *CCS*.

[9] Craig Gentry, Kenny A Goldman, Shai Halevi, Charanjit Julta, Mariana Raykova, and Daniel Wichs. 2013. Optimizing ORAM and using it efficiently for secure computation. In *Privacy Enhancing Technologies*. Springer, 1–18.

[10] Oded Goldreich. 1987. Towards a theory of software protection and simulation by oblivious RAMs. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. ACM, 182–194.

[11] Michael T Goodrich and Michael Mitzenmacher. 2011. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *Automata, Languages and Programming*. Springer, 576–587.

[12] Michael T Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. 2011. Oblivious RAM simulation with efficient worst-case access overhead. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*. ACM, 95–100.

[13] Michael T Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. 2012. Practical oblivious storage. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*. ACM, 13–24.

[14] E. Stefanov J. Dautrich and E. Shi. Burst ORAM: Minimizing ORAM response times for bursty access patterns. *USENIX Security*.

[15] E. Kushilevitz and R. Ostrovsky. 1997. Replication is not needed: single database, computationally-private information retrieval. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*. 364–373.

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

[17] Helger Lipmaa. 2005. An Oblivious Transfer protocol with log-squared communication. In *International Conference on Information Security*. Springer, 314–328.

[18] Steve Lu and Rafail Ostrovsky. 2013. Distributed oblivious RAM for secure two-party computation. In *Theory of Cryptography*. Springer, 377–396.

[19] M. Kantarcioglu M. S. Islam, M. Kuzu. Leakage-abuse attacks against searchable encryption. *NDSS*.

[20] R. Ostrovsky O. Goldreich. Software protection and simulation on oblivious RAMs. *JACM*.

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

[22] Elaine Shi, T-H Hubert Chan, Emil Stefanov, and Mingfei Li. 2011. Oblivious RAM with O(log3(N)) Worst-Case Cost. In *Advances in Cryptology–ASIACRYPT 2011*. Springer, 197–214.

[23] E. Stefanov and E. Shi. Multi-cloud oblivious storage. *CCS*.

[24] E. Stefanov and E. Shi. Oblivistore: High performance oblivious distributed cloud data store. *NDSS*.

[25] Agnes Hui Chan Travis Mayberry, Erik-Oliver Blass. 2014. Efficient Private File Retrieval by Combining ORAM and PIR. *NDSS*.

[26] X. Pan X. Wang V. Bindschaedler, M. Naveed and Y. Huang. 2015. Practicing oblivious access on cloud storage: the gap, the fallacy, and the new way forward. *CCS*, 837–849.

[27] P. Williams and R. Sion. Single round access privacy on outsourced storage. *CCS*.

[28] Peter Williams and Radu Sion. 2012. Single round access privacy on outsourced storage. In *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 293–304.

[29] Z Williams, D Xie, and F Li. 2016. Oblivious RAM: a dissection and experimental evaluation. In *Proceedings of the 2012 ACM conference on Computer and communications security*. VLDB, 1113–1124.

[30] C. Liu T. H. Chan E. Shi E. Stefanov Y. Huang X. S. Wang, K. Nayak. Oblivious data structures. *CCS*.

[31] E. Shi X. Wang, H. Chan. Circuit oram: On tightness of the goldreich-ostrovsky lower bound. *CCS*.

[32] S. Pande X. Zhuang, T. Zhang. 2004. HIDE: an infrastructure for efficiently protecting information leakage on the address bus. *ACM SIGPLAN Notices* 39, 11, 72–84.

[33] S. Tople Y. Jia, T. Moataz and P. Saxena. OblivP2P: An Oblivious Peer-to-Peer Content Sharing System. *USENIX Security*.

A CIPHER ACCUMULATION

In this section, we show how to parallelize the cipher_add operations in order to speed up the cipher_acc operation:

$$\underbrace{E(1) + ... + E(1)}_{m} = E(1) \cdot m = E(m)$$

Figure 12 shows how we implement *Cipher_acc* using cipher_add operations. We use *counter* and *ret* to maintain $E(x)$'s power so that we can improve the effectiveness. In order to realize the PIR implementation , we use the technique of repeating squares instead of adding the value with $\mathsf{E}(x)$ times to reduce the number of addition operations.

```
Cipher_acc(c, cipher):
$l \leftarrow c.length$
$count \leftarrow Enc(c)$
$ret \leftarrow Enc(0)$
for $i$=0 to $l$ do
        if block $cipher$ is odd then
              $ret \leftarrow cipher\_add(ret, count)$
        end if
        $cipher \gg 1$
        if $c \neq 0$ then
              $count \leftarrow cipher\_add(count, count)$
        end if
end for
Return ret
```

Figure 12: Algorithms for PIR implementation.