Constant bandwidth ORAM with small block size using PIR operations

Linru Zhang[1], Gongxian Zeng[1], Yuechen Chen[1], Siu-Ming Yiu[1]
Nairen Cao[2], and Zheli Liu[3]

[1]Department of Computer Science, The University of Hong Kong, HKSAR, China
[2]Department of Computer Science, Georgetown University, US
[3]College of Computer and Control Engineering, Nankai University, China
{lrzhang,gxzeng,ycchen,smyiu}@cs.hku.hk
nc645@georgetown.edu,liuzheli@nankai.edu.cn

Abstract. Recently, server-with-computation model has been applied in Oblivious RAM scheme to achieve constant communication (constant number of blocks). However, existing works either result in large block size $O(log^6 N)$, or have some security flaws. Furthermore, a lower bound of sub-logarithmic bandwidth was given if we do not use expensive fully homomorphic operations. The question of "whether constant bandwidth with smaller block size without fully homomorphic operations is achievable" remains open. In this paper, we provide an affirmative answer. We propose a constant bandwidth ORAM scheme with block size $O(log^3 N)$ using only additive homomorphic operations. Our scheme is secure under the standard model. Technically, we design a non-trivial oblivious clear algorithm with very small bandwidth to improve the eviction algorithm in ORAM for which the lower bound proof does not apply. As an additional benefit, we are able to reduce the server storage due to the reduction in bucket size.

Keywords: ORAM, Constant communication overhead, Oblivious clear algorithm

1 Introduction

Oblivious RAM (ORAM) is a block-based storage structure together with a series of algorithms, which allows a client to outsource storage to an untrusted server, while the server learns nothing about the client's access pattern, i.e., the sequence of data blocks actually needed by the client. The concept of ORAM was first proposed by Goldreich [11] together with a hierarchical construction. Several works of hierarchical structure [2, 10, 12–14, 16, 24] have been proposed to improve the efficiency of ORAM. However, hierarchical structure has a drawback of poor worst-case efficiency. A break-through came from the novel tree-based structure proposed by Shi *et al.* [21], which was further improved by many derivative works (e.g. [6, 8, 22]). In this tree-based structure, the server storage is treated as a binary tree in which each node is a bucket that can hold up to

a fixed number of blocks. It contains an *access algorithm* to fetch the required block and an *eviction algorithm* to reshuffle the data blocks from the root to the leaves. Tree-based ORAM avoids the high worst-case's cost of the hierarchical ORAM and achieves better efficiency.

The above ORAM model assumes that the server acts as a storage device that only allows the client to read and write. Under this model, researchers focus on improving the bandwidth overhead (the amount of communication between the client and the server) from $O(\log^3 N)$ to $O(\log N)$ where N is the number of data blocks. However, due to the simple server setting, constant or sub-logarithmic bandwidth seems not achievable.

Mayberry *et al.* [17] proposed a new model that allows the server to perform some computations. The new model is widely used in cloud setting. Under this model, Devadas *et al.* [7] proposed a tree-based Onion ORAM scheme that achieves $O(1)$ communication overhead (constant number of blocks) by applying fully homomorphic encryption, but it requires a large block size of $O(\log^6 N)$ to bound all intermediate transmitted messages. Recently, two improved schemes C-ORAM and CHf-ORAM were proposed. C-ORAM was proposed by Moataz *et al.* [19] to improve the block size to $O(\log^4 N)$ while keeping $O(1)$ communication overhead in the worst case and it only needs additively homomorphic encryption and Private Information Retrieval (PIR) operation. CHf-ORAM [18] also claims to achieve $O(1)$ bandwidth overhead with even smaller block size of $O(\log^3 N)$ using simple XOR-based PIR and four non-colluding servers. However, [1] found security flaws in both CHf-ORAM and C-ORAM. The *eviction algorithms* in their schemes, which push all blocks from one bucket to its two children, cause bucket's distribution leakage by observing the behaviours of several evictions. [1] further derived a $\Omega(\log_{cD} N)$ bandwidth lower bound for the ORAM model with PIR and PIR-write operations. Here, c is the stash size in the client side and D is the number of blocks on which PIR/PIR-write operations are performed. Practically, c and D can be set to $O(\log N)$, i.e., the lower bound can be interpreted as $\Omega(\frac{\log N}{\log \log N})$, which implies that, in such a model, sub-logarithmic bandwidth is achievable but constant bandwidth seems still impossible. The question of "*whether constant bandwidth is achievable using PIR operations*" remains open.

The lower bound. Let us revisit how the lower bound was derived. Consider the relationship between the *visible* access sequence (the sequence of addresses in the server that has been accessed by ORAM algorithm) and the *actual* request sequence (the sequence of blocks that the client actually wants to retrieve). Each entry in the visible access sequence contributes a read/write operation. Following Goldreich's lower bound model [11], where the client is restricted to do just read and write operations, for each visible access address, the block it reads (or writes back) can be stored in (or be removed from) one of the c registers in the client. Thus, there are c possible ways on a read operation and c possible ways on a write operation, i.e., $2c$ possible requests in total. When extending to the model with PIR operation of size D, besides the two original operations (a similar

analysis was used in [1]), there are cD possible ways on each of the PIR-read and PIR-write operations, i.e., $2c + 2cD$ possible requests in total.

Let t be the size of the actual request sequence. Since there are N blocks, the total number of possible request sequences is N^t. If the size of a visible access sequence is q, we can count the total number of possible request sequences ($Q(q)$) that can be supported. The lower bound of q (the number of operations) can be derived by setting $Q(q) \geq N^t$ as a visible access sequence must be able to support all possible request sequences. According to the analysis above, there can be $2c+2cD$ possible requests for each entry in a specific visible access sequence. Given that the client can stores c blocks, a q length sequence can satisfy at most c^q possible program access sequence (c potential choices in each position). Therefore, $Q(q) = (2c + 2cD)^q \cdot c^q \geq N^t$, i.e., $q = \Omega(\frac{t \log N}{\log(cD)})$. Note that this lower bound is on *the number of operations*. Since each operation incurs at least 1 block of bandwidth, the lower bound of bandwidth overhead (per request) is $\Omega(\log_{cD} N)$ blocks. Please refer to [1] for a complete proof.

1.1 Our contributions

Achieving constant bandwidth. If we are able to overcome the constraint that each operation corresponds to at least 1 block of bandwidth, it is still possible to achieve constant bandwidth while using logarithmic operations. If we can have a new operation OP with bandwidth $V < O(\frac{1}{\log_{cD} N})$ blocks and each visible address can evolve in $b = O(1)$ possible ways for each OP operation. We have $Q(q) = (2c + 2cD + b)^q \cdot c^q \geq N^t$. The lower bound of $q = q_{read/write} + q_{PIR} + q_{OP}$ remains the same, where q_{OP} denotes the number of OP operations. If we set $q_{read/write} = q_{PIR} = O(t)$ and $q_{OP} = O(\frac{t \log N}{\log(cD)})$, we can get the amortized bandwidth overhead $T = \frac{O(t)+O(t)+O(\frac{t \log N}{\log(cD)})V}{t} = O(1)$ blocks, i.e., constant bandwidth is possible. The non-trivial problem is how to design such a small-bandwidth operation that can be applied to ORAM to achieve lower bandwidth by performing this operation frequently.

Avoiding distribution leakage. In order to avoid the distribution leakage (produced by existence of buckets' intersection among different evictions) in C-ORAM and CHf-ORAM, we propose a high-level idea that just moves some blocks from one bucket to one of its child when doing eviction, instead of pushing all blocks in one bucket to its two children. Then, the connections among evictions will be lost. However, some "useless" blocks will reside in the same buckets with "useful" blocks, which will occupy the bucket's space and result in failure if we do not clear them. Thus, we need a new noise-clearing algorithm that supports oblivious clearing of the "useless" slots.

Based on the above two ideas, we introduce a **Two-Server Oblivious Clear Protocol**, whose bandwidth is small and can be used in ORAM scheme to clear the "useless" blocks. Using this algorithm, we propose a new ORAM scheme achieving $O(1)$ bandwidth overhead. Compared with the only existing secure constant communication construction Onion ORAM [7], our scheme only requires a block size of $O(\log^3 N)$ and uses additive homomorphic operations.

Compared with CHf-ORAM [18], our scheme achieves the same block size by using only two servers and avoids the security flaws. Unlike other papers [1,18,19], in order to reduce the block size and the bandwidth, they all tried to use existing techniques to reduce both the number and the size of PIR vectors, but we focus on improving the eviction algorithm to reduce the bucket size. It brings another benefit that the server side storage is reduced to $O(BN)$ where B is the size of block.

The following summarizes our contributions in the paper:

- We propose a novel 2-server oblivious clear protocol based on two non-colluding servers. It brings a new idea to update bucket's content without downloading any block to the client. This clear algorithm can be regarded as the new operation with $O(1)$ bandwidth.
- We propose an efficient new constant ORAM scheme (SC-ORAM) based on the clear algorithm with the properties described in Theorem 1. We believe that the block size we achieved is the lowest possible with existing additive homomorphic encryption schemes since all these schemes require a block size of at least $O(log^3 N)$.

  Theorem 1. *To outsource N blocks with block size $B = O(\gamma)$ database, where γ is a security parameter of encryption scheme, SC-ORAM is secure under the standard model, and costs $O(B)$ bandwidth, $O(BN)$ server storage, $O(\log N)$ client storage, and achieves negligible failure probability in N.*

2 Preliminaries

2.1 Private Information Retrieval

Private information retrieval (PIR) is a useful tool that allows the client to retrieve one data block from an unprocessed database known to a server, revealing nothing to the server about which block is downloaded [5]. There are two categories of PIR algorithms: one processes database in a single server and the other assumes the existence of at least two non-colluding servers. Single server PIR algorithms [3,9,15] designed by applying homomorphic encryption have been used in [7, 19] to reduce bandwidth overhead. However, the length of ciphertext in homomorphic encryption limits its efficiency. PIR on two or more non-colluding servers is based on very simple operations such as XOR operation, which reduces the length of PIR vector significantly.

When applying in ORAM, the contents of one bucket can be considered as $U = (\boldsymbol{u_1}, ..., \boldsymbol{u_p})$, where $\boldsymbol{u_i} = (u_{i1}, ..., u_{iq})^T$ is a column vector that records the encryption of a data block. The database of these records is replicated across two servers S_1 and S_2. For the request to block u_i, the client generates a random bit string of length p, $E_1 = (e_1, e_2, ...e_p)$, and then generates $E_2 = (e'_1, e'_2, ...e'_p)$ by flipping the i-th bit in E and keeping other bits unchanged. The client sends E_1 to S_1 and E_2 to S_2. S_1 computes $\Sigma_j \boldsymbol{u_j} \cdot e_j$ while S_2 computes $\Sigma_j \boldsymbol{u_j} \cdot e'_j$, where Σ donates the XORs. The client then sums up (XORs) the two responses to get $\Sigma_j (e_j \oplus e'_j)\boldsymbol{u_j} = \boldsymbol{u_i}$. The communication overhead is $O(p + |\boldsymbol{u_i}|) = O(|\boldsymbol{u_i}|)$.

2.2 C-ORAM and CHf-ORAM

Both C-ORAM and Chf-ORAM are tree-based ORAMs, and share many common properties with existing schemes. When blocks are added to the ORAM, they start at the root of the tree and are tagged as belonging to one of the leaf nodes. As access operations continues, the ORAM needs an eviction process that pushes blocks towards their tagged leaves. Basically, this is usually accomplished by picking a path in the tree and pushing all the blocks on that path as far as possible towards the leaf node.

They achieved constant bandwidth complexity in the number of ORAM elements without expensive fully homomorphic encryption. Although the client exchanges many pieces of data with the server, the key to having $O(1)$ bandwidth overhead is that the size of one data block, B, dominates all communication, i.e., the bandwidth overhead is $O(B)$. Since a trivial lower bound for the bandwidth is at least one block of data, the literature usually refers this as constant bandwidth overhead. The larger block size, the easier it is to construct the ORAM scheme. So the block size is important to evaluate ORAM schemes.

The main idea behind C-ORAM is an oblivious shuffling based on PIR. ORAM read, write, and flush operations can be performed without the client actually downloading data blocks and doing the merging itself. This saves a huge amount of communication when compared to existing schemes like Path ORAM. Additionally, Onion ORAM introduces a triple eviction that empties all buckets along the path instead of only pushing down some elements down and leaving others at intermediate points in the tree. Elements in any evicted bucket will be pushed towards both children. The authors take advantage of the fact that if the path chosen to evict is by reverse lexicographic order, it is guaranteed during an eviction that the sibling of every node on the path will already be empty from a previous evictions. C-ORAM also uses this technique together with a new oblivious merging to do eviction operation, which is achieved by sending a logarithmic number of permutations to server to adjust the bucket's distribution then merge them by addition. The reason of why this eviction technique is needed in these schemes is that some "useless" (noisy) blocks will appear after we merge two blocks using additive homomorphic encryption, which will occupy the bucket's space, leading to bucket overflow. If all the "useful" (real) blocks in one bucket were moved into its children, then it is easy to clear the noisy blocks by simply clearing the whole bucket.

Chf-ORAM, followed by C-ORAM, uses secret sharing and two-server PIR to reduce the block size. Four non-colluding servers A_1, A_2, B_1 and B_2 are included in their construction. A_1 and A_2 store a pair of secret sharing secrets of each data blocks, so do B_1 and B_2. The two-server PIR is realized by combining the XOR results of the corresponding blocks in A_1, A_2 and B_1, B_2 respectively. In addition, a batch insertion operation is introduced to replace the PIR-write.

Two attacks in [1] on C-ORAM and CHf-ORAM. The first attack focused on the access process in C-ORAM. To access a block b in path $Path(tag)$, C-ORAM generates a copy of $Path(tag)$, denotes as $Path'(tag)$. Then, it uses the shadow oblivious merging to move the block b from its original position

to the leaf $Path(tag, leaf)$. Finally a PIR-read vector is designed for the leaf bucket to retrieve b. In the oblivious merging process, a real block can only be merged with an empty block and a noisy block is prioritized to be merged with another noisy block. The attack shows that if there are a pair of buckets (a parent and its one child) are involved in multiple shadow merging processes while not being evicted, then the adversary can find that certain slots in one bucket will repeatedly prefer certain slots in the other bucket since the contents of two buckets remain the same across these shadow merging. This fact can reveal the number of real blocks in the bucket. And it is easy to see that the buckets in lower levels are more vulnerable to the attack with non-negligible probability.

The second attack focused on the eviction in both C-ORAM and CHf-ORAM. Since the blocks will be moved from one parent to its two children, as the eviction process continues, some buckets can accumulate the blocks from different evict paths. The attack constructs two access sequences such that it can distinguish them by counting the repeated entries in the permutations through eviction processes.

3 Clear Algorithm to break the lower bound

In this section, we propose the formal definition of the Two-Server Oblivious Clear Protocol (2SOC Protocol) we used to achieve constant bandwidth, together with the security definition. The concrete construction will be introduced in the next section.

3.1 Intuition of the algorithm

According to our analysis in Section 1, a new algorithm with small bandwidth is needed to achieve the goal and avoid buckets' distribution leakage. Firstly, we introduce how this algorithm comes up when considering how to avoid these kinds of attacks and reduce the buckets' load.

For the sake of clarity, we first consider the second attack, which focuses on the mixture of permutations in different eviction paths. Essentially, the key idea of avoiding this attack is to prevent this kind of mixture. We can realize it by just moving some blocks from one parent to one child when doing eviction, instead of pushing all blocks to its two children. Then, the connection among permutations in different evictions will be lost. However, the real blocks and noisy blocks will reside in the same bucket after eviction, resulting in failure of the pervious naive clear noisy method. Therefore, a new algorithm that supports oblivious clearing of the noisy blocks while keeping other's plaintext unchanged is needed. It is obvious that such an algorithm could change the system states while being hidden from the server and can be performed frequently, so it can be used to overcome the limitation of the lower bound proof if we can realize it with small bandwidth.

The first kind of attack is easy to avoid by using two-server PIR to the whole path instead of doing traditional PIR to just one leaf bucket. The size of two-server PIR vector is much smaller than the traditional PIR vector, so this will

not affect the final result. This also brings another benefit that we can use two non-colluding servers to design our algorithm.

Now, we can present our 2-server oblivious clear protocol. This protocol can be applied in a two-server model and achieve the function that obliviously clears noisy blocks for one bucket with small bandwidth.

3.2 Algorithm and security definition

Here is the description of 2SOC Protocol.

Definition 1. *A protocol for two-server oblivious clear protocol is a tuple of algorithms based on the a public key encryption scheme* $\mathcal{PKE} = (Setup, Encrypt, Decrypt)$. *It takes as input a bit b from the client. The protocol works as follows:*

KeyGen(1^λ ***):*** *the key generation algorithm takes as input the security parameter* λ. *It outputs a secret key sk and a public key pk by running* $\mathcal{PKE}.Setup(1^\lambda)$. *The algorithm generates an operation sequence* $\mathsf{OP} = \{\mathsf{op_1}, \ldots, \mathsf{op_k}\}$ *and sends it to two servers.*

Encrypt(m,pk): *the encryption algorithm takes as input* pk *and a message* $m \in \mathcal{M}$. *It outputs two ciphertexts* $C^{(1)} = \mathcal{PKE}.Encrypt(pk, f_1(m, r_1))$ *and* $C^{(2)} = \mathcal{PKE}.Encrypt(pk, f_2(m, r_2))$, *where* $f_1, f_2 : \mathcal{M} \times \mathcal{R} \to \mathcal{M}$ *are two transformation functions and* $\mathcal{R}$ *is the randomness space.These two ciphertexts are stored in two servers respectively.*

VecGen(b,pk): *the vector generation algorithm takes as input the bit b. If* $b = 0$, *then the algorithm generates* $(V_0^{(1)}, V_0^{(2)})$ *that supports the clear noisy operation. Else if* $b = 1$, *then the algorithm generates* $(V_1^{(1)}, V_1^{(2)})$ *for the keep real operation. Finally, the algorithm sends* $V_b^{(1)}$ *and* $V_b^{(2)}$ *to two servers respectively.*

Clear Noisy: *For* $i \in 1, 2$, *server i performs* $op_1, \ldots, op_k \in \mathsf{OP}$ *in sequence with the inputs* $C^{(i)}$ *and* $V_0^{(i)}$, *and outputs a new ciphertext* $C^{(i)} = \mathcal{PKE}.Encrypt(pk, f_i(0, r_i'))$.

Keep Real: *For* $i \in 1, 2$, *server i performs* $op_1, \ldots, op_k \in \mathsf{OP}$ *in sequence with the inputs* $C^{(i)}$ *and* $V_1^{(i)}$, *and outputs a new ciphertext* $C^{(i)} = \mathcal{PKE}.Encrypt(pk, f_i(m, r_i'))$.

Informally, a 2SOC protocol should guarantee that any adversary who has access to only one component of both the ciphertext $(C^{(1)}, C^{(2)})$ and $(V_b^{(1)}, V_b^{(2)})$ should not learn anything about both the plaintext and the value of b. We formalize this property using the approach of semantic security. Intuitively, our notion says that as long as the two servers do not collude, each of them does not learn anything about the encrypted messages and operations.

Definition 2. *(2SOC INDISTINGUISHABLE SECURITY) Let 2SOC be a 2SOC protocol as defined above, and* $\mathcal{A}$ *be a PPT adversary. Consider the following experiment:*

Experiment $\boldsymbol{Exp}_{2SOC,\mathcal{A}}^{2S.IND}(\lambda)$

$b \leftarrow 0, 1$; $(pk, sk) \leftarrow 2SOC.KeyGen(1^\lambda)$

$(m, i) \leftarrow \mathcal{A}(pk)$

$(C^{(1)}, C^{(2)}) \leftarrow 2SOC.Encrypt(pk, m)$
$(V_b^{(1)}, V_b^{(2)}) \leftarrow 2SOC.VecGen(b, pk)$
$TM_b^{(1)} \leftarrow 2S_b(c^{(1)}, V_b^{(1)})$; $TM_b^{(2)} \leftarrow 2S_b(c^{(2)}, V_b^{(2)})$
$b' \leftarrow \mathcal{A}(C^{(i)}, V_b^{(i)}, TM_b^{(1)}, TM_b^{(2)})$
If $b' = b$ *return 1. Else, return 0.*

$TM_b^{(1)}, TM_b^{(2)}$ *is the transformation messages(if any) between two servers to complete the clear noisy or keep real operation. Let* $Adv^{2S.IND}_{2SOC,\mathcal{A}}(\lambda) = Pr[Exp^{2S.IND}_{2SOC,\mathcal{A}}(\lambda)] - \frac{1}{2}$. *We say that 2SOC is IND secure if for any PPT* $\mathcal{A}$ *it holds* $Adv^{2S.IND}_{2SOC,\mathcal{A}}(\lambda) = negl(\lambda)$.

By using 2SOC protocol, we can design a new eviction algorithm to avoid the distribution leakage attacks. After showing our construction in Section 4, we will discuss how this algorithm achieves small bandwidth in Section 5.

4 Our Construction

In this section, we will show the construction of 2SOC protocol and our ORAM scheme. Intuitively, 2SOC protocol is used to clear the noisy blocks in one bucket after eviction. When considering the security of ORAM scheme, it cannot be recognized by the servers that which block is noisy. So the 2SOC protocol should be IND-secure (defined in Definition 2). More precisely, for a bucket D, a clear vector W_D is needed, and each item w_i in W_D is corresponding to a data block d_i in D. After some pre-decied computations between w_i and d_i, the date block been either cleared or re-encrypted. The different parts of the vector W_D should be indistinguishable to any adversary. A natural approach is to create W_D as a combination of $Enc(1)$ and $Enc(0)$ if a fully homomorphic encryption is used to encrypt the data. However, fully homomorphic encryption costs so much and has poor efficiency, so an improved method is needed.

4.1 SC-ORAM construction

Stash. When the client reads or writes a block, this block will be added into the stash, which is a linear structure of size $O(\log N)$ in the secure storage on the client side. We will show that our scheme has the same eviction efficiency as Circuit ORAM [23], so the stash in our scheme also has negligible overflow probability has been proved in [23].

Bucket Configuration. Let N be the block number of the out-source database which is the power of 2, our scheme is a binary tree with $L + 1$ levels and $2^L = O(N)$ leaves. Precisely, each bucket contains $\mu \cdot z$ blocks, where z is a constant indicates the number of slots needed to hold actual data blocks and $\mu > 2$ is a multiplicative constant that gives extra room for noisy blocks. Additionally, each bucket contains IND-CPA encrypted meta-information named Headers, including additional information about a bucket's contents. The bucket configuration is shown in Figure 1.

Headers. Bucket headers determine how permutations are generated, which blocks will be moved down and which blocks are supposed to be cleared. A bucket header is comprised of two parts: the first part stores the information whether each block is noisy, real or empty(encryption of 0) data, while the second one keeps the block identifier. Finally, like all tree-based ORAM, each block in a bucket also contains a separate encryption of its address.

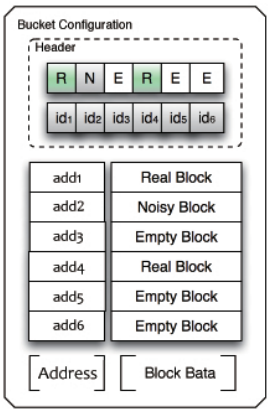

Fig. 1: Bucket Configuration (All encryptions over the data are ignored in this figure)

Two-Server Structure. Our scheme is based on a two-server model. Let A and B are two non-colluding servers with the same size. Both A and B are organized as a binary tree. Two servers share a common position map and always perform the same operation at the same time. For each block b whose data is m, we choose two random values $x, y \leftarrow \mathcal{M}$, where $\mathcal{M}$ is the plaintext space, then store $C_1 = (Enc(x), m+x)$ on server A and $C_2 = (Enc(y), -m+y)$ on server B, where Enc is a appropriate additively homomorphic encryption scheme (such as Paillier cryptosystem [20]). Therefore, any block is always in the same position of both A and B. For simplicity, we only discuss one server in the following text and show the differences between them when necessary.

Access operation. To access a block b in a server, i.e., read or write, the client first fetches the corresponding position tag tag from the position map of the recursive ORAM trees. This tag defines a unique path $Path(tag)$ starting from the root of the ORAM tree and going to a specific leaf given by the tag. The element might reside in any bucket in this path. To retrieve this element, the client finds the position of it through the headers and makes use of a PIRread.

Firstly, the client downloads the headers of all buckets in $Path(tag)$ and searches for the bucket which contains b. Then the client generates two vectors E_1, E_2 according to the two-server PIR protocol. Server A does $\Sigma_i e_i \cdot (m_i + x_i)$ on the second part of block, and does $\bigoplus_i x_i Enc(b_i)$, where $\bigoplus$ represents the

homomorphic addition computations on the first part of block. Server B does the same operations. After receiving responses from two servers, the client also adds the secord part of messages up to get α, and does homomorphic addition on the first part followed by the decryption process to get β. Finally the client will obtain the data b by $m = (e_{1j} - e_{2j})(\alpha - \beta)$, where j is the corresponding position of block b in the path $Path(tag)$. Afterwards, two evict operations according to reverse lexicographic order are performed. Finally, if the number of real blocks in the root is less than z after one evict operation, then a block is written back from stash.

4.2 Evict operation

The last process we need to introduce is the evict operation, which aims at moving blocks from top to bottom along a path. We propose an efficient pre-decided eviction method with small bandwidth overhead between the server and the client, while no entire block needs to be downloaded. In particular, we come up with a novel way to reduce the number of clear vectors by adjusting the buckets' distribution along the eviction path. Since permutation is much smaller than the clear vector, our scheme saves the communication cost. The detail will be shown below and the security analysis is in Section 5.

Eviction Algorithm Since no entire date block cloud be downloaded and only small permutations or vectoros can be sent to drive the eviction operations by the client, the traditional eviction method that downloads all blocks in the evict path and writes them back one by one is not feasible at all. Inspired by Circuit ORAM [23], we can use a pre-processing algorithm to determine which blocks should be moved down and to indicate their destinations according to the header of the evict path$Path(tag)$. The inputs of this algorithm are a set of headers and outputs a sequence of destnations of each block. The detail of the algorithm can be found in Algorithm 1, in which PrepareDeepest and PrepareTarget are two sub-algorithm in [23]. The first outputs an array $deepst[1, \ldots, L]$, where $deepest[i]$ stores the level of the deepest block that can legally store in bucket $Path(tag, i)$. The second outputs an array $target[1, \ldots, L]$, where $target[i]$ stores which level the deepest block in $Path(tag, i)$ will be evicted to. Therefore, the remaining challenge is to implement the move down operation and clear noisy operation obliviously.

Move Down Operation There is at most 1 block that have to be moved done in each bucket along the evict path after the pre-processing process. In order to keep obliviousness, this block should be moved down along the path without skipping any bucket between its beginning and destination, i.e., if we want to move block b from B_i to B_j, then we have to move it to B_{i+1} firstly and then to B_{i+2} and arrive B_j finally. Now we show the approach of how to move block b from B_i to B_{i+1} and other steps are similar. Firstly, the client retrieves a copy of header of B_i, and changes other real blocks' mark from "real" to "noisy" in the duplicate header H'_i. Then, the client generates a permutation Π according to H'_i and H_{i+1}, which ensures all "real" blocks in both buckets corresponding to "empty" blocks. The server performs Π to B_i and merges it

Algorithm 1 Pre-processing algorithm

procedure PRE-PROCESSING($path$)
  Call the PrepareDeepest and PrepareTarget subroutines to pre-process arrays $deepest$ and $target$
  $(flag, pos, des)_L \leftarrow (0, 0, 0)_L$
  for $i \leftarrow 0$**to**L **do**
    if $target[i] \neq \perp$ **then**
      $pos[i] \leftarrow 0$
      for $j \leftarrow 0$**to**μz **do**
        if $path[i][j]$ can be move deeper than $path[i][pos[i]]$ **then**
          $pos[i] \leftarrow j$
        end if
      end for
      $flag[i] \leftarrow 1$
      $des[i] \leftarrow target[i]$
    end if
  end for
  return $(flag, pos, des)_m$
end procedure

into B_{i+1} by adding homomorphic encryption to the blocks. Finally, update the headers H_i, H_{i+1}, and delete the duplicate H'_i.

Clear Noisy Operation After moving down operation, some noisy blocks appear since two buckets are merged. A clear noisy algorithm is needed to guarantee that there are enough empty room to support the following move down operation. We show the simple algorithm first followed by the improvement, both of which are built on a protocol for two-server oblivious clear protocol(2SOC Protocol). This protocol supports performing clear noisy and keeps real operations obliviously by sending two clear vectors to two servers respectively, the detail of which will be introduced in the next section. For convenience, we denote the size of each item in clear vector is α. The length of the clear vector is equal to the bucket size($\mu z = O(1)$) since each item in the clear vector is corresponding to one block. The client designs two clear vectors for each bucket, so $O(\log N)$ vectors are needed in one eviction operation. Then the total communication overhead between the client and servers in clear noisy operation is $O(\mu z \alpha \log N) = O(\alpha \log N)$, which should be bounded by the block size in our result. However, we can see that if another clear algorithm is applied or the overhead size of homomorphic encryption ciphertext is reduced, then this bandwidth overhead is possible to larger than a $O(\gamma)$. The improved method will be introduced to avoid it.

At the beginning of evict operation, the client generates a configuration of bucket D_1 randomly together with a corresponding header H_{D_1} with z real blocks and $\mu z - z$ empty blocks. Similarly, the client generates a configuration of bucket D_2 randomly together with a corresponding header H_{D_2} with $z+1$ real blocks and $\mu z-(z+1)$ empty blocks. Then two pairs of clear vectors (W^A_{D1}, W^B_{D1}) for D_1 and (W^A_{D2}, W^B_{D2}) for D_2 is designed according to the 2S-DCNA Protocol. After the

server merges B_i into B_{i+1} by taking moving down operation, the client generates a permutation Π' according to two headers H_i and D_1 under the condition that all real blocks in B_i are in the same position as D_1 after performing Π' to B_i then do the similar operation to B_{i+1} and D_2. It is easy to know that the real blocks in H_i is at most z and the real blocks in H_{i+1} is at most $z+1$, so the permutations are always existing. Finally we use $(W^A_{D1}, W^B_{D1}), (W^A_{D2}, W^B_{D2})$ for the two buckets to do clear operation, after which at least $\mu z - z$ blocks in B_i and $\mu z - (z+1)$ blocks in B_{i+1} will become an encryption of zero. We will prove that such permutations will never leak any information related to the bucket load to the server, which is guaranteed by the reverse lexicographic eviction order. The size of permutation is $\mu z \log \mu z$, so the total overhead of this improved operation is $O(\mu z \log \mu z \log N + \alpha) = O(\alpha + \log N)$. A $\log N$ factor is saved compared with original one when $\alpha > \log N$ which is always true. In the rest of paper, we will use this improved clear noisy operation.

4.3 2S-DOCA Protocol and Clear Vector

In this section, we will introduce the basic protocol for clear vector construction, two-server oblivious clear protocol(2SOC Protocol), which is inspired by [4], and followed by the construction of the clear vector.

2SOC Protocol is a technique that obliviously performs one of these two operations: change data into $Enc(0)$ or keep the value unchanged and re-encrypt it. All intermediate processes and results are indistinguishable. Using this technique for a certain bucket, we can clear some blocks from noisy to 0 while keep others' value unchanged by sending a clear vectors to servers and performing some pre-decided calculation between the vectors and data blocks. The protocol is based on additively homomorphic encryption, and more specifically, Paillier cryptosystem is used in our ORAM scheme but it also suitable for many other additively homomorphic encryptions.

Before showing the construction, we define the property that an additively homomorphic encryption scheme needs to satisfy. We call such a scheme a public-space homomorphic encryption, and we formalize this notion below.

Definition 3. *An additively homomorphic encryption scheme $\mathcal{HE} = (KeyGen, Enc, Eval, Dec)$ with message space $\mathcal{M}$ is said to be public-space if: (1) $\mathcal{M}$ is a (publicly known) finite and commutative ring with a unity, and (2) it is possible to efficiently sample uniformly distribution elements $m \in \mathcal{M}$.*

We stress that the above is a very mild requirement, and most existing additively homomorphic encryption schemes based on number theory are public-space. Furthermore, we note that also the more recent lattice-based homomorphic encryption schemes satisfy our notion of public-space.

A 2SOC Protocol Our construction builds upon a public-space additively homomorphic encryption, and three parts (server A, server B and client) are included in the construction. The precise description of our scheme follows (based on a public-space additively homomorphic encryption $\mathcal{HE} = (KeyGen, Enc, Eval, Dec)$):

Encryption and storage organization: The randomized encryption algorithm chooses two random value $x, y \leftarrow \mathcal{M}$ and run $\mathcal{HE}.KeyGen(1^\lambda)$ to get the public key pk. Then set $C^{(1)} = (Enc(pk, x), m+x) = (c_1, m+x) \in \mathcal{C} \times \mathcal{M}$, which is stored in A, and set $C^{(2)} = (Enc(pk, y), -m + y) = (c_2, -m + y) \in \mathcal{C} \times \mathcal{M}$, which is stored in B.

Clear Noisy: Client chooses $k_1, k_2 \leftarrow \mathcal{M}$ uniformly under the condition that k_1, k_2 both have inverse in $\mathcal{M}$, while we can make this condition easy to satisfy by choosing suitable additively homomorphic encryption scheme, and then sends $V_0^{(1)} = V_0^{(2)} = (k_1, k_2)$ to A and B respectively. Then, A computes that $C_{k_1}^{(1)} = (c_1^{k_1}, k_1(m + x))$, $C_{k_2}^{(1)} = (c_1^{k_2}, k_2(m + x))$, and sends $C_{k_2}^{(1)}$ to B. B computes that $C_{k_1}^{(2)} = (c_2^{k_1}, k_1(-m + y))$, $C_{k_2}^{(2)} = (c_2^{k_2}, k_2(-m + y))$, and sends $C_{k_1}^{(2)}$ to A. Finally, A does that $C'^{(1)} = (c_1^{k_1} \oplus c_2^{k_1}, k_1(m + x) + k_1(-m + y)) = (Enc(pk, k_1(x + y)), k_1(x + y))$, and B can do the similar computations to get $C'^{(2)} = (Enc(pk, k_2(x + y)), k_2(x + y))$. It is obvious that both $C'^{(1)}$ and $C'(2)$ are representatives of encryption of 0.

Keep Real: Similarly, client chooses $k_1', k_2' \leftarrow \mathcal{M}$ uniformly under the condition that k_1', k_2' both have inverse in $\mathcal{M}$, then, sends $V_1^{(1)} = (k_1', k_2' - 1)$ to A and sends $V_1^{(2)} = (k_1' - 1, k_2')$ to B. A and B perform same computation to get $C'^{(1)} = (Enc(pk, k_1'x + (k_1' - 1)y), m + k_1'x + (k_1' - 1)y)$, $C'^{(2)} = (Enc(pk, (k_2' - 1)x + k_2'y), -m + (k_2' - 1)x + k_2'y)$, which are representatives of re-encryption of m.

Furthermore, we will show that our protocol is semantic secure under our the definition in Section 3. We formalize this property in the following theorem.

Theorem 2. *If $\mathcal{HK}$ is semantically secure, then Our 2S-DOCA protocol is a IND secure protocol under Definition 2.*

The proof of this theorem is rather straightforward. We can easily reduce it to the security of homomorphic encryption scheme.

Clear Vector Construction Now, we give the construction of the clear vector. For any bucket $D = (b_1, ...b_{\mu z})$, we design two clear vectors $W_D^A = (w_{D1}^A, ..., w_{D\mu z}^A)$ for server A and $W_D^B = (w_{D1}^B, ..., w_{D\mu z}^B)$ for server B. If $b_i, i \in \{1, ..., \mu z\}$ is a real block, then set $W_{Di}^A = (k_1', k_2' - 1), W_{Di}^B = (k_1' - 1, k_2')$ according to the above 2SOC protocol. Otherwise if $b_j, j \in \{1, ..., \mu z\}$ is a noisy or empty block, then set $W_{Di}^A = (k_1, k_2), W_{Di}^B = (k_1, k_2)$ similarly. Finally, the client sends W_D^A to server A and W_D^B to server B and let them perform the pre-decided calculation.

5 SC-ORAM Analysis

We will analyze the bandwidth overhead first, which leads to our main result. The proof of correctness and security of SC-ORAM will be given next.

5.1 Bandwidth overhead analysis

Client-Server Bandwidth Evaluations and Block Size. The Access operation is composed of scheduled evict operation, oblivious moving in the cloned path, a PIR read, no more than two PIR writes. The size of headers ($O(\mu z \log N)$) are negligible compared to the PIR read and write vectors. For the sake of clarity, we therefore avoid including them in our asymptotic analysis.

First, the moving down operation of eviction process always involves $O(\log N)$ permutations whose size is $O(\mu z \log \mu z)$ bits, so the total overhead is $O(\log N \mu z \log \mu z)$ bits. The clear noisy operation of eviction process also involves $O(\log N)$ permutations with size $O(\mu z \log \mu z)$ bits and two clear vectors for two servers whose size are $O(\mu z|\mathcal{M}|)$ bits. Therefore, the total amount of overhead of the eviction operation is $O(2 \log N \mu z \log \mu z + 2\mu z|\mathcal{M}|)$. The size of PIR read vector is $O(\mu z \log N)$, and the size of PIR write vector' size is $O(\gamma \mu z)$ bits, where γ is the length of ciphertext in additively homomorphic encryption, which is larger than $|\mathcal{M}|$. Finally, the PIR read operation will return a block with size $O(|B|)$ bits as the result of an access operation. In conclusion, the communication overhead in the whole process is $O(2 \log N \mu z \log \mu z + 2\mu z|\mathcal{M}| + \log N \mu z \log \mu z + \mu z \log N + 2\gamma \mu z + |B|) = O(\log N \mu z \log \mu z + \gamma \mu z + |B|)$.

To have constant bandwidth, the block size should be $|B| \in \Omega(\log N \mu z \log \mu z + \gamma \mu z)$. With $z = O(1)$ and $\mu = O(1)$, we achieve $|B| \in \Omega(\log N + \gamma)$. In practice, we choose $\gamma \in O(\lambda^3)$ and $\lambda \in \omega(\log N)$, the parameter set of which is the same as C-ORAM [19], so γ dominates $\log N$. Therefore, block size is $|B| \in \Omega(\gamma)$.

Communication between Two Servers and Server Storage Size. The communication between two servers mainly occurs in the eviction process. For each bucket in the eviction operation, the two servers will send a ciphertext-form-message to the other according to our 2SOC protocol, the length of which is $O(\mu z(\gamma + |\mathcal{M}|))$. Since there are $O(\log N)$ buckets in one evict path, so the total amount of communication between two servers are $O(\log N \mu z(\gamma + |\mathcal{M}|))$. When applying the above parameters, we get $O(\log N \mu z(\gamma + |\mathcal{M}|)) = O(\log N \mu z \gamma) = O(\log^4 N)$, which is the same as the client-server bandwidth overhead in C-ORAM. The communication between servers is much cheaper than it between the client and the server, so our scheme is more practical than C-ORAM.

The storage structure in the server side is two binary trees (one in each server), whose height is $O(\log N)$, and the size of each bucket is $O(\mu z) = O(1)$ blocks. So the total server side storage is $O(2\mu z \cdot 2^{\log N}) = O(N)$ blocks. But in many other constructions, the size of each bucket is $O(\log N)$ blocks, so the total server side storage is $O(N \log N)$ blocks. Compared with it, we reduced a $logN$ factor in the server storage.

5.2 Correctness analysis

The goal of the correctness analysis is to show that the probability of a failure occurs during the eviction operations is insignificant. To begin with the analysis, we outline two failure types during eviction in SC-ORAM:

- F_1: Blocks with value of encrypted *zero* in the eviction path is less than z

- F_2: Overflow of the stash on the client side

Lemma 1. *If the constant factor $\mu > 2$, the number of empty blocks in each bucket along the eviction path is at least z after the eviction operation.*

The proof of the lemma is in the appendix, and $\mu > 2$ is a basic requirement in SC-ORAM.

Lemma 2. *With $O(\log N)$ lower bound of stash size, the probability of F_2 is negligible.*

The proof of lemma 2 is a straightforward extension from that in Circuit ORAM.

Theorem 3. *SC-ORAM is a correct ORAM scheme.*

Through Lemma 1, we can infer that when a new eviction operation begins, each bucket in the evict path has at least z empty slots and F_1 will not happen. In particular, each bucket should have no more than z real blocks before a eviction operation according to our rules. Therefore it can be proved that the number of empty blocks is enough to perform the eviction operation correctly. Combined with Lemma 2, SC-ORAM is correct.

5.3 Security for SC-ORAM

In this section, we consider the security of the eviction operation and the read/write operation. Intuitively, we show that the adversary cannot gain any knowledge a particular bucket. For the sake of clarity, we use the same assumption as C-ORAM [19] that there is no noisy block in buckets. We can easily extend the proof for the case where we have real, empty and noisy blocks. Assume that A_1 and A_2 are two adjacent buckets whose capacity are m blocks in one evict path, then we can outline the key roundtrips of our eviction operation as follows, which is also depicted in Figure 2.

1. We obtain A_1' by performing a permutation Π_1 to bucket A_1.
2. Bucket A_1' is merged into A_2 by adding the ciphertexts of two corresponding positions, then the new bucket after merging is noted as B_1.
3. B_2 is the result of permutation Π_2 to bucket B_1.
4. Finally, we clear the noisy blocks in B_2 by using the Clear Vector and noted the result bucket by C.

From the descriptions in Figure 2, we elaborate the complete proof by the following steps presented from Lemma 3 to 6, the proof of which can be found in the appendix. To begin with, we will use the adversarial permutation indistinguishability experiment in C-ORAM [19] to show the two kinds of permutations in SC-ORAM are both indistinguishable permutations. We give out the definition and security proof in the appendix B.

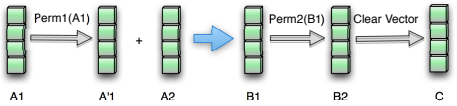

Fig. 2: Permutation and Clear

Lemma 3. *Assumed the distributions of real blocks in A_1 is random, then the real block after Π_1 in A'_1 is randomly distributed.*

The proof of lemma 3 is the extension of that in C-ORAM [19].

Lemma 4. *If real blocks in A_2 is randomly distributed, and independent from the distributions of that in A'_1, then the real data distributions in B_1 is random.*

We will show later that the condition of lemma 4 is always true in SC-ORAM.

Lemma 5. *The distribution of real blocks in B_2 is random and Π_2 is indistinguishable with a randomly permutation*

Lemma 6. *The distribution of real blocks in C is randomly due to the clear operation.*

Proof. The proof of lemma 6 can be devired from lemma 5 easily.

A_1 and A_2 are obtained by two eviction operations. It is easy to know A_1 and A_2 are randomly distributed due to the fact that C (the result of one evict operation) is randomly distributed. On the other hand, when we perform eviction operation according to reverse lexicographic order, we can guarantee that in each evict path, every two adjacent buckets are generated by different pervious eviction operations, so the distribution of these buckets are independent. Therefore, the distribution of A_1 and A_2 are both random and independent, which satisfy with our assumptions in Lemma 4.

Theorem 4. *The real data distribution over each buckets (bucket load) is completed hidden from the adversary.*

Proof. To prove Theorem 4, we complete it in the following cases:

Security over Eviction: Now, the security of Eviction operation has been proved through Lemma 3 to 6.

Security over Access: The only question we left is the security of read/write operation. In one read/write operation on block a, we copy a path P'_i from P_i and perform a oblivious move operation on P'_i which moves block a down to the leaf node L'_iin P'_i, then we perform a PIR read on L'_i. Due to the move operation is obliviously and the PIR read is safe, so the whole read/write operation leak no information to adversary.

Now we move to the proof of our main Theorem 1.

The key point of the proof is to show the probability that the adversary can recover the access sequence is negligible. The details of proof is in the appendix.

6 Conclusions and discussions

In this paper, we propose a secure constant bandwidth ORAM scheme with improved block size. Technically, we propose a new 2-server delegation of oblivious clear algorithm protocol which is proved secure and oblivious, and is applied in our eviction phase. With this improved eviction algorithm, we can reduce the bucket size to $O(1)$ blocks, resulting in reducing both the size of block and server storage by a $O(\log N)$ multiplicative factor. We believe that our scheme achieved the lower bound for block size for existing additively homomorphic encryption schemes.

Although our scheme achieves constant bandwidth overhead, we do so by using two non-colluding servers and AHE. Other schemes without AHE either have been attacked, can only achieve sub-logarithmic bandwidth overhead or with large block size. The problem of having a constant bandwidth ORAM with small block size without using AHE operations remains open.

References

1. Abraham, I., Fletcher, C.W., Nayak, K., Pinkas, B., Ren, L.: Asymptotically tight bounds for composing oram with pir. In: IACR International Workshop on Public Key Cryptography. pp. 91–120. Springer (2017)
2. Boneh, D., Mazieres, D., Popa, R.A.: Remote oblivious storage: Making oblivious ram practical (2011)
3. Cachin, C., Micali, S., Stadler, M.: Computationally private information retrieval with polylogarithmic communication. In: International Conference on the Theory and Applications of Cryptographic Techniques. pp. 402–414. Springer (1999)
4. Catalano, D., Fiore, D.: Using linearly-homomorphic encryption to evaluate degree-2 functions on encrypted data. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. pp. 1518–1529. ACM (2015)
5. Chor, B., Goldreich, O., Kushilevitz, E., Sudan, M.: Private information retrieval. In: Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on. pp. 41–50. IEEE (1995)
6. Chung, K.M., Liu, Z., Pass, R.: Statistically-secure oram with\ tilde {O}(\ logˆ 2 n) overhead. In: Advances in Cryptology–ASIACRYPT 2014, pp. 62–81. Springer (2014)
7. Devadas, S., van Dijk, M., Fletcher, C.W., Ren, L., Shi, E., Wichs, D.: Onion oram: A constant bandwidth blowup oblivious ram. In: Theory of Cryptography, pp. 145–174. Springer (2016)
8. Gentry, C., Goldman, K.A., Halevi, S., Julta, C., Raykova, M., Wichs, D.: Optimizing oram and using it efficiently for secure computation. In: Privacy Enhancing Technologies. pp. 1–18. Springer (2013)
9. Gentry, C., Ramzan, Z.: Single-database private information retrieval with constant communication rate. In: International Colloquium on Automata, Languages, and Programming. pp. 803–815. Springer (2005)
10. Goldreich, O.: Towards a theory of software protection and simulation by oblivious rams. In: Proceedings of the nineteenth annual ACM symposium on Theory of computing. pp. 182–194. ACM (1987)

11. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious rams. Journal of the ACM (JACM) 43(3), 431–473 (1996)
12. Goodrich, M.T., Mitzenmacher, M.: Privacy-preserving access of outsourced data via oblivious ram simulation. In: Automata, Languages and Programming, pp. 576–587. Springer (2011)
13. Goodrich, M.T., Mitzenmacher, M., Ohrimenko, O., Tamassia, R.: Oblivious ram simulation with efficient worst-case access overhead. In: Proceedings of the 3rd ACM workshop on Cloud computing security workshop. pp. 95–100. ACM (2011)
14. Goodrich, M.T., Mitzenmacher, M., Ohrimenko, O., Tamassia, R.: Practical oblivious storage. In: Proceedings of the second ACM conference on Data and Application Security and Privacy. pp. 13–24. ACM (2012)
15. Kushilevitz, E., Ostrovsky, R.: Replication is not needed: Single database, computationally-private information retrieval. In: Foundations of Computer Science, 1997. Proceedings., 38th Annual Symposium on. pp. 364–373. IEEE (1997)
16. Lu, S., Ostrovsky, R.: Distributed oblivious ram for secure two-party computation. In: Theory of Cryptography, pp. 377–396. Springer (2013)
17. Mayberry, T., Blass, E.O., Chan, A.H.: Efficient private file retrieval by combining oram and pir. In: NDSS. Citeseer (2014)
18. Moataz, T., Blass, E.O., Mayberry, T.: Chf-oram: a constant communication oram without homomorphic encryption. Tech. rep., Cryptology ePrint Archive, Report 2015/1116 (2015)
19. Moataz, T., Mayberry, T., Blass, E.O.: Constant communication oram with small blocksize. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. pp. 862–873. ACM (2015)
20. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: Advances in cryptology-EUROCRYPT'99. pp. 223–238. Springer (1999)
21. Shi, E., Chan, T.H.H., Stefanov, E., Li, M.: Oblivious ram with o ((logn) 3) worst-case cost. In: Advances in Cryptology–ASIACRYPT 2011, pp. 197–214. Springer (2011)
22. Stefanov, E., Van Dijk, M., Shi, E., Fletcher, C., Ren, L., Yu, X., Devadas, S.: Path oram: An extremely simple oblivious ram protocol. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. pp. 299–310. ACM (2013)
23. Wang, X., Chan, H., Shi, E.: Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. pp. 850–861. ACM (2015)
24. Williams, P., Sion, R.: Single round access privacy on outsourced storage. In: Proceedings of the 2012 ACM conference on Computer and communications security. pp. 293–304. ACM (2012)

Appendix

A Proofs

Proof of Lemma 1. As we know, each bucket contains at least $2z+1$ blocks. In addition, there are at most $z+1$ real blocks in each bucket along the evict path during each evict operation at any time(at most z origin blocks and at most 1 block is moved down from its parent). Thus the sum of noisy blocks and empty blocks is at least z.

In Oblivious Clear, the client generate a Clear Vector which keeps at most $z+1$ blocks intact and clear other $\mu z-(z+1) \geq (2z+1)-(z+1) = z$ blocks to empty blocks. In particular, the empty block is set the encryption of *zero*. so the number of empty blocks along the evict path is at least z after the eviction operation.

Proof of Lemma 4. We assume that the number of real blocks in A_1' is i and the number of real blocks in A_2 is j. According to the pre-processing algorithm in Circuit ORAM, there is one or zero real block in A_1'(other real blocks were set as noisy at the beginnings of the eviction operation in this bucket), so we have $i = 0$ or $i = 1$. For a randomly selected distribution which is noted by D, we have the results in equation (1).

$$\begin{cases} Pr[B_1 = D] = Pr[A_2 = D] = \frac{1}{\binom{\mu z}{j}}, if\ i = 0 \\ Pr[B_1 = D] = \frac{\binom{j+1}{1}}{\binom{\mu z}{j}\binom{\mu z - j}{1}} = \binom{\mu z}{j+1}, if\ i = 1 \end{cases} \tag{1}$$

From equation (1), we can conclude that the target distribution is random, which completes the proof in lemma 4.

Proof of Lemma 5. We assume that the number of real blocks in B_2 is x, then the number of real blocks in B_1 is x. A randomly selected distribution is noted by D' So we have:

$$Pr[B_2 = D'] = \frac{\binom{\mu z - x}{z-x}}{\binom{\mu z}{z}\binom{z}{x}} = \frac{1}{\binom{\mu z}{x}} \tag{2}$$

Therefore the distribution is randomly.

Regarding Π_2, we can use similar method as C-ORAM to prove it. Firstly, if we use a randomly permutation Π_2' instead of Π_2, then $Pr[X = \pi_0] = \frac{1}{m!}$, so we have to show that the permutations output under such rule in "clear noisy" section is uniformly distributed. Let $|B|$ denote the number of real blocks in bucket B, then we have,

$$\begin{aligned} Pr[X = \pi_1] &= \sum_{i=0}^{z} Pr[X = \pi_1 and |B_1| = i] \\ &= \sum_{i=0}^{z} Pr[X = \pi_1 || B_1| = i] \cdot Pr[|B_1| = i] \end{aligned} \tag{3}$$

We compute the probability is selecting a permutation while the number of real blocks in B_1 is i. The total number of permutations that can be generated under the rule equals $Total = \frac{\binom{\mu z}{i}\binom{\mu z}{z}\binom{z}{i} i!(\mu z - i)!}{\binom{\mu z}{i}\binom{\mu z - i}{z-i}} = \mu z!$. That is:

$$\begin{aligned} Pr[X = \pi_1|] &= \sum_{i=0}^{z} Pr[X = \pi_1 || B_1| = i] \cdot Pr[|B_1| = i] \\ &= \sum_{i=0}^{z} Pr[|B_1| = i] \cdot \frac{1}{\mu z!} = \frac{1}{\mu z!} \end{aligned} \tag{4}$$

Thus for the adversary, permutations output under such rule are randomly distributed, i.e.

$$Pr[X = \pi_0] = Pr[X = \pi_1] = \frac{1}{\mu z!}$$

Proof of the Theorem 1. To prove the security of SC-ORAM, let $\boldsymbol{y}$ be a data request sequence of size t. By the definition of SC-ORAM, the server sees $A(\boldsymbol{y})$ which is a sequence

$$\mathbf{P} = (position_t[a_t], position_{t-1}[a_{t-1}], \ldots, position_1[a_1])$$

, where $position_j[a_j]$ is the position of address a_j indicated by the position map and the header of the path for the j-th access operation, together with a results returned by apply PIR-Read on the leaf node of $position_j[a_j]$. The sequence of results returned by the PIR-Read is computationally indistinguishable from a random sequence of bit strings by the definition of randomized encryption and the obliviousness of the PIRread.

Notice that once $position_i[a_i]$ is revealed to the server, it is remapped to a completely new random label $position'_i[a_i]$, and the positions of different addresses do not affect one another in SCORAM. Moreover, due to we have proved that the distribution of real blocks in any bucket at any stage of one access operation is random, so the evict operation or oblivious move down operation would not reveal any information about the position of a_i, i.e.,

$$\begin{aligned} Pr(position_i[a_i]) = Pr(position_i[a_i]| \\ (Evict(p_i) \cap Move(position_i[a_i]))) \end{aligned} \tag{5}$$

where $Pr(position_i[a_i])$ is the probability that the adversary can guess the address of $position_i[a_i]$ in $\mathbf{P}$ correctly. Therefore, by Bayes rule,

$$\begin{aligned} Pr(\mathbf{P}) &= \prod_{i=1}^{t} Pr(position_i[a_i]) \\ &= \prod_{i=1}^{t} Pr(position_i[a_i]|(Evict(p_i) \cap Move(position_i[a_i]))) \\ &= \prod_{i=1}^{t} \frac{1}{2^L} = (\frac{1}{2^L})^t \end{aligned} \tag{6}$$

This prove that $A(\boldsymbol{y})$ is computationally indistinguishable from a random sequence of bit strings.

Now the security follows from the Lemma 3 that for stash size $O(\log N)$ SCORAM fails (in that it exceeds the stash size) with at most negligible probability.

B Definition of Indistinguishable Permutation and its Proof

Let $\mathcal{E}_1 = (Gen_1, Enc_1, Dec_1)$ and $\mathcal{E}_2 = (Gen_2, Enc_2, Dec_2)$ be an IND-CPA encryption and IND-CPA homomorphic encryption schemes respectively. Assumed that $\mathcal{M}$ is a probabilistic algorithm to generate permutations according to two bucket configurations, and κ is the security parameter. Next we will introduce the experiment $\mathsf{PermG}^{\mathcal{A}}_{\mathcal{M},\mathcal{E}_1,\mathcal{E}_2}(\mathsf{k})$.

- **Setup**(1^k): the Setup algorithm runs $k_1 \leftarrow Gen_1(1^k)$ and $k_2 \leftarrow Gen_2(1^k)$ to output a key pair (k_1, k_2), then chooses n buckets and encrypt them with $Enc_2(k_2, .)$ together with their headers encrypted with $Enc_1(k_1, .)$. Next send the encrypted message to the adversary $\mathcal{A}$
- **Challenge**: the adversary $\mathcal{A}$ chooses two buckets A and B, and sends the encrypted $header(A)$ and $header(B)$. The challenger picks a bit $b \in \{0, 1\}$ randomly. If $b = 1$, calculate $\Pi_1 \leftarrow \mathcal{M}(header(A), header(B))$, else $\Pi_0 \leftarrow Perm$. Then sends Π_b to the adversary $\mathcal{A}$.
- **Query**: the adversary $\mathcal{A}$ access to the oracle $\mathcal{O}_{\mathcal{M}}$, which outputs permutation except for the challenge headers. when $\mathcal{A}$ thinks that the query phase ends, it outputs a bit b'.
- **Output**: If $b = b'$, the experiment will output 1, otherwise 0.

Definition 4. ***Indistinguishable Permutation****: An algorithm $\mathcal{M}$ is said to be an indistinguishable permutation iff for all PPT adversary $\mathcal{A}$ and all possible bucket configurations A and B, there exists a negligible function negl such that:*

$$Pr[PermG^{\mathcal{A}}_{\mathcal{M},\mathcal{E}_1,\mathcal{E}_2}(\lambda, 1) = 1] - Pr[PermG^{\mathcal{A}}_{\mathcal{M},\mathcal{E}_1,\mathcal{E}_2}(\lambda, 0) = 1] \leq negl(\lambda)$$

Theorem 5. *If $\mathcal{E}_1$,$\mathcal{E}_2$ are IND-CPA secure, then the two kinds of permutations in SC-ORAM are indistinguishable permutations.*

The proof of Theorem 5 proceeds according to a series of consecutive games, and it is a easily extension of the proof in C-ORAM [19].