Analysis and Optimization on FlexDPDP: A Practical Solution for Dynamic Provable Data Possession

Ertem Esiner, Alptekin Küpçü, Öznur Özkasap

Koç University,
Department of Computer Engineering
İstanbul, Turkey
{eesiner,akupcu,oozkasap}@ku.edu.tr

Abstract. Security measures, such as proving the integrity of the data, became more important with the increase in popularity of cloud data storage services. Dynamic Provable Data Possession (DPDP) was proposed in the literature to enable the cloud server to prove to the client that her data is kept intact, even in a dynamic setting where the client may update her files. Realizing that variable-sized updates are very inefficient in DPDP (in the worst case leading to uploading the whole file again), Flexible DPDP (FlexDPDP) was proposed.
In this paper, we analyze FlexDPDP scheme and propose optimized algorithms.We show that the initial pre-processing phase at the client and server sides during the file upload (generally the most time-consuming operation) can be efficiently performed by parallelization techniques that result in a speed up of 6 with 8 cores. We propose a way of handling multiple updates at once both at the server and the client side, achieving an efficiency gain of 60% at the server side and 90% in terms of the client's update verification time. We deployed the optimized FlexDPDP scheme on the large-scale network testbed PlanetLab and demonstrate the efficiency of our proposed optimizations on multi-client scenarios according to real workloads based on version control system traces.

1 Introduction

Data outsourcing to the cloud has become popular with the availability of affordable and more satisfying services (e.g., Dropbox, box.net, Google Drive, Amazon S3, iCloud, Skydrive) as well as with several studies in academia [16, 2, 4, 31, 3, 15, 25, 30, 11, 18]. The most important impediment in public adoption of cloud systems is the lack of some security guarantees in data storage services [24, 19, 33]. The schemes providing security guarantees should incur minimal overhead on top of the already available systems in order to promote wide adoption by the service providers.

In this work, we address the integrity of the client's data stored on the cloud storage servers. In a cloud storage system, there are two main parties, namely a server and a client, where the client transmits her files to the cloud storage server and the server stores the files on behalf of the client. For the client to be able to trust the service provider, she should be able to verify the integrity of the data. A trustworthy brand is not sufficient for the client, since hardware/software failures or malicious third parties may also cause data loss or corruption [9].

Solutions for the static cases (i.e., logging or archival storage) such as Provable Data Possession (PDP) [2] were proposed [2, 15, 25, 30, 3]. For the dynamic cases where the client keeps interacting with her data, Scalable PDP was proposed by Ateniese et al. [4], which allows a limited number of operations before a full re-calculation is required to continue providing provable data possession. Extensions of the PDP, using some data structures for dynamic cases, were first studied in Dynamic Provable Data Possession (DPDP) [16] that allows data updates while still providing integrity guarantees. Implementation of DPDP needs rank-based authenticated skip list as the underlying data structure. It is shown that DPDP is not applicable to variable block sized settings (due to the data structure used), hence resulting in unacceptable performance in the dynamic secure cloud storage systems [17]. To solve this issue, a flexible length-based authenticated skip list, called FlexList, and its application to a DPDP scheme allowing variable block-sized updates, called FlexDPDP, were proposed [17]. In this study, we improve the efficiency of the FlexDPDP system by proposing optimized algorithms on FlexList.

Our Contributions are as follows:

- We optimize the first pre-processing phase of the FlexDPDP provable cloud storage protocol by showing that the algorithm to build a FlexList in $O(n)$ time is well parallelizable even though FlexList is an authenticated data structure that generates dependencies over the file blocks. We propose a parallelization algorithm and our experimental results show a speed up of 6 and 7.7, with 8 and 12 cores respectively.
- We provide a multi-block update algorithm for FlexDPDP. Our experiments show 60% efficiency gain at the server side compared to updating blocks independently, when the updates are on consecutive data blocks.
- We provide an algorithm to verify update operations for FlexDPDP. Our new algorithm is applicable to not only modify, insert, and remove operations but also a mixed series of multiple update operations. The experimental results show an efficiency gain of nearly 90% in terms of verification time of consecutive updates.
- We deployed the FlexDPDP implementation on the network testbed PlanetLab and also tested its applicability on a real SVN deployment. The results show that our improved scheme is practically usable in real life scenarios after optimization, namely 4 times faster proof generation for consecutive updates.

1.1 Related Work

Ateniese et al. proposed the first provable storage service named PDP [2] that can only be applied to the static cases. To overcome this problem, Scalable PDP was proposed which allows limited updates [4]. When it consumes its precomputed tokens, Scalable PDP requires a setup phase from scratch. Wang et al. [32] proposed using Merkle tree and Zheng and Zu [34] proposed 2-3 trees as the data structure on top of PDP. Yet, these are also applicable to the static scenarios since there is no efficient algorithm, which keeps the authentication information maintained, is shown for re-balancing neither of these data structures. The authenticated skip lists that are probabilistically balanced in case of any updates were first proposed by [29].

For improving data integrity on the cloud, protocols such as [12, 22, 10, 26, 21, 27] provide Byzantium fault-tolerant storage services based on some server labor. There also exist protocols using quorum techniques, which do not consider the server-client scenarios but works on local systems such as hard disk drives or local storage [23, 20, 1, 13]. A recent protocol using quorum techniques [5] replicates the data on several storage providers to improve integrity of the data stored on the cloud; yet it also considers static data.

Within provable data possession techniques, Erway et al. proposed a skip-list-like data structure called rank based skip list [16] that allows dynamic operations. Yet Esiner et al. [17] showed that updates in DPDP needs to be of fixed block size, and proposed the FlexList data structure that allows variable length dynamic operations with DPDP scheme. Detailed comparison and extended descriptions of these two data structures are provided in [16, 17]. Some distributed versions of the idea have been studied as well [14, 18]. There are also studies showing that a client's file is kept intact in the sense that client can retrieve (recover) it fully whenever she wishes [25, 30, 15, 11, 7].

FlexDPDP, using FlexList, can perform modify, insert, and remove operations one block at a time on the cloud, without any limit on the number of updates and block sizes, while maintaining data possession guarantees. It also provides verification algorithms for update queries on single blocks. In this work, we show that the functions in FlexDPDP are open to optimization, and propose optimized efficient algorithms by evaluating them on the PlanetLab network testbed and with real data update scenarios.

2 Preliminaries

FlexDPDP approach provides variable block sized dynamic provable data possession and uses FlexList as the underlying data structure. We first introduce the intuition behind FlexList and definitions of FlexDPDP to form the basis for describing the proposed optimizations.

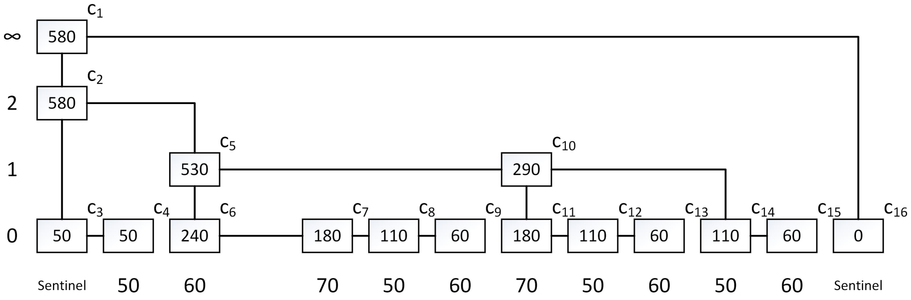


Fig. 1. A FlexList example.

FlexList is a skip-list-like **authenticated** data structure (shown in Figure 1). Each node keeps a *hash* value calculated according to its rank, level, the hash value of below neighbor, and the hash value of the after neighbor, where *rank* indicates the number of bytes that can be reached from the node, and *level* is the height of a node in the FlexList. Note that the hash of the root node is dependent to all leaf level nodes' hashes. Each leaf level node keeps a link to the **data** (the associated block of the file stored) to which it refers, the length of the data, and a **tag** value calculated for the data. **Rank** values are calculated by adding the below and after neighbors' ranks. If a node is at the leaf level, we use the length of its data as below neighbor's rank. FlexList has *sentinel* nodes as the first and the last nodes, which have no data and hence their length value is 0, as shown in Figure 1. These nodes generate no new dependencies but are useful to make algorithms easier and more understandable.

A node's below neighbor's rank shows how many bytes can be reached following the below link. The **search** operation uses this information to find a searched index. We check if the searched index is less than the rank of the below neighbor, if so we follow the below link, otherwise we follow the after link. When we follow an after link, the index we look for is diminished by the amount of bytes passed (rank of the below node). We repeat this procedure to reach the node that includes the search index. A **search path** is the ordered set of nodes visited on the way to reach a searched index by following the above rule starting from the root.

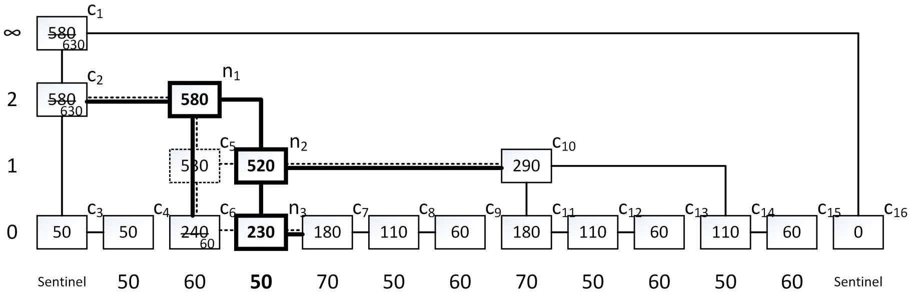


Fig. 2. Insert and remove examples on FlexList.

Insert/Remove operations perform addition/removal of a leaf node by keeping the necessary non-leaf nodes and removing the unnecessary ones, thus preserving the optimality of the structure (definitions and details are provided in [17]). Figure 2 illustrates an example of both **insert and remove** operations. First we insert a data of length 50 to index 110 at level 2. Dashed lines show the nodes and links which are removed, and bold lines show the newly added ones. Node c_5 is removed to keep the FlexList optimal [17]. The old rank values are marked and new values written below them. For the removal of the node at index 110, read the figure in the reverse order, where dashed nodes and lines are newly added ones and strong nodes and lines are to be removed, and the initial rank values are valid again.

Besides search, modify, insert, and remove algorithms, a **build skip list** algorithm was introduced in [17] that generates a FlexList on top of an ordered data using $O(n)$ time. The algorithm takes all data blocks, their corresponding tag values, and levels of prospective nodes as input, and generates the FlexList attaching nodes from right to left, instead of a series of insert method calls (which would cost $O(n \log n)$ in total). In Figure 1, the order of node generation is: c_{16}, c_{15}, c_{14}, c_{13}, c_{12}, c_{11}, c_{10}, c_9, and so on.

FlexDPDP [17] is a FlexList-based secure cloud storage scheme built on DPDP [16]. The scheme starts with the client pre-processing and uploading her data to the server. While pre-processing, both the client and the server build a FlexList over the data blocks. The client keeps the root of the FlexList as her meta data and the server keeps the FlexList as a whole. The server later uses the FlexList to generate proofs of data possession.

For **proof of possession**, Esiner et al. proposed an algorithm named **genMultiProof** [17], which collects all necessary values through search paths of the challenged nodes without any repetition. A multi proof is a response to a challenge of multiple nodes. For instance, a challenge to indices 50, 180, 230 in Figure 1 is replied by a proof vector as in Figure 4, together with a vector of tags of the challenged nodes and the block sum of the corresponding blocks. This proof vector is used to verify the integrity of these specific blocks. We use, in Section 3.3, this proof vector to verify the multiple updates on the server as well.

The client **verifies** the proof by calling **verifyMultiProof** which calculates the hash values from the proof vector one by one until the root's hash value. If the hash value of the root is equal to the meta data that the client keeps, and hashes and tags are verified, the client is satisfied. We use the *verifyMultiProof* method to verify the proof of the nodes on which we perform updates. **Update information** consists of the index of the update, the new data and the corresponding tag.

3 Optimizations on FlexDPDP

In this section, we describe our optimizations on FlexDPDP and FlexList for achieving an efficient and secure cloud storage system. We then demonstrate the efficiency of our optimizations in the next section.

First, we observe that a major time consuming operation in the FlexDPDP scheme is the pre-process operation, where a *build FlexList* function is employed. Previous $O(n)$ time algorithm [17] is an asymptotic improvement, but in terms of actual running times, it is still noticeably slow to build a large FlexList (e.g., half a minute for a 1GB file with 500000 blocks). A parallel algorithm can run as fast as its longest chain of dependent calculations, and in the FlexList structure each node depends on its children for the hash value; yet we show that building a FlexList is surprisingly well parallelizable.

Second, we observe that performing and verifying FlexDPDP updates in batches yield great performance improvements, and also match the real world usage of such a system. The hash calculations of a FlexList take most of the time spent for an update, and performing them in batches may save many hash unnecessary calculations.

Therefore, in this section, we provide a **parallel algorithm for building FlexList**, a **multi-block update algorithm** for the server to perform updates faster, and a **multi-block verification algorithm** for the client to verify the update proofs sent by the server. Notation used in our algorithms is presented in Table 1.

3.1 Parallel Build FlexList

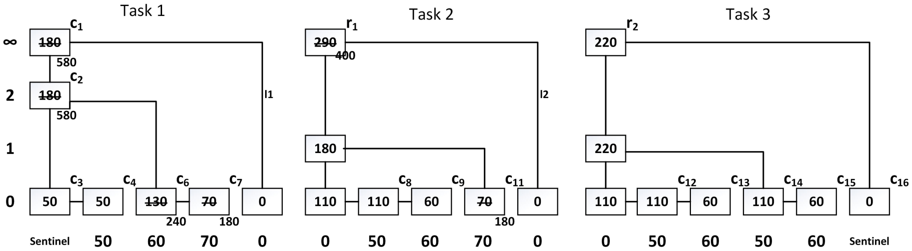


Fig. 3. A build skip list distributed to 3 cores.

We propose a parallel algorithm to generate a FlexList over the file blocks, resulting in the same FlexList as a sequentially generated one. The algorithm has three steps.

Symbol	**Description**
cn / *nn*	current node / new node
$after$ / *below*	node reached by following the after link / by following the below link
C	contains the indices that are challenged (ascending order)
i / $first$ / $last$	index / C's current index / C's end index
rs	The amount of bytes passed with each follow of an $after$ link
$state$	state contains a node, rank state, and $last$ index. These values are used to set the current node *cn* to the point where the algorithm will continue
P / T / M	proof vector / tag vector / block sum
$\sqcup_s$	intersection stack, stores states at intersections
$\sqcup_l$	stores nodes for which a hash calculation is to be done
Method	**Description**
canGoBelow [17]	returns true if the searched index can be reached by following the below link
isIntersection [17]	returns true when the first index can be found following the below link and the second index is found by following the after link. If there are more than one intersection, decrements $last$ for each until finds the closest one
generateIndices	generates an array of indices of the nodes that have been affected. Say the update index is *i*, the algorithm adds *i* for an insert or modify, adds *i* and *i-1* for a remove

Table 1. Symbols and helper methods used in our algorithms.

Figure 3 shows the parallel construction of the same FlexList as in Figure 1 on three cores. We first distribute tasks to threads and generate small FlexLists. Second, to unify them, we connect all roots together with links (c_1 to r_1 and r_1 to r_2, thus eliminate l_1 and l_2) and calculate new rank values of the roots (r_1 and c_1). Third, we use basic remove function to remove the left sentinels, which remain in between each part (to indices 360 and 180: 360 = c_1.rank - r_2.rank and 180 = c_1.rank - r_1.rank). In the example, the remove operation generates c_5 and c_{10} of Figure 1 and connects the remaining nodes to them, and rank values on the search paths of c_2, c_6, c_7, c_{11} are recalculated after the removal of sentinel nodes. As a result, all the nodes of the small FlexLists are connected to their level on the FlexList. After the unify operation, we obtain the same FlexList of Figure 1 generated efficiently in a parallel manner.

3.2 Handling Multiple Updates at Once

We investigated the verifiable updates and inferred that the majority of the time spent is for the hash calculations in each update. We discuss this in detail in Section 4. When a client alters her data and sends it to the server, she generates a vector of updates (U) out of a diff algorithm, which is used used to show the changes between the last and the former versions of a file.

Algorithm 3.1: multiUpdate Algorithm

```
Input: FlexList, U
Output: $P$, $T$, $M$, $newRootHash$

Let $U = (u_0, \ldots, u_k)$ where $u_j$ is the $j^{th}$ update information
$C$ = generateIndices($U$) //According to the nature of the update for each $u \in U$ , we add an index to the vector ($u_j.i$ for insert and modify, $u_j.i$ and $u_j.i - 1$ for remove as it is for a single update proof)
$P$, $T$, $M$= genMultiProof($C$) //Generates the multiProof using the FlexList
for $i = 0$ $to$ $k$ do
    apply $u_i$ to FlexList without any hash calculations
update $C$ to all affected nodes using $U$
calculateMultiHash($C$) // Calculates hash values of the changed nodes
$newRootHash$ = FlexList.root.hash
```

An update information $u \in U$, includes an index i, and (if insert or modify) a block and a tag value. Furthermore, the updates on a FlexList consist of a series of *modify* operations followed by either *insert* or *remove* operations, all to adjacent nodes. This nature of the update operations makes single updates inefficient since they keep calculating the hash values of the same nodes over and over again. To overcome this problem, we propose dividing the task into two: doing a series of **updates without the hash calculations**, and then calculating all **affected nodes' hash values at once**, where affected means that at least an input of the hash calculation of that node has changed. The **multiUpdate** (Algorithm 3.1) gets a FlexList and vector of updates U, and produces proof vector P, tag vector T, block sum M, and new hash value $newRootHash$ of the root after the updates.

hashMulti (Algorithm 3.2), employed in *calculateMultiHash* algorithm, collects nodes on a search path of a searched node. In the meantime, it is collecting the intersection points (which is the lowest common ancestor (lca) of the node the collecting is done for and the next node of which the hash calculation is needed). The repetitive calls from *calculateMultiHash* algorithm for each searched node collects all nodes which may need a hash recalculation. Note that each time, a new call starts from the last intersecting (lca) node.

Algorithm 3.2: hashMulti Algorithm

```
Input: $cn$, $C$, $first$, $last$, $rs$, $\sqcup_l$, $\sqcup_s$
Output: $cn$, $\sqcup_l$, $\sqcup_s$
// Index of the challenged block (key) is calculated according to the current sub skip
list root
$i = C_{first} - rs$
while Until challenged node is included do
    $cn$ is added to $\sqcup_l$
    //When an intersection is found with another branch of the proof path, it is saved
    to be continued again, this is crucial for the outer loop of ``multi'' algorithms
    if isIntersection($cn$, $C$, $i$, $last_k$, $rs$) then
        //note that $last_k$ becomes $last_{k+1}$ in isIntersection method
        $state(cn.after, last_{k+1}, rs + cn.below.r)$ is added to $\sqcup_s$
    if (CanGoBelow($cn$, $i$)) then
        $cn = cn.below$ //unless at the leaf level
    else
        // Set index and rank state values according to how many bytes at leaf nodes are
        passed while following the after link
        $i$ -= $cn.below.r$; $rs$ += $cn.below.r$; $cn = cn.after$
```

calculateMultiHash (Algorithm 3.3) first goes through all changed nodes and collects their pointers, then calculates all their hash values from the largest index value to the smallest, until the root. This order of hash calculation respects all hash dependencies.

We illustrate handling multiple updates with an example. Consider a *multiUpdate* called on the FlexList of Figure 1 and a consecutive modify and insert happen to indices 50 and 110 respectively (insert level is 2). When the updates are done without hash calculations, the resulting FlexList looks like in Figure 2. Since the tag value of c_6 has changed and a new node added between c_6 and c_7, all the nodes getting affected should have a hash recalculation. If we first perform the insert, we need to calculate hashes of n_3, n_2, c_6, n_1, c_2 and c_1. Later, when we do the modification to c_6 we need to recalculate hashes of nodes c_6, n_1, c_2 and c_1. There are 6 nodes to recalculate hashes but we do 10 hash calculations. Instead, we propose performing the insert and modify operations and call *calculateMultiHash* to indices 50 and 110. The first call of *hashMulti* goes through c_1, c_2, n_1, and c_6. On its way, it pushes n_2 to a stack since the next iteration of *hashMulti* starts from n_2. Then, with the second iteration of *calculateMultiHash*, n_2 and n_3

are added to the stack. At the end, we call the nodes from the stack one by one and calculate their hash values. Note that the order preserves the hash dependencies.

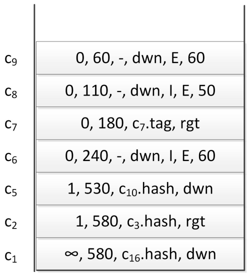
Algorithm 3.3: calculateMultiHash Algorithm

```
Input: $C$
Output:
    Let C$=(i_0,\ldots,i_k)$ where $i_j$ is the $(j+1)^{th}$ altered index; state$_m = (node_m, lastIndex_m,\ rs_m)$
    $cn$ = $root$; $rs$ = 0; $\sqcup_s$, $\sqcup_l$ are empty; $state$= (root, k, $rs$)
    // Call $hashMulti$ method for each index to fill the changed nodes stack $\sqcup_l$
    for $x = 0$ to $k$ do
        $hashMulti(state.\text{node}, C, x, state.end, state.rs, \sqcup_l, \sqcup_s)$
        if $\sqcup_s$ not empty then
            $state = \sqcup_s.\text{pop}()$; $cn = state.\text{node}$ ; $state.rs$ += $cn.below.r$
    for $k = \sqcup_l.size$ to 0 do
        calculate hash of $k^{th}$ node in $\sqcup_l$
```

3.3 Verifying Multiple Updates at Once

When the *multiUpdate* algorithm is used at the server side of the FlexDPDP protocol, it produces a proof vector, in which all affected nodes are included, and a hash value, which corresponds to the root of the FlexList after all of the update operations are performed.

The solution we present to verify such an update is constructed in four parts. First, we **verify the multi proof** both by FlexList verification and tag verification.



A proof Node (a tuple in proof vector) contains the following:

- Level
- Rank
- Hash of neighbor not included in the proof vector
- Direction of the next proof node relative to this one
- Intersection flag
- End flag
- Data length

Fig. 4. An output of a multiProof algorithm.

Second, we **construct a temporary FlexList**, which is constituted of the parts necessary for the updates. Third, we **do the updates** as they are, at the client side. The resulting temporary FlexList has the root of the original FlexList at the server side after performing all updates correctly. Fourth and last, we **check** if the **new root** we calculated is the same as the one sent by the server. If they are the same return accept and update the meta data that is kept by the client.

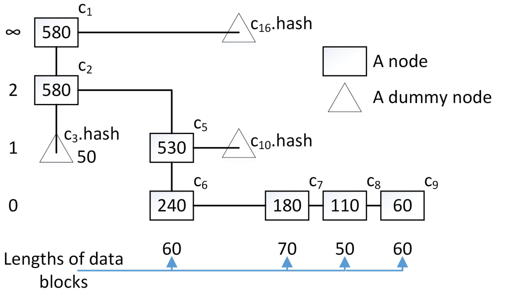


Fig. 5. The temporary FlexList generated out of the proof vector in Figure 4. Note that node names are the same with Figure 1.

Constructing a temporary FlexList out of a multi proof: Building a temporary FlexList is giving the client the opportunity to use the regular FlexList methods to do the necessary changes to calculate the new root. **Dummy nodes** that we use below are the nodes that have some values set and are **never subject to recalculation**.

We explain the algorithm A.1 (see Appendix) using the proof vector presented in Figure 4. The output of the algorithm given the proof vector is the temporary FlexList in Figure 5. First, the algorithm takes the proof node for c_1, generates the root using its values and adds the dummy after, with the hash value (of c_{16}) stored in it. And the nodes are connected to each other depending on their attributes. The proof node for c_2 is used to add node c_2 to the below of c_1 and the c_2's dummy node is connected to its below with rank value of 50, calculated as rank of c_2 minus rank of c_5. Note that the rank values of below nodes are used in regular algorithms so we calculate and set them. The next iteration sets c_5 as c_2's after and c_5's dummy node $c_1$0 is added to c_5's after. The next step is to add c_6 to the below of c_5. c_6 is both an end node and an intersection node, therefore we set its tag (from the tag vector) and its length values. Then we attach c_7 and calculate its length value since it is not in the proof vector generated by *genMultiProof* (but we have the necessary information: the rank of c_7 and the rank of c_8). Next, we add the node for c_8, and set its length value from the proof node and its tag value from the tag vector. Last, we do the same to c_9 as c_8. The algorithm outputs the root of the new temporary FlexList.

Verification: Recall that U is the list of updates generated by the client. An update information $u \in U$, includes an index i, and if the update is an insertion or modification, a block and a tag value. The client calls *verifyMultiUpdate* (Algorithm A.2) with its meta data and the outputs P, T, M of *multiUpdate* from the server. If *verifyMultiProof* returns accept, we call *buildDummyFlexList* with the proof vector P. The resulting temporary FlexList is ready to handle updates. Again we perform the updates without the hash calculations and then call the *calculateMultiHash* algorithm. But, we do not need to track changes to call a *calculateMultiHash* at the end, but instead calculate the hash of all the nodes present in the list. Last, we check if the resulting hash of the root of our temporary FlexList is equal to the one sent by the server. If they are the same, we accept and update the client's meta data.

4 Experimental Evaluation

We used our implementations of the FlexList data structure and the FlexDPDP protocol, that are in C++ with the aid of the *Cashlib* library [28, 8] for cryptography and the *Boost Asio* library [6] for network programming. Our local experiments are run on a 64-bit computer possessing 4 Intel(R) Xeon(R) CPU E5-2640 @ 2.50GHz CPU, 16GB of memory and 16MB of L2 level cache, running Ubuntu 12.04 LTS. The security parameters are as follows: 1024 bit RSA modulus, 80 bit random numbers, SHA1 as hash function resulting with an expected security of 80 bits. Mostly, FlexList operations run on RAM, but we keep each block of a file separately on the hard disk drive and **include the I/O times in our experimental analysis**.

4.1 Parallel buildFlexList Performance

Figure 6 shows the build FlexList function's time as a function of the number of cores used in parallel. The case of one core corresponds to the buildFlexList function proposed in [17]. From 2 cores to 24 cores, we measure the time spent by our parallel build FlexList function. Notice the speed up where parallel build reduces the time to build a FlexList of 4 million blocks from 240 seconds to 30 seconds on 12 coresThe speedup values are reported in Figure 7 where T stands for time for a single core used and T_p

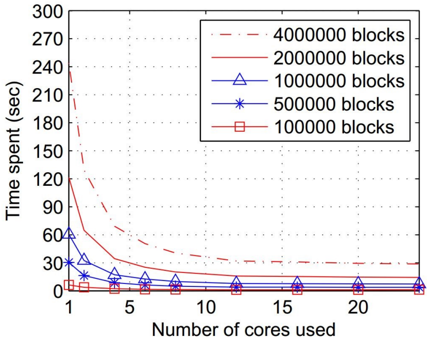


Fig. 6. Time spent while building a FlexList from scratch.

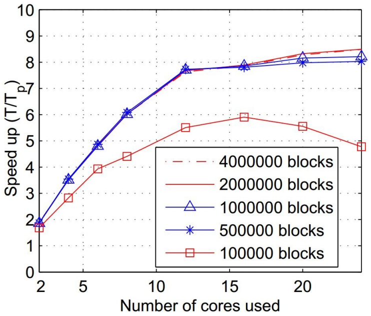


Fig. 7. Speedup values of buildFlexList function with multiple cores.

stands for time with *p* number of cores used. The more sub-tasks created, the more time is required to divide the big task into parts and to combine them. We see that a FlexList of 100000 blocks does not get improved as much, since the sub tasks are getting smaller and the overhead of thread generation starts to surpass the gain of parallel operations. Starting from 12 cores, we observe this side effect for all sizes. For 500000 blocks (i.e., 1GB file) and larger FlexLists, **speed ups of 6 and 7.7** are observed on 8 and 12 cores respectively.

4.2 Server-Side Multi Update Operations

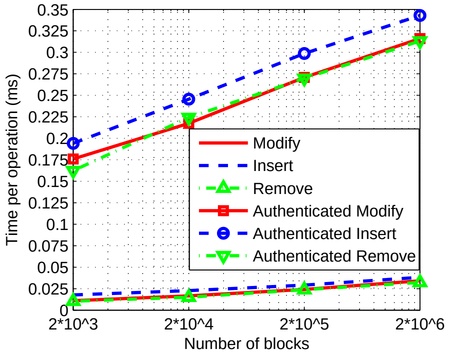


Fig. 8. Time spent for an update operation in FlexList with and without hash calculations.

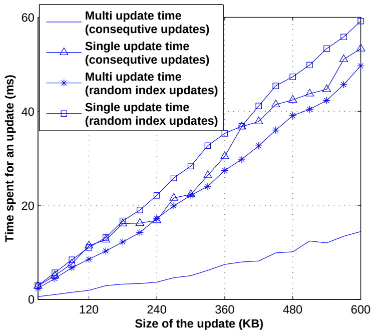


Fig. 9. Time spent on performing multi updates against series of single updates.

Results for the core FlexList methods (insert, remove, modify) with and without the hash calculations for various sizes of FlexList are shown in Figure 8. Even with the I/O time, the operations with the hash calculations take 10 times more time than the simple operations in a 4GB file (i.e., 2000000 nodes). The hash calculations in an update take 90% of the time spent for an update operation. Therefore, this finding indicates the benefit of doing hash calculations only once for multiple updates in the *performMultiUpdate* algorithm.

performMultiUpdate allows using multi proofs as discussed in Section 3. This provides ∼25% time and space efficiency on the verifiable update operations when the update is ∼20KB, and this gain increases up to ∼35% with 200KB updates.The time spent for an update at the server side for various size of updates is shown in Figure 9 with each data point reflecting the average of 10 experiments. Each update is an even mix of modify, insert, and remove operations. If the update locality is high, meaning the updates are on consecutive blocks (a diff operation generates several modifies to consecutive blocks followed by a series of remove if the added data is shorter than the deleted data, or a series of inserts otherwise [17]), using our *calculateMultiHash* algorithm after the updates without hash calculation on a FlexList for a 1GB file, the server time for **300 consecutive update operations** (a 600KB update) decreased **from 53ms to 13ms**.

4.3 Client-Side Multi Update Operations

For the server to be able to use *multiUpdate* algorithm, the client could be able to verify multiple updates at once. Otherwise, as each single verify update requires a root hash value after that specific update, all hash values on the search path of the update should be calculated each time. Also, each update proof should include a FlexList proof alongside them. Verifying multiple updates at once not only diminishes the proof size but also provides time improvements at the client side. Figure 10 shows that a multi verify operation is faster at the client side when compared to verifying all the proofs one by one. We tested two scenarios: One is for the updates randomly distributed along the FlexList, and the other is for the updates with high locality. The client verification time is highly improved. For instance, with a 1GB file and a 300KB update, verification at the client side was reduced from 45ms to less than 5ms. With random updates, the multi verification is still 2 times faster.

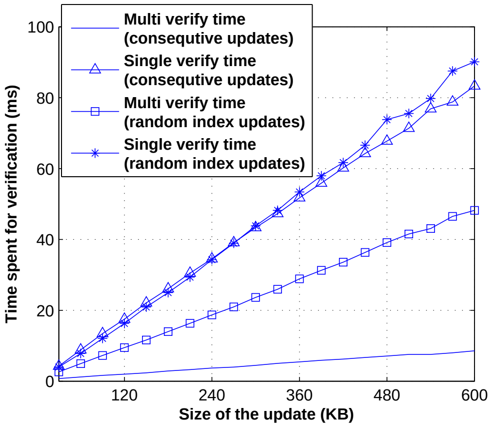


Fig. 10. MultiVerify of an update against standard verify operations.

4.4 Real Usage Performance Analysis via PlanetLab

We deployed the FlexDPDP model on the world-wide network testbed PlanetLab. We chose a node in Wuerzburg, Germany[1] on PlanetLab as the server which has two Intel(R) Core(TM)2 CPU 6600 @ 2.40GHz (IC2) and 48MBit upload and 80MBit download speed. Our protocol runs on a 1GB file, which is divided into blocks of 2KB, having 500000 nodes (for each client). The throughput is defined as the maximum number of

[1] planetlab1.informatik.uni-wuerzburg.de

queries the server can reply in a second. Our results are the average of 50 runs on the PlanetLab with randomly chosen 50 clients from all over the Europe.

Challenge queries :

We measured two metrics, the whole time spent for a challenge proof interaction at the client side and the throughput of the server (both illustrated in Figure 11). As shown in the Figure, the throughput of the server is around 21. When the server limit is reached, we observe a slowdown on the client side where the response time increases from around 500 ms to 1250 ms. Given that preparing a proof of size 460 using the IC2 processor takes 40ms using *genMultiProof* on a single core, we conclude that the bottleneck is not the processing power. The challenge queries are solely a seed, thus the download speed is not the bottleneck neither. A proof of a multi challenge has an average size of 280KB (∼215KB FlexList proof, ∼58KB tags, ∼2KB blocksum), therefore to serve 21 clients in a second a server needs 47 MBit upload speed which seems to be the bottleneck in this experiment. The more we increase the upload speed, the more clients we can serve with such a low end server.

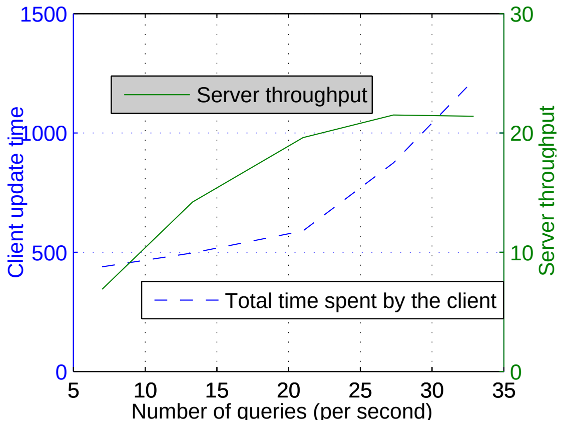


Fig. 11. Clients challenging their data. Two lines present: first, server throughput in count per second and second, whole time for a challenge query of FlexDPDP, in ms.

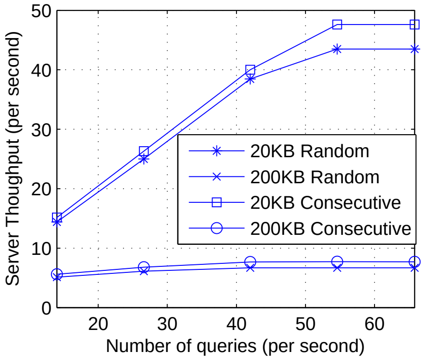


Fig. 12. Server throughput versus the frequency of the client queries.

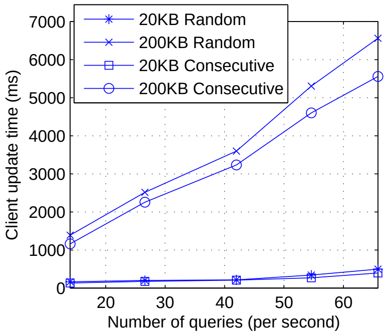


Fig. 13. A client's total time spent for an update query (sending the update, receiving a proof and verifying the proof).

Update queries :

Real life usage analysis over real version control system traces: We have conducted analysis on the SVN server where we have 350MB of data that we have been using for the past 2 years. We examined a sequence of 627 commit calls and provide results for an average usage of a commit function by means of the **update locality**, the **update size** being sent through the network, and the updated **number of blocks**.

We consider the directory hierarchy proposed in [16]. The idea presented is to set root of each file's FlexList (of the single file scheme presented) in the leaf nodes of a dictionary used to organize files. The update locality of the commits is very high. More than 99% of the updates in a single commit occur in the same folder, thus do not affect most parts of the directory, thus FlexList but a small portion of them. Moreover, 27% of the updates are consecutive block updates on a single field of a single file.

With each commit an average of size 77KB is sent, where we have 2.7% commits of size greater than 200KB and 85% commits has size less than 20KB. These sizes are the amounts sent through the network. Erway et al. show analysis on 3 public SVN repositories. They indicate that the average update size is 28KB [16]. Therefore in our experiments on PlanetLab we choose 20KB (to show general usage) and 200KB (to show big commits) as the size sent for a commit call. The average number of blocks affected per commit provided by Erway et al. is 13 [16] and is 57.7 in our SVN repository. They both show the necessity of efficient multiple update operations.

We observe the size variation of the commits and see that the greatest common divisor of the size of all commits is 1, as expected. Thus we conclude that fixed block sized rank-based authenticated skip lists is not applicable to the cloud storage scenario.

Update size and type	Server proof generation time	Corresponding proof size
200KB (100 blocks) randomly dist.	30ms	70KB
20KB (10 blocks) randomly dist.	10ms	11KB
200KB (100 blocks) consecutive	7ms	17KB
20KB (10 blocks) consecutive	6ms	4KB

Table 2. Proof time and size table for various type of updates.

Update queries on the PlanetLab: We perform analysis using the same metrics as a challenge query. The first one is the whole time spent at the client side (Figure 13) and the second one is the throughput of the server (Figure 12), for updates of size ∼20KB and ∼200KB. We test the behavior of the system by varying the query frequency, the update size, and the update type (updates to consecutive blocks or randomly selected blocks). Table 2 shows the measurements for each update type.

Figure 12 shows that a server can reply to ∼45 many updates of size 20KB and ∼8 many updates of size 200KB per second. Figure 13 also approves, that the server is loaded, by the increase in time of a client getting served. Comparing update proofs with the proof size of only challenges (shown in Figure 11), we conclude that the bottleneck for replying update queries is not the upload speed of the server, since a randomly distributed update of size 200KB needs 70KB proof and 8 proof per second is using just 4.5 Mbit of the upload bandwidth or a randomly distributed updates of size 20KB needs a proof of size 11KB and 45 proof per second uses only 4MBit of upload bandwidth. Table 2 shows the proof generation times at the server side. 30ms per 200KB random operation is required so a server may answer up to 110-120 queries per second with IC2 processor and 10ms per 20KB random operation is required, thus a server can reply up to 300 queries per second. Therefore, the bottleneck is not the processing power either. Eventually the amount of queries of a size a server can accept per second is limited, even though the download bandwidth does not seem to be loaded up. But, note that the download speed is checked with a single source and a continuous connection. When a server keeps accepting new connections, the end result is different. This was not a limiting issue in answering challenge queries since a challenge is barely a seed to show the server which blocks are challenged. In our setting, there is one thread at the server side which accepts a query and creates a thread to reply it. We conclude that the bottleneck is the server query acceptance rate of our implementation. These results indicate that with a distributed and replicated server system, a server using FlexDPDP scheme may reply to more queries.

5 Conclusion and Future Work

In this study, we have extended the FlexDPDP scheme with optimized and efficient algorithms, and tested their performance on real workloads in network realistic settings. We obtained a speed up of 6 using 8 cores on the pre-processing step, 60% improvement on the server-side updates, and 90% improvement while verifying them at the client side. We deployed the scheme on the PlanetLab testbed and provided detailed analysis using real version control system workload traces. As future work, we plan to extend FlexDPDP to distributed and replicated servers.

Acknowledgement

We would like to thank Ozan Okumuşoğlu at Koç University, Istanbul, Turkey for his contribution on testing and debugging, working on implementation of server-client side of the project and verification algorithms. We also acknowledge the support of TÜBİTAK (the Scientific and Technological Research Council of Turkey) under project numbers 111E019 and 112E115, Türk Telekom, Inc. under grant 11315-06, the European Union COST Action IC1206, and Koç Sistem, Inc.

References

1. I. Abraham, G. Chockler, I. Keidar, and Malkhi. Byzantine disk paxos: optimal resilience with byzantine shared memory. *Distributed Computing*, 2006.
2. G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *ACM CCS*, 2007.
3. G. Ateniese, S. Kamara, and J. Katz. Proofs of storage from homomorphic identification protocols. In *ASIACRYPT*, 2009.
4. G. Ateniese, R. D. Pietro, L. V. Mancini, and G. Tsudik. Scalable and efficient provable data possession. In *SecureComm*, 2008.
5. A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa. Depsky: dependable and secure storage in a cloud-of-clouds. EuroSys '11. ACM, 2011.
6. Boost asio library. http://www.boost.org/doc/libs.
7. K. D. Bowers, A. Juels, and A. Oprea. Hail: a high-availability and integrity layer for cloud storage. In *ACM CCS*, 2009.

8. Brownie cashlib cryptographic library. http://github.com/brownie/cashlib.
9. C. Cachin, I. Keidar, and A. Shraer. Trusting the cloud. *SIGACT News*, 2009.
10. C. Cachin and S. Tessaro. Optimal resilience for erasure-coded byzantine distributed storage. DSN '06, Washington, DC, USA, 2006. IEEE Computer Society.
11. D. Cash, A. Küpçü, and D. Wichs. Dynamic proofs of retrievability via oblivious ram. In *EUROCRYPT*, 2013.
12. G. Chockler, R. Guerraoui, I. Keidar, and M. Vukolic. Reliable distributed storage. *Computer*, 2009.
13. G. Chockler and D. Malkhi. Active disk paxos with infinitely many processes. In *In Proceedings of PODC02*. ACM Press, 2002.
14. R. Curtmola, O. Khan, R. Burns, and G. Ateniese. Mr-pdp: Multiple-replica provable data possession. In *ICDCS*, 2008.
15. Y. Dodis, S. Vadhan, and D. Wichs. Proofs of retrievability via hardness amplification. In *TCC*, 2009.
16. C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia. Dynamic provable data possession. In *ACM CCS*, 2009.
17. E. Esiner, A. Kachkeev, S. Braunfeld, A. Küpçü, and O. Özkasap. Flexdpdp: Flexlist-based optimized dynamic provable data possession. Cryptology ePrint Archive, Report 2013/645, 2013.
18. M. Etemad and A. Küpçü. Transparent, distributed, and replicated dynamic provable data possession. In *ACNS*, 2013.
19. B. Furht and A. Escalante. *Handbook of Cloud Computing*. Computer science. 2010.
20. E. Gafni and L. Lamport. Disk paxos. *Distrib. Comput.*, 16(1):1–20, Feb. 2003.
21. G. G. M. R. Garth Goodson, Jay Wylie. Efcient byzantine-tolerant erasure-coded storage. DSN '04, 2004.
22. J. Hendricks, G. R. Ganger, and M. K. Reiter. Low-overhead byzantine fault-tolerant storage. SOSP '07. ACM, 2007.
23. P. Jayanti, T. D. Chandra, and S. Toueg. Fault-tolerant wait-free shared objects. *J. ACM*, 1998.
24. M. Jensen, J. Schwenk, N. Gruschka, and L. L. Iacono. On technical security issues in cloud computing. In *Cloud Computing CLOUD'09*. IEEE, 2009.
25. A. Juels and B. S. Kaliski. PORs: Proofs of retrievability for large files. In *ACM CCS*, 2007.
26. B. Liskov and R. Rodrigues. Tolerating byzantine faulty clients in a quorum system. *IEEE 32nd International Conference on Distributed Computing Systems*, 2006.
27. D. Malkhi and M. Reiter. Byzantine quorum systems. *Distrib. Comput.*, 1998.
28. S. Meiklejohn, C. Erway, A. Küpçü, T. Hinkle, and A. Lysyanskaya. Zkpdl: Enabling efficient implementation of zero-knowledge proofs and e-cash. In *USENIX Security*, 2010.
29. C. Papamanthou and R. Tamassia. Time and space efficient algorithms for two-party authenticated data structures. In *ICICS*, 2007.
30. H. Shacham and B. Waters. Compact proofs of retrievability. In *ASIACRYPT*, 2008.
31. P. T. Stanton, B. McKeown, R. C. Burns, and G. Ateniese. Fastad: an authenticated directory for billions of objects. *SIGOPS Oper. Syst. Rev.*, 2010.
32. Q. Wang, C. Wang, J. Li, K. Ren, and W. Lou. Enabling public verifiability and data dynamics for storage security in cloud computing. In *ESORICS*, 2009.
33. P. S. Wooley. Identifying cloud computing security risks. Technical report, 7 University of Oregon Eugene, 2011.
34. Q. Zheng and S. Xu. Fair and dynamic proofs of retrievability. In *CODASPY*, 2011.

A Optimization Algorithms

Algorithm A.1: constructTemporaryFlexList Algorithm

Input: P, T
Output: root (temporary FlexList)

```
Let $P = (A_0, \ldots, A_k)$, where $A_j = ($ $level_j$, $r_j$, hash$_j$, $rgtOrDwn_j$, $isInter_j$, $isEnd_j$, $length_j)$ for
$j = 0, \ldots, k$; $T = (tag_0, \ldots, tag_t)$, where $tag_t$ is tag for challenged $block_t$ and dummy nodes are
nodes including only hash and rank values set on them and they are final once they are
created; //
$root$ = new Node($r_0$, $length_0$) // This node is the root and we keep this as a pointer to return
at the end//
$\sqcup_s$ = new empty stack
$cn$ = $root$
dumN = new dummy node is created with hash$_j$
$cn$.after = dumN
for $i = 0$ to $k$ do
    $nn$ = new node is created with Level$_{i+1}$ and $r_{i+1}$
    if $isEnd_i$ and $isInter_i$ then
        $cn.tag$ = next tag in $T$; $cn.length$ = $length_i$ ; $cn$.after = $nn$; $cn$ = $cn$.after
    else if $isEnd_i$ then
        $cn.tag$ = next tag in $T$; $cn.length$ = $length_i$ ; if $r_i$ != $length_i$ then
            dumN = new dummy node is created with hash$_i$ as hash and $r_i$ - $length_i$ as rank
            $cn$.after = dumN
        if $\sqcup_s$ is not empty then
            $cn$ = $\sqcup_s$.pop() ; $cn$.after = $nn$; $cn$ = $cn$.after
    else if $level_i = 0$ then
        $cn.tag$ = hash$_i$ ; $cn.length$ = $r_i$ - $r_{i+1}$ ; $cn$.after = $nn$ ; $cn$ = $cn$.after
    else if $isInter_i$ then
        $cn$ is added to $\sqcup_s$ ; $cn$.below = $nn$; $cn$ = $cn$.below
    else if $rgtOrDwn_i = rgt$ then
        $cn$.after = $nn$
        dumN = new dummy node is created with hash$_i$ as hash and $r_i$ - $r_{i+1}$ as rank
        $cn$.below = dumN ; $cn$ = $cn$.after
    else
        $cn$.below = $nn$
        dumN = new dummy node is created with hash$_i$ as hash and $r_i$ - $r_{i+1}$ as rank
        $cn$.after = dumN ; $cn$ = $cn$.below
return $root$
```

Algorithm A.2: verifyMultiUpdate Algorithm

Input: P, T ,$MetaData$, U, $MetaData_{byServer}$
Output: accept or reject

```
Let U$= (u_0, \ldots, u_k)$ where $u_j$ is the $j^{th}$ update information
if !verifyMultiProof(P, T, MetaData) then
    return reject
FlexList = buildTemporaryFlexList($P$)
for $i = 0$ to $k$ do
    apply $u_i$ to FlexList without any hash calculations
calculate hash values of all nodes in the temporary FlexList. //A recursive call from the root
if $root.hash$ != $MetaData_{byServer}$ then
    return reject
return accept
```