DivORAM: Towards a Practical Oblivious RAM with Variable Block Size

Zheli Liu[a], Yanyu Huang[a], Bo Li[a], Jin Li[b,*], Chao Shen[c], Xiaochun Cheng[d], Siu-Ming Yiu[e]

[a] *College of Computer and Control Engineering, Nankai University, Tianjin, China*
[b] *School of Computer Science, Guangzhou University, Guangzhou, China*
[c] *School of Electronic and Information Engineering, Xi'an JiaoTong University, Xi'an, China*
[d] *Department of Computer Science, Middlesex University, London, UK*
[e] *Department of Computer Science, University of Hong Kong, Hong Kong, China*

Abstract

Oblivious RAM (ORAM) is proposed to protect the access pattern in untrusted memory. In this paper, we proposed a brand new ORAM protocol, which supports storing blocks of different sizes. Our scheme remodels the tree structure and employs an additively homomorphic encryption scheme (Damgård-Jurik cryptosystem) to achieve constant communication complexity. DivORAM has multiple performance improvements. Firstly, Divoran's network bandwidth is 30% lower than Ring ORAM and 40% lower than HIRB ORAM for small client storage. Secondly, DivORAM greatly saves client computing overhead and improves storage capacity using a third party (trusted proxy). The response time of DivORAM is 10× improvement over Ring ORAM for practical parameters. The analysis of DivORAM is simple. To the best of our knowledge, DivORAM is the best instantiation of a variable block size ORAM in practice.

Keywords: Oblivious RAM, Privacy Protection, Server Computation.

*Corresponding author

Email addresses: liuzheli@nankai.edu.cn (Zheli Liu), onlyerir@163.com (Yanyu Huang), nankailibo@163.com (Bo Li), lijin@gzhu.edu.cn (Jin Li), cshen@sei.xjtu.edu.cn (Chao Shen), X.Cheng@mdx.ac.uk (Xiaochun Cheng), smyiu@cs.hku.hk (Siu-Ming Yiu)

1. Introduction

Most of traditional privacy protection depends on the unobtrusiveness of plaintext. Generally, most work is focused on updating and improving cryptography. However, the attacks caused by the leakage of data access patterns have gradually gained more and more attention. In the modern information service architecture, more and more applications store a great quantity of private data in outsourced devices or servers. Therefore, for an honest and curious server, it can analyze the access pattern to get some important privacy information that could potentially lead to privacy leaks without revealing the plaintext. It will lead to serious consequences. For example, a hospital stores privacy data such as patient information in an outsourced server. When the doctor accesses the data, it is reasonable for the server to estimate that the patient information accessed by the oncologist is mostly a cancer patient. Therefore, the server can recognize which category the stored data belongs to. In the meantime, when the same patient information is accessed more frequently, the server may think that the patient's medical information is given a high priority, which is likely to be a critically ill patient, thereby disclosing the patient's privacy.

Oblivious RAM (ORAM). ORAM first proposed by Goldreich and Ostrovsky [5], is a general cryptographic primitive which allows sensitive data accessed obliviously. The purpose of ORAM is hiding access patterns, through re-encrypting each data and confusing the storage location of data in every access. After the concept of ORAM was proposed, many ORAM mechanisms emerged. These mechanisms can be roughly classified into two types depending on the storage structure employed, layer-based ORAM [24, 26, 27, 28, 29, 30, 32] where data is store in several levels and tree-based ORAM [31, 23, 25, 19] where the storage construction is a binary tree. Meanwhile, more and more ORAM protocol are combined secure multi-party computation [34, 39, 6] or oblivious databases [10, 11] for better performance.

1.1. The vORAM and Challenges

Few works focuses on the variable block-size ORAM (vORAM) like [11]. The simplest way to solve the problem of variable block size is to fill all the blocks to the same length. Apparently, this method will make the server store a lot of invalid information. In the worst case, that is, small size data

Table 1: Comparison with typcial ORAMs with tree-based and variable block-size ORAM scheme. Server storage in all schemes can be set to $O(BN)$, B is the size of one data blocks, N is the maximum number of blocks stored in ORAM schemes.

Scheme	**Block size**	**Bandwidth**	**Computation**	**Client Storage**
Ring ORAM	64 bytes	$O(\log N)$	-	$n(T) \cdot A/2$
HIRB ORAM	$(20\|\tau\| + R)B$	$O(\log N)$	-	$\widetilde{O}(\log N)$
DivORAM	$O(\log^5 N)$	$O(1)$	$O(\log^3 N)$	$O(\log N)$

accounts for the majority of all data while large size data represents only a fraction of the total data, large storage space in the server is wasted to store meaningless information. In practice, this makes the entire ORAM impractical, while causing redundancy in storage space.

In order to solve the above problem, we try to split different size of data into fragments of the same size. Like tree-based ORAM, we choose a binary tree as the storage construction as well. Each node in the tree has several slots to store a fragment as a splinter.

Challenges. When blocks are divided into several splinters, we should use the tree's storage structure as reasonably as possible to store splinters that belong to the same block on the same path. How to store splinter without wasting storage space on the server has become a problem we need to solve. Meanwhile, how to read and write splinters that belong to the same block efficiently and at a fraction of the cost is one challenge we had when designing a solution. In previous tree-based ORAM scheme like [18] and [19], eviction will cause heavy communication overhead between the client and the server. In eviction process, the server must send a great deal of block to the client to re-encrypt and permutate the position of these blocks. In that way, the server knows nothing about whether or not to access the same block. We also hope to find a solution to reduce the computational and communication overheads of shuffle operations in the client.

1.2. Our contribution

In this paper, we proposed DivORAM to deal with the problem caused by variable block size. It is worth noting that we have redesigned the tree-based ORAM to rationalize the operation process. We adopt the PIR technique and set up a trusted proxy, so that the execution time and network bandwidth can

be tolerated by the practical application as much as possible. We compare bandwidth overhead with Ring ORAM and HIRB ORAM in Table1. The main contribution is as follows:

Optimize server storage. We redesign the structure of the tree with different bucket size instead of previously same bucket size. In addition, we "cut" a block to several splinters so that no longer need to padding the block, which results in a waste of server-side storage. In our brand new tree, each node stores several splinters and no longer stores a block. Every splinter which is belong to the same block will spread among the same path. In order for the upper nodes to have enough space to temporarily store splinters which have not been shuffled, we define the size of the parent node to be twice the size of the child node.

Constant communication. In DivORAM protocol, we achieve constant communication $O(1)$ in a complete access process. We adopt Private Information Retrieve (PIR) used in several ORAM schemes [21], [16] to minimize the bandwidth during read operation when we set the block size to $O(\log^5 N)$. The specific analysis is described in Section 5.1. Meanwhile, we transfer the evict operation to a trusted proxy so that the write operation was simplified as much as possible. In this paper, we pay attention to the communication overhead between the client and the server. Therefore, write operation is also constant bandwidth. Since eviction does not involve clients, we do not take into account the bandwidth of eviction.

Lightweight client load. The client undertakes little work during the whole visit. Compared to previous ORAM scheme, the client in DivORAM is no longer involved in shuffle work, and only need a small amount of the vector calculation.

2. Preliminary

In this section, we introduced the features and usage of homomorphic encryption. We also introduced the settings of the trusted proxy and given the standard definition of security.

2.1. Homomorphic Encryption

Definition 1 (Additive homomorphism)**.** *Let x, y denote the plaintext, and $\mathsf{E}(x)$ the homomorphic ciphertext of x, we have*

$$\exists \oplus, \forall x, y, \mathsf{D}(\mathsf{E}(x) \oplus \mathsf{E}(y)) = x + y$$

where $\oplus$ is an efficiently computable function.
In addition, there exists

$$\exists \oplus, \forall x, y, \mathsf{D}(\mathsf{E}(x) \otimes \mathsf{E}(y)) = \mathsf{E}(x \cdot y)$$

where $\otimes$ is scalar multiplication, i.e., repeated $\oplus$ operation.

The above properties can be used to implement Private Information Retrieval (PIR) [36], which use select vectors $\mathsf{E}(1)$ or $\mathsf{E}(0)$ to retrieves an encryption of block u_i from m data blocks u_1, u_2, ..., u_m. Apparently, the index of block u_i will not be revealed.

2.2. Security Definition

The basic standard definition is described in SSS [12], that is, the server as an adversary cannot learn anything about the access pattern. In other words, the following information should not be leaked: 1) which data is being accessed; 2) how old it is (when it was last accessed); 3) whether the same data is being accessed (linkability); 4) access pattern (sequential, random, etc); or 5) whether the access is a read or a write. Similarly, we do not consider about leakage of the timing channel, such as when or how frequently the client makes data requests.

Definition 2. *Let $\vec{y} := ((u_1, data_1), (u_2, data_2), ..., (u_M, data_M))$ denote a data request sequence of length M, where each u_i denotes the identifier of the block being accessed, and and $data_i$ denotes the data being written. Let λ be the security parameter, and $A(\vec{y})$ denote the (possibly randomized) sequence of accesses to the remote storage given the sequence of data requests $\vec{y}$. A oblivious RAM protocol Π is secure only if for any polynomial-time adversary ,*

$$|Pr(A_\Pi(\vec{y})) - Pr(A_\Pi(\vec{z}))| < nelg(\lambda)$$

where $\vec{y}$ and $\vec{z}$ have the same length and $nelg$ is a function negligible in λ.

3. DivORAM Protocol

In this section, we describe the details of DivORAM's protocol. First of all, we introduced the storage structure of each device in DivORAM and then described the concrete realization of each step in this protocol. All notations used throughout the rest of the paper are summarized in Table 2.

Table 2: ORAM parameters and notations

Notation	Meaning
N	Number of real data blocks in ORAM
L	Number of levels in ORAM
$position[u]$	l : The path which u is in
	i : The index of bucket which u in the path
	off : The offset of u in the bucket
$\mathsf{E}(x)$	The AHE encryption of x
s	The size of splinter u
sp_u	All the splinters of block u
Cache	The temporary storage in the client
A	Eviction rate

3.1. Storage construction

3.1.1. Server Storage

We set up two cloud servers as server side, one of which is mainly used to store the outsourced private data as a storage cloud. The other cloud acts as a trusted proxy for executing eviction. In the storage cloud, there are N blocks which stores in a binary tree of $N-1$ nodes where each node is a bucket. Each block of variable size is divided uniformly to at most $\log N$ splinters where a splinter has fixed size s. Levels in the tree are from 1 (the root) to L (the leaves) where $L = \log N$. Each bucket in level i has 2^i slots and each slot store a single splinter. In a steady state, each splinter will be distributed as much as possible in all buckets. Notice that the maximum size of a block is $(\log N) \cdot s$, but for some tiny blocks smaller than s, we pad these to s. For convenience, let sp_u denote all splinters of block u.

As for the trusted proxy (TP), it replaces the client's role in eviction as a third party in traditional tree-based ORAM. Its storage structure may be thought of as an array of many slots to store data sequentially. The proxy can be used to permute items so that the server will know nothing about the position of shuffled blocks. In addition, the trusted proxy can secretly

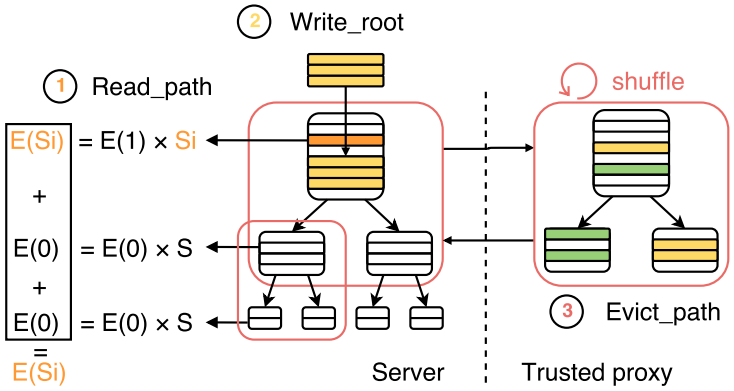

Figure 1: The process of access operations.

transfer the key with the client, and then re-encrypt all the blocks to ensure ciphertext non-repetitive. Our theoretical analysis in Section 5.2 will show that the maximum capacity of proxy is about $O(\log N)$.

3.1.2. Client Storage

The client consists of position map and cache with small storage. The position map is mapping table that maps each splinter of block to a random leaf and specific position in the path of corresponding leaf in the tree. The cache is a temporary storage to divide the blocks. The size of cache will be analyzed in Section 5.3.

3.2. Access process

The construction and process of DivORAM is shown in Figure 1. As in Algorithm 1, each access operation is consist of following three steps:

1. **Read a path** (Lines 2-17). The client looks up the position map to determine where the splinters of interested block store. Then the client sends $\log N$ select vector to retrieve all splinters, where each select vector has $\log^2 N$ items. Finally the client splices all splinters to restore a whole block.
2. **Write to root** (Lines 18-21). After retrieving the block, the client splices those splinters to a whole block. If the operation is write, the client changes this block to the new one. Then the client divides and re-encrypts the block and writes all splinters to the root of ORAM tree.

Algorithm 1 DivORAM Protocol

```
1: function ACCESS(u, op, data')
2:     counter = 0   //global variables
3:     l' ← UniformRandom(0, N)  //choose a fresh path randomly
4:     l ← PositionMap[u]        //get the path of block u
5:     PositionMap[u] ← l'       //reset position map
6:     for i ← 1 to log N do
7:         leaf ← leaf ∪ UniformRandom(0, N/2)
8:     end for
9:     leaf ← leaf ∪ l
10:    for i ← 1 to N/2 do
11:        if i ∈ leaf then
12:            data ← ReadPath(i, u)
13:        end if
14:    end for
15:    if data =⊥ then
16:        return ⊥
17:    end if
18:    if op = write then
19:        data ← data'
20:    end if
21:    WriteRoot(data) //write a block (may modified) back
22:    counter ← counter + 1  mod A
23:    if counter = 0 then
24:        EvictPath()       //evict a path after A access
25:    end if
26:    return data
27: end function
```

3. **Evict a path** (Lines 22-25). Our protocol evicts path in the order of reverse lexicographic like Ring ORAM [18]. Server sends all buckets and sibling buckets in the specified path to the trusted proxy. The proxy reshuffles all splinters in the buckets to the corresponding path and returns these buckets to the server. After A write operation, the ORAM tree will perform evict operation.

3.3. Read Operation

The *ReadPath* function is shown in Algorithm 2. Before reading the block u, the client first looks up position map to determine where sp_u are. Then the client generates $\log N$ select vectors for a path to retrieve $\log N$ splinters, each of which with length of $m \log N$. To select a splinter $sp[i] \in sp_u$, the input is $slot_1, slot_2, ..., slot_m$ and $v_1, v_2, ..., v_m$ where $v_{sp[i]} = 1$ and $v_i = 0$ for all other $i \neq sp[i]$. The server can perform homomorphic computation $sp[i] := (slot_1 \otimes v_1) \oplus (slot_2 \otimes v_2) \oplus ... \oplus (slot_m \otimes v_m)$. Notice that in one read operation, we read $\log N$ paths simultaneously to keep the obliviousness. After that, the client sends these select vectors to the server. For a single path, server performs $\otimes$ operation with corresponding slots and vectors resulting $\log N$ splinters. Server performs $\oplus$ operation with those splinters to get a single splinter. With the help of $\log N$ select vectors, we can get $\log N$ splinters to splice as a whole block. As we read $\log N$ paths, the server will figure out $\log N$ blocks. To achieve constant communication complexity, the server performs $\oplus$ operation with those blocks to get a single block and sends it to the client.

As $\otimes$ operation will resulting the layer of encryption, the client should decrypt the block twice to get the plaintext.

3.4. Write Operation

The *WriteRoot* function is shown in Algorithm 3. Before Writing a block with size B back, it should be divided into $\lceil B/s \rceil$ splinters. If the last splinter is smaller than s, it will be padded to s. After that, all splinters will be stored in cache. In an access operation, the splinters in cache will be written to the root or not, but if the cache is up to $\log N$, these will be written back for certain.

3.5. Evict Operation

The *EvictPath* function is shown in Algorithm 3. We define that *Evict-Path* happens after every A access. To shuffle as many blocks as possible in

Algorithm 2 ReadPath procedure

```
1: function ReadPath(l, u)
2:     splinter, data ←⊥
3:
4:     for i ← 1 to log N do
5:         //retrieve i-th splinter
6:         vector, index ← SelectVector(u, i)
7:         for j ← 1 to L do
8:             offset ← index[i]
9:             temp ← P(l, i, offset)
10:            splinter ← splinter ⊕ (temp ⊗ vector[i])
11:        end for
12:        data ← data|splinter
13:        //splice every splinter to a block
14:    end for
15:
16:    return data
17: end function
18:
19: function SelectVector(u, sp)
20:    l, pos[sp] ← PositionMap[u]
21:    c, off ← pos[sp]
22:    // sp-th splinter is in off offset of bucket c
23:    for i ← 1 to L do
24:        if i = c then
25:            index[i] ← off
26:            vector[i] ← E(1)
27:        else
28:            index[i] ← random
29:            vector[i] ← E(0)
30:        end if
31:    end for
32:    return vector, index
33: end function
```

Algorithm 3 WriteRoot procedure

```
1: function WriteRoot(data)
2:     splinter[ ] ← data   //divide a block to splinters
3:     cache ← cache ∪ splinter[ ]
4:     i ←$ {0,1}
5:     if i = 1 || cache is full then
6:         root ← cache
7:     end if
8: end function
```

a shorter period, the order that evicts the path is the *reverse lexicographic order*, which is first proposed by Gentry et al. [25]. Figure 2 shows the example of *reverse lexicographic order*. This order makes sure that most of the splinters are not shuffled repeatedly in consecutive eviction. Intuitively, the nodes of two paths performed in consecutive eviction will not overlap except for the root. It will help for reducing the frequency of eviction and improving its quality.

In each *EvictPath*, all the buckets and its sibling buckets will be send to the trusted proxy (TP). To minimize the storage of it, we gather a parent bucket, a child bucket and it sibling bucket as a *triad*, then send the *triad* to TP. All splinters will be shuffled thoroughly and each splinter will be assigned to the bucket in the corresponding path. Then the result will be returned to the server and stored in origin buckets. Until every bucket and its sibling bucket in this path are shuffled, *EvictPath* ended. Notice that each splinter must be re-encrypt so that the server cannot tell where the previous block is now.

As only simple decryption and encryption operations carried out in the proxy, the blocks transmitted back to the server will not exist ciphertext expansion caused by an increase in the number of ciphertext layers. Therefore, the blocks we store by default in the server have one and only one layer of encryption. Although there is a ciphertext expansion due to homomorphic computing operations during block reads, these expanded blocks are decrypted multiple times on the client and then encrypted once and written back. This does not affect the number of encrypted layers of shuffle blocks in the server and the trusted proxy.

Algorithm 4 EvictPath procedure

```
1: function EvictPath( )
2:     for i ← 1 to L do
3:         triad ← P(l, i) ∪ P(l, i+1) ∪ P(l+1, i+1)
4:         Send triad to trusted proxy
5:         reshuffle all splinters ∈ triad
6:         send triad back to server
7:     end for
8: end function
```

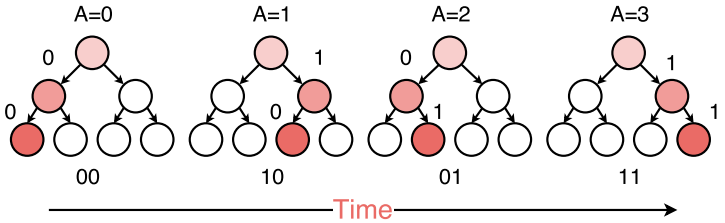

Figure 2: Reverse-lexicographic order in EvictPath. After path A = 3 is evicted to, the order repeats.

4. Security

In this section we analyze the security of each step in the DivORAM protocol to ensure that the entire scheme maintains an obliviousness.

Claim 1. *Read operation in DivORAM protocol is oblivious.*

Proof. In read process, all information that exposed to the server is where and which $\log^3 N$ splinters be accessed. The server retrieves one splinter from $\log N$ splinters without leaking which splinter is the resulted one. Given a data access sequence $\vec{b} = (b_1, b_2, ..., b_m)$, where b_i is the block. Let $s(b_i)$ denotes the splinters belong to block b_i. The probability that an adversary can distinguish which splinter is fetched is

$$Pr[sp(b_i)] = \frac{1}{\log N}, i \in \{1, 2, ..., m\}$$

Notice that only at most $\log N$ splinters forming a whole block. The probability that an adversary can distinguish which block is fetched in a path is

$$Pr[b = b \ in \ a \ path] = \prod_{i=1}^{\log N} Pr[sp(b_i)] = \frac{1}{\log^{\log N} N}$$

In addition, $\log N$ paths will be read in an access behaviours for the consideration of security. The probability that an adversary can distinguish which block is fetched in a read operation is

$$Pr[b = b \ in \ a \ tree] = \frac{1}{\log N} \cdot Pr[b = b \ in \ a \ path]$$

$$= \frac{1}{\log^{\log N+1} N}$$

Therefore, we can conclude that Read operation in DivORAM protocol is oblivious.

Claim 2. *Write and Evict operation in DivORAM protocol leak no information.*

Proof. In write process, the mechanism of eviction makes sure that there always have enough space to store $\log N$ splinters. Apparently the position of those splinters is certain and exposed to the adversary. To break the deterministic position of splinters (or a block), the eviction will shuffle these splinters thoroughly with the help of the trusted proxy. As a path will be sent to the trusted proxy, all splinters in the path will be shuffled and which from the same block will be spread as much as possible while pushing deeper. For the trusted proxy and data storage server are isolated, we can promise that the server will never get any information about where the original splinters are now stored or which path they belong to.

Claim 3. *Given two access sequences in DivORAM protocol, any polynomial-time adversary can not distinguish these sequences.*

Proof. Straightforward conclusion from Claim 1 and Claim 2. It's worth noting that even the eviction is periodically, that is the same block is accessed before being shuffled, the obliviousness of access pattern will be maintain by the read operation.

5. Analysis

In this section, we analyze the bandwidth of read, write and evict operation. At the same time, we calculated the size of the cache for the proxy and client. Due to the best known attack in [36], we set $\lambda = \omega(\log N), \gamma = \Theta(\log^3 N)$ as the security parameters.

5.1. Bandwidth

Read bandwidth. Each read operation requires some select vectors to retrieve the block and happened in $O(\log N)$ paths. To fetch one splinter we need $\log N$ vectors and $\log N$ splinters formed one block. Therefore the communication of select vectors is $O(\log^3 N)$. The Damgård-Jurik cryptosystem [38] will make ciphertext expand to 1.5 times the original. As $\log N$ splinters forming a block, the whole communication complexity is $O(\log^3 N \cdot \nu + 1.5 \cdot B)$, where ν is the vector size. If the plaintext splinter size μ is γs_0, the ciphertext splinter size is $\gamma(s_0 + 1)$ bits. Therefore, the ideal overall bandwidth of DivORAM is $O(\log^2 N \cdot \gamma(s_0+1) \cdot B + 1.5 \cdot B)$. In the usual sense, the way to achieve constant bandwidth is bandwidth limited by B. To achieve constant bandwidth, the block size should be $B \in \Omega(\log^5 N)$. Therefore the constant communication in DivORAM read opearation is $O(1)$.

Write bandwidth. According to the simple operation, we only analyze the worst case that $\log N$ splinters being written. Therefore, we achieve the constant communication in $O(1)$.

Evict bandwidth. With the help of the trusted agent, there is no interaction between the server and the client during the eviction phase. Hence we only analyze the communication between the server and TP. Since each *EvictPath* will happen in every $A = \log N$ access, an average of 3 buckets need to be shuffle in each visit. Due to the different size of bucket, the average communication in an access behaviour is $O(2^{\log N} / \log N)$.

5.2. Proxy storage

For the proper operation of the eviction and shuffle processes, we must consider the worst-case scenario to determine the size of the proxy. As mentioned before that as the depth of the tree increases, the size of the bucket

decreases, the number of splinters to be shuffled reaches the highest value when shuffling the root and its two children buckets. According to that, we have the trusted proxy size $TP = 2^{l+1} \cdot \gamma(s_0 + 1)$.

5.3. Cache Size

In write operation, the block can not be divided and written back immediately due to the leakage of block size. For the sake of server storage and less generation of dummy splinters, we set the cache size is $O(\log N)$. In practice, $2 \log N$ can meet the needs of the correct operation. As mentioned before, each block can be divided to at most $\log N$ splinters. Once a block has been divided and stored in the cache, it will be written back randomly. Consider the worst case, that is, blocks have never been written back to the server so backlog in the cache. The only way to reduce the number of blocks in the cache is to write $\log N$ splinters back to the server deterministically when the number of splinters in the cache reaches $\log N$. When the cache exists in the original $\log N - 1$ splinters without triggering write operation. In the ensuing visit, $\log N$ splinters were deposited. Therefore, the maximum capacity of cache is $2 \log N - 1$. For simplicity, we choose the size of the cache as $2 \log N$.

6. Experiment

In order to more accurately evaluate the performance of the DivORAM solution, we implemented the entire simulator of our construction. We rented two cloud servers, one as the server that stores the primary data and the other as the trusted proxy. At the same time, we set the local PC as a client simulator to simulate the launch of the request. To compare with Ring ORAM, we simulate the main steps of Ring ORAM, such as the process of reading the real block and evict a path, which is compared with our scheme. Roughly speaking, we measured the response time, communication bandwidth, and the amount of computation for each device at each step of each scenario to make statistics and comparisons. In practice, we choose Damgård-Jurik cryptosystem [38] as the AHE-based protocol, which used in Onion ORAM [16]. It will help us to minimize the ciphertext expansion as much as possible.

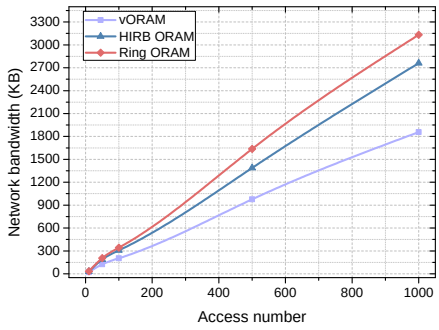

Figure 3: Comparison of network bandwidth among DiDivORAM, Ring ORAM and HIRB ORAM.

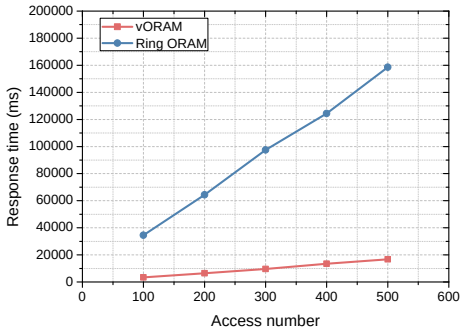

Figure 4: Comparison of response time between DivORAM and Ring ORAM.

ORAM implementations. We deployed our DivORAM in three cloud providers and rented 1 servers per cloud from Alibaba and Baidu cloud. We implement all our experiments on computer of Intel(R) Core(TM) i7-7500U CPU @ 2.70GHZ with Ubuntu OS, 16GB memory and 8 threads for each CPU. We implemented DivORAM in C/C++ with about 1000 lines of codes. We utilized MongoDB as the database to store and access data. In our procedure, we initialize a collection for a binary tree, where each splinter is the ciphertext of 0 encrypted by Damgård-Jurik cryptosystem. We simulate any read/write access with the help of *read* and *updated* operations in MongoDB. For convenience, we specify that the length of the splinter is the key size of Damgård-Jurik cryptosystem. Some experiments are run in the local network environments. The source codes are available at: *https://github.com/liuzheli/SSORAM*.

Network bandwidth. As shown in Figure 3, we can observe that the network bandwidth of DivORAM is reduced by about 30% compared to Ring ORAM and 40% to HIRB ORAM. The reduction is from two aspects: 1) DivORAM achieves constant communication in the whole process of access behaviour; 2) DivORAM employs the trusted proxy to implement eviction to reduce the bandwidth between the server and the client.

Response time. Figure 4 shows the comparison of response time between

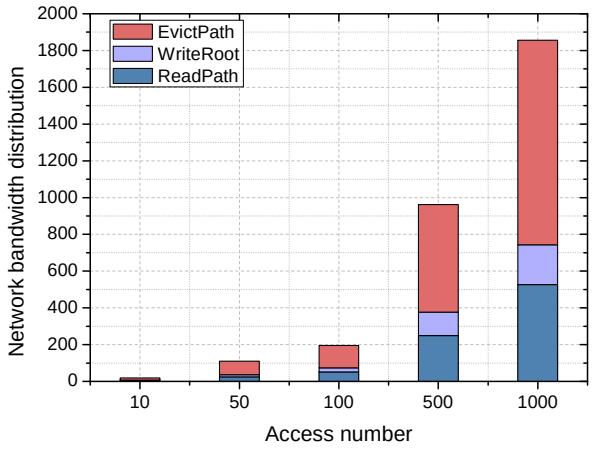

Figure 5: The detailed network bandwidth distribution of data access.

DivORAM and Ring ORAM. We note that DivORAM has better performance than Ring ORAM. In the case of different number of visits, DivORAM is 10× shorter than Ring ORAM. The improvement comes from the followings: 1) the constant communication reduce the response time in transferring data; 2) the eviction process can be carried out in the background without taking the normal access behavior of the visit time..

Load distribution. The detailed network bandwidth distribution of data access in DivORAM is shown in Figure 5. The bandwidth is divided into three parts: *ReadPath*, *WriteRoot* and *EvictPath*. For DivORAM, the bandwidth for the server to trusted proxy accounts for the largest proportion, which accounts for 60% of the total bandwidth. The bandwidth for *ReadPath* is approximately equal to *WriteRoot* because bandwidth can almost be considered only in the transmission of a block in the communication.

7. Conclusion

In this paper, we propose DivORAM protocol to solve the problem caused by variable block-size, which is a bandwidth efficient scheme and has constant communication. DivORAM rebuilds the traditional structure of the tree to accommodate the need to store data of different size. Besides, DivORAM resort to the trusted proxy to reduce the computation in the client

and communication between the client and the server. With the result of experiment, we show that DivORAM improves bandwidth by 4× and response time by 10×. In general, DivORAM is practical to meet the needs of practical applications.

Acknowledgment

This work was supported by the National Natural Science Foundation of China (No. 61672300), and National Natural Science Foundation of Tianjin (No. 16JCYBJC15500).

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

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

[3] Y. Zhang, J. Katz, C. Papamanthou, "All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption", in *USENIX Security*, 2016.

[4] G. Kellaris, G. Kollios, K. Nissim, A. O'Neill, "Generic attacks on secure outsourced databases", in *CCS*, pp. 1329-1340, 2016.

[5] O. Goldreich, R. Ostrovsky, Software protection and simulation on oblivious RAMs, *JACM*, 1997.

[6] E. Stefanov and E. Shi, Multi-cloud oblivious storage, *CCS*, 2013.

[7] E. Stefanov and E. Shi, Oblivistore: High performance oblivious distributed cloud data store, *NDSS*, 2013.

[8] C. W. Fletcher, M. V. Dijk, S. Devadas: A secure processor architecture for encrypted computation on untrusted programs, *STC*, 2012.

[9] Y. Jia, T. Moataz, S. Tople and P. Saxena: OblivP2P: An Oblivious Peer-to-Peer Content Sharing System, *USENIX Security*, 2016.

[10] X. S. Wang, K. Nayak, C. Liu, T. H. Chan, E. Shi, E. Stefanov, Y. Huang: Oblivious data structures, *CCS*, 2014.

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

[12] E. Stefanov, E. Shi, D. Song, Towards practical oblivious RAM, *NDSS*, 2012.

[13] J. Dautrich, E. Stefanov and E. Shi: Burst ORAM: Minimizing ORAM response times for bursty access patterns, *USENIX Security*, 2014.

[14] E. Shi, T.-H. Chan, E. Stefanov, and M. Li: Oblivious RAM with O(log3(N)) Worst-Case Cost, *ASIACRYPT*, 2011.

[15] X. Wang, H. Chan, E. Shi: Circuit oram: On tightness of the goldreich-ostrovsky lower bound, *CCS*, 2015.

[16] S. Devadas, M. van Dijk, C. W. Fletcher, L. Ren, E. Shi and D. Wichs: Onion ORAM: A Constant Bandwidth Blowup Oblivious RAM, *TCC*, 2016.

[17] P. Williams and R. Sion: Single round access privacy on outsourced storage, *CCS*, 2012.

[18] L. Ren, C. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, and S. Devadas: Single round access privacy on outsourced storage, *USENIX Security*, 2015.

[19] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas: Path ORAM: an extremely simple oblivious RAM protocol, *CCS*, 2013.

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

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

[22] Z. Williams, D. Xie and F. Li: Oblivious RAM: a dissection and experimental evaluation, *CCS*, 2012.

[23] Chung, Kai-Min and Liu, Zhenming and Pass, Rafael: Statistically-secure ORAM with $\tilde{O}(\log^2 n)$ Overhead, *Advances in Cryptology–ASIACRYPT 2014*, 2014.

[24] Boneh, Dan and Mazieres, David and Popa, Raluca Ada: Remote oblivious storage: Making oblivious RAM practical, 2014.

[25] Gentry, Craig and Goldman, Kenny A and Halevi, Shai and Julta, Charanjit and Raykova, Mariana and Wichs, Daniel: Optimizing ORAM and using it efficiently for secure computation, *Privacy Enhancing Technologies*, 2013.

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

[27] Goodrich, Michael T and Mitzenmacher, Michael: Privacy-preserving access of outsourced data via oblivious RAM simulation, *Automata, Languages and Programming*, 2011.

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

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

[30] Lu, Steve and Ostrovsky, Rafail: Distributed oblivious RAM for secure two-party computation, *Theory of Cryptography*, 2013.

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

[32] Williams, Peter and Sion, Radu: Single round access privacy on outsourced storage, *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012.

[33] Abraham, I., Fletcher, C.W., Nayak, K., Pinkas, B., Ren, L.: Asymptotically tight bounds for composing oram with pir. In *PKC*, 2017.

[34] Hoang, T., Ozkaptan, C.D., Yavuz, A.A., Guajardo, J., Nguyen, T.: S3oram: A computation-efficient and constant client bandwidth blowup oram with shamir secret sharing. in *CCS*, 2017.

[35] Moataz, T., Mayberry, T., Blass, E.O.: Constant communication oram with small blocksize. In *CCS*, 2015.

[36] H. Lipmaa. An Oblivious Transfer protocol with log-squared communication. In *ISC*, 2005

[37] C. W. Fletcher, M. Naveed, L. Ren, E. Shi, E. Stefanov. Bucket ORAM: Single Online Roundtrip, Constant Bandwidth Oblivious RAM. *IACR Cryptology ePrint Archive*, 2015.

[38] I. Damgard, M. Jurik. A generalisation, a simplification and some applications of Paillier's probabilistic public-key system. In *PKC*, 2001.

[39] J. Doerner, A. Shelat. Scaling ORAM for Secure Computation. in *CCS*, 2017.

[40] Jin Li, Yinghui Zhang, Xiaofeng Chen, Yang Xiang. Secure attribute-based data sharing for resource-limited users in cloud computing. in *Computers & Security*, 2018.

[41] Ping Li, Jin Li, Zhengan Huang, Chong-Zhi Gao, Wen-Bin Chen, Kai Chen. Privacy-preserving outsourced classification in cloud computing. in *Cluster Computing*, 2017.

[42] Jin Li, Xiaofeng Chen, Mingqiang Li, Jingwei Li, Patrick Lee, Wenjing Lou. Secure Deduplication with Efficient and Reliable Convergent Key Management. in *IEEE Transactions on Parallel and Distributed Systems*, 2014.

[43] Jin Li, Xinyi Huang, Jingwei Li, Xiaofeng Chen, Yang Xiang. Securely Outsourcing Attribute-based Encryption with Checkability. in *IEEE Transactions on Parallel and Distributed Systems*, 2014.

[44] Bo Li, Hongyang Yan, Zheli Liu, Tong Li, Jin Li, SiuMing Yiu, "HybridORAM: Practical Oblivious Cloud Storage with constant bandwidth and small storage," in *Information Sciences*, 2018.

[45] Chong-zhi Gao, Qiong Cheng, Xuan Li, Shi-bing Xia, "Cloud-assisted privacy-preserving profile-matching scheme under multiple keys in mobile social network," in *Cluster Computing*, 2018.

[46] Jin Li, Yinghui Zhang, Xiaofeng Chen, Yang Xiang, "Secure attribute-based data sharing for resource-limited users in cloud computing," in *Computers & Security*, 2018.

[47] Ping Li, Jin Li, Zhengan Huang, Tong Li, Chong-Zhi Gao, Siu-Ming Yiu, Kai Chen, "Multi-key privacy-preserving deep learning in cloud computing," in *Future Generation Computer Systems*, 2017.

[48] Ping Li, Jin Li, Zhengan Huang, Chong-Zhi Gao, Wen-Bin Chen, Kai Chen, "Privacy-preserving outsourced classification in cloud computing," in *Cluster Computing*, 2017.

[49] Zhengan Huang, Shengli Liu, Xianping Mao, Kefei Chen, and Jin Li, "Insight of the Protection for Data Security under Selective Opening Attacks," in *Information Sciences*, 2017.

[50] Jin Li, Xiaofeng Chen, Mingqiang Li, Jingwei Li, Patrick Lee, Wenjing Lou, "Secure Deduplication with Efficient and Reliable Convergent Key Management," in *IEEE Transactions on Parallel and Distributed Systems*, 2014.

[51] Jin Li, Xinyi Huang, Jingwei Li, Xiaofeng Chen, Yang Xiang, "Securely Outsourcing Attribute-based Encryption with Checkability," in *IEEE Transactions on Parallel and Distributed Systems*, 2014.