An ORAM Scheme with Improved Worst-Case Computational Overhead

Nairen Cao[1], Xiaoqi Yu[1], Yufang Yang[2], Linru Zhang[3], and SiuMing Yiu[1(✉)]

[1] The University of Hong Kong, Hong Kong, China
caonr@pku.edu.cn, {xqyu,smyiu}@cs.hku.hk
[2] Tsinghua University, Beijing, China
yfyang12@mails.tsinghua.edu.cn
[3] Sun Yat-sen University, Guangzhou, China
zhanglr3@mail2.sysu.edu.cn

Abstract. We construct a statistically secure ORAM with computational overhead of $O(\log^2 N \log\log N)$. Moreover, when accessing continuous blocks, our scheme can achieve an amortized complexity $O(\log N \log\log N)$, which almost matches the theoretical lower bound of the ORAM problem. Our construction is based on a tree-based construction [16]. The technical novelty comes from the idea of combining $O(\log N)$ blocks into a big block together with a more aggressive and efficient "flush" operation, which is the bottleneck of existing ORAM schemes. All in all, we can achieve better amortized overhead in our new scheme.

1 Introduction

ORAM (oblivious random access memory) was first studied in [4,11,14] by Goldreich and Ostrovsky *et al.* aiming at protecting software privacy against reverse-engineering by hiding the memory access pattern. Recently, researchers (e.g. [12]) also gave examples on the possibility of revealing trading transactions if data access pattern is not protected. Due to advancement of cloud technologies, more and more users host their data in a third-party cloud system (out-sourcing). The privacy problem of access pattern becomes one of the key concerns in these applications.

It is obvious that hiding the access pattern would increase the computational overhead for accessing the data. Many schemes evolve in recent years aiming at achieving better results in efficiency (e.g. [5–8,12,18]). Improvements involving techniques such as using Cuckoo hash functions [1] and randomised Shell sort have also been proposed. However, the complexity of these techniques hingers the practicality of the schemes in real applications. Luckily a breakthrough, which significantly simplifies the ORAM structure, was proposed in [16] by using a binary tree layout. Our scheme also follows its structure of storage. In fact, this tree-based structure triggers several important results in this area (e.g. [3,10,15,17]).

Roughly speaking, in a tree-based ORAM structure, we rely on a linear structure called position map which maps the indexes of data items to the indexes of

S. Qing et al. (Eds.): ICICS 2015, LNCS 9543, pp. 393–405, 2016.
DOI: 10.1007/978-3-319-29814-6_33

tree leaves [16] so that the corresponding path (from root to a particular leaf) can be retrieved to obtain the data item. The security of the approach lies on two facts. First, the server has no idea which node contains the data item as all nodes will be retrieved and some contain dummy values which are not distinguishable from the server point of view. Also, which path corresponds to which data item is also unknown to the server. Second, after each access of a path, the retrieved data items will be randomly assigned to another path of the tree to increase the difficulty of the server to identify the data item. This operation is usually referred as the "flush" operation. This flush operation is usually the bottleneck of the schemes. Moreover, since the position map is important, some schemes assume that this map is stored in the client side. However, this extra space for storing the map would mean a storage requirement of $N \log N$ bits which would be a big burden to the client. The situation will be worse if N, the number of data block (the basic item for operations) is huge (e.g. in a big data application) and the size of the index $\log(N)$ may not be considered as a constant. On the other hand, the position map cannot be stored in plaintext in the server side, otherwise the access pattern privacy may be comprised since server can learn some information of the logical and physical positions such as which path a particular data item is stored. Existing solutions try to make use of the technique of recursive ORAM to store the position map also in the server side, i.e., the position map is also stored in smaller ORAM structures layer by layer in the server side. This, however, affects the block size, for example, a block size of $\omega\log(N)$ is required in [16]. Consequently, the large block size will lead to large overhead of $O(\log^3 N)$, which is one element to bring up the complexity of this problem. To explore a solution that can achieve better efficiency without comprising security requirements is of great significance to apply ORAM to practical situations and better utilise the tree structures in this problem.To our knowledge, it seems that the best results of existing solutions sore both the data items and the position map in the server side to provide a scheme with worst-case overhead complexity of $\varnothing(log^3 N)$. In our work, we propose an improved structure and a more aggressive and efficient flush algorithm, which is the key step in $PathORAM$ related structures [16], that can achieve better performance.

ORAM Recursive and Problems. The recursive ORAM idea in PathOram [16] is to outsource the position map to a smaller ORAM. We have $ORAM_0, ORAM_1, ...$ to represent the ORAM structures. The size of latter ORAM is smaller than the previous one which is the trick for the success of the recursive ORAM idea. Intuitively, the "smaller" idea comes from larger block size. Particularly, they set block size as $O(\log N)$ with constant ≥ 2. Obviously, this setting will result in heavy overhead and bandwidth of the scheme.

While in work [2], Chung *et al.* claim that the number of recursion rounds is $O(\log N)$ in order to reduce the position map to constant, involving the process of a Markov chain-like procedure [9]. However, due to their constant block settings, their scheme may cause recursion failure because they can not reduced the size of position map in the following layers [2].

In our scheme, we resolve this by combining several blocks into one in the position map and hence reducing its size instead of relying on large block size, and show that $O(\log_\alpha N)$ rounds of recursion is sufficient, where α is the compress factor in each recursion layer.

Idea of Our Scheme. While we follow the basic concepts in [16], our construction has some special properties, resulting in higher efficiency of our construction. Roughly speaking, we conceptually combine Q (to be decided later) *consecutive* blocks into a big block, called *trunk* in this paper, and always assign consecutive Q blocks in a trunk to consecutive S leaf nodes. Therefore we only store N/Q position in range $(1, N)$ each. This idea comes with a more aggressive (will be explained in details in the following sections) and efficient flush operation.[1] Thus, we improve the overhead to $O(\log^2 N \log\log N)$ instead of $log^3 N$. Also, our scheme achieves better performance than [16] when consecutive blocks are accessed which will be introduced in details in Sect. 3.6.

1.1 Our Contribution

Our scheme makes the following contributions.

Lower Computation Overheads. In Path Oram [16], they provide the scheme with $O(\log^2 N)$ overhead with $O(\log N)$ block size, and $O(\log N)$ overhead of $O(\log^2 N)$ block size. Therefore, it is indeed summed up to $O(\log^3 N)$ overhead. However, Our scheme can decrease the time overhead from $O(\log^3 N)$ to $O(\log^2 N \log\log N)$ mainly from by replacing the linear storage with searchable tree structure for the stash and to avoid $O(\log N)$ block size.

Better Performance in Continuous Access. Our scheme achieves better performance when continuous data blocks are read. In the construction of [16], if we read $\log N$ blocks *once*, the time overhead will be $O(\log^3 N) \cdot \log N$. In contrast, in our construction, we can reduce the overhead to as small as $O(\log^2 N \log\log N)$ when they are in the same trunk and we only have to read one path interval. In this case, we can reach amortizing $O(\log N \log\log N)$ overhead in average, which is nearly the asymptotic lower bound according to [4].

2 Preliminaries

Notations. We use N to denote the total number of blocks that can be stored in our ORAM. The block size is denoted by $C = O(\log N)$, and the capacity of

[1] Moreover, in our experiments, we found a problem in our experiments about an assertion (Lemma 3) made by [16]. When Lemma 3 claims that a N times, no duplicated block access can maximize the probability of stash size exceeding R (R can be any number). In other words, Lemma 3 claims that maximum number of stash will appear with one round after accessing all indexes. However, we find that the stash size is probably not maximal when each index is visited once, but will continue increasing until $O(\log N)$ number of rounds to converge.

buckets is denoted by Z, which is a constant. The height of the binary tree on the server's side, is denoted by h which is of size $O(\log N)$ (and the total number of leaves is $L = O(N)$). `T`, `root`, `stash` and `PM` denote the binary tree, the root of the tree, the stash and the position map respectively. The number of blocks and number of big blocks are denoted by $N(b)$ and $N(B)$ respectively. For any node `n`, the Z-sized bucket associated to it is denoted by `n.bucket`.

Trunk: For Q consecutive blocks whose position is stored in the same slot in position map, we call the big block formed by this consecutive blocks a Trunk.

Paths: For any leaf node l_i where $i \in \{0, 1, \ldots, 2^h - 1\}$, define the i-th path $P(i)$ to be the path from node l_i to the root. It is trivial that in a binary tree, this path is well-defined and unique, and contains $h + 1 = O(\log N)$ nodes. We call two paths to be *consecutive*, if and only if their corresponding leaf nodes are consecutive (i.e. their number differ by 1 modulo 2^h).

Path Intervals: Given the interval length $S = O(\log N)$ and interval start position $x \in \{0, 1, \ldots, 2^h - 1\}$, a path interval $PI(x, s)$ is defined to be the union of the S consecutive paths $P(x), P((x + 1) \mod 2^h), \ldots, P((x + s - 1) \mod 2^h)$. We further define the path interval on the i-th level, $PI(x, s, i)$, to be the intersection set of nodes of the i-th level and $PI(x, s)$.

We consider a client storing data at a remote untrusted server while preserving its privacy. While traditional encryption methods protect the content of the data, they fail to hide the data access pattern, which may reveal information to untrusted servers. We assume that the server is curious but honest (*i.e.*, do not modify the data), and small amount of memory is available at the client. Our ORAM scheme completely hides the data access pattern (the sequence of blocks read/write) from the server. From the server's perspective, the data access pattern from two sequence of read/write operations with the same length should be indistinguishable. Now we provide the followings:

Definition 1 (Access Pattern). *Let* $A(\mathtt{y})$ *denote the sequence of access to the remote server storage given the data request sequence* `y` *from client. Specifically,* $\mathtt{y} = (\mathtt{op}_{L'}, \mathtt{a}_{L'}, \mathtt{data}_{L'}), ..., (\mathtt{op}_1, \mathtt{a}_1, \mathtt{data}_1)(L' = |\mathtt{y}|)$*, 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) For any two data request* `y` *and* `z` *of the same length, their access pattern* $A(\mathtt{y})$ *and* $A(\mathtt{z})$ *are computationally indistinguishable by anyone but the client, (2) the ORAM construction is correct in the sense that it returns correct results for any input* `y` *with probability* $\leq 1 - negl(|\mathtt{y}|)$.

3 Our Scheme

3.1 Overview

Our Scheme follows the tree-based ORAM framework in [16], by building a binary tree with N leaves nodes. Each node is a bucket with capacity $Z = O(1)$

blocks. Furthermore, the client stores limited amount of local data in a stash. The blocks to be outsourced are denoted by $b_1, b_2, ..., b_N$. We combine every contiguous $Q = O(\log N)$ blocks into a trunk, denoted by $B_1, B_2, ..., B_M$, where $M = \frac{N}{\log N}$. *i.e.*, $b_{kQ+1}, b_{kQ+2}, ..., b_{(k+1)Q}$ are contained in B_k.

We maintain the invariant that at any time, each trunk is mapped to a uniformly random interval of contiguous leaf buckets in the range of $(1, N)$, and every block b_{iQ+j} in the binary tree is always placed in some node of the Path Interval $PI(pos(i), S)$.

Whenever a block is read from the server, the nodes in the Path Interval is read into the stash and the trunk which contains the requested block is remapped to a new path interval. When the nodes are written back to the server, additional blocks in the stash may be evicted into the path as close to leaf level as possible there exists available spaces in the buckets.

3.2 Server Storage

Our server storage consist of three parts:

Tree: The server stores a binary tree data structure of height h and 2^h leaves, and we need $h = \lceil \log_2(N) \rceil$. the levels of the tree are numbered 0 to h where level 0 denotes the root of the tree and level h denotes the leaves. Each node in the tree contains one bucket that contains up to Z blocks. If a bucket has less than Z real blocks, it is padded with dummy blocks to always be of size Z.

Server Storage Size: There are $O(N)$ buckets in the tree and the total server storage used is about $Z \cdot O(N) = O(N)$ blocks.

3.3 Client Storage

Stash: When we write the blocks back to the tree after a read/write operation, a small number of blocks might overflow from the tree buckets and they would stay in the stash. We will show in the following sections that $O(\log^2 N)$ size of a shared stash for all recursive ORAM is enough to bound the overflow blocks with high probability. Specifically, we store the tuple of $(index, pos, data)$ for each block in the stash arranged. If we store stash as a binary searchable tree with *pos* as the key, which still has $O(\log^2 N)$ storage size. Therefore each access operation in the stash will cost $O(\log \log^2 N) = O(\log \log N)$ overhead.

Position Map: In our scheme, we have to keep a record of which *Path Interval* each *Trunk* is mapped to. We use p_i to denote the index of the start leaves and assume that a trunk B_i will be mapped to the leaves in $[p_i, p_i + S]$. $O(\log N)$ bits are needed to mark the leaves, so the size of position map is $\frac{N}{Q} \cdot O(\log N) = O(N)$ bits, which is $\frac{N}{O(\log N)} \cdot O(\log N) = O(N)$ bits when $Q = O(\log N)$. We can use recursion method to save position map in the server just like previous schemes to achieve constant position map storage on client side when we can achieve compress factor α in each recursion layer as much as $Q = O(\log N)$

Input: Tree `T`, position map `PM`, client stash `stash`

1 Construct binary tree `T` with height h and fill all buckets with Z dummies each;
2 $\emptyset \leftarrow$ `stash`;
3 **for** $i = 1$ **to** $N(B) - 1$ **do**
4 $\quad PM[i] \leftarrow UniformRandom(0, 1, \ldots, L-1)$;
5 **end**

Algorithm 1. `Init(T, PM, stash)`

Client Storage Size: The size of stash is $O(\log^2 N)$, which will be discussed later, and position map will be reduced to constant by the idea of recursive ORAM on server side. Therefore, the client storage size if $O(\log^2 N)$.

3.4 Detailed Scheme Description

Initialization. We require that the client stash S is initially empty, and each server buckets are initialized to contain Z random encryptions of the dummy block while the client's position map is filled with independent random numbers between 0 and $L-1$. The algorithm is described in details in pseudocode in Algorithm 1.

Access Operation (a). In our construction, reading and writing a block to ORAM is done via a single protocol called `Access`. Specifically, to read block `a`, the client performs `data` $\leftarrow$ `Access(read, a, None)` and to write `data`* to a block `a`, the client performs `Access(write, a, data`*`)`. The `Access` protocol can be summarized in following simple steps:

1. `Remap block` : We determine that which trunk B_i contains `a` by $i = a/Q$, and get the $PositionInterval$ $p_i = PM(i)$ from position map. Then randomly remap the position of B_i to a new random position in range $(0, L-1)$, and update the position map.
2. `Read path interval` : Read the path interval $PI(p_i, S)$ containing block `a`, and store all the blocks in this path interval in stash on the client side.
3. `Search Stash(`$\mathtt{p_i}, .., \mathtt{p_i} + \mathtt{S}$`)`: As described in Sect. 3.3, stash is the searchable binary tree with size of $O(\log^2 N)$ sorted by key pos. In our access operation, we have retrieved the positions of $Path\ Interval$ of the target block in step 1. Supposed it contains the paths of $p_i, ..., p_i + S$ in the $Path\ Interval$, then we can search each path in the stash, which cost $O(\log \log^2 N) = O(\log \log N)$ for each access. Since S is set as constant in our scheme, the overhead for this step is $O(\log \log N)$.
4. `Update block` : If the operation is $write$, update the data stored for block a.

We provide the detailed algorithm in pseudocode in Algorithm 2.

Input: The index `index` of the entry required by the client; new value `data`* if it is a *write* operation
Output: The content at `index`

```
1  found ← false;
2  index_trunk ← index/Q //the index of the trunk index belongs to;
3  curPos ← PM[index_trunk];
4  PM[index_trunk] ← UniformRandom(0, 1, ..., L − 1);
5  I ← PI(curPos, S);//the path interval we are going to search
6  for each node n in I do
7      if index is in the bucket of n then
8          b ← the required block; data ← contents of b;
9          found ← true;
10         if the operation is write then
11             b.content = data*;
12         end
13         delete b from bucket of n and add b to stash;
14     end
15     stash ← stash + bucket(n); set bucket(n) to empty;
16 end
17 if found != true then
18     search for index in stash;
19     if found then
20         data← the value found; found←true;
21     end
22     else
23         data← ⊥ and add it to stash;
24     end
25 end
26 return data;
```

Algorithm 2. `Access(index)`

Flush Operation. After an `Access` operation, we have to flush the elements in the stash to the tree in order to avoid overflow. Hence any `Access` operation is invariably succeeded by a `flush` operation. In a `flush` operation, we flush the elements in the stash to the tree within a random path interval `PI`. When the `Access` operation and the `flush` operation is considered as a whole, the path interval we use in the `flush` operation is precisely the same interval we search in the tree for the index we have just had accessed to. Before describing the details of the algorithm, we first introduce a property of a node n in the tree called *compatible*. Specifically, $n.compatible(i)$ is true if index i belongs to the *Path Interval* PI, and there exists one path in PI whose ancestor is n. Pseudo-code of flush operation is presented in Algorithm 3.

Intuitively, supposed we will evict the *Path Interval* $PI(p_i, S)$. Then we will traverse nodes of this *Path Interval* from leaf level to root. For each node, we will write back the blocks in stash that are compatible with the node. Specifically, to write back a node n in $PI(p_i, s)$. It is easy to decide the path range (p_i, p_j)

that node n covers. Then choose a position $p_r \in (p_i, p_j)$ randomly, and search the stash for key p_r to retrieve a block subset S'. Obviously, the subset S' constructed in this way contain the blocks whose indexes are compatible with node n. Then write back Z blocks from S' to the node n if size of S' is larger than Z.

For example, the green nodes in Fig. 1 is the path interval $PI(4, 2)$. To flush this *Path Interval*, we start with node 13, we search the stash with key 13, and write back the blocks to the node. It is similar for node 14. Next, we move to node 6. Supposed that $S = 2$, then the position range for node 6 is 12–15. Then all blocks whose index mapped to $PI(3, 2)$,$PI(4, 2)$,$PI(5, 2)$ are compatible with the node 6. In this case, we will randomly pick one position in range of 12–15. For example, we will call search (13) in the binary searchable tree in stash if we pick $p_r = 13$. Then write back the retrieved blocks in S' to node 6. Likewise, we can get the position range (denoted as leaf id) for node 3 and node 1 is 12–16 and 8–16 respectively. Then pick random position and write back the corresponding nodes.

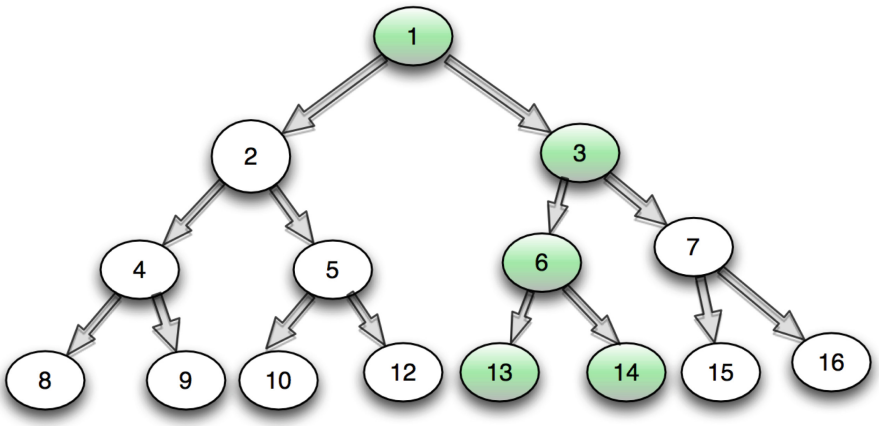

Fig. 1. Path interval

Combining the `Access` operation and `Flush` operation, we can now readily give our whole set of *read/write* operations. The pseudo-code for the complete protocol is written in Algorithm 4.

3.5 Security Analysis

To prove the security of our scheme, let $\boldsymbol{y}$ be a data request sequence of size M. The server sees $\boldsymbol{A}(\boldsymbol{y})$ which is a sequence

$$\boldsymbol{p} = (position_M[a_M], position_{M-1}[a_{M-1}], ..., position_1[a_1])$$

Input: PI, which is the path interval to be flushed

for i=h **to** *0* **do**
  $IL \leftarrow PI(i)$;//the set of i-levelled nodes in the path interval
  for *each node* n *in* IL **do**
    $S' \leftarrow$ all blocks in stash that are compatible with n;
    TEMP $\leftarrow$ any $\max\{Z, |S'|\}$ blocks from S';
    add all blocks in TEMP into n.bucket, and fill with dummies if not full;
    stash $\leftarrow$ stash $-$ TEMP;
  end
end

Algorithm 3. Flush(PI)

Input: index, the index required by the client; v′ which is the new value written if it is a *write* operation
Output: v, the value associating with the index

PI $\leftarrow$ the path interval corresponding to the big block index belongs to;
v $\leftarrow$ Access(index);
$Flush(PI)$;
return v;

Algorithm 4. RWWithFlush(index)

where $position_j[a_j]$ is the position of address a_j indicated by the position map for the j-th Access operation. Note that once $position_i[a_i]$ is revealed to the server, it is re-mapped to a completely new random label, hence, $position_i[a_i]$ is statistically independent of $position_j[a_j]$, when the i-th address and the j-th address correspond to the same trunk.

As for $a_j \neq a_i$, the i-th and j-th address are mapped to distinct trunks, with high probability. Their positions $position_i[a_i]$ and $position_j[a_j]$ are independent (*i.e.*, never affect each other) according to our flush mechanism. Therefore, the position sequence $\{position_j[a_j]\}_{j=0}^{M-1}$ consists of independent random integers from 0 to $L-1$. So the server, after seeing the position sequence of $\boldsymbol{y}$, guesses correctly with probability at most

$$\Pr[\boldsymbol{p}] \leq \prod_{j=0}^{M-1} \Pr[position_j[a_j]] = (\frac{1}{2^h})^M = 2^{-hM} \tag{1}$$

which is negligible in h, where $\Pr[X]$ denotes the probability of a correct guess on the corresponding access step made by the server after seeing the corresponding position X. This indicates that $\boldsymbol{A}(\boldsymbol{y})$ is computationally indistinguishable from random coins.

Now the security follows from the statement above and the fact that the $O(\log^2 N)$-sized stash will not overflow with high probability, which we will discuss in the following section.

3.6 Parametres and Complexities

Now we analyze our parametre settings and corresponding complexities.

Parametre Settings. We choose block size C as a constant, e.g. $C = \log N$ and $Z = 5$ as in [16]. Furthermore, the size of the trunk is set to $\frac{1}{2}\log N \leq S \leq \log N$(e.g. when $N = 2^{19}$ to 2^{24}, we have $S = 16$). Now we analyze the complexity of our scheme.

Recursion. We can follow the similar recursion in [16] to reduce the client space usage to $O(1)$(*i.e.*, constant). In each recursion, suppose initially we have N indexes in total, which make up $N * \log N$ bits originally. With the help of trunk, we should actually store $\frac{N*\log N}{Q}$ bits, which is $\frac{N*\log N}{Q*C} = O(\frac{N}{\alpha})$ blocks. α is denoted the compress factor in the recursive layer, and obviously $\alpha \geq 1$, which is decided by the size of trunk Q. The number of recursion layer is $\log_{\alpha} N$. Particularly, the number of recursion is $\log N / \log\log N$ when $Q = O(\log N)$. Compared with the previous work [16] that rely on large block to achieve the compress factor $\alpha \geq 1$, our method can achieve lower overhead, and avoid the burden of bandwidth overhead caused by big block size, simultaneously, reduce the storage of position map to constant.

Time Complexity. First we consider time complexity. To analyze the time complexity of *read/write* operation, we have to figure out the number of blocks fetched from the path interval in each read/write operation. First we introduce the following theorem:

Theorem 1. *The total number of nodes in a path interval PI of length S (S is constant) is still $O(\log N)$.*

Proof. For a path interval PI of length $O(\log N)$, we consider $PI(j)$ for each level $j \in \{0, 1, \ldots, h\}$. Without loss of generality, we suppose the length of PI is $C \log N$ where C is a constant. By the definition of path interval, the size(*i.e.*, number of nodes) in $PI(h)$ is $C \log N$. Note that for any $PI(j)$, the nodes in $PI(j)$ are the parents of nodes in $PI(j+1)$. Since nodes in $PI(j+1)$ are consecutive, except for at most the leftmost and rightmost ones, every two adjacent nodes in $PI(j+1)$ share the same parent in $PI(j)$. So, the number of nodes in $PI(j)$ is at most

$$\frac{PI(j+1) - 2}{2} + 2 = \frac{PI(j+1)}{2} + 1 \tag{2}$$

Solving this recursion yields

$$PI(j) \leq 2^{-(h-j)}(PI(h) - 2) + 2 \tag{3}$$

Hence, adding them together, the total number of nodes is

$$\sum_{j=0}^{h} PI(j) \leq \sum_{j=0}^{h} 2^{-(h-j)}(PI(h) - 2) + 2h \leq 2PI(h) + 2h - 4 = O(\log N) \tag{4}$$

which comes from the fact that $PI(h) = O(\log N)$ and $h = O(\log N)$. □

Time Overhead Analysis. An access in each recursive level consist of three sub-steps: 1. Read path in the binary tree totally visit $O(\log N)$ nodes (buckets) in a random path interval according to Theorem 1. Since the number of blocks held by a bucket is constant Z, the total time usage is $O(\log N)$. 2. Scan local stash: Supposed stash is maintained as a binary searchable tree with size $O(\log^2 N)$, the access of stash in total make up $O(\log \log N)$. 3. Flush cost: In a *flush* operation, according to Theorem 1, we visit $O(\log N)$ number of nodes.Adding up the three steps, we get the overhead for each layer to $O(\log N \log \log N)$.

$\log \alpha N$ is the number of recursive layer since we can achieve compress factor $\alpha = O(\log N)$. Take recursion into consideration , we can get overhead of $O(\log^2 N \log \log N)$.

Continuous Access. Our scheme is extremely suitable for continuous access. While accessing one block needs $O(\log^2 N \log \log N)$ overhead, the overhead will be at most $O(\log^2 N \log \log N)$ if we access continuous Q blocks, thus the amortized complexity will be $O(\log N \log \log N)$, nearly the asymptotic lower bound. The $O(\log \log N)$ comes from the flush operation: for each 'flushing' node, we need to find an appropriate block, taking an extra $O(\log \log N)$.

Bandwidth. Similar to the analysis in [16], in each read-write operation, by Theorem 1, the client reads a total number of $O(\log N)$ blocks from the tree and writes them back, inducing a bandwidth of $O(\log N)$.

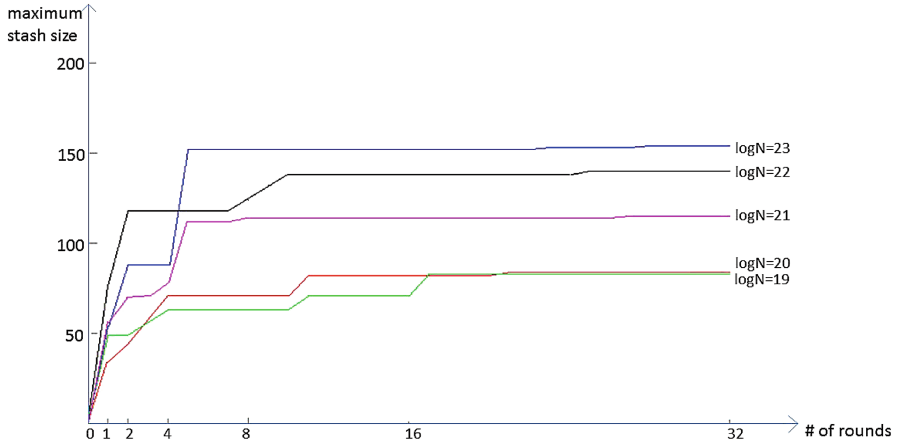

Fig. 2. Experiment results for stash size. As *log N* increase, so does the stash size. When the *log N* increases by one, the stash size increases nearly *log N*. Another observation is the maximum stash size usually does not appear at first round. Since the Path Oram [16] has a similar distribution depend on round, we think maybe one round is not enough for proof.

4 Discussion

In this section we show that our bound for stash size is $O(\log^2 N)$. Intuitively this complexity can be formed by results in [13], but we require some more actual evidences. In our experiment, we tested a number of cases with parameters $Z = 5$ and $\log N$ ranging from 19 to 23. The results show that the stash size is approx. $O(\log^2 N)$ and the maximum stash size will not increase after $O(\log N)$ rounds. We present the actual results in Fig. 2.

Note that the distance between any two adjacent lines is at most $O(\log N)$, indicating that the stash usage is bounded by $O(\log^2 N)$. We can also notice that after visiting all indexes once, the maximum stash usage is approximately half the total maximum stash usage, and only when approximately 16 rounds have been performed, then maximum stash usage reaches a balance.

5 Conclusion

We propose a new scheme of ORAM achieving $O(\log^2 N \log\log N)$ time overheads per access, and most importantly, attaining almost asymptotically lower bound time complexity if *continuous* data are requested. However, it is still an open question whether the lower bound can be achieved even when *random* addresses are visited, or, on the other hand, if the lower bound can be improved for general cases.

Acknowledgement. This work is supported in part by National High Technology Research and Development Program of China (No. 2015AA016008), NSFC/RGC Joint Research Scheme (N_HKU 729/13), and Seed Funding Programme for Basic Research of HKU (201411159142).

References

1. Arbitman, Y., Naor, M., Segev, G.: Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. CoRR, abs/0912.5424 (2009)
2. Chung, K.-M., Liu, Z., Pass, R.: Statistically-secure ORAM with $\tilde{O}(\log^2 n)$ overhead. CoRR, abs/1307.3699 (2013)
3. Damgård, I., Meldgaard, S., Nielsen, J.B.: Perfectly secure oblivious RAM without random oracles. In: Ishai, Y. (ed.) TCC 2011. LNCS, vol. 6597, pp. 144–163. Springer, Heidelberg (2011)
4. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious rams. J. ACM **43**(3), 431–473 (1996)
5. Goodrich, M.T.: Randomized shellsort: a simple data-oblivious sorting algorithm. J. ACM **58**(6), 27: 1–27: 26 (2011)
6. Goodrich, M.T., Mitzenmacher, M.: Privacy-preserving access of outsourced data via oblivious RAM simulation. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part II. LNCS, vol. 6756, pp. 576–587. Springer, Heidelberg (2011)

7. Goodrich, M.T., Mitzenmacher, M., Ohrimenko, O., Tamassia, R.: Oblivious ram simulation with efficient worst-case access overhead. In: Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop, CCSW 2011, New York, pp. 95–100. ACM (2011)
8. Goodrich, M.T., Mitzenmacher, M., Ohrimenko, O., Tamassia, R.: Privacy-preserving group data access via stateless oblivious ram simulation. In: Proceedings of the Twenty-third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, pp. 157–167. SIAM (2012)
9. Liu, Z., Chung, K.M., Lam, H., Mitzenmacher, M.: Chernoff-hoeffding bounds for markov chains: generalized and simplified. In: ACM (1998)
10. Chung, K.-M., Pass, R.: A simple oram (2013)
11. Goldreich, M.T.: Towards a theory of software protection and simulation by oblivious rams. STOC (1987)
12. Pinkas, B., Reinman, T.: Oblivious RAM revisited. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 502–519. Springer, Heidelberg (2010)
13. Raab, M., Steger, A.: "Balls into bins" - a simple and tight analysis. In: Proceedings of the Second International Workshop on Randomization and Approximation Techniques in Computer Science, RANDOM 1998, London, pp. 159–170. Springer-Verlag (1998)
14. Ostrovsky, R.: Efficient computation on oblivious rams. STOC (1990)
15. Shi, Elaine, Chan, T-HHubert, Stefanov, Emil, Li, Mingfei: Oblivious RAM with $o((\log n)^3)$ worst-case cost. In: Lee, Dong Hoon, Wang, Xiaoyun (eds.) ASIACRYPT 2011. LNCS, vol. 7073, pp. 197–214. Springer, Heidelberg (2011)
16. Stefanov, E., van Dijk, M., Shi, E., Fletcher, C., Ren, L., Xiangyao, Y., Devadas, S., Path oram: An extremely simple oblivious ram protocol. In: Proceedings of the ACM SIGSAC Conference on Computer & Communications Security, CCS 2013, New York, pp. 299–310. ACM (2013)
17. Wang, X., Chan, H., Shi, E.: Circuit oram: On tightness of the goldreich-ostrovsky lower bound. Cryptology ePrint Archive, Report /672 (2014). http://eprint.iacr.org/
18. Williams, P., Sion, R., Carbunar, B.: Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In: Proceedings of the 15th ACM Conference on Computer and Communications Security, CCS 2008, New York, pp. 139–148. ACM (2008)