Dynamic Proofs of Retrievability with Improved Worst Case Overhead

Xiaoqi Yu
The University of Hong Kong
xqyu@cs.hku.hk

Nairen Cao
Peking University
caonr@pku.edu.cn

Jun Zhang
The University of Hong Kong
zhangjun_sdu@163.com

Siu-Ming Yiu
The University of Hong Kong
smyiu@cs.hku.hk

***Abstract*—Proofs of Retrievability ($\mathcal{POR}$) is a scheme to build a verifiable storage on a remote server that a client can access the data randomly, and periodically execute an efficient Audit protocol to ensure that the data is intact. In dynamic $\mathcal{POR}$, the difficulties are to maintain the latest version of the data while achieving efficient Update and Audit. In this paper, we propose $\mathcal{PDPOR}$ that achieves the best worse-case overhead $O(\log N)$ for both Write and Audit protocols together with the best worse-case bandwidth. We also prove the security of $\mathcal{DPOR}$ using simpler techniques.**

1. Introduction

More and more users deposit their data on a third-party (untrusted) cloud storage. It is important to have a verifiable storage scheme to make sure that the remote database is intact. The verification process is referred as the Audit protocol, which will be executed by a client with the server. As long as the server can pass the Audit check, the data should be intact with an overwhelming probability. This problem is called Proofs of Retrievability ($\mathcal{POR}$) [1]. A trivial solution is to download the entire database, but this is not efficient. Existing $\mathcal{POR}$ schemes aim at developing an efficient Audit protocol. These schemes follow a similar idea. They apply an erasure code to generate redundant data to guarantee that the whole set of data can be recovered unless the server deletes a "significant" portion of it. Then, by randomly request a small fraction of the data and checks the correctness of this subset of data, the client can be sure that the entire database can be retrieved [2].

Dynamic Proofs of Retrievability ($\mathcal{DPOR}$): However, the aforementioned techniques will induce a heavy cost of updating large portions of the data on the server even when the client only flips one bit. To tackle this weakness, a common idea is to apply local erasure code, thus the client only have to update a small portion of the data. When the database is encoded with blocks of size k, and the client challenge t locations in the Audit check. It is only secure when k and t are $\Omega(\sqrt{l})$, where l is the size of data [2], otherwise, the server can delete one target encoded block and still have a significant probability to pass an Audit, whilst the block cannot be recovered. Further solution involves applying a pseudo-random permutation $\pi : [l_{code}] \to [l_{code}]$ to output $\mathcal{C}' = (c[\pi(1)], ..., c[\pi[l_{code}])$ before they are outsourced to the server. Yet it will lead to the same attack when Write is executed and the locations of a block must be revealed.

Difficulties of Efficient $\mathcal{DPOR}$: A secure $\mathcal{DPOR}$ scheme requires that the server always keeps the latest version of the database, or Audit will end with failure. However, an efficient update can only access a small portion of the data, thus the server can easily ignore this part without being detected. In this case, we need an Audit that should check sufficient locations to hit the "bad" data, which leads to a time-consuming Audit. To solve this dilemma, we should design a smart Update or Audit algorithm, otherwise we must enlarge update parameter k or Audit parameter t to achieve security of $\mathcal{DPOR}$.

Prior works: In $\mathcal{PORAM}$ [2], Cash *et al.* built a $\mathcal{DPOR}$ scheme with the $\mathcal{ORAM}$ primitive [3] [4], which is applied as a new server-side storage layout that allows randomly access of the data in a secret way. Via $\mathcal{ORAM}$, $\mathcal{PORAM}$ prevents untrusted server from inferring all the n encoded words accessed and deleting many entries of the block or ignoring most of the recent updates. However, $ORAM$ induces a heavy computation and bandwidth overhead. $\mathcal{DPOR}$ scheme in E. Shi *et al.* [5], on the other hand, applies fast Fourier transform (FFT) algorithm to solve the heavy bandwidth problem compared with that of $\mathcal{ORAM}$. Precisely, they maintain the FFT encoded blocks with increasing size of $2^0, 2^1, ..., 2^{\log l+1}$. However, it will lead to linear complexity of the Write operation when the largest block is accessed.

Overview of Our Ideas: Our core idea is to develop a shuffling process to distribute the subset of data being updated uniformly in the entire database, the server cannot learn extra information of relations of the data. In this case, the Audit protocol that checks t words can easily hit the latest modified locations in the last update. Meanwhile, the shuffling process should also achieve comparative sublinear efficiency. Roughly speaking, we store the outsourced data as a binary tree, along with smart Update and Audit protocols. Table 1 shows that our scheme is better than existing schemes and can achieve the best worst case bandwidth and overhead for both Write and Audit protocols. Details of the analysis will be shown in Section 4.

In the first step, we also apply an erasure code algorithm to generate the encoded data $c = \Sigma^{l_{code}}$, then store each c_i in a random path of the binary tree on the server side (the

Scheme	Write cost		Audit cost	
	Server cost(Average/Worst Case)	BW	Server cost(Average/Worst Case)	BW
Cash [2]	$O(k\lambda(\log N)^2)$/$O(k\lambda N^2)$	$O(k\lambda(\log N)^2)$	$O(k\lambda(\log N)^2)$ /$O(k\lambda(\log N)N)$	$O(k\lambda(\log N)^2)$
Pratical DPoR in [5]	$O(k\log N)$/ $O(kN)$	$\beta(1+\varepsilon)+O(\lambda\log N)$	$O(k\lambda\log N)$ / $O(k\lambda N)$	$O(k\lambda\log N)$
PathDPoR	$O(k\log N)$ /$O(k\log N)$	$O(k\lambda\log N)$	$O(k\lambda\log N)$ /$O(k\lambda\log N)$	$O(k\lambda\log N)$

N: Block Numbers; k: Block Size over dictionary $\Sigma = 1^w$; λ: Security Parameter; s: constant, branch number of the tree in improved scheme;

TABLE 1. PERFORMANCE COMPARISON OF EXISTING DPoR SCHEMES

idea of using binary tree is from [7]). By this tree-based structures, the client will always do Read/Write operations within a path. However, the server have no ideas which path an encoded word is stored, which is the secret information to the client. After each access that revealing the path of an encoded word c_i, it will be assigned to a new path immediately, and resides in a local stash S on the client side. By uniformly distribution of the data in the tree, checking small t paths is sufficient in the Audit check to achieve security. As the size of stash S increases, we need an Evict algorithm to write the data back to the sever.

Efficient Evict: Recall that to Read/Write each encoded word c_i, it will be assigned to a $new\ path$ and stored in the local stash S. For each Evict, we will pick a path randomly named $evict\ path$, and only the words that are assigned to $evict\ path$ will be considered. Through this process, it can ensure that our shuffle consumes only $\log l$ overhead. Intuitively, the idea is to write c_i to the node that $new\ path$ and $evict\ path$ interacted with. However, when the size of each node is set to a small constant Z, it will easily become full and cause an overflow failure. Instead of setting larger node size, we improve the Evict algorithm by checking the availability of the proceeding nodes of the intersection nodes.

Deal with Overflow Failure: When a word fails to be stored in server's binary tree and cause an overflow failure, it will reside in the stash S, and be pushed back to the binary tree in **Evict** protocol. In addition, we also elaborate that the size of stash S can be bounded by $O(\log l_{code})$.

Contributions In this paper, we call our scheme $\mathcal{PDPOR}$, which costs $O(\log l_{code})$ for communication and computation asymptotically in both average and worst cases. Additionally, our improved scheme can accelerate an extra factor of s to $O(\log l_{code}/\log s)$, where s is the branch number of the tree. Both of them only consume constant client storage and linear server storage. Moreover, we prove our scheme with simpler techniques, and implement our construction in a practical way. Our tree-based scheme can be easily adapted to satisfy pattern hiding privacy, and be built as a hash tree.

2. Our $\mathcal{DPOR}$ scheme

To begin with, we introduce the scheme of dynamic proofs of retrievability in Definition 2.1. In terms of the detailed security definitions, we follow the definitions in [2].

Definition 2.1. *A dynamic proofs of retrievability scheme* $\mathcal{DPOR} = (\mathsf{Init}, \mathsf{Read}, \mathsf{Write}, \mathsf{Audit})$*, consists of four protocols between a stateful client* $\mathcal{C}$ *and a stateful server* $\mathcal{S}$*. In this scheme,* $\mathcal{C}$ *can be randomized, while the honest server* $\mathcal{S}$ *is assumed to be deterministic.*

$(st, \bar{\mathcal{M}}) \leftarrow$ ***Init***$(1^\lambda, l, w, \mathcal{M})$ *: On input the security parameter* λ *and the memory* $\mathcal{M} = \Sigma^w$ *, it outputs the client state* st *and the server state* $\bar{\mathcal{M}}$.

$\{m_i, \textbf{\textit{reject}}\} \leftarrow$ ***Read***$(i, st, \bar{\mathcal{M}})$ *: Server will read back the* i *th word* m_i *from the outsourced data, otherwise it will return reject.*

$\{(st', \bar{\mathcal{M}}'), \textbf{\textit{reject}}\} \leftarrow$ ***Write***$(i, m_i, st, \bar{\mathcal{M}})$ *: Client sends write request with new tuple* (i, m_i)*, server will update the corresponding word and return new state or output reject when the write fails.*

$\{\textbf{\textit{accept}}, \textbf{\textit{reject}}\} \leftarrow$ ***Audit***$(st, \bar{\mathcal{M}})$ *: In the audit, client will challenge the server on some random subset, then output accept if verify pass, else output reject.*

2.1. Our $\mathcal{PDPOR}$ Construction

Before we give the detailed descriptions of our construction, we list the key notations and terms in the followings:

Server State $\bar{\mathbf{M}}$: $\mathbf{T}$ is a full binary tree with height H, which is determined by the data size l_{code}. Each node of T is represented as a bucket with constant size Z, containing either real data or dummy value $\perp$. $\mathbf{Path}$ is a contiguous sequence of buckets from the root to a leaf. Let x denote a leaf identity, $P(x)$ is the path from the root to the leaf node x in T. In particular, $P(x, L)$ denotes the node at level L along the path $P(x)$, and $InterPath(x_1, x_2)$ will return the intersection node of $P(x_1)$ and $P(x_2)$. $\mathbf{PM}$ is encrypted position map, with each tuple (i, p_i), storing the map from logical index for each data cell to the path it resides in T.

Client State st: An array $\mathbf{S}$ is consisted of tuples (idx, pos, val). When overflow occurs in any node of T, the tuple will reside in S. $\mathbf{Encoder}$ is the primitive that encode k input words into n encoded words array via an efficient erasure encode algorithm.

Init: Given the original database $\mathcal{M}$ of length l that is encrypted under IND-CPA-secure encryption algorithm, the client first divides $\mathcal{M}$ into N blocks $\mathcal{M} = (\mathcal{M}_1, ..., \mathcal{M}_N)$, and each contains k entries. Let b be the block index, $\forall \mathcal{M}_b$, call $\mathsf{Encoder}(\mathcal{M}_\mathsf{b})$ and output $\mathcal{C}_b$ of n words. After that, we get the encoded database $\mathcal{C} = (\mathcal{C}_1, ..., \mathcal{C}_N)$. Let $l_{code} = n * N$, we can also represent the encoded database as $\mathcal{C} = (c_1, ..., c_{l_{code}})$. Assume that each entry has been attached by a corresponding message authentication code(MAC) for checking the correctness and authenticity of the outsourced database. Let $p_{a_i} \in (1, l_{code})$ denote the position assigned randomly for the encoded entry c_i, c_i will reside in some node of path $\mathsf{P}(\mathsf{p}_{\mathsf{a}_\mathsf{i}})$. Finally the client initializes stash S as an empty array.

On the server side, it will allocate the binary tree of height $H = \lceil \log l_{code} \rceil + 1$, and each node with Z dummy values $\perp$. Then store each tuple (i, p_{a_i}, c_i) of $\mathcal{C}$ in the leaf node p_{a_i}. At the same time, it stores the encrypted map $(\mathsf{i}, \mathsf{p}_{\mathsf{a_i}})$ in the position map PM.

Read(i): To read an element m_i, the client will first determine the block identity b that m_i belongs to and offset u that m_i resides in this block. Then read the encoded block $\mathcal{C}_b = (c_{b \cdot n+1} ... c_{b \cdot n+n})$ by retrieving from stash first else call Fetch(i). If the size of stash exceeds some threshold S', it will call Evict to write back some data in the stash S to the server. Then decode $\mathcal{C}_b$=($c_{b \cdot n+1}$, ..., $c_{b \cdot n+n}$) to obtain $\mathcal{M}_b$=($m_{b \cdot k+1}$, ..., $m_{b \cdot k+k}$) and return $m_{b \cdot k+u}$ as the requested data entry. Next re-assign a new random position p'_{a_i} to all the encoded data entries $c_{b \cdot n+1}$ to $c_{b \cdot n+n}$, and store them in the stash S, at the same time updating the position map PM on the server side.

Write(i, m'_i): To write a new tuple (i, m'_i), it is identical to Read(i), plus updating m_i to m'_i in stash and re-encode the block.

Evict: When the size of the stash S exceeds a threshold S', we will call Evict. Additionally, we first choose an evict path $p_e \in (1, l_{code})$ randomly, and then fetch all real data in the path p_e to the stash S and delete them in the tree. Next we will write back the tuples in the stash S to the root of the tree and try best to push them down along the $P(p_e)$ as deep as possible. Specifically, for any tuple (i', c'_i) in S, re-locate (i', c'_i) in the position between root and $InterPath(p_e, p_{a_i})$. If there is available space in this bucket, we will store (i', c'_i) in the next available space of the bucket in InterPath($\mathsf{p_e}, \mathsf{p_{a_i}}$), else try to store it in the ancestor of $InterPath(p_e, p_{a_i})$ until the root. If the traverse process fails, (i', c'_i) will remain in stash S.

Fetch(i): In Fetch, we first read from the position map PM to get $p_{a_i} = PM[i]$. Then traverse the path from the root to leaf p_{a_i} until the target entry is found. Specifically, for level L from 0 to H, server will read bucket in node $P(p_{a_i}, L)$. Read Z tuples (i, c_i, p_{a_i}) in each bucket, output the tuple and write back dummy value $\perp$ to the node when the target data cell is found in the bucket and return. Finally return the tuple to the client.

Audit: In Audit, the client or third party will call Fetch λ times, and read back λ random paths. Then check the correctness of all retrieved data, if all checking is correct, then it will output *accept*, else output *reject*.

2.2. Efficient Shuffle

Through randomly assigned position map and efficient Evict, we can avoid the server from deleting most data in some blocks which leads to failure of recovery for this block or loss the latest version. We use a simple example to illustrate the idea. Assumed that the height of our tree is 3, we set threshold for stash S′ as the stash size 3, and c_i is located in i th leaf after initialization. Firstly we call Fetch(c_1) and Fetch(c_3), the stash S still has available space, thus there is no need to call Evict at this moment. After Fetch(c_5), stash S becomes full, therefore we call Evict to rebuild T. Without loss of generality, we assume that the evict path is $P(5)$. Next we will discuss three results for Evict(5) with different path assignment to the entries whose positions have been revealed. The results of three cases, namely NonConflict Nodes, Improved Evict, Overflow Failure are presented in Figure 1. After Evict(5), most of the space has been deleted, and leave more room for the coming read sequence, which is depicted in Figure 1.

Non-Conflict Nodes: If the new paths assigned to entries c_1, c_3, c_5 are $5, 7, 7$, we can find that c_1 is assigned to $Path(5)$, obviously, $InterPath(5, 5) = P(5, 4)$, which is the empty leaf. Next c_3 and c_5 both have the intersection node with the evict path P(5) in the node P(5, 2), which can be occupied with these two entries.

Improved Evict: If the new paths assigned to entries c_1, c_3, c_5 are $7, 7, 8$, c_1 are also assigned to $P(7)$, and $P(5, 2)$ will be occupied by c_1 and c_3. When it comes to c_5, it will cause an overflow in Evict where the only candidate node for c_5 is $P(5, 2)$. However, we extend the path from the intersection node upwards to the root in our improved Evict idea. Therefore, c_5 will be located in the root, instead of output a overflow failure.

Overflow Failure: Now we consider the case that cause an overflow if all c_1, c_3 and c_5 are re-assigned to the left subtree (P(1) to P(4)). Then c_1, c_3 will be added to the root while c_5 will cause an overflow failure, and remains in the stash S, as depicted in the right bottom tree in Figure 1.

3. Performance Analysis

3.1. Security Proof

$\mathcal{PDPOR}$ has applied basic memory checking techniques, *i.e.* the message authenticity code attached in each entry. Therefore, the correctness and authenticity properties of $\mathcal{PDPOR}$ follows immediately from the underlying memory checking mechanism. Next we will show the main challenges to achieve retrievability of our construction.

Achieving retrievability Roughly speaking, there are two core steps in our approach in order to guarantee the retrievability. The first step is to retrieve sufficient correct data $\mathcal{C}$ (*e.g.* $\frac{3}{4} \cdot l_{code}$) from the whole index space, and the second one is that each block can be recovered in $\mathcal{C}$, which means that it is uniformly distributed. At the first step, we will start the proof with lemma 3.1.

Lemma 3.1. *In $\mathcal{PDPOR}$, if the server can pass the* Audit *protocol with probability in security parameter λ, then the server stores at least $\sigma \cdot l_{code}$ intact paths, where σ is a constant, being set as $(1 + k/n)/2$.*

Proof. Intuitively, if λ paths are verified successfully, then the server must at least maintain $\sigma \cdot l_{code}$ correct paths with significant probability. If the server's intact path number is fewer than $\sigma \cdot l_{code}$, then it can pass the Audit with probability $(1 - \sigma)^\lambda = negl(\lambda)$, which will contradict the claim that the server can pass the test with non-negligible probability.

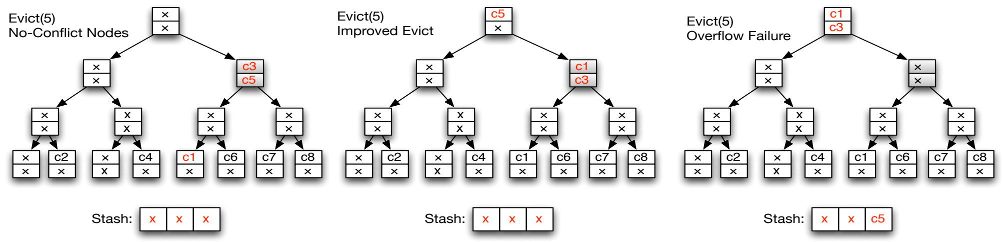

Figure 1. Example of Evict

More precisely, we define an event A to denote that the server at some state $\mathcal{S}'$ can pass Audit, and that the sever maintains at least $\sigma \cdot l_{code}$ paths in entirety as event P. Supposed that the proportion of correct paths is σ', we can easily have:

$$\Pr[\mathsf{A} \cap \bar{\mathsf{P}}] = \Pr[\mathsf{A}|\bar{\mathsf{P}}] \cdot \Pr[\bar{\mathsf{P}}] < \sigma'^{\lambda} < \sigma^{\lambda} \tag{1}$$

In addition, we also know that $\Pr[\mathsf{A}] \geq 1/\mathsf{poly}(\lambda)$. Along with Equation (1), we can imply:

$$\begin{aligned} &\Pr[\bar{\mathsf{P}}|\mathsf{A}] = \Pr[\mathsf{A} \cap \bar{\mathsf{P}}]/\Pr[\mathsf{A}] \leq \mathsf{poly}(\lambda) \cdot \sigma^{\lambda} \\ &= (\frac{1}{\sigma})^{-(\lambda + \frac{\log poly(\lambda)}{\log \sigma})} \\ &< (\frac{1}{\sigma})^{\lambda} = negl(\lambda) \end{aligned} \tag{2}$$

From Equation (2), we can have:

$$\Pr[\mathsf{P}|\mathsf{A}] = 1 - \Pr[\bar{\mathsf{P}}|\mathsf{A}] = 1 - \mathsf{negl}(\lambda) \tag{3}$$

From Equation (3) we see that event P occurs with significant probability on the condition that A happens, which means that server stores at least $\sigma \cdot l_{code}$ intact paths. □

Lemma 3.2. *Let $k = \Omega(\lambda)$, and $k/n = (1 - \Omega(1))$, all blocks can be recovered if server passes an* Audit *with probability larger than $\frac{1}{p}$, when $p = poly(\lambda)$.*

Proof. Based on Lemma 3.1, it is not difficult to prove lemma 3.2 since each data entry c_i is randomly located on a path, which means that all the encoded data entries are uniformly distributed among the l_{code} locations. More formally, denote that the intact path sequence is $\mathsf{P} = (p_1, p_2, ..., p_{\sigma'})$, where p_i is path identity and $p_i < p_{i+1}$. Based on Lemma 3.1, we know that $\sigma' >= \sigma$. Now we consider some special data cell m_i, it is easy to see that server cannot learn which path that m_i resides in. Then we have:

$$\Pr[\mathsf{PM}[\mathsf{m_i}] \in \mathsf{P}] = \sigma' >= \sigma \tag{4}$$

From Equation (4), we can infer that the probability the server holds m_i is at least σ.

Based on the application of erasure code, we can say that a block can be decoded if the server stores more than k entries in the block. To prove that all blocks can be decoded correctly, we first assume that block $\mathcal{C}_{\mathsf{j}}$ fails to be recovered. Then we assume that $\mathsf{X_i}$ ($i \in [1, n]$) is a random variable where $\mathsf{X_i}$ is 1 if the server stores the i th entry of block $\mathcal{C}_{\mathsf{j}}$, otherwise is 0. Let $\overline{\mathsf{X}} = \frac{1}{\mathsf{n}} \sum_{\mathsf{i}=1}^{\mathsf{n}} \mathsf{X_i}$, we can compute the probability that the server does not maintain block $\mathcal{C}_{\mathsf{j}}$ by Hoeffding's bound:

$$\begin{aligned} &\Pr[\mathsf{Fail}((\mathsf{C_j})] \\ &= \Pr[\overline{\mathsf{X}} < \frac{\mathsf{k}}{\mathsf{n}}] = \Pr[\overline{\mathsf{X}} < \sigma - (\sigma - \frac{\mathsf{k}}{\mathsf{n}})] \\ &< \exp(-2\mathsf{n}(\sigma - \frac{\mathsf{k}}{\mathsf{n}})^2) = \mathsf{negl}(\lambda) \end{aligned} \tag{5}$$

Based on Equation (5), we can take a union-bound for all blocks, and conclude that the server will not store full knowledge of a block only with negligible probability. □

Theorem 3.3. *Let $k = \Omega(\lambda)$, and $k/n = (1 - \Omega(1))$, then* PathDPoR=(Init, Read, Write, Audit) *is a dynamic* PoR *scheme.*

Proof. From Lemma 3.1 and 3.2, we can conclude that all blocks of the client's original database can be decoded correctly. Thus full knowledge of the outsourced database can be retrieved by Extract function defined above if the server can pass an Audit with probability larger than $poly(\lambda)$. This completes our proof of Retrievability. □

3.2. Efficiency Analysis

We first show the efficiency analysis for our basic $\mathcal{PDPOR}$, recap that l_{code} is the length of outsourced data.

Write cost: For each Write, the client should first retrieve the whole block, which costs $O(\log l_{code})$. After updating the value of the block, Enc in the Encoder will be called to re-encode the updated block by the client. Next the server will call Evict with overhead of $O(\log l_{code})$ if stash size reaches some threshold, which sums up to $O(\log l_{code})$ on the server-side. Particularly, we can achieve better-worst case of $O(\log l_{code})$ since it never leads to rebuild of $O(l_{code})$ data size as previous schemes [6] [2].

Audit cost: For each Audit process, we will randomly check λ paths with a total of $O(\lambda \log l_{code})$ overhead, which averaged to be $O(\log l_{code})$ for the Audit protocol.

Upper Bound of Evict Frequency Stash size can be bounded as $O(\log l_{code})$ if Evict is called after each Read [7]. However, we will show that the frequency to call Evict can be bounded to some limited number that is fewer than or equal to the number of Read if we set a reasonable stash threshold S'. Let Evict Frequency = |Evict| \ |Read|, we can get Theorem 3.4. Because of page limit, the proof of Theorem 3.4 will be included in our full version.

Theorem 3.4. *In* PathDPoR, *we set the threshold of stash size* $S' = \log l_{code}$, *then* $0 <$ Evict Frequency ≤ 1, *the upper bound of the* Evict Frequency *is 1.*

4. Experimental Results and Analysis

We implement both our basic scheme and the scheme in [6] as a comparison for the write cost with C program. We list the main results in the followings, and include more comparisons and results in our full version.

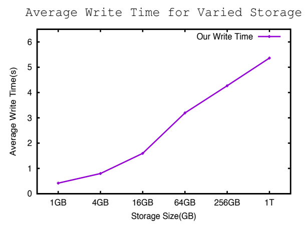

Figure 2. Server time for 1K block size and varied storage size

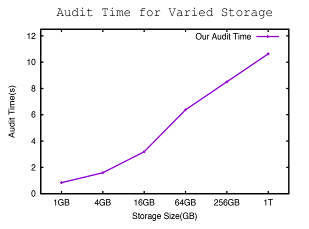

Figure 3. Audit time(ms) for varied storage size

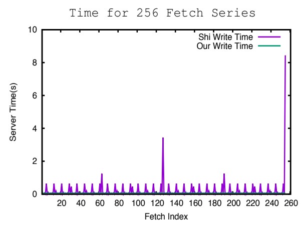

Figure 4. Server time(s) for 256 write sequence with rebuild

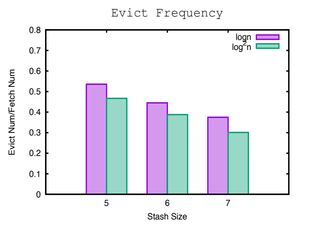

Figure 5. Evict Frequency with stash size of $\log n$ and $\log^2 n$

Server Cost: Figure 2 depicts the Write cost for varying storage size (1GB – 1TB) with fixed block size of 1KB. In Figure 4, we show that the server time cost for each Write in a sequence of length 256. The cost of our scheme is almost a flat line which is about the size of $\log l_{code}$ except for a small fluctuation caused by random overflow and Evict cost. However, in Shi *et al.* [6], the curve shows big fluctuations which will cause high cost of $O(l_{code})$ in the request of 2^L, and also periodical high cost of rebuild. Figure 3 shows the time cost for Audit on varying storage size from 1GB to 1TB. We can see the curve is nearly straight under our parameter settings.

Client-Server Bandwidth: We record the bandwidth cost for each Write by operating one block of 4KB(4096Bytes), with the cost for transferring a 4096bytes block as baseline. Since our Evict protocol needs data transfer between the server and the client and has about 0.6 Evict Frequency from our simulation result, we have about 1.8 bandwidth cost compared to Shi *et al.* [6] in which their scheme only have to transfer the block once. They pay less for the client-server bandwidth, but more for server computation. Even though our bandwidth results are not comparable with that in Shi *et al.* [6], we still achieve better bandwidth cost than other works' [2], which is 100X cost more than Shi *et al.* [6]

Evict Frequency: To show that the stash size and Evict Frequency are both upper-bounded, we depict the results in Figure 5. More precisely, we compare Evict Frequency of different stash size settings in Figure 5, which shows that Evict will be called less than 6^6 times out of 10^7 times of Read. Therefore the worst Evict Frequency is less than 0.6 in our experiment.

5. Conclusions

In the paper, we propose a secure and efficient dynamic proof of retrievability scheme $\mathcal{PDPOR}$. In particular, it achieves the best worst-case overhead. We also prove that $\mathcal{PDPOR}$ satisfy the security properties of $\mathcal{DPOR}$ with simple techniques. It is easy to extend the binary tree structure to a tree with s branches to improve the overhead from $O(\log l_{code})$ to $O(\log l_{code} / \log s)$.

Acknowledgments

This work was supported in part by NSFCRGC Joint Research Scheme (N_HKU 72913) of Hong Kong, Seed Funding Programme for Basic Research of HKU (201511159034, 201411159142), and National High Technology Research and Development Program of China (2015AA016008).

References

[1] A. Juels and B. S. Kaliski Jr, "Pors: Proofs of retrievability for large files," in *Proceedings of the 14th ACM conference on Computer and communications security*. Acm, 2007, pp. 584–597.

[2] D. Cash, A. Küpçü, and D. Wichs, "Dynamic proofs of retrievability via oblivious ram," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2013, pp. 279–295.

[3] M. T. Goodrich and M. Mitzenmacher, "Privacy-preserving access of outsourced data via oblivious ram simulation," in *Automata, Languages and Programming*. Springer, 2011, pp. 576–587.

[4] E. Kushilevitz, S. Lu, and R. Ostrovsky, "On the (in) security of hash-based oblivious ram and a new balancing scheme," in *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*. SIAM, 2012, pp. 143–156.

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

[6] E. Shi, E. Stefanov, and C. Papamanthou, "Practical dynamic proofs of retrievability," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 325–336.

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