FlexDPDP: FlexList-based Optimized Dynamic Provable Data Possession

Ertem Esiner[1], Adilet Kachkeev[1], Samuel Braunfeld[2], Alptekin Küpçü[1], and Öznur Özkasap[1]

[1]Department of Computer Engineering, Koç University, İstanbul, TURKEY
{eesiner,akachkeev,akupcu,oozkasap}@ku.edu.tr
[2]Department of Electrical Engineering and Computer Science, University of California, Berkeley, USA
sbraunfeld@berkeley.edu

Abstract

With popularity of cloud storage, efficiently proving the integrity of data stored at an untrusted server has become significant. Authenticated Skip Lists and Rank-based Authenticated Skip Lists (RBASL) have been used in cloud storage to provide support for provable data update operations. In a dynamic file scenario, an RBASL falls short when updates are not proportional to a fixed block size; such an update to the file, however small, may translate to O(n) many block updates to the RBASL, for a file with n blocks.

To overcome this problem, we introduce FlexList: Flexible Length-Based Authenticated Skip List. FlexList translates even variable-size updates to O(u) insertions, removals, or modifications, where u is the size of the update divided by the block size. We present various optimizations on the four types of skip lists (regular, authenticated, rank-based authenticated, and FlexList). We compute one single proof to answer multiple (non-)membership queries and obtain efficiency gains of 35%, 35% and 40% in terms of proof time, energy, and size, respectively. We also deployed our implementation of FlexDPDP (DPDP with FlexList instead of RBASL) on PlanetLab, demonstrating that FlexDPDP performs comparable to the most efficient static storage scheme (PDP), while providing dynamic data support.

Keywords: Cloud storage, skip list, authenticated dictionary, provable data possession, data integrity.

I. Introduction

Data outsourcing has become quite popular in recent years both in industry (e.g., Amazon S3, Dropbox, Google Drive) and academia [2], [4], [11], [14], [15], [21], [30], [31]. A client outsources her data to the third party data storage provider (server), which is supposed to keep data intact and make it available to her. The problem is that the server may be malicious, and even if the server is trustworthy, hardware/software failures may cause data corruption. The client should be able to efficiently and securely check the integrity of her data without downloading the entire data from the server [2].

One such model proposed by Ateniese et al. is *Provable Data Possession* (PDP) [2] for provable data integrity. In this model, the client can challenge the server on random blocks and verify the data integrity through a proof sent by the server. PDP and related static schemes [2], [3], [14], [21], [30] show poor performance for blockwise update operations (insertion, removal, modification). While the static scenario can be applicable to some systems (e.g., archival storage at the libraries), for many applications it is important to take into consideration the dynamic scenario, where the client keeps interacting with the outsourced data in a read/write manner, while maintaining the data possession guarantees. Ateniese et al. [4] proposed Scalable PDP, which overcomes this problem with some limitations (only a pre-determined number of operations are possible within a limited set of operations). Erway et al. [15] proposed a solution called *Dynamic Provable Data Possession* (DPDP), which extends the PDP model and provides a dynamic storage scheme. Implementation of the DPDP scheme requires an underlying authenticated data structure based on a skip list [29].

Authenticated skip lists were presented by Goodrich and Tamassia [19], where skip lists and commutative hashing are employed in a data structure for authenticated dictionaries. A skip list is a key-value store whose leaves are sorted by keys. Each node stores a hash value calculated with the use of its own fields and the hash values of its neighboring nodes. The hash value of the root is the authentication information (meta data) that the client stores in order to verify responses from the server. To insert a new block into an authenticated skip list, one must decide on a key value for insertion since the skip list is sorted according to the key values. This is very useful if one, for example, inserts files into directories, since each file will have a unique name within the directory, and searching by this key is enough. However, when one considers blocks of a file to be inserted into a skip list, the blocks do not have unique names; they have indices. Unfortunately, in a dynamic scenario, an insertion/deletion would necessitate incrementing/decrementing the keys of all the blocks till the end of the file, resulting in degraded performance. DPDP [15] employs Rank-based Authenticated Skip List (RBASL) to overcome this limitation. Instead of providing *key* values in the process of insertion, the *index* value where the new block should be inserted is given. These indices are imaginary and no node stores any information about the indices. Thus, an insertion/deletion does not propagate to other blocks.

Theoretically, an RBASL provides dynamic updates with O($\log n$) complexity, assuming the updates are multiples of the fixed block size. Unfortunately, a variable size update leads to the propagation of changes to other blocks, making RBASL inefficient in practice. Therefore, one variable size update may affect $O(n)$ other blocks. We discuss the problem in detail in Section III. We propose FlexList to overcome the problem in DPDP. With our FlexList, we use the same idea but instead of the indices of blocks, indices of bytes of data are used, enabling searching, inserting, removing, modifying, or challenging a specific block containing the byte at a specific index of data. Since in practice a data alteration occurs starting from an index of the file, not necessarily an index of a block of the file, our DPDP with FlexList (FlexDPDP) performs much faster than the original DPDP with RBASL. Even though Erway et al. [15] presents the idea where the client makes updates on a range of bytes instead of blocks, we show that a naive implementation of the idea leads to a security gap in the storage system, as we discuss in Section VI-A. Our optimizations result in a dynamic cloud storage system whose efficiency is comparable to the best known static systems, and its security directly follows from the DPDP security proof and the security of authenticated skip lists.

Our contributions are as follows:

- Our implementation uses the *optimal* number of links and nodes; we created optimized algorithms for basic operations (i.e., insertion, deletion). These optimizations are applicable to all skip list types (skip list, authenticated skip list, rank-based authenticated skip list, and FlexList).
- Our FlexList translates a variable-sized update to $O(u)$ insertions, removals, or modifications, where u is the size of the update divided by the block size, while an RBASL requires $O(n)$ block updates.
- We provide multi-prove and multi-verify capabilities in cases where the client challenges the server for multiple blocks using authenticated skip lists, rank-based authenticated skip lists and FlexLists. Our algorithms provide an *optimal* proof, without any repeated items. The experimental results show efficiency gains of 35%, 35%, 40% in terms of proof time, energy, and size, respectively.
- We provide a novel algorithm to build a FlexList from scratch in $O(n)$ time instead of $O(n \log n)$ (time for n insertions). Our algorithm assumes the original data is already sorted, which is the case when a FlexList is constructed on top of a file in secure cloud storage.
- We deployed our client-server implementation on PlanetLab. Our results demonstrate that FlexDPDP performs comparable to the most efficient static storage scheme (PDP), while providing dynamic data support.

II. Related Work

	Hash Map (whole file)	Hash Map (block by block)	PDP[2]	Merkle Tree[33]	Balanced Tree (2-3 Tree)[35]	RBASL[15]	FlexList
Storage (client)	$O(1)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Proof Complexity (time and size)	$O(n)$	$O(1)$	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Dynamic (insert, remove, modify)	-	-	-	-	+ (balancing issues)	+ (fixed block size)	+

TABLE I
COMPLEXITY AND CAPABILITY TABLE OF VARIOUS DATA STRUCTURES FOR PROVABLE CLOUD STORAGE. N: NUMBER OF BLOCKS

Skip Lists and Other Data Structures: Table I provides an overview of different data structures proposed for the secure cloud storage setting. Among the structures that enable dynamic operations, the advantage of skip list is that it keeps itself balanced probabilistically, without the need for complex operations [29]. It offers search, modify, insert, and remove operations with *logarithmic* complexity with high probability [28]. Skip lists have been extensively studied [1], [5], [12], [15], [20], [22], [27]. They are used as authenticated data structures in two-party protocols [25], in outsourced network storage [20], with authenticated relational tables for database management systems [5], in timestamping systems [6], [7], in outsourced data storages [15], [18], and for authenticating queries for distributed data of web services [27].

In a skip list, not every edge or node is used during a search or update operation; therefore those unnecessary edges and nodes can be omitted. Similar optimizations for authenticated skip lists were tested in [32]. Furthermore, as observed in DPDP [15] for an RBASL, some corner nodes can be eliminated to decrease the overall number of nodes. Our FlexList contains all these optimizations, and many more, analyzed both formally and experimentally.

A binary tree-like data structure called rope is similar to our FlexList [8]. It was originally developed as alternative to the strings, bytes can be used instead of the strings as in our scheme. Since a rope is tree-like structure, it requires rebalancing operations. Moreover, a rope needs further structure optimizations to eliminate unnecessary nodes.

Cloud Storage Related Work: PDP was one of the first proposals for provable cloud storage [2]. PDP does not employ a data structure for the authentication of blocks, and is applicable to only static storage. A later variant called Scalable PDP [4] allows a limited number of updates. Wang et al. [33] proposed the usage of Merkle tree [24] which works perfectly for the static scenario, but has balancing problems in a dynamic setting. For the dynamic case we would need an authenticated balanced tree such as the data structure proposed by Zheng and Xu [35], called range-based 2-3 tree. Yet, there is no algorithm that has been presented for rebalancing either a Merkle tree or a range-based 2-3 tree while efficient updating and maintaining authentication information. Nevertheless, such algorithms have been studied in detail for the authenticated skip list [25]. Table I summarizes this comparison.

For dynamic provable data possession (DPDP) in a cloud storage setting, Erway et al. [15] were the first to introduce the new data structure *rank-based authenticated skip list (RBASL)* which is a special type of the authenticated skip list [20]. In the DPDP model, there is a client who wants to outsource her file and a server that takes the responsibility for the storage of the file. The client pre-processes the file and maintains meta data to verify the proofs from the server. Then she sends the file to the server. When the client needs to check whether her data is intact or not, she challenges some random blocks. Upon receipt of the request, the server generates the proof for the challenges and sends it back. The client then verifies the data integrity of the file using this proof. Many other static and dynamic schemes have been proposed [21], [30], [14], [11] including multi-server optimizations on them [9], [13], [16].

An RBASL, unlike an authenticated skip list, allows a search with indices of the blocks. This gives the opportunity to efficiently check the data integrity using block indices as proof and update query parameters in DPDP. To employ indices of the blocks as search keys, Erway et al. proposed using authenticated ranks. Each node in the RBASL has a *rank*, indicating the number of the leaf-level nodes that are reachable from that particular node. Leaf-level nodes having no $after$ links have a rank of 1, meaning they can be used to reach themselves only. Ranks in an RBASL handle the problem with block numbers in PDP [2], and thus result in a dynamic system.

Nevertheless, in a realistic scenario, the client may wish to change a part of a block, not the whole block. This can be problematic to handle in an RBASL. To partially modify a particular block in an RBASL, we not only modify a specified block but also may have to change all following blocks. This means the number of modifications is $O(n)$ in the worst case scenario for DPDP as well.

Another dynamic provable data possession scheme was presented by Zhang et al. [34]. They employ a new data structure called a balanced update tree, whose size grows with the number of the updates performed on the data blocks. Due to this property, they require extra rebalancing operations. The scheme uses message authentication codes (MAC) to protect the data integrity. Unfortunately, since the MAC values contain indices of data blocks, they need to be recalculated with insertions or deletions. The data integrity checking can also be costly, since the server needs to send all the challenged blocks with their MAC values, because the MAC scheme is not homomorphic (see [3]). In our scheme we send only tags and a block sum, which is approximately of a single block size. At the client side, there is an overhead for keeping the update tree.

Our proposed data structure FlexList, based on an authenticated skip list, performs dynamic operations (modify, insert, remove) for cloud data storage, having efficient variable block size updates.

III. Definitions

Skip List is a probabilistic data structure presented as an alternative to balanced trees [29]. It is easy to implement without complex balancing and restructuring operations such as those in AVL or Red-Black trees [1], [17]. A skip list keeps its nodes ordered by their *key* values. We call a leaf-level node and all nodes directly above it at the same index a $tower$.

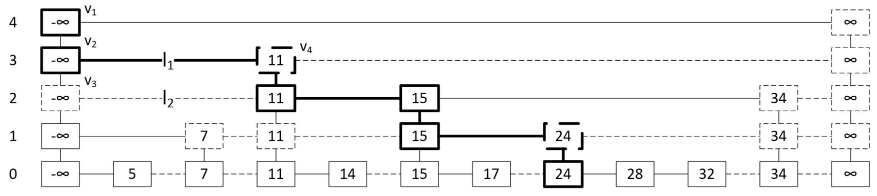


Fig. 1. Regular skip list with search path of node with key 24 highlighted. Numbers on the left represent levels. Numbers inside nodes are key values. Dashed lines indicate unnecessary links and nodes.

Figure 1 demonstrates a search on a skip list. The search path for the node with key 24 is highlighted. In a basic skip list, the nodes include *key*, *level*, and data (only at leaf level nodes) information, and $below$ and $after$ links (e.g., $v_2.below$ = v_3 and $v_2.after$ = v_4). To perform the search for 24, we start from the root (v_1) and follow the link to v_2, since v_1's $after$ link leads it to a node which has a greater key value than the key we are searching for ($\infty > 24$). Then, from v_2 we follow link l_1 to v_4, since the key value of v_4 is smaller than (or equal to) the searched key. In general, if the key of the node where $after$ link leads is smaller or equal to the key of the searched node, we follow that link, otherwise we follow the $below$ link. Using the same decision mechanism, we follow the highlighted links until the searched node is found at the leaf level (if it does not exist, then the node with key immediately before the searched node is returned).

We observe that some of the links are never used in the skip list, such as l_2, since any search operation with key greater or equal to 11 will definitely follow link l_1, and a search for a smaller key would never advance through l_2.

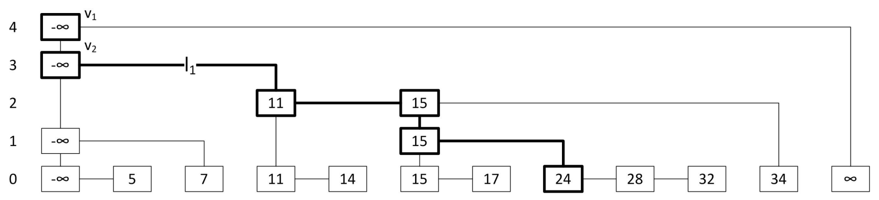


Fig. 2. Skip list of Figure 1 without unnecessary links and nodes.

Thus, we say links that are not present on any search path, such as l_2, are unnecessary. When we remove unnecessary links, we observe that some nodes, which are left without $after$ links (e.g., v_3), are also unnecessary since they do not provide any new dependencies in the skip list. Although it does not change the asymptotic complexity, it is beneficial not to include them for time and space efficiency. An optimized version of the skip list from Figure 1 can be seen in Figure 2 with the same search path highlighted. Formally:

- A **link** is **necessary** if and only if it is on any search path.
- A **node** is **necessary** if and only if it is at the leaf level or has a necessary $after$ link.

Assuming existence of a collision-resistant hash function family H, we randomly pick a hash function h from H and let || denote concatenation. Throughout our study we will use: $hash(x_1, x_2, ..., x_m)$ to mean $H(x_1||x_2||...||x_m)$.

An **authenticated skip list** is constructed with the use of a collision-resistant hash function and keeps a hash value in each node. Nodes at level 0 keep links to file blocks (may link to different structures e.g., files, directories, anything to be kept intact) [20]. A hash value is calculated with the following inputs: *level* and *key* of the node, and the hash values of the node $after$ and the node $below$. Through the inputs to the hash function, all nodes are dependent on their *after* and *below* neighbors. Thus, the root node is dependent on every leaf node, and due to the collision resistance of the hash function, knowing the hash value of the root is sufficient for later integrity checking. Note that if there is no node $below$, data or a function of data (which we will call *tag* in the following sections) is used instead of the hash of the $below$ neighbor. If there is no $after$ neighbor, then a dummy value (e.g., null) is used in the hash calculation.

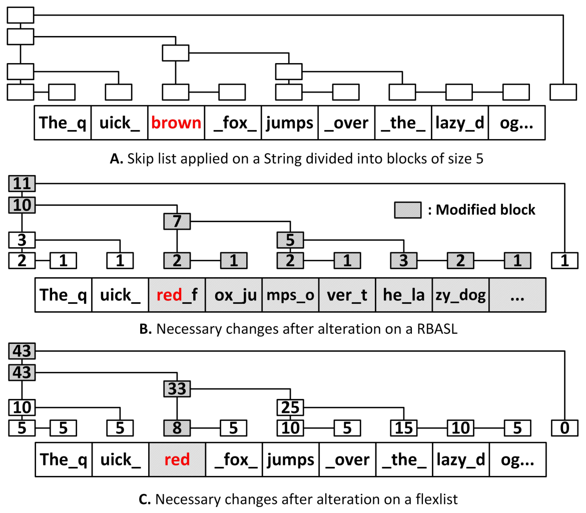


Fig. 3. Skip list alterations depending on an update request.

A **rank-based authenticated skip list (RBASL)** is different from an authenticated skip list by means of how it indexes data [15]. An RBASL has *rank* information (used in hashing instead of the *key* value), meaning how many nodes are reachable from that node. An RBASL is capable of performing all operations that an authenticated skip list can in the cloud storage context.

IV. FlexList

A FlexList supports variable-sized blocks whereas an RBASL is meant to be used with fixed block size since a search (consequently insert, remove, modify) by index of data is not possible with the rank information of an RBASL. For example, Figure 3-A represents an outsourced file divided into blocks of fixed size.

In our example, the client wants to change “brown” in the file composed of the text “The quick brown fox jumps over the lazy dog...” with “red” and the diff algorithm returns [delete from index 11 to 15] and [insert “red” from index 11 to 13]. Apparently, a modification to the 3^{rd} block will occur. With a rank-based skip list, to continue functioning properly, a series of updates is required as shown in Figure 3-B which asymptotically corresponds to $O(n)$ alterations. Otherwise, the beginning and the ending indices of each block will be complicated to compute, requiring $O(n)$ time to translate a diff algorithm output to block modifications at the server side. It also leaves the client unable to verify that the index she challenged is the same as the index of the proof by the server (this issue is explained in Section V-B with the *verifyMultiProof* algorithm). Therefore, for instance a FlexList having 500000 leaf-level nodes needs an expected 250000 update operations for a single variable-sized update. Besides the modify operations and related hash calculations,

Symbol	Description
cn	current node
pn	previous node, indicates the last node that current node moved from
mn	missing node, created when there is no node at the point where a node has to be linked
nn	new node
dn	node to be deleted
$after$	the after neighbor of a node
$below$	the below neighbor of a node
r	rank value of a node
i	index of a byte
npi	a boolean which is always true except in the inner loop of *insert* algorithm
$\sqcup_n$	stack (initially empty), filled with all visited nodes during $search$, $modify$, $insert$ or $remove$ algorithms

TABLE II
SYMBOL DESCRIPTIONS OF SKIP LIST ALGORITHMS.

this also corresponds to 250000 new tag calculations either on the server side, where the private key (order of the RSA group) is unknown (thus computation is very slow) or at the client side, where the new tags should go through the network. Furthermore, a verification process for the new blocks is also required (that means a huge proof, including half of the data structure used, sent by the server and the verified by the client, where she needs to compute an expected 375000 hash values). With our FlexList, only one modification suffices as indicated in Figure 3-C.

Due to the lack of providing variable block sized operations with an RBASL, we present FlexList which overcomes this problem and serves our purposes in the cloud data storage setting. A FlexList stores, at each node, how many *bytes* can be reached from that node, instead of how many *blocks* are reachable. The *rank* of each leaf-level node is computed as the sum of the *length* of its data and the *rank* of the $after$ node (0 if *null*). The *length* information of each data block is added as a parameter to the hash calculation of that particular block. We discuss the insecurity of an implementation that does not include the length information in the hash function calculation in Section VI-A. Note that when the length of data at each leaf is considered as a unit, the FlexList reduces to an RBASL (thus, ranks only count the number of reachable blocks). Therefore all our optimizations are also applicable to RBASL, which is indeed a special case of FlexList.

A. Preliminaries

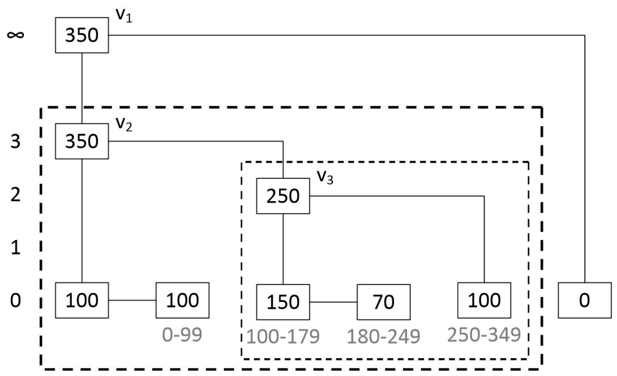


Fig. 4. A FlexList example with 2 sub skip lists indicated.

Algorithm IV.1: nextPos Algorithm

```
    Input: $pn$, $cn$, $i$, $level$, $npi$
    Output: $pn$, $cn$, $i$, $\sqcup_n$
1   $\sqcup_n$ = new empty Stack
2   while $cn$ can go $below$ OR $after$ do
3       if canGoBelow(cn, i) AND $cn.below.level \geq level$ AND
        $npi$ then
4           $cn = cn.below$
5       else if canGoAfter(cn, i) AND $cn.after.level \geq level$
        then
6           $i = i - cn.below.r$; $cn = cn.after$
7       add $cn$ to $\sqcup_n$
```

In this section, we introduce the helper methods required to traverse the skip list, create missing nodes, delete unnecessary nodes, delete nodes, and decide on the level to insert at, to be used in the essential algorithms (*search, modify, insert, remove*). Note that all algorithms are designed to fill a stack $\sqcup_n$ where we store nodes which may need a recalculation of hash values if authenticated, and rank values if using FlexList. All algorithms that move the current node immediately push the new current node to the stack $\sqcup_n$ as well. Further notations are shown in Table II.

We first define a concept called **sub skip list** to make our FlexList algorithms easier to understand. An example is illustrated in Figure 4. Let the search index be 250 and the current node start at the root (v_1). The current node follows its $below$ link to v_2 and enters a sub skip list (big dashed rectangle). Now, v_2 is the root of this sub skip list and the searched node is still at index 250. In order to reach the searched node, the current node moves to v_3, which is the root of another sub skip list (small dashed rectangle). Now, the searched byte is at index 150 in this sub skip list. Therefore the searched index is updated accordingly. The amount to be reduced from the search index is equal to the difference between the rank values of v_2 and v_3, which is equal to the rank of $below$ of v_2. Whenever the current node follows an $after$ link, the search index should be updated. To finish the search, the current node follows the $after$ link of v_3 to reach the node containing index 150 in the sub skip list with root v_3.

nextPos (Algorithm IV.1): The $nextPos$ method moves the current node cn repetitively until the desired position according to the method ($search$, $insert$, $remove$) from which it is called. There are 4 cases for $nextPos$:

- $insert$ - moves current node cn until the closest node to the insertion point.
- $remove$ or $search$ - moves current node cn until it finds the searched node's tower.
- loop in $insert$ - moves cn until it finds the next insertion point for a new node.
- loop in $remove$ - moves current node cn until it encounters the next node to delete.

Algorithm IV.2: createMissingNode Algorithm

```
Input: $pn$, $cn$, $i$, $level$
Output: $pn$, $cn$, $i$, $\sqcup_n$
1 $\sqcup_n$ = new empty Stack
2 $mn$ = new node is created using $level$ //Note that rank value for missing node is given $\infty$
3 if $canGoBelow(cn,i)$ then
4     $mn.below$ = $cn.below$; $cn.below$ = $mn$
5 else
6     $mn.below$ = $cn.after$; $cn.after$ = $mn$
7     $i$ = $i$ - $cn.below.r$ //Since current node is going $after$, $i$ value should be updated
8 $pn$ = $cn$; $cn$ = $mn$; then $cn$ is added to $\sqcup_n$
```

createMissingNode (Algorithm IV.2) is used in both the *insert* and *remove* algorithms. Since in a FlexList there are only necessary nodes, when a new node needs to be connected, this algorithm creates any missing node to make the connection.

deleteUNode (Algorithm IV.3) is employed in the *remove* and *insert* algorithms to delete an unnecessary node (this occurs when a node loses its *after* node) and maintain the links. It takes the previous node and current node as inputs, where the current node is unnecessary and meant to be deleted. The purpose is to preserve connections between necessary nodes after the removal of the unnecessary one. This involves deletion of the current node if it is not at the leaf level. It sets the previous node's $after$ or $below$ to the current node's $below$. As the last operation of deletion, we remove the top node from the stack $\sqcup_n$, as its rank and hash values no longer need to be updated.

Algorithm IV.3: deleteUNode Algorithm

```
Input: $pn$, $cn$
Output: $pn$, $cn$, $\sqcup_n$
1 $\sqcup_n$ = new empty Stack
2 if $cn.level$ == $0$ then
3     $cn.after$ = NIL
4 else
5     if $pn.below$ == $cn$ then
6         $pn.below$ = $cn.below$
7     else
8         $pn.after$ = $cn.below$
9     $\sqcup_n$.pop(); $cn$ = $pn$
```

deleteNode method, employed in the *remove* algorithm, takes two consecutive nodes, the previous node and the current node. By setting $after$ pointer of the previous node to current node's $after$, it detaches the current node from the FlexList.

tossCoins: Probabilistically determines the level value for a new node tower. A coin is tossed until it comes up heads. The output is the number of consecutive tails.

B. Methods of FlexList

FlexList is a particular way of organizing data for secure cloud storage systems. Some basic functions must be available, such as search, modify, insert and remove. These functions are employed in the verifiable updates. All algorithms are designed to fill a stack for the possibly affected nodes. This stack is used to recalculate of rank and hash values accordingly. A search path, which is the basic idea of a proof path, is visible in the stack in the basic algorithms.

search (Algorithm IV.4) is the algorithm used to find a particular byte. It takes the index i as the input, and outputs the node at index i with the stack $\sqcup_n$ filled with the nodes on the search path. Any value between 0 and the file size in bytes is valid to be searched. It is not possible for a valid index not to be found in a FlexList.

In algorithm IV.4, the current node cn starts at the root. The $nextPos$ method moves cn to the position just before the top of the tower of the searched node. Then cn is taken to the searched node's tower and moved all the way down to the leaf level.

modify: By taking index *i* and new data, we make use of the *search* algorithm for the node, which includes the byte at index *i*, and update its data. It then we recalculate hash values along the search path. The input of this algorithm contains the index i and new data. The outputs are the modified node and stack $\sqcup_n$ filled with nodes on the search path.

Algorithm IV.4: search Algorithm

```
Input: $i$
Output: $cn$, $\sqcup_n$
1  $\sqcup_n$ = new empty Stack
2  $cn = root$
   // $cn$ moves until $cn.after$ is a tower node of
   the searched node
3  call $nextPos$
4  $cn$ = $cn.after$ then $cn$ is added to $\sqcup_n$
   // $cn$ is moved below until the node at the
   leaf level, which has data
5  while $cn.level \neq 0$ do
6      $cn$ = $cn.below$ then $cn$ is added to $\sqcup_n$
```

insert (Algorithm IV.5) is run to add a new node to the FlexList with a random level by adding new nodes along the insertion path. The inputs are the index i and data. The algorithm generates a random *level* by tossing coins, then creates the new node with given data and attaches it to index i, along with the necessary nodes until the *level*. Note that this index should be the beginning index of an existing node, since inserting a new block inside a block makes no sense.[1] As output, the algorithm returns the stack $\sqcup_n$ filled with nodes on the search path of the new block.

Algorithm IV.5: insert Algorithm

```
Input: $i$, data
Output: $nn$,$\sqcup_n$
1  $\sqcup_n$ = new empty Stack
2  $pn = root$; $cn = root$; $level = tossCoins()$
3  call $nextPos$// $cn$ moves until it finds a missing node or $cn.after$ is where $nn$ is to be inserted
   // Check if there is a node where new node will be linked. if not, create one.
4  if !$CanGoBelow(cn, i)$ or $cn.level \neq level$ then
5      call $createMissingNode$;
   // Create new node and insert after the current node.
6  $nn$ = new node is created using $level$
7  $nn.after$ = $cn.after$; $cn.after = nn$ and $nn$ is added to $\sqcup_n$
   // Create insertion tower until the leaf level is reached.
8  while $cn.below \neq null$ do
9      if $nn$ already has a non-empty after link then
10         a new node is created to the $below$ of $nn$; $nn = nn.below$ and $nn$ is added to $\sqcup_n$
11     call $nextPos$ // Current node moves until we reach an $after$ link that passes through the tower.
       That is the insertion point for the new node.
       // Create next node of the insertion tower.
12     $nn.after$ = $cn.after$; $nn.level$ = $cn.level$
       // $cn$ becomes unnecessary as it looses its $after$ link, therefore it is deleted
13     $deteletUNode(pn, cn)$;
   // Done inserting, put data and return this last node.
14 $nn.data = data$
   // For a FlexList, call $calculateHash$ and $calculateRank$ on the nodes in the $\sqcup_n$ to compute their
   (possibly) updated values.
```

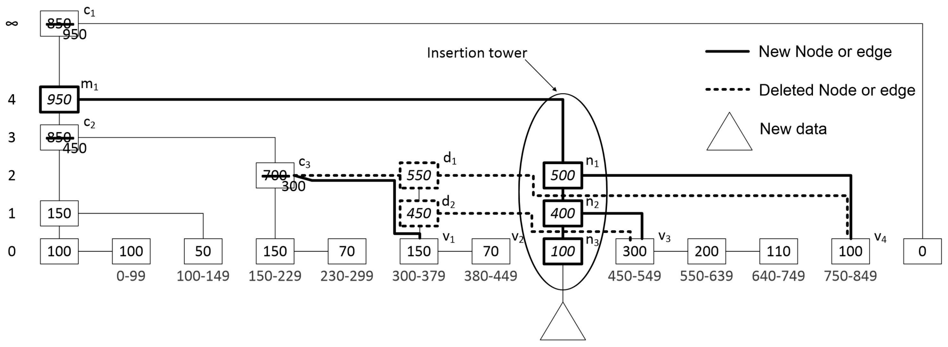


Fig. 5. Insert at index 450, level 4 (FlexList).

Figure 5 demonstrates the insertion of a new node at index 450 with $level$ 4. $nextPos$ brings the current node to the closest node to the insertion point with level greater than or equal to the insertion level (c_1 in Figure 5). Lines 3-4 create any missing node at the $level$, if there was no node to connect the new node to (e.g., m_1 is created to connect n_1 to). Within the while loop, during the first iteration, n_1 is inserted to level 2 since nodes at levels 3 and 4 are unnecessary in the insertion tower. Inserting n_1 makes d_1 unnecessary, since n_1 stole its after link. Likewise, the next iteration results in n_2 being inserted at level 1 and d_2 being removed. Note that removal of d_1 and d_2 results in c_3 getting connected to v_1. The last iteration inserts n_3, and places data. Since this is a FlexList, hashes and ranks of all the nodes in the stack will be recalculated ($c_1, m_1, n_1, c_2, c_3, n_2, n_3, v_1, v_2$). Those are the only nodes whose hash and rank values might have changed.

remove (Algorithm IV.6) is run to remove the node which starts with the byte at index i. As input, it takes the index i. The algorithm detaches the node to be removed and all other nodes above it while preserving connections between

[1]In case of an addition inside a block we can do the following: search for the block including the byte where the insertion will take place, add our data in between the first and second part of data found to obtain new data and employ *modify* algorithm (if new data is long, we can divide it into parts and send it as one modify and a series of inserts).

the remaining nodes. As output, the algorithm returns the stack $\sqcup_n$ filled with the nodes on the search path of the left neighbor of the node removed.

Figure 6 demonstrates removal of the node having the byte with index 450. The algorithm starts at the root c_1, and the first $nextPos$ call on line 2 returns d_1. Lines 4-7 check if d_1 is necessary. If d_1 is necessary, d_2 is deleted and we continue deleting from d_3. Otherwise, if d_1 is unnecessary, then d_1 is deleted, and we continue searching from c_1. In our example, d_1 is unnecessary, so we continue from c_1 to delete d_2. Within the while loop, the first call of $nextPos$ brings the current node to c_3. The goal is to delete d_2, but this requires creating of a missing necessary node m_1. Note that, m_1 is created at the same level as d_2. Once m_1 is created and d_2 is deleted, the while loop continues its next iteration starting from m_1 to delete d_3. This next iteration creates m_2 and deletes d_3. The last iteration moves the current node to v_2 and deletes d_4 without creating any new nodes, since we are at the leaf level. The output stack contains nodes $(c_1, c_2, c_3, m_1, m_2, v_1, v_2)$. Rank and hash values of those nodes could have changed, those values will be recalculated.

Algorithm IV.6: remove Algorithm

```
  Input: $i$
  Output: $dn$,$\sqcup_n$
1   $\sqcup_n$ = new empty Stack $pn = root$; $cn = root$
2   call $nextPos$ // Current node moves until $after$ of the current node is the node at the top of deletion tower
3   $dn$ = $cn.after$
    // Check if current node is necessary,if so it can steal $after$ of the node to delete, otherwise delete current node
4   if $cn.level$ = $dn.level$ then
5       $deleteNode(cn, dn)$; $dn$ = $dn.below$; // unless at leaf level
6   else
7       $deleteUNode(pn, cn)$;
    // Delete whole deletion tower until the leaf level is reached
8   while $cn.below \neq null$ do
9       call $nextPos$// Current node moves until it finds a missing node
        // Create the missing node unless at leaf level and steal the $after$ link of the node to delete
10      call $createMissingNode$; $deleteNode(cn, dn)$
11      $dn$ = $dn.below$ // move $dn$ to the next node in the deletion tower unless at leaf level
    // For a FlexList, call $calculateHash$ and $calculateRank$ on the nodes in the $\sqcup_n$ to compute their (possibly) updated values.
```

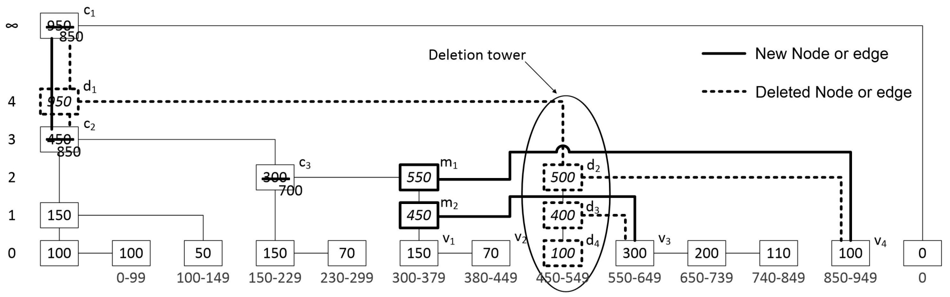


Fig. 6. Remove block at index 450(FlexList).

C. Novel Build from Scratch Algorithm

Algorithm IV.7: buildFlexList Algorithm

```
  Input: $B$, $L$, $T$
  Output: $root$
    // $H$ will keep pointers to tower heads
1   $H$ = new vector is created of size $L_0$ + 1
    // Loop will iterate for each block
2   for $i = B.size - 1$ to $0$ do
3       $pn$= $null$
4       for $j = 0$ to $L_i$+$1$ do
            // Enter only if at level 0 or $H_j$ has an element
5           if $H_j \neq null$ or $j = 0$ then
6               $nn$ = new node is created with level $j$ //if $j$ is 0, $B_i$,$T_i$ are included to the creation of $nn$
7               $nn.below$ = $pn$; $nn.after$ = $H_j$ // Connect tower head at $H_j$ as an $after$ link
8               call $calculateRank$ and $calculateHash$ on $nn$
9               $pn$ = $nn$; $H_j = null$
        // Add a tower head to $H$ at $H_{L_i}$
10      $H_{L_i}$ = $pn$
11  $root = H_{L_0}$ //which is equal to $pn$
12  $root.level = \infty$; call $calculateHash$ on root
13  return $root$
```

The usual way to build a skip list (or FlexList) is to perform n insertions (one for each item). When original data is already sorted, one may insert them in increasing or decreasing order. Such an approach will result in $O(n \log n)$ total time complexity. But, when data is sorted as in the secure cloud storage scenario (where blocks of a file are

already sorted), a much more efficient algorithm can be developed. Observe that a skip list contains $2n$ nodes in total, in expectation [29]. This is an $O(n)$ value, and thus spending $O(n \log n)$ time for creating $O(n)$ nodes is an overkill, since creation of nodes take a constant time only. We present our novel algorithm for building a FlexList from scratch in just $O(n)$ time. To the best of our knowledge, such an efficient build algorithm did not exist before.
buildFlexList (Algorithm IV.7) is an algorithm that generates a FlexList over a set of sorted data in time complexity $O(n)$. It has the small space complexity of $O(l)$ where l is number of levels in the FlexList ($l = O(\log n)$ with high probability). As the inputs, the algorithm takes blocks B on which the FlexList will be generated, corresponding (randomly generated) levels L and tags T. The algorithm assumes data is already sorted. In cloud storage, the blocks of a file are already sorted according to their block indices, and thus our optimized algorithm perfectly fits our target scenario. The algorithm attaches one link for each tower from right to left. For each leaf node generated, its tower follows in a bottom up manner. As output, the algorithm returns the root node.

Figure 7 demonstrates the building process of a FlexList where the insertion levels of blocks are 4, 0, 1, 3, 0, 2, 0, 1, 4, in order. Labels v_i on the nodes indicate the generation order of the nodes. Note that the blocks and the tags for the sentinel nodes are null values. The idea of the algorithm is to build towers of a given level for each block. As shown in the figure, all towers have only one link from left side to its tower head (the highest node in the tower). Therefore, we need to store the tower heads in a vector, and then make necessary connections. The algorithm starts with the creation of the vector H to hold pointers to the tower heads at line 1. At lines 6-9 for the first iteration of the inner loop, the node v_1 is created which is a leaf node, thus there is no node below. Currently, H is empty; therefore there is no node at H_0 to connect to v_1 at level 0. The hash and the rank values of v_1 are calculated. Since H is still empty, we do not create new nodes at levels 1, 2, 3, 4. At line 10, we put v_1 to H as H_4. The algorithm continues with the next block and the creation of v_2. H_0 is still empty, therefore no $after$ link for v_2 is set. The hash and the rank values of v_2 are calculated. The next iterations of the inner loop skip the lines 6-9, because H_1 and H_2 are empty as well. At line 10, v_2 is inserted to H_2. Then, v_3 is created and its hash and rank values are calculated. There is no element at H_0 to connect to v_3. Its level is 0, therefore it is added to H as H_0. Next, we create the node v_4; it takes H_0 as its $after$. The hash and the rank values are calculated, then v_4 is added to H at index 0. The algorithm continues for all elements in the block vector. At the end of the algorithm, the root is created, connected to the top of the FlexList, then its hash and rank values are calculated.

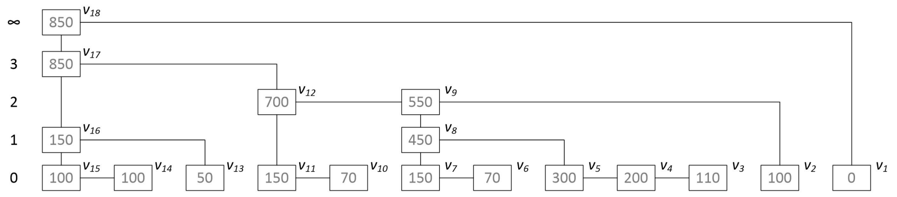


Fig. 7. buildFlexList example.

V. FlexList-based Dynamic Secure Cloud Storage

In this section, we describe the application of our FlexList to integrity checking in secure cloud storage systems according to the DPDP model [15]. The DPDP model has two main parties: the client and the server. The cloud server stores a file on behalf of the client. Erway et al. showed that an RBASL can be created on top of the outsourced file to provide proofs of integrity (see Figure 8). The following are the algorithms used in the DPDP model for secure cloud storage [15]:

- $Challenge$ is a probabilistic function run by the client to request a proof of integrity for randomly selected blocks.
- $Prove$ is run by the server in response to a challenge to send the proof of possession.
- $Verify$ is a function run by the client upon receipt of the proof. A return value of accept ideally means the file is kept intact by the server.
- $prepareUpdate$ is a function run by the client when she changes some part of her data. She sends the update information to the server.
- $performUpdate$ is run by the server in response to an update request to perform the update *and* prove that the update performed reliably.
- $verifyUpdate$ is run by the client upon receipt of the proof of the update. Returns accept (and updates her meta data) if the update was performed reliably.

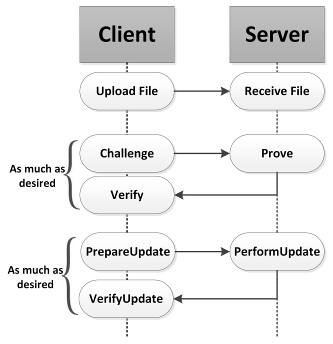


Fig. 8. Client Server interactions in FlexDPDP.

We construct the above model with FlexList as the authenticated data structure. We provide new capabilities and efficiency gains as discussed in Section IV and call the resulting scheme **FlexDPDP**. In this section, we describe our corresponding algorithms for each step in the DPDP model.

The FlexDPDP scheme uses *homomorphic verifiable tags* (as in DPDP [15]); multiple tags can be combined to obtain a single tag that corresponds to combined blocks [3]. Tags are small compared to data blocks, enabling storage in memory. Authenticity of the skip list guarantees integrity of tags, and tags protect the integrity of the data blocks.

A. Preliminaries

Before providing optimized proof generation and verification algorithms, we introduce essential methods to be used in our algorithms to determine intersection nodes, search multiple nodes, and update rank values. Table III shows additional notation used in this section.

Symbol	Description
hash	hash value of a node
rs	rank state, indicates the byte count to the left of current node and used to recover i value when roll-back to a state is done
$state$	state, created in order to store from which node the algorithm will continue, contains a node, rank state, and last index
C	challenged indices vector, in ascending order
V	verify challenge vector, reconstructed during verification to check if the proof belongs to challenged blocks, in terms of indices
p	proof node
P	proof vector, stores proof nodes for all challenged blocks
T	tag vector of challenged blocks
M	block sum
$\sqcup_s$	intersection stack, stores states at intersections in $searchMulti$ algorithm
$\sqcup_h$	intersection hash stack, stores hash values to be used at intersections
$\sqcup_i$	index stack, stores pairs of integer values, employed in $updateRankSum$
$\sqcup_l$	changed nodes' stack, stores nodes for later hash calculation, employed in $hashMulti$
$start$	start index in $\sqcup_i$ from which $updateRankSum$ should start
end	end index in $\sqcup_i$
$first$	current index in C
$last$	end index in $\sqcup_s$

TABLE III
SYMBOLS USED IN OUR ALGORITHMS.

isIntersection: This function is used when *searchMulti* checks if a given node is an intersection. A node is an intersection point of proof paths of two indices when the first index can be found following the $below$ link and the second index is found by following the $after$ link (the challenged indices will be in ascending order). There are two conditions for a node to be called an intersection node:

- The current node follows the $below$ link according to the index we are building the proof path for.
- The current node needs to follow the $after$ link to reach the element of challenged indices at index $last$ in the vector C.

If one of the above conditions is not satisfied, then there is no intersection, and the method returns false. Otherwise, it decrements $last$ and continues trying until it finds a node which cannot be found by following the $after$ link and returns $last'$ (to be used in the next call of *isIntersection*) and true (as the current node cn is an intersection point). Note that this method directly returns false if there is only one challenged index.

Algorithm V.1: searchMulti Algorithm

```
   Input: $cn$, $C$, $first$, $last$, $rs$, $P$, $\sqcup_s$
   Output: $cn$, $P$, $\sqcup_s$
   // Index of the challenged block (key) is calculated according to the current sub skip list root
1  $i = C_{first} - rs$
   // Create and put proof nodes on the search path of the challenged block to the proof vector
2  while Until challenged node is included do
3      $p$ = new proof node with $cn.level$ and $cn.r$
       // End of this branch of the proof path is when the current node reaches the challenged node
4      if $cn.level = 0$ and $i < cn.length$ then
5          $p.setEndFlag()$; $p.length = cn.length$
       //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
6      if $isIntersection(cn, C, i, last_k, rs)$ then
           //note that $last_k$ becomes $last_{k+1}$ in $isIntersection$ method
7          $p.setInterFlag()$; $state(cn.after, last_k, rs+cn.below.r)$ is added to $\sqcup_s$ // Add a state for $cn.after$ to
           continue from there later
       // Missing fields of the proof node are set according to the link current node follows
8      if $(CanGoBelow(cn, i))$ then
9          $p$.hash = $cn.after$.hash; $p.rgtOrDwn = dwn$
10         $cn = cn.below$ //unless at the leaf level
11     else
12         $p$.hash = $cn.below$.hash; $p.rgtOrDwn = rgt$
           // Set index and rank state values according to how many bytes at leaf nodes are passed
           while following the $after$ link
13         $i$ -= $cn.below.r$; $rs$ += $cn.below.r$; $cn = cn.after$
14     $p$ is added to $P$
```

Proof node is the building block of a proof, used throughout this section. It contains level, data length (if level is 0), rank, hash, and three boolean values $rgtOrDwn$, end flag and intersection flag. Level and rank values belong to the node for which the proof node is generated. The hash is the hash value of the neighbor node, which is not on the proof path. There are two scenarios for setting hash and $rgtOrDwn$ values:

(1) When the current node follows $below$ link, we set the hash of the proof node to the hash of the current node's $after$ and its $rgtOrDwn$ value to dwn.

(2) When the current node follows $after$ link, we set the hash of the proof node to the hash of the current node's $below$ and its $rgtOrDwn$ value to rgt.

searchMulti (Algorithm V.1): This algorithm is used in *genMultiProof* to generate the proof path for multiple nodes without unnecessary repetitions of proof nodes. Figure 9, where we challenge the node at the index 450, clarifies how the algorithm works. Our aim is to provide the proof path for the challenged node. We assume that in the search, the current node cn starts at the root (w_1 in our example). Therefore, initially the search index i is 450, the rank state rs and $first$ are zero, the proof vector P and intersection stack $\sqcup_s$ are empty.

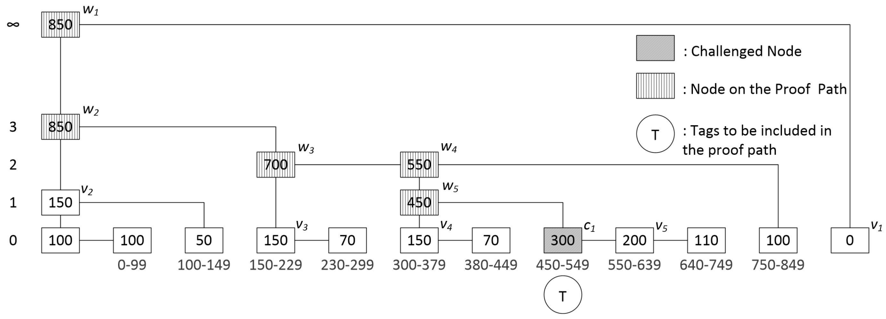


Fig. 9. Proof path for challenged index 450 in a FlexList.

For w_1, a proof node is generated using scenario (1), where p.hash is set to v_1.hash and $p.rgtOrDwn$ is set to dwn. For w_2, the proof node is created as described in scenario (2) above, where p.hash is set to v_2.hash and $p.rgtOrDwn$ is set to rgt. The proof node for w_3 is created using scenario (2). For w_4 and w_5, proof nodes are generated as in scenario (1). The last node c_1 is the challenged leaf node, and the proof node for this node is also created as in scenario (1). Note that in the second, third, and fifth iterations of the while loop, the current node is moved to a sub skip list (at line 13 in Algorithm V.1). Lines 4-5 (setting the end flag and collecting the data length) and 6-7 (setting intersection flag and saving the state) in Algorithm V.1 are crucial for generation of proof for multiple blocks. We discuss them later in this section.

updateRankSum: This algorithm, used in *verifyMultiProof*, is given the rank difference as input, the verify challenge vector V, and indices $start$ and end (on V). The output is a modified version of the verify challenge vector V'. The procedure is called when there is a transition from one sub skip list to another (larger one). The method updates entries starting from index $start$ to index end by rank difference, where rank difference is the size of the larger sub skip list minus the size of the smaller sub skip list.

Finally, tags and combined blocks will be used in our proofs. For this purpose, we use an RSA group Z_N^*, where N = pq is the product of two large prime numbers, and g is a high-order element in Z_N^* [15]. It is important that the server does not know p and q. The tag t of a block m is computed as $t = g^m \mod N$. The block sum is computed as $M = \sum_{i=0}^{|C|} a_i m_{C_i}$ where C is the challenge vector containing block indices and a_i is the random value for the i^{th} challenge.

B. Handling Multiple Challenges at Once

Client server interaction (Figure 8) starts with the client pre-processing her data (creating a FlexList for the file and calculating tags for each block of the file). The client sends the random seed she used for generating the FlexList to the server along with a public key, data, and the tags. Using the seed, the server constructs a FlexList over the blocks of data and assigns tags to leaf-level nodes. Note that the client may request the root value calculated by the server to verify that the server constructed the correct FlexList over the file. When the client checks and verifies that the hash of the root value is the same as the one she had calculated, she may safely remove her data and the FlexList. She keeps the root value as meta data for later use in the proof verification mechanism.

To challenge the server, the client generates two random seeds, one for a pseudo-random generator that will generate random indices for bytes to be challenged, and another for a pseudo-random generator that will generate random coefficients to be used in the block sum. The client sends these two seeds to the server as the challenge, and keeps them for verification of the server's response.

1) Proof Generation: **genMultiProof** (Algorithm V.2): Upon receipt of the random seeds from the client, the server generates the challenge vector C and random values A accordingly and runs the *genMultiProof* algorithm in order to get tags, file blocks, and the proof path for the challenged indices. The algorithm searches for the leaf node of each challenged index and stores all nodes across the search path in the proof vector. However, we have observed that regular searching for each particular node is inefficient. If we start from the root for each challenged block, there will be a lot of replicated proof nodes. In the example of Figure 9, if proofs were generated individually, w_1, w_2, and w_3 would be replicated 4 times, w_4 and w_5 3 times, and c_3 2 times. To overcome this problem we save states at each intersection node. In our *optimal* proof, only one proof node is generated for each node on any proof path. This is beneficial in terms of not only space but also time. The verification time of the client is greatly reduced since she computes less hash values.

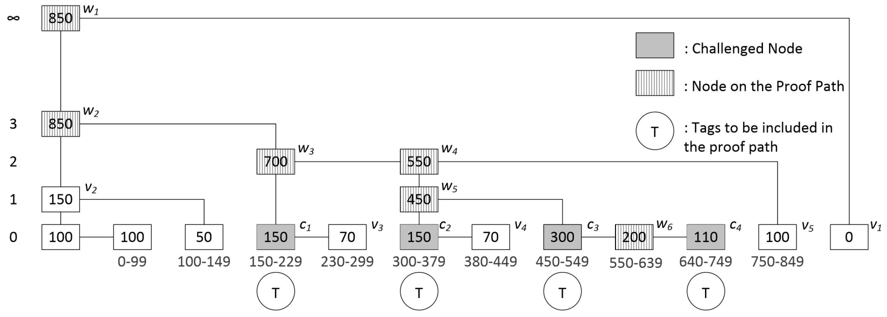
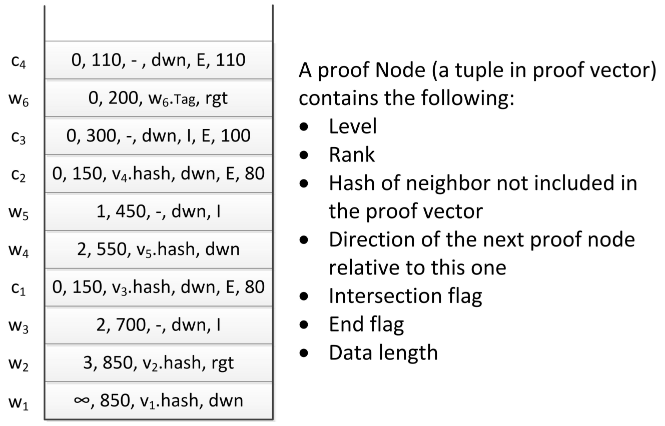


Fig. 10. Multiple blocks are challenged in a FlexList.

We explain *genMultiProof* (Algorithm V.2) using Figure 10 and notations in Table III. By taking the index array of challenged nodes as input (challenge vector C generated from the random seed sent by the client contains [170, 320, 470, 660] in the example), the *genMultiProof* algorithm generates the proof P, collects the tags into the tag vector T, calculates the block sum M at each step, and returns all three. The algorithm starts traversing from the root (w_1 in our example) by retrieving it from the intersection stack $\sqcup_s$ at line 3 of Algorithm V.2. Then, in the loop, we call *searchMulti*, which returns the proof nodes for w_1, w_2, w_3 and c_1. The state of node w_4 is saved in the stack $\sqcup_s$ as it is the $after$ of an intersection node, and the *intersection* flag for proof node for w_3 is set. Note that proof nodes at the intersection points store no hash value. The second iteration starts from w_4, which is the last saved state. New proof nodes for w_4, w_5 and c_2 are added to the proof vector P, while c_3 is added to the stack $\sqcup_s$. The third iteration starts from c_3 and *searchMulti* returns P, after adding c_3 to it. Note that w_6 is added to the stack $\sqcup_s$. In the last iteration, w_6 and c_4 are added to the proof vector P. As the stack $\sqcup_s$ is empty, the loop is over. Note that all proof nodes of the challenged indices have their *end* flags and length values set (line 5 of Algorithm V.1). When *genMultiProof* returns, the output proof vector should be as in Figure 11. At the end of the *genMultiProof* algorithm the proof and tag vectors and the block sum are sent to the client for verification.

Algorithm V.2: genMultiProof Algorithm

```
    Input: $C$, A
    Output: $T$, $M$, $P$
      Let C= $(i_0,\dots,i_k)$ where $i_j$ is the $(j+1)^{th}$ challenged index; $A=(a_0,\dots,a_k)$ where $a_j$ is the $(j+1)^{th}$
      random value; state$_m$ = $(node_m, lastIndex_m,\ rs_m)$
1     $cn$ = $root$; $rs$ = 0; $M$ = 0; $\sqcup_s$, $P$ and $T$ are empty; state(root, k, $rs$) added to $\sqcup_s$
      // Call $searchMulti$ method for each challenged block to fill the proof vector $P$
2     for $i=0$ to $k$ do
3         $state$ = $\sqcup_s$.pop()
4         $cn$ = $searchMulti$($state$.node,$C$,$i$,$state.end$,$state.rs$,$P$,$\sqcup_s$)
          // Store tag of the challenged block and compute the block sum
5         $cn.tag$ is added to $T$ and $M$ += $cn$.data*a$_i$
```



c4
0, 110, - , dwn, E, 110
w6
0, 200, w6.Tag, rgt
c3
0, 300, -, dwn, I, E, 100
c2
0, 150, v4.hash, dwn, E, 80
w5
1, 450, -, dwn, I
w4
2, 550, v5.hash, dwn
c1
0, 150, v3.hash, dwn, E, 80
w3
2, 700, -, dwn, I
w2
3, 850, v2.hash, rgt
w1
∞, 850, v1.hash, dwn
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. 11. Proof vector for Figure 10 example.

2) Verification: **verifyMultiProof** (Algorithm V.3): Remember that the client keeps random seeds used for the challenge. She generates the challenge vector C and random values A according to these seeds. If the server is honest, these will contain the same values as the ones the server generated. There are two steps in the verification process: tag verification and FlexList verification.

Tag verification is done as follows: Upon receipt of the tag vector T and the block sum M, the client calculates $tag = \prod_{i=0}^{|C|} T_i^{a_i} \mod N$ and accepts iff $tag = g^M \mod N$. By this, the client checks the integrity of file blocks by tags. Later, when tags are proven to be intact by FlexList verification, the file blocks will be verified. **FlexList verification** involves calculation of hashes for the proof vector P. The hash for each proof node can be calculated in different ways as described below using the example from Figure 10 and Figure 11.

The hash calculation always has the *level* and *rank* values stored in a proof node as its first two arguments.

- If a proof node is marked as *end* but *not intersection* (e.g., c$_4$, c$_2$, and c$_1$), this means the corresponding node was challenged (to be checked against the challenged indices later), and thus its tag must exist in the tag vector. We compute the corresponding hash value using that tag, the hash value stored in the proof node (null for c$_4$ since it has no $after$ neighbor, the hash value of v$_4$ for c$_2$, and the hash value of v$_3$ for c$_1$), and the corresponding length value (110 for c$_4$, 80 for c$_2$ and c$_1$).
- If a proof node is not marked and $rgtOrDwn = rgt$ or $level = 0$ (e.g., w$_6$, w$_2$), this means the $after$ neighbor of the node is included in the proof vector and the hash value of its $below$ is included in the associated proof node (if the node is at leaf level, the tag is included instead). Therefore we compute the corresponding hash value using the hash value stored in the corresponding proof node and the previously calculated hash value (hash of c$_4$ is used for w$_6$, hash of w$_3$ is used for w$_2$).
- If a proof node is marked as *intersection* and *end* (e.g., c$_3$), this means the corresponding node was both challenged (thus its tag must exist in the tag vector) and is on the proof path of another challenged node; therefore, its $after$ neighbor is also included in the proof vector. We compute the corresponding hash value using the corresponding tag from the tag vector and the previously calculated hash value (hash of w$_6$ for c$_3$).
- If a proof node is marked as *intersection* but *not end* (e.g., w$_5$ and w$_3$), this means the node was not challenged but both its $after$ and $below$ are included in the proof vector. Hence, we compute the corresponding hash value using the previously calculated two hash values (the hash values calculated for c$_2$ and for c$_3$, respectively, are used for w$_5$, and the hash values calculated for c$_1$ and for w$_4$, respectively, are used for w$_3$).

- If none of the above is satisfied, this means a proof node has only $rgtOrDwn = dwn$ (e.g., w_4 and w_1), meaning the $below$ neighbor of the node is included in the proof vector. Therefore we compute the corresponding hash value using the previously calculated hash value (hash of w_5 is used for w_4, and hash of w_2 is used for w_1) and the hash value stored in the corresponding proof node.

We treat the proof vector (Figure 11) as a stack and do necessary calculations as discussed above. The calculation of hashes is done in the reverse order of the proof generation in *genMultiProof* algorithm. Therefore, we perform the calculations in the following order: c_4, c_6, c_3, c_2, w_5, ... until the hash value for the root (the last element in the stack) is computed. Observe that to compute the hash value for w_5, the hash values for c_3 and c_2 are needed, and this reverse (top-down) ordering always satisfies these dependencies. Finally, we compute the corresponding hash values for w_2 and w_1. When the hash for the last proof node of the proof path is calculated, it is compared with the meta data that the client possesses (in line 22 of Algorithm V.3).

The check above makes sure that the nodes, whose proofs were sent, are indeed in the FlexList that correspond to the meta data stored at the client. But the client also has to make sure that the server indeed proved storage of data that she challenged. The server may have lost those blocks but may instead be proving storage of some other blocks at different indices. To prevent this, the verify challenge vector, which contains the start indices of the challenged nodes (150, 300, 450, and 460 in our example), is generated by the rank values included in the proof vector (in lines 5, 9, 10, 13, 14, and 18 of Algorithm V.3). With the start indices and the lengths of the challenged nodes given, we check if each challenged index is included in a node that the proof is generated for (as shown in line 22 of Algorithm V.3). For instance, we know that we challenged index 170, c_1 starts from 150 and is of length 80. We check if $0 \leq 170 - 150 < 80$. Such a check is performed for each challenged index and each proof node with an *end* mark.

Algorithm V.3: verifyMultiProof Algorithm

```
   Input: $C$, $P$, $T$, $MetaData$
   Output: accept or reject
     Let $P = (A_0, \ldots, A_k)$, where $A_j = (level_j,\ r_j,\ \text{hash}_j,\ rgtOrDwn_j,\ isInter_j,\ isEnd_j,\ length_j)$ for $j = 0, \ldots, k$; $T = (tag_0, \ldots, tag_n)$, where $tag_m =$ tag for challenged $block_m$ for $m = 0, \ldots, n$;
1    $start = n$; $end = n$; $t = n$; $V = 0$; hash = 0; $\text{hash}_{prev} = 0$; $startTemp = 0$; $\sqcup_h$ and $\sqcup_i$ are empty stacks
     // Process each proof node from the end to calculate hash of the root and indices of the challenged blocks
2    for $j = k$ to $0$ do
3        if $isEnd_j$ and $isInter_j$ then
4            hash = $hash(level_j, r_j, tag_t, \text{hash}_{prev}, length_j)$; decrement(t)
5            $updateRankSum(length_j, V, start, end)$; decrement($start$) // Update index values of challenged blocks on the leaf level of current part of the proof path
6        else if $isEnd_j$ then
7            if $t \neq n$ then
8                $\text{hash}_{prev}$ is added to $\sqcup_h$
9                $(start, end)$ is added to $\sqcup_i$
10               decrement($start$); $end = start$
11           hash = $hash(level_j, r_j, tag_t, \text{hash}_j, length_j)$; decrement(t)
12       else if $isInter_j$ then
13           $(startTemp, end) = \sqcup_i$.pop()
14           $updateRankSum(r_{prev}, V, startTemp, end)$ // Last stored indices of challenged block are updated to rank state of the current intersection
15           hash = $hash(level_j, r_j, \text{hash}_{prev}, \sqcup_h.pop())$
16       else if $rgtOrDwn_j = rgt$ or $level_j = 0$ then
17           hash = $hash(level_j, r_j, \text{hash}_j, \text{hash}_{prev})$
18           $updateRankSum(r_j - r_{prev}, V, start, end)$ // Update indices of challenged blocks, which are on the current part of the proof path
19       else
20           hash = $hash(level_j, r_j, \text{hash}_{prev}, \text{hash}_j)$
21       $\text{hash}_{prev}$ = hash; $r_{prev} = r_j$
     //$endnodes$ is a vector of proof nodes marked as End in the order of appearance in $P$
22   if $\forall a$, $0 \leq a \leq n$, $0 \leq C_a - V_a < endnodes_{n-a}.length$ OR $hash \neq MetaData$ then
23       return reject
24   return accept
```

C. Verifiable Variable-size Updates

The main purpose of the insert, remove, and modify operations (update operations) of our FlexList being employed in the cloud setting is that we want the update operations to be verifiable. The purpose of the following algorithms is to verify the update operation and compute new meta data to be stored at the client through the proof sent by the server.

1) Performing an Update: performUpdate is run at the server side upon receipt of an update request to the index i from the client. We consider it to have three parts: *proveModify, proveInsert, proveRemove*. The server runs *genMultiProof* algorithm to acquire a proof vector in a way that it covers the nodes which may get affected from the update. For a modify operation the modified index (*i*), for an insert operation the left neighbor of the insert position (*i-1*), and for a remove operation the left neighbor of the remove position and the node at the remove position (*i-1*, *i*) are to be used as challenged indices for *genMultiProof* Algorithm. Then the server performs the update operation as it is using the regular FlexList algorithms, and sends the new meta data to the client.

2) Verifying an Update: The algorithm *verifyUpdate* of the DPDP model, in our construction, not only updates her meta data but also verifies if it is correctly updated at the server by checking whether or not the calculated meta data and the received one are equal. It makes use of one of the following three algorithms due to the nature of the update, at the client side.

verifyModify (Algorithm V.4) is run at the client to approve the modification. The client alters the last element of the received proof vector and calculates temp meta data accordingly. Later she checks if the new meta data provided by the server is equal to the one that the client has calculated. If they are the same, then modification is accepted, otherwise rejected.

Algorithm V.4: verifyModify Algorithm

```
Input: $C$, $P$, $T$, $tag$, $data$, MetaData, $MetaData_{byServer}$
Output: accept or reject, $MetaData'$
Let C= $(i_0)$ where $i_0$ is the modified index; $P = (A_0, \ldots, A_k)$, where $A_j = ($ $level_j$, $r_j$, hash$_j$, $rgtOrDwn_j$,
$isInter_j$, $isEnd_j$, $length_j)$ for $j = 0, \ldots, k$; $T = (tag_0)$, where $tag_0$ is tag for $block_0$ before modification;
$P$, $T$ are the proof and tag before the modification; $tag$ and $data$ are the new tag and data of the
modified block
1  if !VerifyMultiProof(C, P, T, MetaData) then
2      return reject;
3  else
4      $i$ = size($P$) - 1
5      hash = $hash$(A$_i$.level, A$_i$.rank - A$_i$.length + $data$.length, $tag$, A$_i$.hash, $data$.length)
       // Calculate hash values until the root of the Flexlist
6      $MetaData_{new}$ = calculateRemainingHashes( i-1, hash, $data$.length - A$_i$.length, $P$)
7      if $MetaData_{byServer}$ = $MetaData_{new}$ then
8          $Metadata$ = $MetaData_{new}$
9          return accept
10     else
11         return reject
```

verifyInsert (Algorithm V.5) is run to verify the correct insertion of a new block to the FlexList, using the proof vector and the new meta data sent by the server. It calculates the temp meta data using the proof P as if the new node has been inserted in it. The inputs are the challenged block index, a proof, the tags, and the new block information. The output is accept if the temp root calculated is equal to the meta data sent by the server, otherwise reject.

Algorithm V.5: verifyInsert Algorithm

```
Input: $C$, $P$, $T$, $tag$, $data$, $level$, MetaData, $MetaData_{byServer}$
Output: accept or reject, $MetaData'$
Let C= $(i_0)$ where $i_0$ is the index of the left neighbor; $P = (A_0, \ldots, A_k)$, where $A_j = ($ $level_j$, $r_j$, hash$_j$,
$rgtOrDwn_j$, $isInter_j$, $isEnd_j$,$length_j)$ for $j = 0, \ldots, k$; $T = (tag_0)$ where $tag_0$ is for precedent node of
newly inserted node; $P$, $T$ are the proof and tag before the insertion; $tag$, $data$ and $level$ are the
new tag, data and level of the inserted block
1  if !VerifyMultiProof(C, P, T, MetaData) then
2      return reject;
3  else
4      i = size($P$) - 1; rank = A$_i$.length; rankTower = A$_i$.rank - A$_i$.length + $data$.length
5      hashTower = $hash$(0, rankTower, $tag$, A$_i$.hash, $data$.length)
6      if $level \neq 0$ then
7          hash = $hash$(0, A$_i$.length, $tag_0$, 0);
8      decrement(i)
9      while $A_i.level \neq level$ or ($A_i.level = level$ and $A_i.rgtOrDwn = dwn$) do
10         if $A_i.rgtOrDwn = rgt$ then
11             rank += A$_i$.rank - A$_{i+1}$.rank
               // A$_i$.length is added to hash calculation if A$_i$.level $= 0$
12             hash = $hash$(A$_i$.level, rank, A$_i$.hash, hash)
13         else
14             rankTower += A$_i$.rank - A$_{i+1}$.rank
15             hashTower = $hash$(A$_i$.level, rankTower, hashTower, A$_i$.hash)
16         decrement(i)
17     hash = $hash$($level$, rank + rankTower, hash, hashTower)
18     $MetaData_{new}$ = calculateRemainingHashes(i, hash, $data$.length, $P$)
19     if $MetaData_{byServer}$ = $MetaData_{new}$ then
20         MetaData = $MetaData_{new}$
21         return accept
22     return reject
```

The algorithm is explained using Figure 12 as an example where a verifiable insert at index 450 occurs. The algorithm starts with the computation of the hash values for the proof node n_3 as *hashTower* at line 5 and v_2 as *hash* at line 7. Then the loop handles all proof nodes until the intersection point of the newly inserted node n_3 and the precedent node v_2. In the loop, the first iteration calculates the hash value for v_1 as *hash*. The second iteration yields a new *hashTower* using the proof node for d_2. The same happens for the third iteration but using the proof node for d_1. Then the hash value for the proof node c_3 is calculated as *hash*, and the same operation is done for c_2. The hash value for the proof node m_1 (intersection point) is computed by taking *hash* and *hashTower*. Following this, the algorithm calculates all remaining hash values until the root. The last hash value computed is the hash of the root, which is the temp meta data. If the server's meta data for the updated FlexList is the same as the newly computed temp meta data, then the meta data stored at the client is updated with this new version.

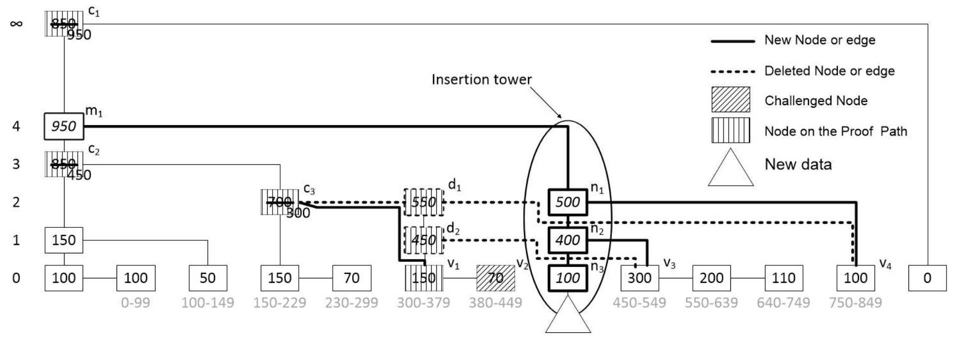


Fig. 12. Verifiable insert example.

verifyRemove (Algorithm V.6) is run to verify the correct removal of a block in the FlexList, using the proof and the new meta data by the server. Proof vector P is generated for the left neighbor and the node to be deleted. It calculates the temp meta data using the proof P as if the node has been removed. The inputs are the proof, a tag, and the new block information. The output is accept if the temp root calculated is equal to the meta data from the server, otherwise reject.

Algorithm V.6: verifyRemove Algorithm

```
Input: $C$, $P$, $T$, $MetaData$, $MetaData_{byServer}$
Output: accept or reject, $MetaData'$
    Let C= $(i_0, i_1)$ where $i_0$, $i_1$ are the index of the left neighbor and the removed index respectively; $P = (A_0, \ldots, A_k)$, where $A_j = (level_j,\ r_j,\ \text{hash}_j,\ rgtOrDwn_j,\ isInter_j,\ isEnd_j,\ length_j)$ for $j = 0, \ldots, k$; $T = (tag_0, tag_1)$ where $tag_1$ is tag value for deleted node and $tag_0$ is for its precedent node ; $P$, $T$ are the proof and tags before the removal;
1   if $!VerifyMultiProof(C, P, T, MetaData)$ then
2       return reject
3   else
4       dn = size($P$) - 1; i = size($P$) - 2; last = dn
5       while $!A_i.isEnd$ do
6           decrement(i)
7       rank = $A_{dn}$.rank; hash = $hash$(0, rank, $tag_0$, $A_{dn}$.hash, $A_{dn}$.length)
8       decrement(dn)
9       if $!A_{dn}.isEnd$ or $!A_i.isInter$ then
10          decrement(i)
11      while $!A_{dn}.isEnd$ or $!A_i.isInter$ do
12          if $A_i.level < A_{dn}.level$ or $A_{dn}.isEnd$ then
13              rank += $A_i$.rank - $A_{i+1}$.rank
                // $A_i$.length is added to hash calculation if $A_i$.level $= 0$
14              hash = $hash$($A_i$.level, rank, $A_i$.hash, hash)
15              decrement(i)
16          else
17              rank += $A_{dn}$.rank - $A_{dn+1}$.rank
18              hash = $hash$($A_{dn}$.level, rank, hash, $A_{dn}$.hash)
19              decrement(dn)
20      decrement(i)
21      $MetaData_{new}$ = calculateRemainingHashes(i, hash, $A_{last}$.length, $P$)
22      if $MetaData_{byServer}$ = $MetaData_{new}$ then
23          MetaData = $MetaData_{new}$
24          return accept
25      return reject
```

The algorithm will be discussed through the example in Figure 13, where a verifiable remove occurs at index 450. The algorithm starts by placing iterators i and dn at the position of v_2 (line 6) and d_4 (line 4), respectively. At line 7, the hash value (*hash*) for the node v_2 is computed using the hash information at d_4. dn is then updated to point at node d_3 at line 8. The loop is used to calculate the hash values for the newly added nodes in the FlexList using the hash information in the proof nodes of the deleted nodes. The hash value for v_1 is computed by using *hash* in the first iteration. The second and third iterations of the loop calculate the hash values for m_2 and m_1 by using hash values stored at the proof nodes of d_3 and d_2 respectively. Then the hash calculation is done for c_3 by using the hash of m_1. After the hash of c_2 is computed using the hash of c_3, the algorithm calculates the hashes until the root. The hash of the root is the temp meta data. If the server's meta data for the updated FlexList is verified using the newly computed temp meta data, then the meta data stored at the client is updated with this new version.

VI. Analysis

A. Security Analysis

Note that within a proof vector, all nodes which are marked with the end flag "E" contain the length of their associated data. These values are used to check if the proof in the process of verification is indeed the proof of the block corresponding

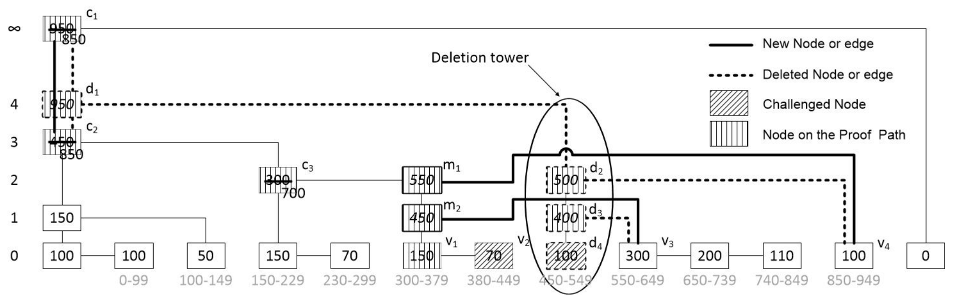


Fig. 13. Verifiable remove example.

to the challenged index. A careless implementation may not consider the authentication of the length values. To show the consequence of not authenticating the length values, we will use Figure 10 and Figure 11 as an example.

The scenario starts with the client challenging the server on the indices $\{170, 400, 500, 690\}$ that correspond to nodes c_1, v_4, c_3, and c_4 respectively. The server finds out that he does not possess v_4 anymore, and therefore, instead of that node, he will try to deceive the client by sending a proof for c_2. The proof vector will just be the same as the proof vector illustrated in Figure 11 with a slight change done to deceive the client. The change is done to the fourth entry from the top (the one corresponding to c_2): Instead of the original length 80, the server puts 105 as the length of c_2. The verification algorithm (without authenticated length values) at the client side will accept this fake proof as follows:

- The block sum value and the tags get verified since both are prepared using genuine tags and blocks of the actual nodes. The client cannot realize that the data of c_2 counted in the block sum is not 105 bytes, but 80 bytes instead. This is because the largest challenged data (the data of c_4 of length 110 in our example) hides the length of the data of c_2.
- Since the proof vector contains genuine nodes (though not necessarily all the challenged ones), when the client uses *verifyMultiProof* algorithm on the proof vector from Figure 11, the check on line 22 (Algorithm V.3), "$hash \neq$ MetaData" will be passed.
- The client also checks that the proven nodes are the challenged ones by comparing the challenge indices with the reconstructed indices by "$\forall$a, $0 \leq$ a $\leq$ n , $0 \leq C_a - V_a < endnodes_{n-a}.length$" (Algorithm V.3 on line 22). This check will also be passed because:
  - c_1 is claimed to start at index 150 and contain 80 bytes, and hence includes the challenged index 170 (verified as $0 < 170 - 150 < 80$).
  - c_2 is claimed to start at index 300 and contain **105** bytes, and hence includes the challenged index 400 (verified as $0 < 400 - 300 < 105$).
  - c_3 is claimed to start at index 450 and contain 100 bytes, and hence includes the challenged index 500 (verified as $0 < 500 - 450 < 100$).
  - c_4 is claimed to start at index 640 and contain 110 bytes, and hence includes the challenged index 690 (verified as $0 < 690 - 640 < 110$).

There are two possible solutions. We may include either the authenticated rank values of the right neighbors of the end nodes to the proofs, or use the length of the associated data in the hash calculation of the leaf nodes. We choose the second solution, which is authenticating the length values, since adding the neighbor node to the proof vector also adds a tag and a hash value, for each challenged node, to the communication cost.

Lemma 1. *If there exists a collision resistant hash function family, FlexList is an authenticated dictionary.*

Proof: The only difference between FlexList and RBASL is the calculation of the rank values at the leaf levels. All rank values, which are used in the calculation of the start indices of the challenged nodes, are used in hash calculations as well. Therefore, both *length* and *rank* values contribute to the calculation of the hash value of the root. To deceive the client, the adversary should fake the rank or length value of at least one of the proof nodes. By Theorem 1 of [25], if the adversary sends a verifying proof vector for any node other than the challenged ones, we can break the collision resistance of the hash function, using a simple reduction. Therefore, we conclude that our FlexList protects the integrity of the tags and data lengths associated with the leaf-level nodes. ■

Remember that the tags verification protects the integrity of the data itself, based on the factoring assumption, as shown by DPDP [15]. Combining this with Lemma 1 concludes security of FlexDPDP.

Theorem 1. *If the factoring problem is hard and a collision resistant hash function family exists, then FlexDPDP is secure.*

Proof: Consider the proof by Erway et al. for Theorem 2 of [15]. Replacing Lemma 2 of [15] in that proof with our Lemma 1 yields an identical challenger, and the exact proof shows the validity of our theorem. ■

B. Performance Analysis

We have developed a prototype implementation of an optimized FlexList (on top of our optimized skip list and authenticated skip list implementations). We used C++ and employed some methods from the *Cashlib* library [23], [10]. The local experiments were conducted on a 64-bit machine with a 2.4GHz Intel 4 core CPU (only one core is active), 4GB main memory and 8MB L2 cache, running Ubuntu 12.10. As security parameters, we used 1024-bit RSA modulus, 80-bit random numbers, and SHA-1 hash function, overall resulting in an expected security of 80-bits. All our results are the average of 10 runs. The tests include I/O access time since **each block of the file is kept on the hard disk drive separately**, unless it stated otherwise. The size of a FlexList is suitable to keep a lot of FlexLists in RAM.

For energy efficiency tests (Figure 15 and 18), we used Watts up Pro meter. It measures the total energy consumption of the connected device. We conducted the tests and took their energy consumption measurements. Then, we measured the average energy cost for the idle time when no tests were taking place. The difference between these two measurements were used in the calculation of the results. Energy consumption and time (CPU) gain results are close for both graphs. For the energy efficiency tests we do not take I/O delay into account. Therefore, we argue that energy efficiency of our algorithms is directly impacted by the CPU usage time of them. Therefore, our algorithms that optimize operations in the FlexList creation and multi-challenge proof generation are efficient in terms of both time and energy.

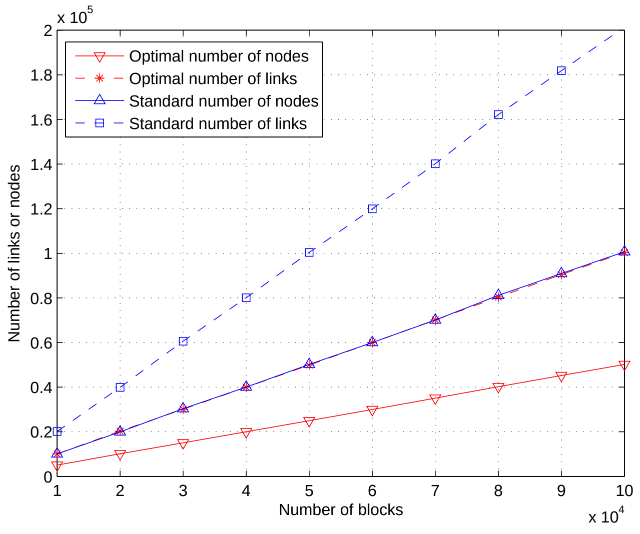


Fig. 14. The number of nodes and links used on top of leaf level nodes, before and after optimization.

1) Core FlexList Algorithm Performance: One of the core optimizations in a FlexList is done in terms of the structure. Our optimization, removing unnecessary links and nodes, ends up with 50% less nodes and links on top of the leaf nodes, which are always necessary since they keep the file blocks. Figure 14 shows the number of links and nodes used before and after optimization. The expected number of nodes in a regular skip list is $2n$ [29] (where n represents the number of blocks): n leaf nodes and n non-leaf nodes. Each non-leaf node makes any left connection below its level unnecessary as described in Section III. Since in a skip list, half of all nodes and links are at the leaf level in expectation, this means half of the non-leaf level links and half of the leaf level links are unnecessary, making a total on n unnecessary links. Since there are $n/2$ non-leaf unnecessary links, it means that there are $n/2$ non-leaf unnecessary nodes as well, according to unnecessary node definition (Section III). Hence, there are $n - n/2 = n/2$ non-leaf necessary nodes. Since each necessary node has 2 links, in total there are $2 * n/2 = n$ necessary links above the leaf level. Therefore, in Figure 14, there is an overlap between the standard number of non-leaf nodes (n) and the optimal number of the non-leaf links (n). Therefore, we eliminated approximately **50% of all nodes and links** above the leaf level (and 25% of all).

Moreover, we presented a novel algorithm for the efficient building of a FlexList. Figure 15 demonstrates energy consumption and time ratios between the *buildFlexList* algorithm and building FlexList by means of insertion (in sorted order). The time ratio is calculated by dividing the time spent for the building FlexList using insertion method by the time needed by the *buildFlexList* algorithm. The same ratio equation is applied to the energy consumption ratio calculation. In our energy-time ratio experiments, we do not take into account the disk access time; therefore there is no delay for I/O switching. The energy and time ratio values are close to each other because of the same reason: the more time,

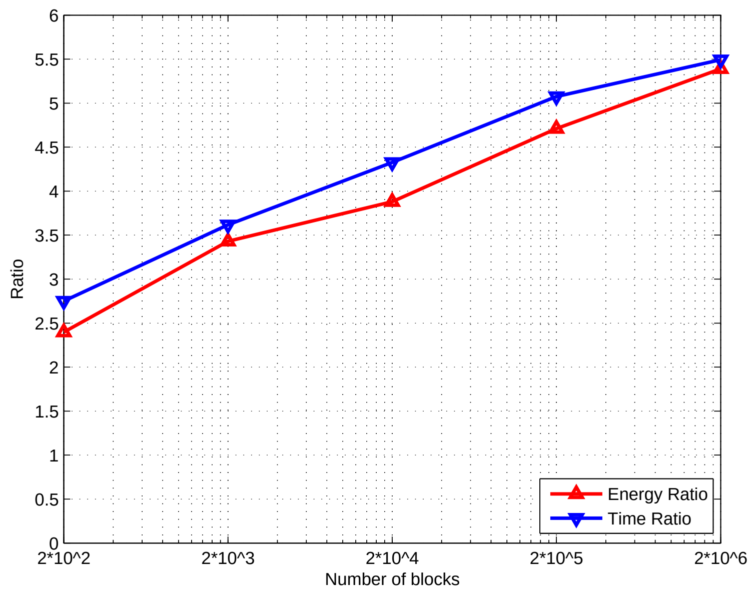


Fig. 15. Time and energy ratios on buildFlexList algorithm against insertions.

the algorithm executes, the more energy is spent. As expected, *buildFlexList* algorithm outperforms the regular insertion method, since in the *buildFlexList* algorithm the expensive hash calculations are performed only once for each node in the FlexList. So practically, the *buildFlexList* algorithm reduced the time to build a FlexList for a **file of size 400MB** with 200000 blocks **from 12 seconds to 2.3 seconds** and for a **file of size 4GB** with 2000000 blocks **from 128 seconds to 23 seconds**.

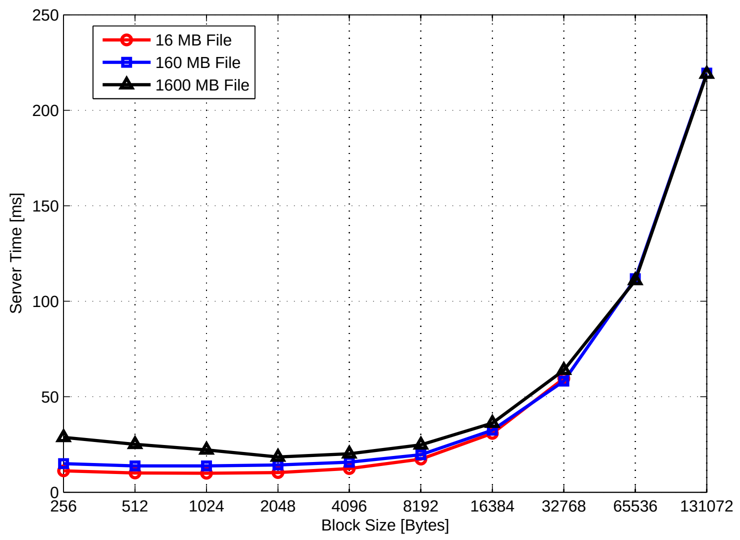


Fig. 16. Server time for 460 random challenges as a function of block size for various file sizes.

2) FlexDPDP Performance: **Proof Generation Performance** : Figure 16 shows the server proof generation time for FlexDPDP as a function of the block size by fixing the file size to 16MB, 160MB, and 1600MB. As shown in the figure, with the increase in block size, the time required for the proof generation increases, since with a higher block size, the block sum generation takes more time. Interestingly though, with extremely small block sizes, the number of nodes in the FlexList become so large that it dominates the proof generation time. Since 2KB block size worked best for various file sizes, our other tests employ 2KB blocks. These 2KB blocks are kept **on the hard disk drive**, on the other hand the FlexList nodes are much smaller and subject to be **kept in RAM**. While we observed that *buildFlexList* algorithm runs faster with bigger block sizes (since there will be fewer blocks), the creation of a FlexList happens only once. On the other hand, the proof generation algorithm runs periodically depending on the client, therefore we chose to optimize for its running time.

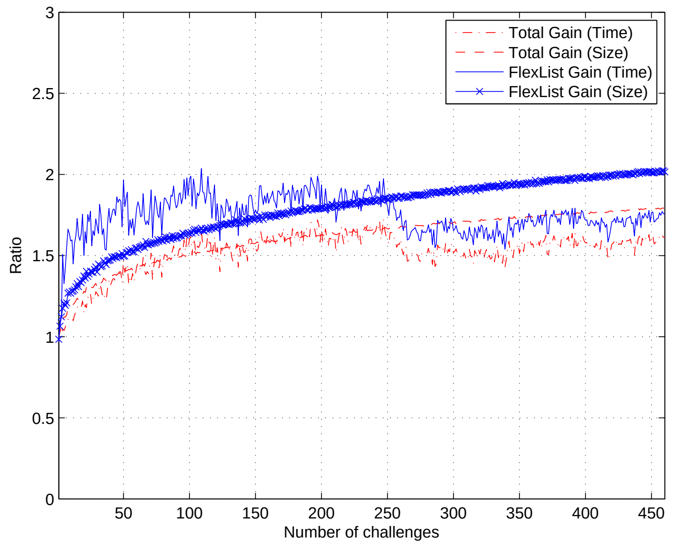


Fig. 17. Performance gain graph ([460 single proof / 1 multi proof] for 460 challenges).

The performance of our optimized implementation of the proof generation mechanism is evaluated in terms of communication and computation. We take into consideration the case where the client wishes to detect with more than 99% probability if more than a 1% of her 1GB data is corrupted by challenging 460 blocks; the same scenario as in PDP and DPDP [2], [15]. In our experiment, we used a FlexList with 500,000 nodes, where the block size is 2KB.

In Figure 17 we plot the ratio of the unoptimized proofs over our optimized proofs in terms of the **FlexList proof** size and computation, as a function of the number of challenged nodes. The unoptimized proofs correspond to proving each block separately, instead of using our *genMultiProof* algorithm for all of them at once. Our multi-proof optimization results in **40% computation and 50% communication gains** for FlexList proofs. This corresponds to FlexList proofs being up to **1.75 times as fast** and **2 times as small**.

We also measure the gain in the total size of a **FlexDPDP proof** and computation done by the server in Figure 17. With our optimizations, we clearly see a gain of about **35%** and **40% for the overall computation and communication**, respectively, corresponding to proofs being up to **1.60 times as fast** and **1.75 times as small**. The whole proof roughly consists of 213KB FlexList proof, 57KB of tags, and 2KB of block sum. Thus, for 460 challenges as suggested by PDP and DPDP [2], [15], we obtain a decrease in total **proof size from 485KB to 272KB**, and the computation is **reduced from 19ms to 12.5ms** by employing our *genMultiProof* algorithm. We could have employed gzip to eliminate duplicates in the proof, but it does not perfectly handle the duplicates and our algorithm also provide computation (proof generation and verification) time optimization as well. Compression is still beneficial when applied on our optimal proof.

Furthermore, we tested the performance of *genMultiProof* algorithm in terms of energy efficiency. The time and energy ratio graph for the *genMultiProof* algorithm is shown in Figure 18. We have tested the algorithm in different file size scenarios, starting a file size from 4MB to 4GB (where block size is 2KB, and thus the number of blocks increase with the file size). In *constant* scenario we applied the same challenge size of 460. Our results showed a relative decline in the performance of the *genMultiProof* as the number of blocks in the FlexList increases. This is caused by the number of challenges being constant. Because as the number of blocks in the FlexList grows, the number of repeated proof nodes in the proof decreases. In *proportional* scenario, we have the time and energy ratio for 5, 46 and 460 challenges for the block number of 20000, 200000 and 2000000 respectively. The graph shows a relative incline in the performance of *genMultiProof* for the proportional number of challenges to the number of blocks in a file. The algorithm has a clear efficiency gain in the computation time in comparison to the generating each proof individually.

Provable Update Performance: In FlexDPDP, we have optimized algorithms for verifiable update operations. The results for the basic functions of the FlexList ($insert, modify, remove$) against their verifiable versions are shown in Figure 19. The regular $insert$ method takes more time than any other method, since it needs extra time for the memory allocations and I/O delay. The $remove$ method takes less time than the $modify$ method, because there is no I/O delay and at the end of the $remove$ algorithm there are less nodes that need recalculation of the hash and rank values. As

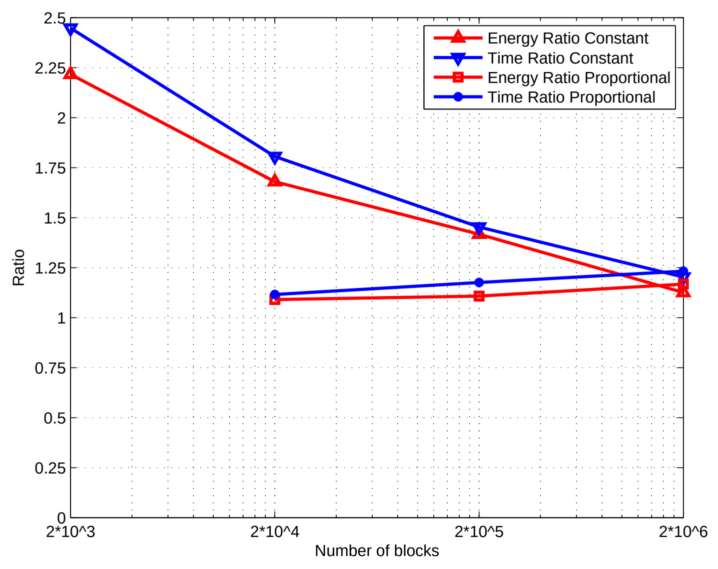


Fig. 18. Time and energy ratios on *genMultiProof* algorithm.

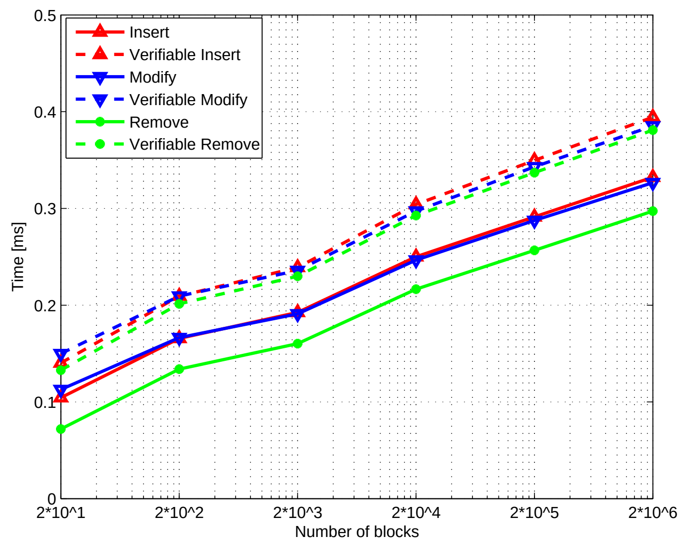


Fig. 19. Performance evaluation of FlexList methods and their verifiable versions.

expected, the complexity of the FlexList operations increase logarithmically. The verifiable versions of the functions require an average overhead of 0.05 ms for a single run. For a **single verifiable insert**, the server **needs less than 0.4ms** to produce a proof in a FlexList with 2 million blocks (corresponding to a 4GB file). These results show that the verifiable versions of the updates can be employed with only little overhead.

3) Comparison with Static Cloud Storage on the PlanetLab: We compare static PDP with FlexDPDP, which is a dynamic system. The server in PDP computes the sum of the challenged blocks and the multiplication and exponentiation of their tags. FlexDPDP server only computes the sum of the blocks and FlexList proof, but not the multiplication and exponentiation of their tags, which are expensive cryptographic computations. In such a scenario the FlexDPDP server outperforms such a naive PDP server, since the multiplication of tags in PDP takes much longer than the FlexList proof generation in FlexDPDP. This result is in contract to the fact that PDP proofs take $O(1)$ time and space whereas FlexDPDP proofs require $O(\log n)$ time and space, due to a huge difference in the constants in the Big-Oh notation.

We note that it is possible for PDP to be implemented by the server sending the tags to the client and the client computing the multiplication and exponentiation of the tags. If this is done in a PDP implementation, even though the proof size grows, the PDP server can respond to challenges faster than FlexDPDP. Therefore, we realize that where to handle the multiplication and exponentiation of tags is an implementation decision for PDP.

	PDP	PDP*	FlexDPDP
Local Server Computation	413.19	12.97	38.60
Close Client Total	466.82	557.49	649.11
Mid-range Client Total	496.856	714.47	874.63
Distant Client Total	551.376	986.98	1023.25

TABLE IV
TIME SPENT FOR A CHALLENGE OF SIZE 460, IN MILLISECONDS. PDP* IS THE MODIFIED PDP SCHEME, WHERE WE SEND ALL CHALLENGED TAG VALUES TO THE CLIENT INSTEAD OF MULTIPLYING THEM.

We deployed FlexDPDP, together with original and modified PDP versions, on a world-wide network test bed, PlanetLab. On PlanetLab, a node has minimum requirements of having 6x Intel Xeon E5 cores @ 2.2GHz processor, 24 GB of RAM, and 2TB shared hard disk space. The nodes are also required to have minimum of 400kbps of bi-directional bandwidth to the Internet [26] As a central point in Europe, we chose a node in Berlin, Germany[2] as the server. We measured the whole time spent for one challenge at both the client and the server side (Table IV). We moved our client location and tested serving a close range client in Munich, Germany[3], a mid-range client in Koszalin, Poland[4], and a distant client in Lisbon, Portugal[5]. We used a single core at each side. The protocols are run on a 1GB file, which is divided into blocks of 2KB, having 500000 nodes.

Inducting from Table IV, we conclude that using 6 cores (usual core count in PlanetLab nodes), a **PDP server can answer 14.5 queries per second** whereas a server using **PDP* and FlexDPDP can serve 462 queries and 155.5 queries per second** respectively. We discern that, for the server, tag multiplication is the most time consuming task in each challenge. It is clear that to increase the server throughput, tag multiplication should be delegated to the client. This delegation increases the total time spent by the client a bit more than it saves from the server, since the tags should be sent over the network. However, the outcome is the dramatic increase in the server throughput. Note that, when one considers the total time a client spends for sending a challenge, obtaining the proof, and verifying it, the overhead of being dynamic (FlexDPDP vs. PDP*) is around 40 to 90 ms, which is a barely-visible difference for a real-life application (especially considering that the whole process takes on the order of a second).

VII. Conclusion and Future Work

The security and privacy issues are significant obstacles toward the cloud storage adoption [36]. With the emergence of cloud storage services, data integrity has become one of the most important challenges. Early works have shown that the static solutions with optimal complexity [2], [30], and the dynamic solutions with logarithmic complexity [15] are within reach. However, a DPDP [15] solution is not applicable to real life scenarios since it supports only fixed block size and therefore lacks flexibility on the data updates, while the real life updates are likely not of constant block size. We have extended earlier studies in several ways and provided a new data structure (FlexList) and its optimized implementation for use in the cloud data storage. A FlexList efficiently supports variable block sized dynamic provable updates, and we showed how to handle multiple proofs and updates at once, greatly improving scalability. As future work, we plan to further study parallelism and energy efficiency. We also aim to extend our system to peer-to-peer settings.

Acknowledgements

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 number 112E115, and Türk Telekom, Inc. under grant 11315-06, and Koç Sistem, Inc.

References

[1] A. Anagnostopoulos, M. T. Goodrich, and R. Tamassia. Persistent authenticated dictionaries and their applications. In *ISC*, 2001.
[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.

[2]planetlab01.tkn.tu-berlin.de
[3]planetlab1.lkn.ei.tum.de
[4]ple2.tu.koszalin.pl
[5]planetlab1.di.fct.unl.pt

[4] G. Ateniese, R. D. Pietro, L. V. Mancini, and G. Tsudik. Scalable and efficient provable data possession. In *SecureComm*, 2008.

[5] G. D. Battista and B. Palazzi. Authenticated relational tables and authenticated skip lists. In *DBSec*, 2007.

[6] K. Blibech and A. Gabillon. Chronos: an authenticated dictionary based on skip lists for timestamping systems. In *SWS*, 2005.

[7] K. Blibech and A. Gabillon. A new timestamping scheme based on skip lists. In *ICCSA (3)*, 2006.

[8] H.-J. Boehm, R. Atkinson, and M. Plass. Ropes: an alternative to strings. *Software: Practice and Experience*, 25, 1995.

[9] K. D. Bowers, A. Juels, and A. Oprea. Hail: a high-availability and integrity layer for cloud storage. In *ACM CCS*, 2009.

[10] Brownie cashlib cryptographic library. http://github.com/brownie/cashlib.

[11] D. Cash, A. Küpçü, and D. Wichs. Dynamic proofs of retrievability via oblivious ram. In *EUROCRYPT*, 2013.

[12] S. A. Crosby and D. S. Wallach. Authenticated dictionaries: Real-world costs and trade-offs. *ACM TISSEC*, 2011.

[13] R. Curtmola, O. Khan, R. Burns, and G. Ateniese. Mr-pdp: Multiple-replica provable data possession. In *ICDCS*, 2008.

[14] Y. Dodis, S. Vadhan, and D. Wichs. Proofs of retrievability via hardness amplification. In *TCC*, 2009.

[15] C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia. Dynamic provable data possession. In *ACM CCS*, 2009.

[16] M. Etemad and A. Küpçü. Transparent, distributed, and replicated dynamic provable data possession. In *ACNS*, 2013.

[17] C. C. Foster. A generalization of avl trees. *Commun. ACM*, 1973.

[18] M. T. Goodrich, C. Papamanthou, R. Tamassia, and N. Triandopoulos. Athos: Efficient authentication of outsourced file systems. In *ISC*, 2008.

[19] M. T. Goodrich and R. Tamassia. Efficient authenticated dictionaries with skip lists and commutative hashing. Technical report, Johns Hopkins Information Security Institute, 2001.

[20] M. T. Goodrich, R. Tamassia, and A. Schwerin. Implementation of an authenticated dictionary with skip lists and commutative hashing. In *DARPA*, 2001.

[21] A. Juels and B. S. Kaliski. PORs: Proofs of retrievability for large files. In *ACM CCS*, 2007.

[22] P. Maniatis and M. Baker. Authenticated append-only skip lists. *Acta Mathematica*, 2003.

[23] S. Meiklejohn, C. Erway, A. Küpçü, T. Hinkle, and A. Lysyanskaya. Zkpdl: Enabling efficient implementation of zero-knowledge proofs and electronic cash. In *USENIX Security*, 2010.

[24] R. Merkle. A digital signature based on a conventional encryption function. *LNCS*, 1987.

[25] C. Papamanthou and R. Tamassia. Time and space efficient algorithms for two-party authenticated data structures. In *ICICS*, 2007.

[26] Planetlab node requirements. https://planet-lab.org/node/225 [Online; accessed 20-March-2013].

[27] D. J. Polivy and R. Tamassia. Authenticating distributed data using web services and xml signatures. In *In Proc. ACM Workshop on XML Security*, 2002.

[28] W. Pugh. A skip list cookbook. Technical report, 1990.

[29] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 1990.

[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] R. Tamassia and N. Triandopoulos. On the cost of authenticated data structures. In *ESA, LNCS*, 2003.

[33] 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.

[34] Y. Zhang and M. Blanton. Efficient dynamic provable data possession of remote data via balanced update trees. *ASIA CCS*, 2013.

[35] Q. Zheng and S. Xu. Fair and dynamic proofs of retrievability. In *CODASPY*, 2011.

[36] M. Zhou, R. Zhang, W. Xie, W. Qian, and A. Zhou. Security and privacy in cloud computing: A survey. In *IEEE SKG*, 2010.