OC-ORAM: Constant Bandwidth ORAM with Smaller Block Size using Oblivious Clear Algorithm

Linru Zhang[1], Gongxian Zeng[1], Yuechen Chen[1], Nairen Cao[2], Siu-Ming Yiu[1] and Zheli Liu[3]
[1]*Department of Computer Science, The University of Hong Kong, HKSAR, China*
[2]*Department of Computer Science, Georgetown University, U.S.A.*
[3]*College of Computer and Control Engineering, Nankay University, Tianjin, China*

Keywords: ORAM, Constant Communication Overhead, Oblivious Clear Algorithm.

Abstract: Oblivious RAM has been studied extensively. A recent direction is to allow the server to perform computations instead of being a storage device only. This model substantially reduces the communication between the server and the client, making constant bandwidth communication (the number of blocks transmitted) feasible. It is obvious that the larger the block size, the easier it is to construct a constant bandwidth ORAM scheme. Also, a lower bound of sub-logarithmic bandwidth was given if we do not use expensive homomorphic multiplications. The question of "whether constant bandwidth with smaller block size without homomorphic multiplications is achievable" remains open. In this paper, we show that the block can be further reduced to $O(log^3N)$ using only additive homomorphic operations. 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.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

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 accessed by the client. It is one of efficient approaches to protect access pattern(di Vimercati et al., 2011; di Vimercati et al., 2016). The concept of ORAM, with a hierarchical structure, was first proposed by Goldreich (Goldreich and Ostrovsky, 1996). Several works of hierarchical structure(Boneh et al., 2011; Goldreich, 1987; Goodrich and Mitzenmacher, 2011; Goodrich et al., 2011; Goodrich et al., 2012; Lu and Ostrovsky, 2013; Williams and Sion, 2012) have been proposed to improve the efficiency of ORAM. However, hierarchical structure has a drawback of poor worst-case efficiency. A breakthrough came from the novel tree-based structure proposed by Shi *et al.* (Shi et al., 2011), which was further improved by many subsequent works (e.g. (Chung et al., 2014; Gentry et al., 2013; Stefanov et al., 2013)). 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 cost of the hierarchical ORAM and achieves better efficiency.

The original 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 to serve a client request) from $O(\log^3 N)$ to $O(\log N)$ (Path ORAM (Stefanov et al., 2013)) where N is the number of data blocks. However, due to the simple server setting, constant or sub-logarithmic bandwidth seems not achievable.

Server-with-Computation ORAM. In order to overcome this bandwidth barrier, Mayberry *et al.* (Mayberry et al., 2014) proposed a new model that allows the server to perform some computations. The new model is widely used in cloud setting. In this model, the computations are outsourced to the server. And the client just sends messages to drive the computations and retrieve the required blocks. Intuitively, the client can instruct the server to "compress" all transmitted blocks and "hide" the required blocks inside before sending over to decrease the bandwidth.

Zhang, L., Zeng, G., Chen, Y., Cao, N., Yiu, S. and Liu, Z.
OC-ORAM: Constant Bandwidth ORAM with Smaller Block Size using Oblivious Clear Algorithm.
DOI: 10.5220/0007924701490160
In *Proceedings of the 16th International Joint Conference on e-Business and Telecommunications (ICETE 2019)*, pages 149-160
ISBN: 978-989-758-378-0
Copyright © 2019 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

But then at least one data block B must be returned from the server to the client. If we use the number of blocks to evaluate the bandwidth of ORAM schemes, constant bandwidth (i.e., $O(1)$ blocks) is the natural target. The size of a block, $|B|$, becomes an issue. The client sends instructions to drive operations on server. Aiming for obliviousness, each instruction should drive actions on all blocks in one path. So the size of an instruction is always related to the number of blocks in one path (i.e., related to the number of blocks in one bucket times the number of buckets). When considering constant bandwidth, the size of instructions should be bounded by the block size $|B|$. The larger the bucket size, the larger block size it is.

Under this model, Devadas *et al.* (Devadas et al., 2016) proposed a tree-based Onion ORAM scheme that achieves $O(1)$ communication overhead (constant number of blocks) by applying homomorphic multiplications, but it requires a large block size of $O(\log^5 N)$ to bound all intermediate transmitted messages. An open question is how small a block of an ORAM scheme can be while still achieving constant bandwidth.

Two improved schemes C-ORAM and CHf-ORAM were proposed. C-ORAM was proposed by Moataz *et al.* (Moataz et al., 2015b) 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 operation and Private Information Retrieval (PIR) operation. CHf-ORAM (Moataz et al., 2015a) also claimed 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, (Abraham et al., 2017) 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 a leakage of buckets' distribution by observing the behaviour of several evictions. (Abraham et al., 2017) later 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 read/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.

Very recently, a new distributed tree-based ORAM scheme (S^3ORAM) was proposed (Hoang et al., 2017) which achieves $O(1)$ client-server bandwidth. S^3ORAM does not rely on the direct computation of PIR, so the lower bound in (Abraham et al., 2017) does not apply. Technically, it uses Shamir Secret Sharing along with a secure multi-party multiplication protocol using more than one server to perform the eviction operations in Onion ORAM in a more efficient way. It is a breakthrough to use more than one server to reduce the communication bandwidth and further reduce the block size. Based on the communication analysis of (Hoang et al., 2017)[1], we conclude that the block size is $O(\log^4 N)$, and at least 3 servers are needed.

An open problem is whether the block size (the best result is $O(\log^4 N)$) can be further reduced while maintaining constant bandwidth. Additionally, notice that all schemes (Devadas et al., 2016; Moataz et al., 2015b; Abraham et al., 2017; Hoang et al., 2017; Moataz et al., 2015a) use the same eviction strategy, which leads to $O(\log N)$ bucket size. So the total server storage in these schemes is $O(N\log N)$ blocks. Whether the server storage can be reduced (the lower bound should be $O(N)$) is another open question.

1.1 Our Contributions

Over the past four years, researchers have come a long way to achieve constant bandwidth and reduce the block size in the new server-with-computation ORAM model. (Devadas et al., 2016) introduces an eviction algorithm, which first pushes all useful (real) blocks down on a chosen path from the root to the leaf node and then clears all the noisy blocks on the path except the leaf node. All previous works(Moataz et al., 2015b; Moataz et al., 2015a; Hoang et al., 2017) try to apply several technologies (PIR, Secret Sharing) to perform the eviction algorithm in (Devadas et al., 2016) more efficiently, but none of them try to improve the eviction algorithm itself. On the other hand, we notice that such an eviction algorithm causes a large bucket size ($O(\log N)$), which would finally cause a '$\log N$' factor in both block size and server storage. To further reduce the block size for constant bandwidth, we need to solve two problems: (i) de-

[1]The client-server communication complexity in (Hoang et al., 2017) is at least $O(|B| + (H+1)(2Z^2\lceil\log_2 p\rceil))$, where $|B|$ is the size of block, $H = O(\log N)$ is the height of binary tree, $Z = O(\log N)$ is the bucket size and p is a prime used in Shamir Secret Sharing. Note that when B is larger than $\lceil\log_2 p\rceil$, the client would split the data into equal-sized chunks. While each block B should contain an index and the size of the index is $O(\log N)$, $\lceil\log_2 p\rceil$ times the number of chunks should be at least $O(\log N)$. The paper does not count the number of chunks into bandwidth analysis (i.e., it is viewed as a constant). Actually, the $\lceil\log_2 p\rceil$ parts in its analysis means $\lceil\log_2 p\rceil$ times the number of chunks. Combining this parameters together, we can get $|B| \in O(|B| + (H+1)(2Z^2\lceil\log_2 p\rceil)) = O(\log^4 N)$.

riving an improved eviction strategy; and (ii) bypassing the lower bound barrier in (Abraham et al., 2017). The followings show how we tackle these two problems. By combining these two solutions, we come up with a novel two-server oblivious clear protocol which is the core component of our proposed OC-ORAM scheme.

1) Deriving an Improved Eviction Strategy. 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 $O(1)$ bucket size is proved large enough to achieve negligible overflow probability. 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.

2) Bypassing the Lower Bound Barrier. Recall that the lower bound in (Abraham et al., 2017) is derived by computing the number of operations (read, write, PIR-read and PIR-write). So this lower bound is on the number of operations. Under the condition that each of the 4 kinds of operations results in at least 1 block's bandwidth, the lower bound on the number of blocks is equal to the lower bound on the number of operations. If we can overcome the constraint that each operation corresponds to at least 1 block of bandwidth, it is possible to achieve constant bandwidth while using logarithmic number of operations. More precisely, if we can have a new operation OP with bandwidth $V < O(\frac{1}{\log_{cD} N})$ blocks, by calling this operation frequently ($O(\log N)$ times) constant bandwidth is achievable.

Combining the two ideas, we introduce a **Two-server Oblivious Clear Protocol (2SOC Protocol)**. 1) The bandwidth overhead of this protocol is $V = O(\log N + |\mathcal{M}|)$ bits, where $|\mathcal{M}|$ is the length of plaintext. It will be proved later that such bandwidth is obvious smaller than $O(\frac{1}{\log_{cD} N})$ blocks. 2) 2SOC Protocol can be used in ORAM scheme to clear the 'useless' blocks. So our improved eviction algorithm can be performed efficiently.

Applying this algorithm, we propose a new ORAM scheme achieving $O(1)$ bandwidth overhead. Compared with the existing secure constant communication construction Onion ORAM(Devadas et al., 2016) and S^3ORAM(Hoang et al., 2017), our scheme only requires a block size of $O(\log^3 N)$ with a small constant factor. Different from others, we focus on improving the eviction algorithm to reduce the bucket size. It brings another benefit that the server side storage is reduced to $O(N)$ blocks. Table 1 summarizes our improvements when compared to existing schemes in server-with-computation model. Note that besides reducing the block size, we achieve the optimal server storage, which closes the gap between the upper bound and the lower bound.

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 (OC-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 $O(log^3 N)$. We remark that we can also achieve optimal server storage of $O(N)$.

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

2 PRELIMINARIES

2.1 ORAM Basics

We now present the basics of ORAM, to illustrate its important features.

We build on the tree-based ORAM framework of (Shi et al., 2011), which organizes server storage as a binary tree of nodes. In all tree-based ORAMs, each block is mapped to a random path and each block can only live in a bucket along that path or in client side at any time. The client maintains a local position map to store the path mapped to each block. In most cases, there is a stash in the client side, which is used to store the blocks returned by the server temporarily.

When blocks are added to the ORAM, they start at the root of the tree. As access operations continue, the ORAM needs an eviction process that pushes blocks

Table 1: Comparison with existing sever-with-computation ORAM schemes.

Scheme	Bandwidth overhead	total # of bits (bandwidth)	Block size (bits)	Server storage	# of servers	Secure?
Onion ORAM	$O(1)$	$O(\log^5 N)$	$O(\log^5 N)$	$O(N \log N)$	1	Yes
C-ORAM	$O(1)$	$O(\log^4 N)$	$O(\log^4 N)$	$O(N \log N)$	1	No
CHf-ORAM	$O(1)$	$O(\log^3 N)$	$O(\log^3 N)$	$O(N \log N)$	4	No
ORAM in (Abraham et al., 2017)	$O(\frac{\log N}{\log\log N})$	$O(\frac{\log^4 N}{\log\log N})$	$O(\log^3 N)$	$O(N \log N)$	2	Yes
S^{3}ORAM	$O(1)$	$O(\log^4 N)$	$O(\log^4 N)$	$O(N \log N)$	≥ 3	Yes
OC-ORAM	$O(1)$	$O(\log^3 N)$	$O(\log^3 N)$	$O(N)$	2	Yes

towards their tagged leaves. Usually, this is accomplished by picking a path in the tree and pushing all the blocks on that path as far as possible towards the leaf node.

To avoid incurring large client storage, the position map should be recursively stored in other smaller ORAMs (Shi et al., 2011). When the data block size is $\Omega(\log^2 N)$ bits for an N block ORAM (which is the case for our result), the asymptotic costs of recursion are insignificant relative to the main ORAM(Stefanov et al., 2013). So we no longer consider the bandwidth cost of recursion in the rest of the paper.

We show the definition of Access Pattern and security of ORAM refer to (Stefanov et al., 2013)

Definition 1. Access Pattern: Let $A(\vec{y})$ denote the sequence of access to the remote server storage given the data request sequence $\vec{y}$ from client. Specifically, $\vec{y} = \{(op_{L'}, a_{L'}, data_{L'}), ..., (\mathsf{op}_1, \mathsf{a}_1, \mathsf{data}_1)\}$, $L' = |\vec{y}|$, and op_i denotes a operation of either $read(a_i)$ or $write(a_i, data_i)$. In addition, a_i is logical identifiers of the block.

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

2.2 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(Chor et al., 1995). 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(Cachin et al., 1999; Gentry and Ramzan, 2005; Kushilevitz and Ostrovsky, 1997) designed by applying homomorphic encryption have been used in (Devadas et al., 2016; Moataz et al., 2015b) to reduce bandwidth overhead. PIR on two or more non-colluding servers is based on very simple operations such as XOR operation, which has been used in (Moataz et al., 2015a). In this paper, we will use a variant of two-server PIR and we will show the details in Section 4.

3 A NEW EVICTION AND CLEAR ALGORITHM

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

We introduce how this algorithm comes up when considering how to derive new eviction algorithm and reduce the buckets' load.

Aiming for $O(1)$ bucket size, inspired by Path ORAM(Stefanov et al., 2013) and Circuit ORAM(Wang et al., 2015), we propose a new eviction algorithm that just move some blocks (at most 1 block from each bucket) to one of its child along the eviction path. However, after this eviction, the real blocks and noisy blocks will reside in the same buckets on the path. So we cannot directly clear all the buckets as in (Moataz et al., 2015b). Therefore, we design a new algorithm that supports oblivious clearing of the noisy blocks while keeping other's plaintext unchanged[2]. It is obvious that this algorithm could change the system states while being hidden from the server and can be performed frequently, so it can be

[2]Oblivious clearing means that change the data in the slot into $Enc(0)$, and keeping other unchanged means that re-encrypt the data in the slots.

used to overcome the limitation of the lower bound proof if we can realize it with small bandwidth.

This clear algorithm can be easily realized if fully homomorphic encryption (FHE) is used to encrypt data. We can send a sequence of $Enc(0)$ and $Enc(1)$, where each $Enc(0)$ is corresponding to the noisy block and $Enc(1)$ is corresponding to the real block. However, FHE is too expensive and far from practical. When considering only use additively homomorphic encryption (AHE), we find that it is hard to achieve.

Inspired by (Catalano and Fiore, 2015), we use two non-colluding servers A and B to design our algorithm. Novelly, we store an encryption of m on server A and an encryption of $-m$ on server B, where m is the data of one block. Then, by multiplying same coefficient k on m, $-m$ and summing them up we can get 0. And by multiplying different coefficient k, $k-1$ on m, $-m$ and summing them up we can get m. Additionally, in order to preserve obliviousness, the message sent by the client to drive the clear operation should not leak any information about which block is cleared.

Based on the above ideas, we store $Enc(x), m+x$ on server A and $Enc(y), -m+y$ on server B, where x, y are two random numbers and Enc is an AHE. As proved in (Catalano and Fiore, 2015), as long as A and B are non-colluding, each server cannot recover m. For each noisy block, the client sends the same random numbers (k_1, k_2) to A and B, then A will get $Enc(k_1(x+y)), k_1(x+y)$ and B will get $Enc(k_2(x+y)), k_2(x+y)$, both of which can be viewed as an encryption of 0. For each real block, the client sends $(k_1', k_2'-1)$ to A and $(k_1'-1, k_2')$ to B, then A will get $Enc(k_1'x+(k_1'-1)y), m+k_1'x+(k_1'-1)y$ and B will get $Enc(k_2'y+(k_2'-1)x), -m+k_2'y+(k_2'-1)x$, both of which can be viewed as a re-encryption of m.

Now, we can present our 2-server oblivious clear protocol. This protocol can be applied to 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 comes the definitions of 2SOC Protocol.

Definition 3. 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 the security parameter λ and outputs a secret key **sk** and a public key **pk** by running $\mathcal{PKE}.Setup(1^\lambda)$. The algorithm generates an operation sequence $\mathbf{OP} = \{op_1, \dots, op_k\}$ 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 $(op_1, \dots, op_k) \in \mathsf{OP}$ 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 $(op_1, \dots, op_k) \in \mathsf{OP}$ 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 the pair $(C^{(i)}, V_b^{(i)})$, $i = 1$ or 2 should not learn anything about both the plaintext and the value of b. We formalize this property using the approach of indistinguishable security.

Definition 4. (2SOC Indistinguishable Security) Let 2SOC be a 2SOC protocol as defined above, and $\mathcal{A}$ be a PPT adversary. Consider the following experiment:

```
Experiment $\mathbf{Exp}^{2S.IND}_{2SOC,\mathcal{A}}(\lambda)$
$(pk, sk) \leftarrow 2SOC.KeyGen(1^\lambda)$
$(m, i) \leftarrow \mathcal{A}(pk)$
$(C^{(1)}, C^{(2)}) \leftarrow 2SOC.Encrypt(pk, m)$
$(V_b^{(1)}, V_b^{(2)}) \leftarrow 2SOC.VecGen(b, pk)$
$TM_b^{(1)} \leftarrow 2S_b(C^{(1)}, V_b^{(1)})$; $TM_b^{(2)} \leftarrow 2S_b(C^{(2)}, V_b^{(2)})$
$b' \leftarrow \mathcal{A}(C^{(i)}, V_b^{(i)}, TM_b^{(1)}, TM_b^{(2)})$
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. $2S_0$ means the protocol performing ‘clear noisy’ operation and $2S_1$ means the protocol performing ‘keep real’ operation. Let $Adv^{2S.IND}_{2SOC,\mathcal{A}}(\lambda) = Pr[\mathrm{Exp}^{2S.IND}_{2SOC,\mathcal{A}}(\lambda)] - \frac{1}{2}$. 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 reduce bucket size. The constructions of 2SOC and OC-ORAM will be shown in section 4. We will discuss how 2SOC bypasses the lower bound.

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 buckets after eviction. When considering the security of ORAM scheme, the server cannot distinguish which block is noisy. So the 2SOC protocol should be IND-secure (defined in Definition 4). 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 b_i in D. After some computations between w_i and b_i, the data block is either cleared or re-encrypted. The different parts of the vector W_D should be indistinguishable to any adversary. We will first show our overall OC-ORAM construction and then follow by the 2SOC protocol construction.

4.1 OC-ORAM Construction

Our ORAM builds on the tree-based ORAM framework and shares many similar features with most tree-based constructions.

Stash. When the client reads or writes a block, this block will be added into the stash. The stash is a linear structure of size $R = O(\log N)$ in the secure storage on the client side.

Position Map. The client stores a position map. $x := position[b]$ means that block b is currently mapped to the x-th leaf node, i.e., block b resides in some bucket in the path from the root to the x-th leaf node. The size of position map is $N \log N$ bits. As shown in Section 2.1, we can leverage the recursion technique to reduce the client side storage.

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 $Z = \mu \cdot z$ blocks, where z is a constant indicates the number of slots needed to hold real 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.

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.

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 (Paillier, 1999)). Therefore, each 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. 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. Denote that b_i is that i-th block in the path $Path(tag)$ and j is the corresponding position of block b in this path.
- Then the client generates two vectors $E_1 = (e_{11}, e_{12}, \ldots, e_{1n})$, $E_2 = (e_{21}, e_{22}, \ldots, e_{2n}) \in \{0,1\}^n$, where $e_{1i} = e_{2i}$ for all $i \neq j$ and $e_{1j} \neq e_{2j}$. Then the client sends E_1 to server A and E_2 to server B.
- After receiving vector from the client, server A does $\alpha_1 = \sum_i e_{1i}(m_i + x_i)$ on the second part of block, and does $\beta_1 = \oplus_i e_{1i} Enc(x_i)$ on the first part of block, where $\oplus$ represents the homomorphic addition computation and m_i, x_i is the corresponding content in block b_i. Similarly, server B does $\alpha_2 = \sum_i e_{2i}(-m_i + y_i)$ and $\beta_2 = \oplus_i e_{2i} Enc(y_i)$. Then server A returns α_1, β_1 to the client and server B returns α_2, β_2 to the client.
- After receiving responses from two servers, the client does $\alpha = \alpha_1 + \alpha_2 = \sum_i (m_i(e_{1i} - e_{2i})) + \sum_i (e_{1i}x_i + e_{2i}y_i)$ and $\beta = Dec(\beta_1 \oplus \beta_2) = \sum_i (e_{1i}x_i + e_{2i}y_i)$, where Dec is the decryption algorithm. Finally, the client does $(e_{1j} - e_{2j})(\alpha - \beta) = (e_{1j} - e_{2j})m_j(e_{1j} - e_{2j}) = m_j$. So the client can recover the data m_j of requested block b.
- 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 predetermined 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 details will be shown below and the security analysis is in Section 5.

Eviction Algorithm. Since no more than constant date blocks could be downloaded and only small permutations or vectors 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(Wang et al., 2015), 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 $Path(tag)$. Then the client can drive the move down process by small bandwidth messages (such as permutations).

Pre-processing Algorithm. The inputs of the pre-processing algorithm are a set of headers. The outputs of the algorithm is a sequence of destinations of each block. The details of the algorithm can be found in Algorithm 1(Appendix A), in which PrepareDeepest and PrepareTarget are two sub-algorithms in (Wang et al., 2015). The first outputs an array $deepest[1,\ldots,L]$, 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. Due to the space limitation, we omit the details of these two sub-algorithms. Readers can refer to (Wang et al., 2015) for details. 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 has to be moved down in each bucket along the evict path based on the results from the pre-processing algorithm. And there is no intersection bucket between the moving ways of any two data blocks. In order to keep obliviousness, this block should be moved down along the path one bucket by one bucket, i.e., if we want to move block b from B_i to B_{i+j}, then we have to move it to B_{i+1} firstly and then to B_{i+2} and arrive B_{i+j} after j moves. Without loss of generality, we show the approach of how to move block b from B_i to B_{i+1}.

- 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'.
- Secondly, 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" slots.
- Thirdly, the server performs Π to B_i and merges it into B_{i+1} by homomorphic additions.
- Finally, update the headers H_i, H_{i+1}, and delete the duplicate H_i'.

Clear Noisy Operation. After the 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 for the following move down operation. We show the simple algorithm and its improvement. Both of them are built on a two-server oblivious clear protocol(2SOC Protocol).

By sending two clear vectors to two servers respectively, 2SOC protocol supports performing clear noisy and keeps real operations obliviously. For convenience, we use α to denote the size of each element in clear vector. The length of the clear vector is equal to the bucket size($\mu z = O(1)$) since each element 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.

Next, we introduce an improved clear noisy algorithm to further reduce the bandwidth:

- At the beginning of an evict operation, the client generates a configuration of bucket D_1 randomly together with a corresponding header H_{D_1}. D_1 includes z real blocks and $\mu z - z$ empty blocks. Similarly, a configuration of bucket D_2 randomly together with a corresponding header H_{D_2}. D_2 includes $z+1$ real blocks and $\mu z - (z+1)$ empty blocks is also generated by the client. Then two pairs of clear vectors (W_{D1}^A, W_{D1}^B) for D_1 and (W_{D2}^A, W_{D2}^B) 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 same 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 always exist.

- Finally we use $(W^A_{D1}, W^B_{D1}), (W^A_{D2}, W^B_{D2})$ for the two buckets to do the 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 empty block.

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 the original one when $\alpha > \log N$, which is always true. In the rest of paper, we will use this improved clear noisy operation. In Section 5, we will prove that such permutations will never leak any information related to the bucket load to the server, with the help of the reverse lexicographic eviction order.

4.3 2SOC 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 (Catalano and Fiore, 2015), and followed by the construction of the clear vector.

2SOC Protocol is a technique that obliviously performs one of these two operations: 1) Change data into $Enc(0)$; 2) Keep the value unchanged and re-encrypt it. By sending clear vectors to servers and performing some computations between the vectors and data blocks, we can clear some blocks from noisy to 0 while keeping others' value unchanged. The protocol is based on any additively homomorphic encryption.

Before showing the construction, there is a very mild property (i.e. public-space homomorphic encryption) that the additively homomorphic encryption needs to satisfy. The details can be referred to (Catalano and Fiore, 2015). 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 also satisfy our notion of public-space.

A 2SOC Protocol. Our construction builds upon a public-space additively homomorphic encryption. The construction includes three parts, server A, server B and client. Now comes the precise descriptions of our scheme (based on a public-space additively homomorphic encryption $\mathcal{HE} = (KeyGen, Enc, Dec)$), where $\oplus$ is the homomorphic addition:

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 $\mathcal{M}$. We can make this condition easy to satisfy by choosing suitable additively homomorphic encryption scheme. Then send $V^{(1)}_0 = V^{(2)}_0 = (k_1, k_2)$ to A and B respectively. Then, A computes that $C^{(1)}_{k_1} = (k_1 \cdot c_1, k_1(m+x))$, $C^{(1)}_{k_2} = (k_2 \cdot c_1, k_2(m+x))$, and sends $C^{(1)}_{k_2}$ to B. B computes that $C^{(2)}_{k_1} = (k_1 \cdot c_2, k_1(-m+y))$, $C^{(2)}_{k_2} = (k_2 \cdot c_2, k_2(-m+y))$, and sends $C^{(2)}_{k_1}$ to A. Finally, A does that $C'^{(1)} = ((k_1 \cdot c_1) \oplus (k_1 \cdot c_2), k_1(m+x) + k_1(-m+y)) = (Enc(pk, k_1(x+y)), k_1(x+y))$, 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 send $V^{(1)}_1 = (k'_1, k'_2 - 1)$ to A and sends $V^{(2)}_1 = (k'_1 - 1, k'_2)$ 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 IND secure under our the definition in Section 3. We formalize this property in the following theorem.

Theorem 2. *If $\mathcal{HE}$ is IND-CPA secure, then Our 2S-DOCA protocol is a IND secure protocol under Definition 4.*

Proof sketch. We proof Theorem 2 via a sequence of hybrid experiments from the IND-security game, and then we show the experiments are indistinguishable. Through the sequence of experiments, we show that it gives the adversary the same view by replacing b by $1-b$ under the condition that the encryption scheme is IND-secure. The details of the proof can be found in Appendix B

4.3.1 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 $W^A_D = (w^A_{D1}, ..., w^A_{D\mu z})$ for server A and $W^B_D =$

$(w^B_{D1},...,w^B_{D\mu z})$ for server B. If $b_i, i \in \{1,...,\mu z\}$ is a real block, then set $W^A_{Di} = (k'_1, k'_2 - 1), W^B_{Di} = (k'_1 - 1, k'_2)$ according to the above 2SOC protocol. Otherwise if $b_j, j \in \{1,...,\mu z\}$ is a noisy or empty block, then set $W^A_{Di} = (k_1,k_2), W^B_{Di} = (k_1,k_2)$ similarly. Finally, the client sends W^A_D to server A and W^B_D to server B and lets them perform the computations defined in the protocol.

5 OC-ORAM ANALYSIS

We analyze the bandwidth overhead and server storage first, which leads to our main result. The proof of correctness and security (which are defined in Definition 2) of OC-ORAM will be given in Section 5.2 and full version.

5.1 Bandwidth Overhead and Storage Analysis

In this section, we discuss the bandwidth overhead and the server storage size of OC-ORAM. The bandwidth in OC-ORAM contains two parts: Client-Server communication bandwidth and Server-Server communication bandwidth. And the block size can be derived by Client-Server communication bandwidth overhead.

Client-server Bandwidth Evaluations and Block Size. The bandwidth overhead is defined as the total amount of communication in one Access process. The Access is composed of scheduled evict operations, a PIR read, and no more than two PIR writes. Compared to the PIR read and write vectors, the size of headers ($O(\mu z \log N)$) are negligible. For the sake of clarity, we therefore avoid including them in our asymptotic analysis.

The moving down operation of eviction process always involves $O(\log N)$ permutations with size $O(\mu z \log \mu z)$ 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 obvious 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 $O(2\log N \mu z \log \mu z + 2\mu z|\mathcal{M}| + \mu z \log N + 2\gamma \mu z + |B|) = O(\log N \mu z \log \mu z + \gamma \mu z + |B|)$.

To have constant bandwidth, the block size should be $|B| = O(\log N \mu z \log \mu z + \gamma \mu z)$. With $z = O(1)$ and $\mu = O(1)$, we achieve $|B| = O(\log N + \gamma)$. In practice, we choose $\gamma \in O(\lambda^3)$ and $\lambda \in \omega(\log N)$, the parameter set of which is the same as Onion ORAM(Devadas et al., 2016), so γ dominates $\log N$. Therefore, block size is $|B| = O(\gamma) = O(\log^3 N)$. And the constant factor of $|B|$ is not large, since we will show that $\mu > 2$ and $z \geq 4$ is enough to avoid overflowing in section 5.2.

Additionally, notice that the bandwidth overhead of the clear noisy operation is $O(\mu z \log \mu z \log N + \mu z|\mathcal{M}|) = O(\log N + |\mathcal{M}|)$ bits. When setting $|\mathcal{M}| = \log N$, the bandwidth is $O(\log N)$ bits, which is smaller than $O(\frac{1}{\log_{cD} N})$ blocks ($O(\frac{1}{\log_{cD} N})$ blocks equals to $O(\frac{1}{\log_{cD} N}) \times |B| > \log N$ bits). So our new clear algorithm can bypass the lower bound in (Abraham et al., 2017).

Server-server Bandwidth Evaluations. The communication between two servers mainly occurs in the evict operation. For each bucket in the evict 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 (Moataz et al., 2015b; Hoang et al., 2017). The communication between servers is much cheaper than it between the client and the server, so our scheme is more practical than their schemes.

Server Storage Size. 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 existing constructions (Moataz et al., 2015b; Devadas et al., 2016; Moataz et al., 2015a; Abraham et al., 2017; Hoang et al., 2017), the size of each bucket is $O(\log N)$ blocks, so the total server side storage is $O(N \log N)$ blocks. Compared with them, we reduce a *logN* factor in the server storage.

5.2 Correctness Analysis for OC-ORAM

In this section, we discuss the correctness of OC-ORAM. As defined in Definition 2, the correctness states that the ORAM returns correct results for any input sequence $\vec{y}$ with probability $\leq 1 - negl(|\vec{y}|)$. Alternatively, we can prove the correctness by showing the probability of a failure occurs is negligible. To begin with the analysis, we outline two failure types in OC-ORAM:

- F_1: Blocks with value of encrypted *zero* in the evict path is less than z
- F_2: Overflow of the stash on the client side

Lemma 1. *When the constant factor $\mu > 2$ (which is always true in OC-ORAM), the number of blocks with value of encrypted zero in each bucket along the evict path is at least z after the eviction.*

The proof of Lemma 1 can be found in Appendix B.

Lemma 2. *Let $z \geq 4$. Let $st(ORAM^{\mu z}[\vec{s}])$ be a random variable denoting the stash size after access sequence $\vec{s}$ for OC-ORAM with bucket size μz ($\mu > 2$). Then, for any access sequence $\vec{s}$,*

$$Pr[st(ORAM^{\mu z}[\vec{s}]) > R] \leq e^{-R}$$

where probability is taken over the ORAM algorithm's randomness.

The proof of Lemma 2 can be refer to (Wang et al., 2015).

Theorem 3. *OC-ORAM is a correct ORAM scheme by Definition 2 (1).*

The proof of Theorem 3 can be found in Appendix B.

6 CONCLUSIONS

In this paper, we propose a secure constant bandwidth ORAM scheme (OC-ORAM) with smaller block size. Recently, (Abraham et al., 2017) gives the lower bound in number of operations is $O(\log_{cD} N)$ when combining ORAM with PIR operations. However, the operations (read, write, PIR read and PIR write) involved in lower bound computation in existing schemes have large bandwidth (at least $O(1)$ blocks). Therefore the lower bound in bandwidth is also $O(\log_{cD} N)$. According to our analysis, if we can design a new operation that has small bandwidth, then it is possible to achieve constant bandwidth while use logarithmic operations. Technically, we propose a new 2-server oblivious clear protocol (2SOC Protocol) which is proved IND-secure, and is applied in our eviction phase to achieve constant bandwidth ORAM. 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.

REFERENCES

Abraham, I., Fletcher, C. W., Nayak, K., Pinkas, B., and Ren, L. (2017). Asymptotically tight bounds for composing oram with pir. In *IACR International Workshop on Public Key Cryptography*, pages 91–120. Springer.

Boneh, D., Mazieres, D., and Popa, R. A. (2011). Remote oblivious storage: Making oblivious ram practical.

Cachin, C., Micali, S., and Stadler, M. (1999). Computationally private information retrieval with polylogarithmic communication. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 402–414. Springer.

Catalano, D. and Fiore, D. (2015). 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*, pages 1518–1529. ACM.

Chor, B., Goldreich, O., Kushilevitz, E., and Sudan, M. (1995). Private information retrieval. In *Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on*, pages 41–50. IEEE.

Chung, K.-M., Liu, Z., and Pass, R. (2014). Statistically-secure oram with\ tilde {O}(\ logˆ 2 n) overhead. In *Advances in Cryptology–ASIACRYPT 2014*, pages 62–81. Springer.

Devadas, S., van Dijk, M., Fletcher, C. W., Ren, L., Shi, E., and Wichs, D. (2016). Onion oram: A constant bandwidth blowup oblivious ram. In *Theory of Cryptography*, pages 145–174. Springer.

di Vimercati, S. D. C., Foresti, S., Moretti, R., Paraboschi, S., Pelosi, G., and Samarati, P. (2016). A dynamic tree-based data structure for access privacy in the cloud. In *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 391–398. IEEE.

di Vimercati, S. D. C., Foresti, S., Paraboschi, S., Pelosi, G., and Samarati, P. (2011). Efficient and private access to outsourced data. In *2011 31st International Conference on Distributed Computing Systems*, pages 710–719. IEEE.

Gentry, C., Goldman, K. A., Halevi, S., Julta, C., Raykova, M., and Wichs, D. (2013). Optimizing oram and using it efficiently for secure computation. In *Privacy Enhancing Technologies*, pages 1–18. Springer.

Gentry, C. and Ramzan, Z. (2005). Single-database private information retrieval with constant communication rate. In *International Colloquium on Automata, Languages, and Programming*, pages 803–815. Springer.

Goldreich, O. (1987). Towards a theory of software protection and simulation by oblivious rams. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 182–194. ACM.

Goldreich, O. and Ostrovsky, R. (1996). Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)*, 43(3):431–473.

Goodrich, M. T. and Mitzenmacher, M. (2011). Privacy-preserving access of outsourced data via oblivious ram simulation. In *Automata, Languages and Programming*, pages 576–587. Springer.

Goodrich, M. T., Mitzenmacher, M., Ohrimenko, O., and Tamassia, R. (2011). Oblivious ram simulation with efficient worst-case access overhead. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 95–100. ACM.

Goodrich, M. T., Mitzenmacher, M., Ohrimenko, O., and Tamassia, R. (2012). Practical oblivious storage. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, pages 13–24. ACM.

Hoang, T., Ozkaptan, C. D., Yavuz, A. A., Guajardo, J., and Nguyen, T. (2017). S3oram: A computation-efficient and constant client bandwidth blowup oram with shamir secret sharing. Cryptology ePrint Archive, Report 2017/819. http://eprint.iacr.org/2017/819.

Kushilevitz, E. and Ostrovsky, R. (1997). Replication is not needed: Single database, computationally-private information retrieval. In *Foundations of Computer Science, 1997. Proceedings., 38th Annual Symposium on*, pages 364–373. IEEE.

Lu, S. and Ostrovsky, R. (2013). Distributed oblivious ram for secure two-party computation. In *Theory of Cryptography*, pages 377–396. Springer.

Mayberry, T., Blass, E.-O., and Chan, A. H. (2014). Efficient private file retrieval by combining oram and pir. In *NDSS*. Citeseer.

Moataz, T., Blass, E.-O., and Mayberry, T. (2015a). Chf-oram: a constant communication oram without homomorphic encryption. Technical report, Cryptology ePrint Archive, Report 2015/1116.

Moataz, T., Mayberry, T., and Blass, E.-O. (2015b). Constant communication oram with small blocksize. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 862–873. ACM.

Paillier, P. (1999). Public-key cryptosystems based on composite degree residuosity classes. In *Advances in cryptology-EUROCRYPT'99*, pages 223–238. Springer.

Shi, E., Chan, T.-H. H., Stefanov, E., and Li, M. (2011). Oblivious ram with o ((logn) 3) worst-case cost. In *Advances in Cryptology–ASIACRYPT 2011*, pages 197–214. Springer.

Stefanov, E., Van Dijk, M., Shi, E., Fletcher, C., Ren, L., Yu, X., and Devadas, S. (2013). Path oram: An extremely simple oblivious ram protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 299–310. ACM.

Wang, X., Chan, H., and Shi, E. (2015). Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 850–861. ACM.

Williams, P. and Sion, R. (2012). Single round access privacy on outsourced storage. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 293–304. ACM.

APPENDIX

A PRE-PROCESSING ALGORITHM IN SECTION 4.2

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$\mu z$ 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
```

B PROOFS

B.1 Proof of Theorem 2

Proof. We proof the security via a sequence of hybrid experiments, and then we show they are indistinguishable.

Hybrid H1: This is the IND game (Table 2).

Table 2: H1: IND-Security Game.

```
Experiment $\mathbf{Exp}^{2S.IND}_{2SOC,\mathcal{A}}(\lambda)$
$(pk, sk) \leftarrow 2SOC.KeyGen(1^{\lambda})$
$(m, i) \leftarrow \mathcal{A}(pk)$
$(C^{(1)}, C^{(2)}) \leftarrow 2SOC.Encrypt(pk, m)$
$(V_b^{(1)}, V_b^{(2)}) \leftarrow 2SOC.VecGen(b, pk)$
$TM_b^{(1)} \leftarrow 2S_b(C^{(1)}, V_b^{(1)}); TM_b^{(2)} \leftarrow 2S_b(C^{(2)}, V_b^{(2)}$
$b' \leftarrow \mathcal{A}(C^{(i)}, V_b^{(i)}, TM_b^{(1)}, TM_b^{(2)})$
If $b' = b$ return 1. Else return 0.
```

Hybrid H2: This is like H2 except that the vectors are generated by invoking the algorithm $H2.VecGen$ defined as follows:
$H2.VecGen(b, pk)$: The algorithm calls $V_b^{(i)} = (k_1, k_2) \leftarrow 2SOC.VecGen(b, pk)$. If $b = 0$, then sets $k_1' = k_1 + (i-1)$, $k_2' = k_2 + (2-i)$. And returns $V_b^{'(i)} = (k_1' - (i-1), k_2' - (2-i))$. Else if $b = 1$, then sets $k_1' = k_1$, $k_2' = k_2$. And returns $V_b^{'(i)} = (k_1', k_2')$.

Table 3: H2 Experiment.

```
Experiment $\mathbf{Exp}^{2S.IND}_{2SOC,\mathcal{A}}(\lambda)$
$(pk, sk) \leftarrow 2SOC.KeyGen(1^{\lambda})$
$(m, i) \leftarrow \mathcal{A}(pk)$
$(C^{(1)}, C^{(2)}) \leftarrow 2SOC.Encrypt(pk, m)$
$(V_b^{(1)}, V_b^{(2)}) \leftarrow H2.VecGen(b, pk)$
$TM_b^{(1)} \leftarrow 2S_b(C^{(1)}, V_b^{(1)}); TM_b^{(2)} \leftarrow 2S_b(C^{(2)}, V_b^{(2)}$
$b' \leftarrow \mathcal{A}(C^{(i)}, V_b^{(i)}, TM_b^{(1)}, TM_b^{(2)})$
If $b' = b$ return 1. Else return 0.
```

Notice that if $b = 0$, $k_1' - (i-1) = (k_1 + (i-1)) - (i-1) = k_1$, $k_2' - (2-i) = (k_2 + (2-i)) - (2-i) = k_2$. Else if $b = 1$, we also have $k_1' = k_1$ and $k_2' = k_2$. So, H1 and H2 are indistinguishable.

However, after calling $H2.VecGen$, the function of b (clear noisy or keep real) has been reversed, i.e. if $b = 0$ (which means than the operation is supposed to be 'clear noisy'), the experiment does 'keep real' operation (based on k_1', k_2') actually and vice versa.

Hybrid H3: This is like H2 except that the vectors are generated by invoking the algorithm $2SOC.VecGen$. Notice that the elements in vectors (k_1, k_2) are generated randomly, and other computations are independent of b, so H2 and H3 are indistinguishable. Through this sequence of experiments, we show that it gives the adversary the same view by replacing b by $1-b$. Therefore, when the additive homomorphic encryption is IND-CPA secure, the advantage of the adversary is negligible. □

Table 4: H3 Experiment.

```
Experiment $\mathbf{Exp}^{2S.IND}_{2SOC,\mathcal{A}}(\lambda)$
$(pk, sk) \leftarrow 2SOC.KeyGen(1^{\lambda})$
$(m, i) \leftarrow \mathcal{A}(pk)$
$(C^{(1)}, C^{(2)}) \leftarrow 2SOC.Encrypt(pk, m)$
$(V_{1-b}^{(1)}, V_b^{(2)}) \leftarrow H2.VecGen(1-b, pk)$
$TM_{1-b}^{(1)} \leftarrow 2S_{1-b}(C^{(1)}, V_{1-b}^{(1)});$
$TM_{1-b}^{(2)} \leftarrow 2S_{1-b}(C^{(2)}, V_{1-b}^{(2)}$
$b' \leftarrow \mathcal{A}(C^{(i)}, V_{1-b}^{(i)}, TM_{1-b}^{(1)}, TM_{1-b}^{(2)})$
If $b' = b$ return 1. Else return 0.
```

B.2 Proof of Lemma 1

Proof. Each bucket contains at least $2z+1$ slots. In addition, there are at most $z+1$ real blocks in each bucket at any time in eviction (at most z original real 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 Clear noisy operation, the client generates clear vectors which keep 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 as the encryption of *zero*. so the number of blocks with value of encrypted *zero* along the evict path is at least z after the eviction operation. □

B.3 Proof of Theorem 3

Proof. By Lemma 1, we can infer that when a new evict operation begins, each bucket in the evict path has at least z empty slots. Each bucket should contain no more than z real blocks before a evict operation. So F_1 will never happen. Therefore it can be proved that the number of empty blocks is enough to perform the evict operation correctly. By Lemma 2, the probability of stash overflow is negligible. Combining them together, OC-ORAM is correct. □

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS