Efficient Dynamic Provable Possession of Remote Data via Balanced Update Trees

Yihua Zhang and Marina Blanton
Department of Computer Science and Engineering
University of Notre Dame
{yzhang16,mblanton}@nd.edu

ABSTRACT

The emergence and availability of remote storage providers prompted work in the security community that allows a client to verify integrity and availability of the data she outsourced to an untrusted remove storage server at a relatively low cost. Most recent solutions to this problem allow the client to read and update (insert, modify, or delete) stored data blocks while trying to lower the overhead associated with verifying data integrity. In this work we develop a novel and efficient scheme, computation and communication overhead of which is orders of magnitude lower than those of other state-of-the-art schemes. Our solution has a number of new features such as a natural support for operations on ranges of blocks, and revision control. The performance guarantees that we achieve stem from a novel data structure, termed *balanced update tree*, and removing the need to verify update operations.

Categories and Subject Descriptors

K.6 [**Management of Computing and Information Systems**]: Security and Protection; E.1 [**Data Structures**]: Trees; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*distributed systems*

General Terms

Security, Algorithms, Verification

Keywords

Authentication, dynamic provable data possession, balanced update tree, outsourced storage, proof of retrievability

1. INTRODUCTION

Cloud computing and storage services today enable convenient on-demand access to computing and data storage resources, which make them attractive and economically sensible for clients with limited computing or storage resources. Security and privacy, however, have been suggested to be the top impediment on the way of harnessing full benefits of these services (see, e.g., [1]). For that reason, there has been an increased interest in the research community in securing outsourced data storage and computation, and in particular, in verification of remotely stored data.

The line of work on proofs of retrievability (POR) or provable data possession (PDP) was initiated in [4, 20] and consists of many results [11, 25, 5, 24, 19, 10, 29, 6, 8, 9, 16, 26, 27, 13, 28, 30, 21, 17, 22] that allow for integrity verification of large-scale remotely stored data. At high level, the idea consists of partitioning a collection of data into data blocks and storing the blocks together with metadata at a remote storage server. Periodically, the client issues integrity verification queries (normally in the form of challenge-response protocols), which allow the client to verify a number of data blocks independent of the overall number of outsourced blocks using the metadata to ensure with high probability that all stored blocks are intact and available. Schemes that support dynamic operations [5, 19, 16, 26, 27, 30, 21, 17, 22], DPDP, additionally allow the client to issue modify, insert, and delete requests, after each of which the integrity of the newly stored data is verified.

The motivation for this work comes from (i) improving the performance of the existing schemes when modifications to the data are common, and (ii) extending the available solutions with new features such as support for revision control and multi-user access to shared data[1]. Toward this end, we design and implement a novel mechanism for efficient verification of remotely stored data with support for dynamic operations. Our solution uses a new data structure, which we call a balanced *update tree*. The size of the tree is independent of the overall size of the outsourced storage, but rather depends on the number of updates to the remote blocks. The data structure is designed to provide a natural support for handling ranges of blocks (as opposed to always processing individual blocks) and is balanced allowing for very efficient operations. Unlike all prior work with support for dynamic operations where each dynamic operation is followed by verification of its correct execution by the server, our scheme eliminates such checks. Instead, verification is performed only at the time of retrieving the data or through periodic challenge queries (both of which are also present in prior work). This distinctive feature of our scheme therefore results in substantial communication and computation savings.

Today many services outsource their storage to remote servers or the cloud, which can include web services, blogs, and other applications in which there is a need for multiple users to access and update the data, and modifications to the stored data are common. For example, many subscribers of a popular blog hosted by a cloud-based server are allowed to upload, edit, or remove blog content ranging from a short commentary to a large video clip. This demands support for multiple user access while maintaining data

[1]Due to space constraints, multi-user access is described in the full version of this work.

ASIA CCS'13, May 8–10, 2013, Hangzhou, China.

consistency and integrity, which current schemes do not provide. In addition to supporting this feature, our solution provides support for revision control which can be of value for certain applications as well. Because in the existing solutions the server maintains only the up-to-date values of each data block, support for revision control can be added by means of additional techniques (such as [3]), but they result in noticeable overhead per update. In our solution, on the other hand, there is no additional cost for enabling retrieval and verification of older versions of data blocks beyond the obvious need for the server to store them with small metadata. Finally, because the size of the maintained data structure grows with the number of dynamic operations, by issuing a commit command, the client will be able to keep the size of the maintained update tree below a desired threshold if necessary.

To summarize, our solution enjoys the following features:

- *Improved efficiency in handling dynamic operations.* In our solution there is no need to verify integrity of updates (e.g., a data block which is modified a large number of times and is subsequently deleted), while prior schemes invest resources in verifying correct implementation of each user's action by the storage server.
- *Support for range operations.* The natural support and use of range operations allows for additional performance improvement compared to the existing solutions.
- *Balanced data structure.* The update tree used for verifying correctness of the stored data blocks is always balanced regardless of the number and order of dynamic operations on the storage. This results in similar performance for locating information about each data block in the tree and is logarithmic in the size of the tree.
- *Size of the maintained data structure.* In our solution the size of the maintained update tree is independent of the outsourced data size, while it is linear for other solutions that support dynamic operations. The size of the update tree grows with the number of dynamic operations, but can be reduced by issuing a commit command.
- *Support for revision control.* We provide natural support for revision control and allow clients to retrieve previous versions of data and efficiently verify their integrity. There is no additional overhead for either the client or the server for enabling this feature (besides the obvious need to maintain different versions of data by the server).
- *Public verifiability.* Our scheme can be easily modified to support public verifiability, which allows the client to outsource periodic verification of storage integrity to a third party auditor (who is different from the server).

These features come at the cost of increased storage (compared to other schemes) at the client who in our solution maintains the update tree locally. Because the data structure size is not large (and is independent of the outsourced data size), we believe it is a reasonable tradeoff for other improvements that we achieve. In particular, any PC-based client will not be burdened by the local storage even if it reaches a few MB. Other weaker clients (such as mobile users) and battery-operated devices in particular are power-bound and benefit from the reduced computation in our scheme while still are able to store the data structure locally. A detailed comparison of the performance of our and other DPDP schemes is given in section 8, which shows that both computation and communication overhead of our scheme is orders of magnitude lower than those of other existing solutions.

2. RELATED WORK

We next review selected PDP/POR schemes from prior literature and their difference with the proposed solution. In particular, we are interested in the schemes that support dynamic operations on outsourced storage.

One line of research [20, 5, 26] relies on so-called sentinels which are outsourced together with the client's data and are used to verify remotely stored blocks. Such scheme, however, poorly scale when blocks need to be updated and allow only for a limited number of audits.

Another line of research with support for dynamic operations utilizes specialized data structures such as Merkle hash trees or chained hashes [27, 21, 22] or skip lists [16, 19, 17] to organize the data blocks outsourced to a server. When a Merkle hash tree is used, each leaf node corresponds to the hash of a data block, and the client locally keeps the root value of the tree. Correctness of a read request on the ith block is verified in a standard way by reconstructing the root value from the ith block and hashes of the sibling nodes on the path from the block to the root. For an update request on the ith block, the client retrieves and verifies the same information as that of the read request. The client then computes a new root value based on the new ith block, substitutes it for the previously stored root value, and uses it afterwards.

A disadvantage of Merkle hash tree based solutions is that the tree becomes unbalanced after a series of insert and delete requests. That is, a block insert request at position i is handled by replacing the $(i-1)$th block's node with a new node that has two children: a node for the previously stored $(i-1)$th and a node for the newly inserted ith block. Similarly, a deletion request is handled by removing the corresponding node from the tree and making its sibling take the place of its parent. As access patterns normally do not uniformly span across the stored data, inserting multiple blocks at the same position will result in the height of the tree growing for each inserted data block. This will result in a large variance in the time to locate different blocks in the tree. Using an advanced tree structure (e.g., red-black tree) can help mitigate this problem by balancing the tree when necessary, but it comes with an additional cost of recomputing the hash of nodes affected by the balancing process.

To support the dynamic operations, [16] develops a scheme based on a skip list, which extends the original skip list [23] by incorporating *label* [18] and *rank* information to enable efficient authentication of client's updates. This allows for each update to be verified in expected $O(\log n)$ time with high probability, where n is the size of the skip list. The skip list remains balanced regardless of the client's access or update patterns. The authors also propose support for variable-sized blocks, which is achieved by handling a number of fixed size blocks as a single block and performing standard operations on it. While this approach guarantees the integrity of variable-sized data blocks in their entirety, it becomes impossible to verify an individual block upon receiving a request on it. Furthermore, the time to locate a fixed size block is linear in the number of blocks stored in a node, which may dominate the overall time when a node contains many blocks.

The original Merkle hash tree and skip list schemes maintain only the most recent copy of data. To incorporate revision control capabilities, [3] used a persistent authenticated data structure, which adds $O(\log n)$ extra space for each update, where n is the number of nodes in the data structure.

The data structure that we build has three properties that make it favorably compare to the existing schemes. First, each node in our update tree corresponds to a range of block indices (instead of a specific index as in prior work) defined by a dynamic operation per-

formed on a range of consecutive blocks. The reason for assigning a range of block indices to a node is motivated by a study [15] on user's file access patterns that observed that a large number of file accesses are sequential. Second, unlike maintaining a data structure of size linear in the number of outsourced data blocks, in our solution it is independent of the size of the stored data. Previously, the large size required the client to outsource the data structure to the cloud while locally maintaining only a constant-size data for integrity verification. In our update tree, on the other hand, a node represents a user-triggered update, and multiple updates issued on the same range of blocks can also be condensed into a single node. Due to its moderate size, the client can maintain the data structure locally, which makes the verification process more efficient. Third, we can specify requirements that define when the data structure should be rebalanced. Once the requirement is violated, the tree is re-organized to satisfy the constraint. As an example, the constraint of AVL trees [2] can be used that requires that the heights of a node's subtrees must differ by at most 1.

Prior to this work, the notion of a range tree was used in the databases to deal with range queries [7]. The range tree data structure, however, is majorly dissimilar to our update trees. For instance, range trees store one record per node as opposed to a range, are static as opposed to be dynamically updated and balanced throughout system operation, etc. Similarly, interval trees [12] cannot support insertions and deletion of block ranges which require partitioning of existing ranges/intervals in the tree and index changes. The operational details of update trees are therefore very different from those of interval trees. One of most significant challenges of this work was to design a dynamic update tree that can be rebalanced at low cost after arbitrary changes to it. A balanced update tree is therefore one of the novel aspects of this work.

3. PROBLEM DEFINITION

We consider the problem in which a resource-limited client is in possession of a large amount of data partitioned into blocks. Let N denote the initial number of blocks and m_i denote the data block at index i, where $1 \leq i \leq N$. The client C outsources her data to a storage or cloud server S and would like to be able to update and retrieve her data in a way that integrity of all returned data blocks can be verified. If the data is sensitive and its secrecy is to be protected from the server, the client should encrypt each data block using any suitable encryption mechanism prior to storing it at the remote server. In that case, each data block m_i corresponds to encrypted data, and the solution should be oblivious to whether data confidentiality is protected or not. We assume that the client and the server are connected by (or establish) a secure authenticated channel for the purposes of any communication.

The primary feature that we would like a scheme to have is *support for dynamic operations*, which include modifying, inserting, or deleting one or more data blocks. We also consider minimal-overhead *support for revision control* which allows the client to access and verify previous versions of its data, as a desirable feature to have, but it is not strictly necessary for a PDP scheme. Our scheme achieves this property at no extra cost beyond maintaining previous versions by the server, and we defer any additional discussion of this feature to the full version of this work.

We define a dynamic provable data possession scheme (DPDP) in terms of the following procedures:

- $\mathsf{KeyGen}(1^\kappa) \rightarrow \{\mathsf{sk}\}$ is a probabilistic algorithm run by C that on input a security parameter 1^κ produces key sk.
- $\mathsf{Init}(\langle \mathsf{sk}, m_1, \ldots, m_N \rangle, \langle \perp \rangle) \rightarrow \{\langle \mathsf{M}_C \rangle, \langle \mathsf{M}_S, D \rangle\}$ is a protocol run between C and S during which C uses sk to encode the initial data blocks $m_1, \ldots, m_N$ and store them at S who maintains all data blocks outsourced by the client in D. C's and S's metadata are maintained in M_C and M_S, resp.
- $\mathsf{Update}(\langle \mathsf{sk}, \mathsf{M}_C, \mathsf{op}, \mathsf{ind}, \mathsf{num}, m_{\mathsf{ind}}, \ldots, m_{\mathsf{ind}+\mathsf{num}-1} \rangle, \langle \mathsf{M}_S, D \rangle) \rightarrow \{\langle \mathsf{M}'_C \rangle, \langle \mathsf{M}'_S, D' \rangle\}$ is a protocol run between C and S, during which C prepares num blocks starting at index ind and updates them at S. The operation type op is either modification (0), insertion (1), or deletion (-1), where no data blocks are communicated for deletion.
- $\mathsf{Retrieve}(\langle \mathsf{sk}, \mathsf{M}_C, \mathsf{ind}, \mathsf{num} \rangle, \langle \mathsf{M}_S, D \rangle) \rightarrow \{\langle m_{\mathsf{ind}}, \ldots, m_{\mathsf{ind}+\mathsf{num}-1} \rangle, \langle \perp \rangle\}$ is a protocol run between C and S, during which C requests num data blocks starting from index ind, obtains them from S and verifies their correctness.
- $\mathsf{Commit}(\langle \mathsf{sk}, \mathsf{M}_C, \mathsf{ind}, \mathsf{num}, m_{\mathsf{ind}}, \ldots, m_{\mathsf{ind}+\mathsf{num}-1} \rangle, \langle \mathsf{M}_S, D \rangle) \rightarrow \{\langle \mathsf{M}'_C \rangle, \langle \mathsf{M}'_S, D' \rangle\}$ is a protocol run between C and S, during which C re-stores metadata of num data blocks starting from index ind at S. S erases all previous copies of the data blocks in the range as well as previously deleted by C blocks that fall into the range if they were kept for revision control.

Our formulation of the scheme has minor differences with prior definitions of DPDP, e.g., as given in [16]. First, update and retrieve operations are defined as interactive protocols rather than several algorithms run by either the client or the server. Second, in addition to using the Retrieve protocol for reading data blocks, in the current formulation it is also used to execute periodic audits. That is, verification of each read is necessary to ensure that correct blocks were received even if the integrity of the overall storage is assured through periodic challenges, and the verification is performed similar to periodic audits. In particular, because the Retrieve protocol is executed on a range of data blocks and can cover a large number of blocks, verification is performed probabilistically by checking a random sample of blocks of sufficient (but constant) size to guarantee the desired confidence level. (And if the number of requested blocks is below the constant, all of them are verified.) This protocol can then be easily adapted to implement periodic audits denoted as $\mathsf{Challenge}(\langle \mathsf{sk}, \mathsf{M}_C \rangle, \langle \mathsf{M}_S, D \rangle) \rightarrow \{\langle m_{i_1}, \ldots, m_{i_c} \rangle, \langle \perp \rangle\}$ during which a random subset of blocks at indices $i_1, \ldots, i_c$ is verified. To implement Challenge, we simply call Retrieve on the entire storage with the difference that data blocks which are not being verified are not returned. In other words, during each Challenge query, the client receives and verifies correctness of c data blocks. We stress that defining audit Challenge queries in terms of Retrieve requests is the matter of notational preference: the functionality and probabilistic nature of verification of both Retrieve and Challenge requests is the same in our and other DPDP schemes. We obtain that prior work requires verification for each block update operation and a constant number of verifications per audit or read request. In our proposed scheme, only read and audit requests need to be verified by checking a constant number of blocks per request.

This constant c is computed in our and prior work by using detection probability of $1 - ((\mathsf{num} - t)/\mathsf{num})^c$, where num is the number of blocks being checked, from which the server tampers with t. Then, say, using $c = 460$ the client can detect the problem with 99% probability if the server tampers with 1% or more of the data regardless of the data size. This means that during Retrieve or Challenge calls, $\min(c, \mathsf{num})$ data blocks need to be verified.

To show security, we follow the definition of secure dynamic PDP from prior literature. In particular, we base our definition on the original definition of secure DPDP from [16] and introduce logical changes to account for the slightly different setting. In this

context, the client should be able to verify the integrity of any data block returned by the server. This includes the verification that the returned data block corresponds to the most recent version of it (or, when revision control is used, a specific previous version, including deleted content, as requested by the client). The server is considered fully untrusted and can modify the stored data in any way it wishes (including deleting the data). Our goal is to design a scheme in which any violations of data integrity or availability will be detected by the client. More precisely, in the single-user setting the security requirements are formulated as a game between a challenger (who acts as the client) and any probabilistic polynomial time (PPT) adversary $\mathcal{A}$ (who acts as the server):

Setup: the challenger runs $\mathsf{sk} \leftarrow \mathsf{KeyGen}(1^\kappa)$. $\mathcal{A}$ specifies the data blocks $m_1, \ldots, m_N$ and their number N for the initialization and obtains initial transmission from the challenger.

Queries: The adversary $\mathcal{A}$ specifies what type of a query to perform and on what data blocks. The challenger prepares the query and sends it to $\mathcal{A}$. If the query requires a response, $\mathcal{A}$ sends it to the challenger, who informs $\mathcal{A}$ about the result of verification. The adversary can request any polynomial number of queries of any type, participate in the corresponding protocols, and be informed of the result of verification.

Challenge: At some point, $\mathcal{A}$ decides on the content $m_1, \ldots, m_R$ on which it wants to be challenged. The challenger prepares a query that replaces the current storage with the requested data blocks and interacts with $\mathcal{A}$ to execute the query. The challenger and adversary update their metadata according to the verifying updates (non-verifying updates are considered not to have taken place), and the challenger and $\mathcal{A}$ execute $\mathsf{Challenge}(\langle \mathsf{sk}, \mathsf{M}_C \rangle, \langle \mathsf{M}_\mathsf{S}, D \rangle)$. If verification of $\mathcal{A}$'s response succeeds, $\mathcal{A}$ wins. The challenger has the ability to reset $\mathcal{A}$ to the beginning of the $\mathsf{Challenge}$ query a polynomial number of times with the purpose of data extraction. The challenger's goal is to extract the challenged portions of the data from $\mathcal{A}$'s responses that pass verification.

DEFINITION 1. *A DPDP scheme is secure if for any PPT adversary $\mathcal{A}$ who can win the above game with a non-negligible probability, there exists an extractor that allows the client to extract the challenged data in polynomial time.*

The existence of an extractor in this definition means that the adversary that follows any strategy can win the game above with probability negligibly larger than the probability with which the client is able to extract correct data. In our case, the probability of catching a cheating server is the same as in prior literature, and its security is analyzed in section 6.

Besides security, efficient performance of the scheme is also one of our primary goals. Toward that goal, we would like to minimize all of the client's local storage, communication, and computation involved in using the scheme. We also would like to minimize the server's storage and computation overhead when serving the client's queries. For that reason, the solution we develop has a natural support for working with ranges of data blocks which is also motivated by users' sequential access patterns in practice.

4. PROPOSED SCHEME

Building blocks. In this work we rely on a message authentication code (MAC) scheme, defined by three algorithms:

1. The key generation algorithm Gen, which on input a security parameter 1^κ produces a key k.
2. The tag generation algorithm Mac, which on input key k and message $m \in \{0,1\}^*$, outputs a fixed-size tag t.
3. The verification algorithm Verify, which on input a key k, message m, and tag t outputs a bit b, where $b = 1$ iff verification was successful.

For compactness, we write $t \leftarrow \mathsf{Mac}_k(m)$ and $b \leftarrow \mathsf{Verify}_k(m,t)$. The correctness requirement is such that for every κ, every $k \leftarrow \mathsf{Gen}(1^\kappa)$, and every $m \in \{0,1\}^*$, $\mathsf{Verify}_k(m, \mathsf{Mac}_k(m)) = 1$. The security property of a MAC scheme is such that every PPT adversary $\mathcal{A}$ succeeds in the game below with at most negligible probability in κ:

1. A random key k is generated by running $\mathsf{Gen}(1^\kappa)$.
2. $\mathcal{A}$ is given 1^κ and oracle access to $\mathsf{Mac}_k(\cdot)$. $\mathcal{A}$ eventually outputs a pair (m,t). Let Q denote the set of all of $\mathcal{A}$'s queries to the oracle.
3. $\mathcal{A}$ wins iff both $\mathsf{Verify}_k(m,t) = 1$ and $m \notin Q$.

Overview of the scheme. To mitigate the need for performing verifications for each update on the outsourced data, in our solution both the client and the server maintain metadata in the form of a binary tree of moderate size. We term the new data structure a *block update tree*. In the update tree, each node corresponds to a range of data blocks on which an update (i.e., insertion, deletion, or modification) has been performed. The challenge with constructing such a tree was to ensure that (i) a data block or a range of blocks can be efficiently located within the tree and (ii) we can maintain the tree to be balanced after applying necessary updates caused by client's queries. With our solution, we obtain that all operations on the remote storage (i.e., retrieve, insert, delete, modify, and commit) involve only work logarithmic in the tree size.

Each node in the update tree contains several attributes, one of which is the range of data blocks $[\mathsf{L}, \mathsf{U}]$. Each time the client requests an update on a particular range, the client and the server first need to find all nodes in the update tree with which the requested range overlaps (if any). Depending on the result of the search and the operation type, either 0, 1, or 2 nodes might need to be added to the update tree per single-block request. Operating on ranges helps to lower the size of the tree. For any given node in the update tree the range of its left child always covers data blocks at strictly lower indices than L, and the range of the right child always contains a range of data blocks with indices strictly larger than U. This allows us to efficiently balance the tree using standard algorithms such as that of AVL trees [2]. Furthermore, because insert and delete operations affect indices of the existing data blocks, in order to quickly determine (or verify) the indices of the stored data blocks after a sequence of updates, we store an offset value R with each node of the update tree which indicates how the ranges of the blocks stored in the subtree rooted at that node need to be adjusted. Lastly, for each range of blocks stored in the update tree, we record the number of times the blocks in that range have been updated. This information will allow the client to verify that the data she receives corresponds to the most recent version and integrity of the data (or, alternatively, to any previous version requested by the client).

At the initialization time, the client computes a MAC of each data block together with its index and version number (which is initially set to 0). The client stores the blocks and their corresponding MACs at the server. If no updates take place, the client will be able to retrieve a data block by its index number and verify its integrity. To support dynamic operations, the update tree is first initialized to empty. To modify a range of existing blocks, we insert a node in the tree that indicates that the version of the blocks in the range has increased. To insert a range of blocks, the client creates a node in the tree with the new blocks and also indicates that the indices of the blocks that follow need to be increased by the num-

ber of inserted blocks. A node's offset affects its entire subtree, which removes the need to touch many nodes. To delete a range of blocks, the deleted blocks are marked with operation type "-1" in the tree and the offset of blocks that follow is adjusted accordingly. Then to perform an update (insert, delete, or modify), the client first modifies the tree, computes the MACs of the blocks to be updated, and communicates the blocks (for insertion and modification only) and the MACs to the server. Upon receiving the request, the server also modifies the tree according to the request and stores the received data and MACs. If the server behaves honestly, the server's update tree will be identical to the client's update tree (i.e., all changes to the tree are deterministic). To retrieve a range of blocks, the client receives a number of data blocks and their corresponding MACs from the server and verifies their integrity by using information stored in the tree.

Update tree attributes. Before we proceed with the description of our scheme, we outline the attributes stored with each node of the update tree, as well as global parameters. Description of the update tree algorithms is deferred to Section 5.

With our solution, the client and the server maintain two global counters together with the update tree, GID and CID, both of which are initially set to 0. GID is incremented for each insertion operation to ensure that each insert operation is marked with a unique identifier. This allows the client to order the blocks that have been inserted into the same position of the file through different operations. CID is incremented for each commit operation and each commit is assigned a unique identifier. For a given data block, the combination of its version number and commit ID will uniquely identify a given revision of the block. In addition, each node in the update tree stores several attributes:

Node type Op represents the operation type associated with the node, where values -1, 0, and 1 indicate deletion, modification, and insertion, respectively.

Range L, U specifies the start and end indices of the data blocks, information about which is stored at the node.

Version number V indicates the number of modifications performed on the data blocks associated with the node. The version number is initially 0 for all data blocks (which are not stored in the update tree), and the version is also reset to 0 during a commit operation for all affected data blocks (at which point information about them is combined into a single node).

Identification number ID of a node has a different meaning depending on the node type. For a node that represents an insertion, ID denotes the value of GID at the time of the operation, and for a node that represents a modification or deletion, ID denotes the value of CID at the time of the last commit on the affected data blocks (if no commit operations were previously performed on the data blocks, the value is set to 0). In order to identify the type of ID (i.e., GID or CID) by observing its value, we use non-overlapping ranges for the values from which IDs for the two different types are assigned.

Offset R indicates the number of data blocks that have been added to, or deleted from, the range of data block indices that precede the range of the node (i.e., $[0, \mathsf{L}-1]$). The offset value affects all data blocks information about which is stored directly in the node as well as all data blocks information about which is stored in the right child subtree of the node.

Pointers $\mathsf{P_l}$ and $\mathsf{P_r}$ point to the left and right children of the node, respectively, and $\mathsf{P_p}$ points to the node's parent.

In addition to the above attributes, each node in the server's update tree also stores pointers to the data blocks themselves (and tags used for their verification).

Construction. We next provide the details of our construction. Because the solution relies on our update tree algorithms, we outline them first, while their detailed description is given in Section 5.

- $\mathsf{UTInsert}(\mathsf{T}, \mathsf{s}, \mathsf{e})$ inserts a range of new blocks into the update tree T, where the range starts from index s and consists of $(\mathsf{e}-\mathsf{s}+1)$ data blocks. It returns a node v that corresponds to the newly inserted block range.
- $\mathsf{UTDelete}(\mathsf{T}, \mathsf{s}, \mathsf{e})$ marks blocks in the range $[\mathsf{s}, \mathsf{e}]$ as deleted in the update tree T and adjusts the indices of the data blocks that follow. It returns an array of nodes C from T that correspond to the deleted data blocks.
- $\mathsf{UTModify}(\mathsf{T}, \mathsf{s}, \mathsf{e})$ updates the version of the blocks in the range $[\mathsf{s}, \mathsf{e}]$ in the tree T. If some of blocks in the range have not been modified in the past (and therefore are not represented in the tree), the algorithm inserts necessary nodes with version 1. The function returns all the nodes in T that correspond to the modified data blocks.
- $\mathsf{UTRetrieve}(\mathsf{T}, \mathsf{s}, \mathsf{e})$ returns the nodes in T that correspond to the data blocks in the range $[\mathsf{s}, \mathsf{e}]$.
- $\mathsf{UTCommit}(\mathsf{T}, \mathsf{s}, \mathsf{e})$ replaces nodes in T that correspond to the data blocks in the range $[\mathsf{s}, \mathsf{e}]$ with a single node and balances the remaining tree.

The protocols that define our solution are as follows:

1. $\mathsf{KeyGen}(1^\kappa) \rightarrow \{\mathsf{sk}\}$: C calls $\mathsf{sk} \leftarrow \mathsf{Gen}(1^\kappa)$.

2. $\mathsf{Init}(\langle \mathsf{sk}, m_1, \ldots, m_N \rangle, \langle \perp \rangle) \rightarrow \{\langle \mathsf{M}_C \rangle, \langle \mathsf{M}_S, D \rangle\}$: C and S initialize the update tree T to empty and set $\mathsf{M}_C = \mathsf{T}$ and $\mathsf{M}_S = \mathsf{T}$, respectively. For $1 \leq i \leq N$, C computes $t_i = \mathsf{Mac_{sk}}(m_i||i||0||0||0)$, where "||" denotes concatenation and the three "0"s indicate the version number, CID, and operation type, resp. C sends each $\langle m_i, t_i \rangle$ to S who stores this information in D.

3. $\mathsf{Update}(\langle \mathsf{sk}, \mathsf{M}_C, \mathsf{op}, \mathsf{ind}, \mathsf{num}, m_{\mathsf{ind}}, \ldots, m_{\mathsf{ind}+\mathsf{num}-1} \rangle, \langle \mathsf{M}_S, D \rangle) \rightarrow \{\langle \mathsf{M}'_C \rangle, \langle \mathsf{M}'_S, D' \rangle\}$: the functionality of this protocol is determined by the operation type op and is defined as:
   (a) **Insert** $\mathsf{op} = 1$**:** C executes $u \leftarrow \mathsf{UTInsert}(\mathsf{M}_C, \mathsf{ind}, \mathsf{ind}+\mathsf{num}-1)$.
   Delete $\mathsf{op} = -1$**:** C executes $U \leftarrow \mathsf{UTDelete}(\mathsf{M}_C, \mathsf{ind}, \mathsf{ind}+\mathsf{num}-1)$.
   Modify $\mathsf{op} = 0$**:** C executes $U \leftarrow \mathsf{UTModify}(\mathsf{M}_C, \mathsf{ind}, \mathsf{ind}+\mathsf{num}-1)$.
   C stores the updated update tree in M'_C.
   (b) For each $u \in U$ (or a single u in case of insertion), C locates the data blocks corresponding to the node's range from the m_i's, for $\mathsf{ind} \leq i \leq \mathsf{ind}+\mathsf{num}-1$, and computes $t_i \leftarrow \mathsf{Mac_{sk}}(m_i||u.\mathsf{L}+j||u.\mathsf{V}||u.\mathsf{ID}||\mathsf{op})$, where $j \geq 0$ indicates the position of the data block within the node's blocks. C sends op, ind, and num together with the $\langle m_i, t_i \rangle$ pairs to S, except that for deletions the data blocks themselves are not sent.
   (c) **Insert** $\mathsf{op} = 1$**:** S executes $u \leftarrow \mathsf{UTInsert}(\mathsf{M}_S, \mathsf{ind}, \mathsf{ind}+\mathsf{num}-1)$.
   Delete $\mathsf{op} = -1$**:** S executes $U \leftarrow \mathsf{UTDelete}(\mathsf{M}_S, \mathsf{ind}, \mathsf{ind}+\mathsf{num}-1)$.
   Modify $\mathsf{op} = 0$**:** S executes $U \leftarrow \mathsf{UTModify}(\mathsf{M}_S, \mathsf{ind}, \mathsf{ind}+\mathsf{num}-1)$.
   S stores the updated tree in M'_S and combines D with received data (using returned u or U) to obtain D'.

Recall that the protocol does not involve integrity verification for each dynamic operation, which removes a round of interaction between the client and server. Instead, the server records the operation in its metadata, which will be used for proving the integrity of returned blocks at retrieval time.

4. $\mathsf{Retrieve}(\langle \mathsf{sk}, \mathsf{M}_C, \mathsf{ind}, \mathsf{num}\rangle, \langle \mathsf{M}_S, D\rangle) \rightarrow \{\langle m_{\mathsf{ind}}, \ldots, m_{\mathsf{ind}+\mathsf{num}-1}\rangle, \langle \perp\rangle\}$:
   (a) C sends ind and num to S.
   (b) S executes $U \leftarrow \mathsf{UTRetrieve}(\mathsf{M}_S, \mathsf{ind}, \mathsf{ind}+\mathsf{num}-1)$. For each $u \in U$, S retrieves the attributes (L, U, and pointer to the data blocks) from u, locates the blocks and their tags $\langle m_i, t_i\rangle$ in D, and sends them to C.
   (c) Upon receiving $\langle m_i, t_i\rangle$, C executes $U \leftarrow \mathsf{UTRetrieve}(\mathsf{M}_C, \mathsf{ind}, \mathsf{ind}+\mathsf{num}-1)$. C chooses a random subset of data blocks of size $\min(c, \mathsf{num})$. For each chosen data block m_i, C locates the corresponding $u \in U$ and computes $b_i \leftarrow \mathsf{Verify}_{\mathsf{sk}}(m_i||u.\mathsf{L}+j||u.\mathsf{V}||u.\mathsf{ID}||u.\mathsf{Op}, t_i)$, where $j \geq 0$ is the data block position within the node's data blocks. If $b_i = 1$ for each verified m_i, C is assured of integrity of returned data.

5. $\mathsf{Challenge}(\langle \mathsf{sk}, \mathsf{M}_C\rangle, \langle \mathsf{M}_S, D\rangle) \rightarrow \{\langle m_{i_1}, \ldots, m_{i_c}\rangle, \langle \perp\rangle\}$:
   (a) C chooses c distinct indices $i_1, \ldots, i_c$ at random from the range $[1, N]$ and sends them to S.
   (b) If none of the indices are adjacent, S executes $U_j \leftarrow \mathsf{UTRetrieve}(\mathsf{M}_S, i_j, i_j)$ for each $j \in [1, c]$. Otherwise, S combines adjacent indices in ranges and executes UTRetrieve for each range.
   (c) For each U_j, S retrieves the attributes (L, U, and pointer to the data block) from M_S, locates the blocks and their tags $\langle m_{i_j}, t_{i_j}\rangle$ in D, and sends them to C.
   (d) Upon the receipt of c data blocks and their corresponding tags $\langle m_{i_1}, t_{i_1}\rangle, \ldots, \langle m_{i_c}, t_{i_c}\rangle$, C executes $U_j \leftarrow \mathsf{UTRetrieve}(\mathsf{M}_C, i_j, i_j)$ for each j (or for each range when some indices are adjacent). For each data block m_{i_j}, C verifies its tag using the same computation as in Retrieve.

6. $\mathsf{Commit}(\langle \mathsf{sk}, \mathsf{M}_C, \mathsf{ind}, \mathsf{num}, m_{\mathsf{ind}}, \ldots, m_{\mathsf{ind}+\mathsf{num}-1}\rangle, \langle \mathsf{M}_S, D\rangle) \rightarrow \{\langle \mathsf{M}'_C\rangle, \langle \mathsf{M}'_S, D'\rangle\}$:
   (a) C executes $u \leftarrow \mathsf{UTCommit}(\mathsf{M}_C, \mathsf{ind}, \mathsf{ind}+\mathsf{num}-1)$ and stores updated metadata in M'_C. C next computes $t_{\mathsf{ind}+i} \leftarrow \mathsf{Mac}_{\mathsf{sk}}(m_{\mathsf{ind}+i}||\mathsf{L}+i||0||\mathsf{CID}||0)$ for $0 \leq i \leq \mathsf{num}-1$, and sends the tags and parameters ind and num to S (the blocks are assumed to be unchanged).
   (b) S executes $u \leftarrow \mathsf{UTCommit}(\mathsf{M}_S, \mathsf{ind}, \mathsf{ind}+\mathsf{num}-1)$ and updates its metadata to M'_S. S updates the affected blocks' tags in D to obtain D'.

Public verifiability. To enable outsourcing of periodic challenge queries to a third party auditor, instead of choosing a private key sk, the client creates a public-private key pair (pk, sk). The client then replaces MAC computation with a signature produced using sk. An auditor with an up-to-date copy of the update tree will be able to perform challenge queries on behalf of the client and verify them using pk.

5. UPDATE TREE OPERATIONS

In this section we describe all operations on the new type of data structure, balanced update tree, that allow us to achieve attractive performance of the scheme. The need to maintain several attributes associated with a dynamic operation and the need to keep the tree balanced add complexity to the tree algorithms. Initially, the tree is empty and new nodes are inserted upon dynamic operations triggered by the client. All data blocks information about which is not stored in the tree have not been modified and their integrity can be verified by assuming version number and commit ID to be 0.

When traversing the tree with an up-to-date range $[\mathsf{s}, \mathsf{e}]$ of data blocks, the range will be modified based on the R value of the nodes lying on the traversal path. By doing that, we are able to access the original indices of the data blocks (prior to any insertions or deletions) to either correctly execute an operation or verify the result of a read request. We illustrate the tree operations on the example given in Figure 1, in which the leftmost tree corresponds to the result of three modify requests with the ranges given in the figure. We highlight modifications to the tree after each additional operation.

The first operation is an insertion, the range of which falls on left side of node A's range and overlaps with the range of node B. To insert the blocks, we partition B's range into two (by creating two nodes) and make node D correspond to an insertion ($\mathsf{Op} = 1$). Note that the offset R of node A is updated to reflect the change in the indices for the blocks that follow the newly inserted blocks. Furthermore, for the insertion operation, only the position at which the new blocks are inserted matters, not its range. For instance, executing an insertion with a block range [61, 300] or [61, 70] will have the same impact on the tree structure. The second operation is a modification, the range of which lies on the right to node A's range. When going down the tree, we modify the block range contained in the original request based on A's offset R (for the right child only), which now overlaps with node C's range. To accommodate the request, we increment the version of C's blocks and insert two new nodes with ranges before and after C's range. The last operation is a deletion, the range of which likewise fall on the right to A's range and the indices in the original request are adjusted. Because the adjusted range falls before all ranges in C's subtree, it is inserted as the left child of E_1 with type $\mathsf{Op} = -1$ and the offset R of both C and E_1 is adjusted to reflect the change in block indices for these nodes and their right children.

We first list sub-routines called by the main algorithms followed by an outline of main operations. Due to space constraints, pseudocode and detailed explanations are given in the full version.

Sub-routines:

$\mathsf{UTInsertNode}(\mathsf{u}, \mathsf{w}, \mathsf{dir})$ inserts a node w into a (sub-)tree rooted at node u. The routine is called only in the cases when after the insertion, w becomes either the leftmost ($\mathsf{dir} = \mathsf{left}$) or the rightmost ($\mathsf{dir} = \mathsf{right}$) node of the subtree.

When the node w is inserted into the left subtree of node u, the offset R of each node on the path should be updated according to the range of indices of w when the operation is insertion or deletion, because the new range lies to the left of the blocks of the current u. When the node w is inserted into the right subtree of node u, the range of node w should also be modified as it traverses the tree, since we need to store the original indices of data blocks .

$\mathsf{UTFindNode}(\mathsf{u}, \mathsf{s}, \mathsf{e}, \mathsf{op})$ searches the tree rooted at node u for a node corresponding to block range $[\mathsf{s}, \mathsf{e}]$ for the purposes of executing operation op on that range.

After the function is invoked on range $[\mathsf{s}, \mathsf{e}]$ and that range does not overlap with the ranges of any of the existing nodes, the function creates a new node and returns it. Otherwise, the function needs to handle the case of range overlap, defined as follows: (i) op is insertion and the index s lies within the range of a tree node or (ii) op is modification or deletion and the range $[\mathsf{s}, \mathsf{e}]$ overlaps with the range of at least one existing tree node.

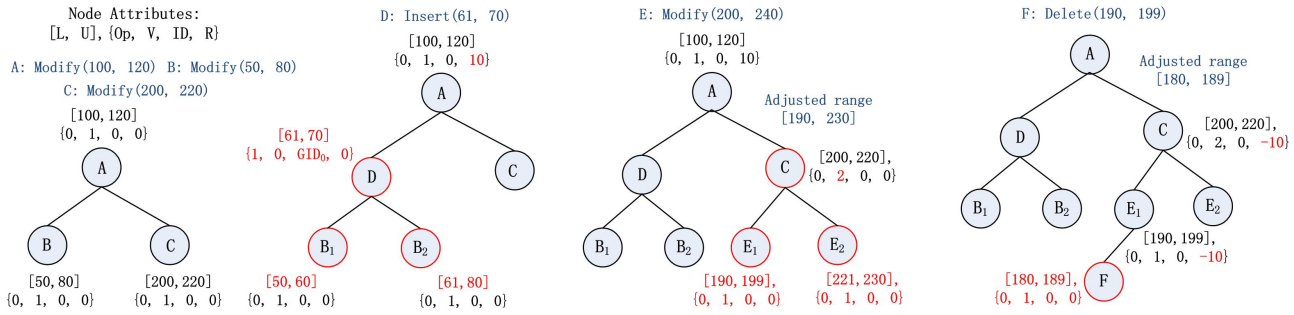


Figure 1: Example of update tree operations.

The tricky part of the algorithm is to avoid returning nodes that correspond to deleted block ranges. If such a node is found, we should ignore it and keep searching until we find a node that represents either an insertion or modification operation. The function can be invoked for any dynamic operation and takes $O(\log n)$ time.

UTUpdateNode$(\mathsf{u}, \mathsf{s}, \mathsf{e}, \mathsf{op})$ is called by a modification or deletion routine on a sub-tree rooted at node u when the range $[\mathsf{s}, \mathsf{e}]$ of data blocks needs to be updated and falls into the range of u.

The function handles four different situations based on the type of intersection of ranges $[\mathsf{s}, \mathsf{e}]$ and $[\mathsf{u.L}, \mathsf{u.U}]$. If the two ranges are identical several attribute of u (i.e., V, ID and Op) will be reset with values that depend on the operation type. If only the lower (only the upper) bound of the two ranges coincide, we reset the range of the current root node to $[\mathsf{s}, \mathsf{e}]$, fork a new node corresponding to the remaining range, and insert it into the right (resp., left) subtree of current root node. If neither the lower nor the upper bound matches with each other, we fork two child nodes corresponding to the head and tail remaining ranges, and insert each of them into the left or right subtree of current root node respectively. As can be expected, the node generated with the remaining range will become either a leftmost or a rightmost node of the subtree, and we use UTInsertNode sub-routine to achieve it. The time complexity of the sub-routine is $O(\log n)$.

UTBalance(u) balances the tree rooted at node u using AVL trees method and returns the root of a balanced structure. This function will only be called on trees both direct child sub-trees of which are already balanced rather than on arbitrarily unbalanced trees. The time complexity of this function is linear in the height difference of u's child sub-trees.

Main routines:

UTInsert$(\mathsf{T}, \mathsf{s}, \mathsf{e})$ updates the tree T for an insert request with the block range $[\mathsf{s}, \mathsf{e}]$ by inserting a node in the tree.

The main functionality of the routine is (i) to find a position for node insertion (using UTFindNode sub-routine), and (ii) to insert a new node into the tree. When the range $[\mathsf{s}, \mathsf{e}]$ does not overlap with any existing nodes, UTFindNode inserts a new node into the tree and no other action is necessary. Otherwise, an existing node u' that overlaps with $[\mathsf{s}, \mathsf{e}]$ is returned and determines the number of nodes that need to be created. That is, if the (adjusted) insertion position s equals to the lower bound of u', u' is substituted with a new node and is inserted into the right subtree of the new node. Otherwise, u' is split into two nodes, which are inserted into the left and right subtrees of u', respectively while u' itself is set to correspond to the insertion. The insert request in Figure 1 corresponds to the scenario.

UTModify$(\mathsf{u}, \mathsf{s}, \mathsf{e})$, when called with $\mathsf{u} = \mathsf{T}$, updates the tree T based on a modification request with the block range $[\mathsf{s}, \mathsf{e}]$ and returns the set of nodes that correspond to the range. The algorithm creates a node for the range if T is empty, and otherwise invokes UTFindNode to locate the positions of nodes to be modified. After finding them, the algorithm distinguishes between three cases based on how the (adjusted) range $[\mathsf{s}, \mathsf{e}]$ overlaps with the range of a found node u_i (i.e., $[\mathsf{u}_i.\mathsf{L}, \mathsf{u}_i.\mathsf{U}]$):

1. If the adjusted range is contained in u_i's range, u_i is the only node to be modified, and this is handled by UTUpdateNode. The modify request in Figure 1 corresponds to the scenario.
2. If the adjusted range overlaps with the ranges of u_i and its one subtree, the algorithm updates the range of u_i and then recursively calls itself to update the remaining nodes.
3. If the adjusted range overlaps with the range of u_i and both of its subtrees, the algorithm updates u_i and calls UTModify twice to handle changes to its child subtrees.

UTDelete$(\mathsf{u}, \mathsf{s}, \mathsf{e})$, when called with $\mathsf{u} = \mathsf{T}$, updates the update tree T based on a deletion request with the block range $[\mathsf{s}, \mathsf{e}]$. It does not delete any node from T, but rather finds all nodes whose ranges fall into $[\mathsf{s}, \mathsf{e}]$, sets their operation types to -1, and returns them to the caller. UTDelete works similar to UTModify.

UTRetrieve$(\mathsf{u}, \mathsf{s}, \mathsf{e})$, when called with $\mathsf{u} = \mathsf{T}$, returns the nodes whose ranges fall into $[\mathsf{s}, \mathsf{e}]$. Its high-level structure follows that of UTModify.

UTCommit$(\mathsf{T}, \mathsf{s}, \mathsf{e})$ replaces all nodes in tree T whose ranges falls into $[\mathsf{s}, \mathsf{e}]$ with a single node with the range $[\mathsf{s}, \mathsf{e}]$. The goal of a commit operation is to reduce the tree size, but in the process it may become unbalanced or even disconnected. Thus, to be able to maintain the desired performance guarantees, we must restructure and balance the remaining portions of the tree. To achieve that, we first search for two nodes that contain the lower and upper bounds s and e, respectively, and make the adjusted s and e (denoted s' and e', respectively) become the left or right bound of the nodes that contain them. We then traverse T from the two nodes to their least common ancestor T', remove the nodes with ranges falling into the range $[\mathsf{s}', \mathsf{e}']$, and apply UTBalance sub-routine to balance the tree if necessary. Lastly, we traverse T from T' to the root, and balance the tree if necessary. We also add a node with $[\mathsf{s}, \mathsf{e}]$ and new CID. The routine returns adjusted lower bound s' and updated CID.

To illustrate how the tree is being traversed and balanced in the process, let us consider an example in Figure 2. In the figure, u_1 and u_2 correspond to the tree nodes that incorporate the smallest and largest block indices falling in the commit range (i.e., s and e), respectively, and T' is their lowest common ancestor. The nodes and their subtrees shown using dotted lines corresponds to the nodes whose entire subtrees are to be removed. To remove all nodes in the subtree of T' with block indices larger than adjusted s (located in the left child's subtree), we traverse the path from u_1 to T'. Ev-

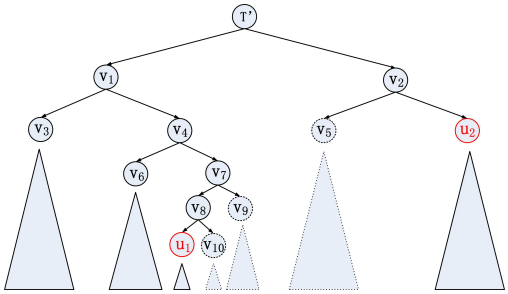


Figure 2: Illustration of the commit algorithm.

ery time u_1 is the left child of its parent, we remove u_1's right sibling and its subtree, remove u_1's parent node, and make u_1 take the place of its parent. For the example in the figure, it means that nodes v_{10} and v_9 are removed together with their subtrees, nodes v_8 and v_7 are also removed, and u_1 takes the place of v_7. At this point u_1 becomes the right child of its parent, and we balance the subtree rooted at u_1's parent and make u_1 point to its parent node (by calling $\mathsf{UTBalance}(\mathsf{u})$ sub-routine). This rebalancing procedure is continued by traversing up to v_7, v_4, and v_1, until the left subtree of T' is completely balanced.

The same process applies to the right child's tree of T' that contain u_2 with the difference that node removal is performed when u_2 is the right child of its parent and rebalancing is performed when u_2 is the left child of its parent. For the example in Figure 2, we obtain that node v_5 is removed together with its subtree, node v_2 is removed, and u_2 takes the place of v_2.

The last step that remains is to rebalance the subtree rooted at T' and the subtrees of all other nodes on the path from T' to the root. This is accomplished by making T' point to its parent after each rebalancing procedure. We obtain a balanced tree T with all nodes in the range $[\mathsf{s}, \mathsf{e}]$ removed and insert one single node corresponding to this range that indicates that the commit number CID of all blocks in the range $[\mathsf{s}, \mathsf{e}]$ has been increased.

6. ANALYSIS OF THE SCHEME

Complexity analysis. In what follows, we analyze the complexity of main update tree algorithms and the protocols that define the scheme. Each $\mathsf{UTInsert}$ adds one or two nodes to the tree, and all operations are performed during the process of traversing the tree. Therefore, its time complexity is $O(\log n)$, where n is the current number of nodes in the tree. Both $\mathsf{UTModify}$ and $\mathsf{UTDelete}$ can add between 0 and $O(\min(n, \mathsf{e} - \mathsf{s}))$ nodes to the tree, but as our experiments suggest, a constant number of nodes is added per range on average. Their time complexity is $O(\log n + \min(n, \mathsf{e} - \mathsf{s}))$, and both the size of the block range and the number of nodes in the tree form the upper bound on the number of returned nodes. $\mathsf{UTRetrieve}$ does not add nodes to the tree and its complexity is also $O(\log n + \min(n, \mathsf{e} - \mathsf{s}))$. Lastly, $\mathsf{UTCommit}$ removes between 0 and $O(\min(n, \mathsf{e} - \mathsf{s}))$ nodes from the tree and its time complexity is $O(\log n + \min(n, \mathsf{e} - \mathsf{s}))$. While the function calls $\mathsf{UTBalance}$, the worst case complexity of which is $O(\log n)$, at most $O(\log n)$ number of times, due to the careful construction of the tree and the commit function, we are able to achieve $O(\log n)$ node rearrangement time (plus, node deallocation time). This is due to the fact that balancing a tree, both subtrees of which are themselves balanced, but their heights differ by a constant, requires only a constant number of operations. A detailed proof can be found in the full version.

Next, we analyze the complexity of the protocols themselves. It is clear that Init has time and communication complexity of N, i.e., the number of transmitted blocks. Update for any operation type has time complexity of $O(\log n + \mathsf{num})$ and communication complexity of $O(\mathsf{num})$. $\mathsf{Retrieve}$ has the same complexities as Update, unless it is used for integrity verification rather than block retrieval. In the latter case, its computation and communication complexities become $O(\log n + \min(\mathsf{num}, c))$ and $O(\min(\mathsf{num}, c))$, respectively, where constant c bounds the number of $\langle m_i, t_i \rangle$ pairs used for the purpose of probabilistic verification. Lastly, the complexities of Commit are $O(\log n + \mathsf{num})$ and $O(\mathsf{num})$, because the client needs to communicate num MACs to the server.

Security analysis. Security of our scheme can be shown according to the definition of DPDP in Section 3.

THEOREM 1. *The proposed update tree scheme is a secure DPDP scheme assuming the security of MAC scheme.*

PROOF SKETCH. Suppose that the adversary $\mathcal{A}$ wins the data possession game with a non-negligible probability. Then the challenger can either extract the challenged data blocks (i.e., if $\mathcal{A}$ has not tampered with them) or break the security of the MAC scheme (i.e., if $\mathcal{A}$ tampered with the data). In particular, in the former case, the challenger can extract the genuine data blocks from $\mathcal{A}$'s response. In the latter case, if the adversary tampers with a data block (by possibly substituting it with a previously stored data for the same or a different block), it will have to forge a MAC for it, which the challenger can use to win the MAC forgery game. This is because our solution is designed to ensure that any two MACs communicated by the client to the server are computed on unique parameters. That is, two different versions of the same data block i will have either their version, CID, or operation type differ, while two different blocks that at different points in time assume the same index i (e.g., a deleted block and a block inserted in its place) can be distinguished by the value of their ID (i.e., at least one of them will have a GID, and two GIDs or a GID and CID are always different). □

The probability that a cheating server is caught on a $\mathsf{Retrieve}$ or $\mathsf{Challenge}$ request of size $\mathsf{num} < c$ is 1, and otherwise the probability is $1 - ((\mathsf{num} - t)/\mathsf{num})^c$, where t is the number of tampered blocks among the challenged blocks.

7. ENABLING REVISION CONTROL

In this section, we sketch how our scheme can be extended to support revision control. The exact algorithms are omitted and can be found in the full version. To enable versioning functionality, we need to specify (i) how a user can retrieve a specific version of a data block or range and (ii) how a user can retrieve deleted data blocks (prior to a commit on them, which permanently removes them from the server).

Specific version retrieval can be realized by modifying the $\mathsf{Retrieve}$ protocol to add version number V to the set of parameters sent to the server with the request. After receiving the request, the server executes $\mathsf{UTRetrieve}$ as usual, but returns to the client the data blocks and their tags that correspond to version V. To verify the response, the client verifies the returned tags using the intended version V and other attributes obtained from $\mathsf{UTRetrieve}$.

Deleted data retrieval can be realized by extending the $\mathsf{Retrieve}$ protocol's interface with a flag that indicates that deleted data is to be returned and modifying $\mathsf{UTRetrieve}$ that currently skips all deleted data. The difficulty in specifying what deleted data to retrieve is caused by the fact that deleted blocks no longer have indices associated with them. To remedy the problem, we propose to

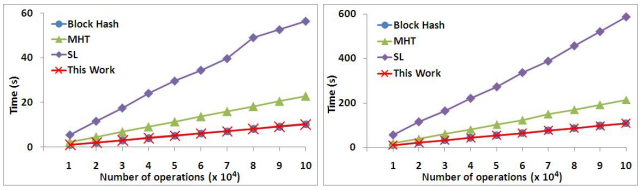
specify in the request a range [s, e] that contains the desired deleted range and includes one or more non-deleted blocks before and after the deleted range being requested. We also need to use an alternative UTRetrieve function for retrieval of deleted data, which instead of ignoring nodes that represent deletions will return them as the output and will allow the server to locate the deleted blocks and their tags.

8. PERFORMANCE EVALUATION

To evaluate performance of our scheme and provide a comparison with prior solutions, we designed experiments which measure the computation, communication, and storage requirements of three different schemes. The schemes that we compare are: (i) our update tree (UTree) solution, (ii) the solution based on Merkle hash tree (MHT) [27], and (iii) the solution based on skip lists (SL) [16, 19]. The asymptotic complexities of the schemes being compared are given in Table 1. The table provides storage complexities for both the server (in addition to the data blocks themselves, i.e., space for maintaining metadata) and the client, as well as computation complexities per operation. For a retrieve, update, insert, or delete operation, it is assumed that the operation is executed on a range consisting of t blocks. For MHT and SL schemes, the integrity of a dynamic operation on t blocks is verified by executing the appropriate verification procedure for each of the t blocks. Verification of retrieve and challenge operations is assumed to be probabilistic in all schemes. For a retrieve operation, $\min(c, t)$ blocks are verified for constant c, while transmitting all t blocks to the client. The complexity of the challenge query of the *entire outsourced storage* is that of verifying c data blocks for constant c. In the table, N denotes the number of data blocks stored at the server, and M denotes the number of dynamic operations on the stored blocks. The worst-case complexity of an operation with the MHT-based solutions is linear in the size of the repository because after arbitrary insertions and deletions the height of the tree is $O(N)$ in the worst case.

We evaluate the performance of the schemes in three different settings: the first uses 1GB of outsourced storage with 4KB data blocks, the second uses 256GB of storage with 4KB blocks, and the third uses 256GB of storage with 64KB blocks. The first 1GB+4KB setting was chosen for consistency with experiments in prior work and the other two allow us to examine the systems' behavior when one parameter remains fixed while the other changes (i.e., 4KB block size in the first two settings and 256GB overall storage in the last two settings). As another important observation about the chosen settings, notice that the number of blocks are 2^{18}, 2^{26}, and 2^{22}, respectively, which allows us to test the performance with respect to its dependence on the number of outsourced data blocks. In situations when the amount of outsourced storage is significantly larger than in our experiments, we expect the data block size to become larger as well leaving the number of blocks within the range that we evaluate. Lastly, we evaluate the performance of the schemes in the realistic scenarios by utilizing the file traces gathered by Microsoft Research data centers for a week time period. We believe the access patterns we observed in the trace are representative of a large number of small to medium size enterprise data centers.

We implement our and MHT schemes in C, while the SL scheme was implemented as in [16] in Java. Despite the programming language difference, the time to compute a hash is similar in both implementations. Because the overall computation of the SL scheme is dominated by hash function evaluation, we consider the performance of all implementations to be comparable. We use SHA-224 for hash function evaluation and HMAC with SHA-224 for MAC computation. The experiments were run on 2.4GHz Linux machines (both the client and the server).



(a) single-block mixed operations (b) multiple-block mixed operations

Figure 3: Aggregate client's computation time after n operations with 1GB outsourced storage and 4KB block.

Computation. To evaluate computation, we measure the client's time after executing n client's requests for n between 10^4 and 10^5. The server's overhead in all schemes is similar to that of the respective client's overhead. The initial cost of building the data structures in MHT and SL schemes or computing MACs in our solution is not included in the measured times.

In the first experiment, we choose one of four operations (insert, delete, modify, and retrieve) at random and execute it on a randomly selected single data block. From the schemes that we compare, only our solution provides a natural support for querying ranges of blocks (the SL solution in [16] can provide a limited support for block ranges as previously described). Then because in practice accesses are often consecutive in their nature (see, e.g., [15]), the performance of our scheme is expected to be even better in practice. For all of the above operations except deletion, the client needs to compute a hash (or MAC) of the data block used in the request. Because this computation is common to all three schemes[2], we separately measure it and also provide the times for the remaining processing that the client needs to do.

Because of drastic differences in performance of the schemes, we present many results in tables instead of displaying them as plots. This allows us to convey information about the growth of each function. For that reason, Figure 3(a) plots the aggregate and Figure 4 plots average performance of all schemes for the first 1GB+4KB setting, while Table 2 provides average computation for all three settings. Notice that in the figure the overhead of each scheme is added to the common hash computation work, while in the table the common and additional computation are shown separately. We were unable to complete 256GB+4KB experiments for SL due to its extensive computation (primarily to build the data structure) and memory requirements. As can be seen from the results, the overhead of UTree scheme is 2 to 3 orders of magnitude lower for the settings used in the experiment. The majority of the total work in a UTree scheme comes from MAC computation, while in MHT and SL the proof often dominates the cost. Also note that the overhead of SL is larger than that of MHT due to the use of longer proofs and commutative hashing in the former, where the majority of the difference comes from the hashing. As expected, the number of data blocks in the storage affects performance of MHT and SL schemes (the proof sizes of which are logarithmic in the total number of blocks), while the average time per operation remains near a constant for each setting. In our scheme, on the other hand, the time grows slowly with the number of operations, but does not increase with the total storage size.

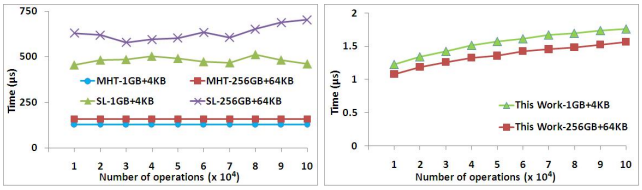
[2]In MHT and SL schemes the client needs to compute the hash of a data block, while in our scheme the client computes a MAC of it. Because MAC computation is slightly more expensive than the hash, we include MAC's additional overhead into the performance of our scheme.

Scheme	Cost per operation (Server and Client)								Storage	
	Update or Insert		Delete		Retrieve		Challenge		Server	Client
	Computation	Communic.	Computation	Communic.	Computation	Communication	Computat.	Communic.		
MHT [27]	$O(Nt)$	$O(Nt)$	$O(Nt)$	$O(Nt)$	$O(\min(c,t)N+t)$	$O(\min(c,t)N+t)$	$O(cN)$	$O(cN)$	$O(N)$	$O(1)$
SL [16, 19]	$O(t\log N)$	$O(t\log N)$	$O(t\log N)$	$O(t\log N)$	$O(\min(c,t)\log N+t)$	$O(\min(c,t)\log N+t)$	$O(c\log N)$	$O(c\log N)$	$O(N)$	$O(1)$
This work	$O(\log M+t)$	$O(t)$	$O(\log M+t)$	$O(1)$	$O(\log M+t)$	$O(t)$	$O(c\log M)$	$O(c)$	$O(N)$	$O(M)$

Table 1: Asymptotic complexities of DPDP schemes.

Setting	File size	Block size	Scheme	Total number of operations n					Block hash
				20000	40000	60000	80000	100000	
Single-block mixed random operations	1GB	4KB	UTree	1.34	1.51	1.62	1.70	1.77	134
			MHT	127	127	127	127	127	
			SL	481	502	473	512	462	
	256GB	64KB	UTree	1.19	1.33	1.43	1.49	1.57	1972
			MHT	148	147	148	147	148	
			SL	619	595	633	652	703	
	256GB	4KB	UTree	1.20	1.32	1.43	1.50	1.56	134
			MHT	180	179	179	178	183	
Multi-block mixed random operations	1GB	4KB	UTree	1.68	2.14	2.48	2.78	2.99	1340
			MHT	944	978	1000	1050	1090	
			SL	4750	4460	4570	4640	4800	
Single block same position insertions	1GB	4KB	UTree	1.13	1.17	1.21	1.26	1.76	134
			MHT	77400	155000	232000	308000	466000	
			SL	250	235	242	258	212	

Table 2: Average client's computation time per operation measured after n operations for various schemes in μsec.



(a) MHT and SL (b) UTree

Figure 4: Average client's computation time measured after n single-block randomly chosen operations with 1GB storage and 4KB block (without block hash computation).

File size	Block size	Scheme	Total number of operations n				
			20000	40000	60000	80000	100000
1GB	4KB	UTree	0.4	0.8	1.2	1.6	2
		MHT	9.6	19.3	28.9	38.6	48.3
		SL	21	39.6	62.2	77.5	98.3
		Data	60	120	180	240	300
256GB	64KB	UTree	0.4	0.8	1.2	1.6	2
		MHT	11.7	23.5	35.3	47.0	58.8
		SL	25.1	54.7	79.2	101	126
		Data	960	1920	2880	3840	4800
256GB	4KB	UTree	0.4	0.8	1.2	1.6	2
		MHT	13.9	27.8	41.7	55.5	69.4
		Data	60	120	180	240	300

Table 3: Aggregate communication size after n operations for various schemes measured in MB.

For the second experiment, we changed the first experiment to execute each operation on a range of data blocks of size between 1 and 20. To be able to verify correctness of individual blocks, we do not implement the variable-sized data block approach suggested in [16], but rather repeat the single-block operation multiple times for each operation. However, to improve efficiency of MHT scheme, for a range insertion operation we construct an independent tree from the blocks specified in the request first and then insert it into the MHT in the same way as a single node. The performance results are given in Table 2 and Figure 3(b).

Compared to the single-block operations, performance of MHT and SL schemes deteriorates by a factor of 9–10, while for our UTree, which was designed to work on ranges, there is only a modest (20%–70%) increase in the performance. The increase in the tree size can be explained by using more nodes to partition an existing node. We expect that in the other settings with a larger number of blocks (256GB+4KB and 256GB+64KB), the tree will have a smaller size as the ranges are spread out over a larger space.

For the third experiment, we used a new access pattern that inserts data blocks at the same position in a file. The goal is to demonstrate the effect of such operation on an unbalanced data structure. The results are shown in Table 2. As we can see, UTree and SL schemes exhibit stable performance due to their balanced data structures, while performance of MHT grows significantly and linearly with the number of operations (as its height in that case exhibits linear in the number of operations growth). There is no easy way to remedy the situation by balancing MHT in a similar way to our solution, as the client does not have complete information about the MHT.

Communication. To evaluate communication, we measure the amount of data exchanged between the client and the server after executing a number of single-block requests. The data transferred in each operation consists of a data block (except for deletion) and corresponding auxiliary data. The former cost is common to all schemes, while the latter varies in its format and size. In particular, for UTree the auxiliary data consists of a single MAC, while for MHT and SL it is the proof linear in the height of the data structure. Another difference is that UTree involves a unidirectional communication for all except one operation (i.e., Retrieve which returns a response), while all operations in MHT and SL require bidirectional communication. For that reason, we measured the aggregated data exchanged for each operation, without considering the direction of data transfer. The results are given in Table 3.

Because deletion does not involve data block transfer in all three schemes, the average size of data block communication per oper-

ation is $3/4$ of the block size. As can be observed from Table 2, UTree scheme's communication is independent of the data structure size or the setting and is constant per operation. For MHT and SL scheme, on the other hand, performance depends on the data structure size. For data blocks of small size, the proof overhead of MHT and SL schemes constitutes a significant portion of the overall communication volume (14–30%), which could be a fairly large burden for a user constrained by a limited network bandwidth. The overhead of UTree scheme, on the other hand, is no more than 0.6%. Lastly, the difference in performance of MHT and SL schemes can be explained by the length of the proof and the size of elements within the proof.

Storage. To evaluate storage, we measure the size of data structures after executing client's requests on single blocks as well as ranges. In both MHT and SL schemes, the server maintains a data structure while the client keeps only constant-sized data for integrity verification. In our scheme, both the server and the client maintain a data structure, but it should be moderate in size for a variety of operating environments (and can be reduced using commit).

The data structures maintained in the schemes consists of a static portion that corresponds to the initially uploaded data and a dynamic portion that corresponds to dynamic operations issued afterwards. In MHT and SL schemes, the static component is linear in the number of outsourced blocks and is expected to be fairly large. In our scheme, on the other hand, there is no static component.

As far as dynamic component goes, the size of the data structure in our solution grows upon executing dynamic operations according to the analysis in Section 6. The growth is always constant per single-block operation, and the use of ranges allows us to reduce the overall growth. With MHT and SL schemes, the size of the data structure remains at the same level as long as the number of insertions is similar to the number of deletions. Block modifications do not affect the data structure size. Lastly, because MHT and SL scheme do not support versioning functionality, to enable it, they can be upgraded using persistent authenticated data structure [3]. The use of persistent data structures increase the data structure size by $O(\log n)$ per single-block update, where n is the number of nodes within the data structure. Therefore, considering both static and dynamic components, UTree inevitably leads to a more compact data structure, and its size is also the reason why the client can store the data structure locally.

The results of our experiments are given in Table 4. For each setting, the first row for UTree corresponds to single-block mixed dynamic operations at random locations, while the second row corresponds to similar range operations (1–20 blocks per operations). The performance is estimated based on the number of nodes (measured using UTree, MHT, and SL implementations) in the data structures and the approximate node size of 50 bytes for each scheme. It does not correspond to the memory measurement at the run time. Clearly, there is a large difference in the performance of our scheme and other approaches for the tested settings.

The experiments correspond to the original solutions, without the support of versioning functionality. This means that a deletion operation actually deletes a node and a modification does not contribute additional nodes to the data structure, and the size of the data structure remains constant after executing an equal number of different types of updates. As expected, the size of UTree grows linearly with the number of dynamic operations. Another observation that aligns with our experiments above is that the additional UTree size of range operations compared to the single-block case is significantly reduced with a larger number of outsourced blocks (compare, e.g., 112%, 0.8%, and 1.1% overhead at 100000 operations). As before, it is caused by fewer range overlaps, which results in fewer node partitioning and smaller tree size.

File size	Block size	MHT any n	SL any n	UTree for n operations				
				20000	40000	60000	80000	100000
1GB	4KB	25	24	0.70	1.37	2.03	2.66	3.28
				0.97	2.26	3.74	5.32	6.94
256GB	64KB	400	391	0.71	1.43	2.15	2.85	3.57
				0.71	1.46	2.15	2.89	3.60
256GB	4KB	6400	6206	0.71	1.43	2.15	2.86	3.57
				0.72	1.44	2.16	2.88	3.61

Table 4: The size of data structures for various schemes after n single-block or range mixed operations in MB.

Volume	Max offset	Operations ($\times 10^6$)	MHT	SL	UTree
Proj-0	170 GB	4.2	6400	21000	8.4
Proj-1	880 GB	24	29000	95000	1200
Proj-2	880 GB	29	48000	120000	2200
Proj-3	240 GB	2.2	670	2300	3
Proj-4	240 GB	6.5	3400	12000	150

Table 5: Client's aggregate computation time measured in seconds for each volume (without block hash computation).

Real life data. In the above experiments, we evaluated the efficiency of the schemes on synthetic randomly generated data. To provide more convincing results, we also conduct experiments on real life data sets from [14], which consist of file traces gathered from Microsoft data centers for a period of a week. We believe that the access patterns in the traces are representative of data usage types seen in practice. The traces were collected per volume below the file system cache and capture all block-level reads and writes performed on 36 volumes. Out of these volumes, for our experiments we select five that belong to a single server (a research project server) and contain 66 million events. Each event contains information such as a timestamp, a disk number, the start logical block number (i.e., offset), the number of blocks transferred, and its type (i.e., read or write).

For the purposes of MHT and SL schemes, we find the maximum offset that appears in the trace of a volume, consider it as the size of "outsourced data," and use it construct the corresponding data structure. (In contrast, the operation of our scheme does not need that information.) We then map a "read" or "write" operation in a event will to a respective "retrieve" or "modify" operation in all three schemes. Because there are no insertions or deletions, each operation takes the same amount of time in MHT and SL schemes.

Because the communication overhead of our scheme is always smaller than those of MHT and SL schemes in all cases, here we concentrate on computation and storage overhead. As before, we leave out the initial cost of building the data structures in MHT and SL schemes and exclude the time for computing MACs for our solution. We also do not measure the time of computing a hash (or a MAC) of a data block in the request. The results are presented in table 5 and assume a block size of 4KB. As can be observed from the table, out of five volumes our solution is almost two orders of magnitude faster than the other two schemes. For the storage overhead, we measure the data structure size after executing all operations appeared in a file trace. In case of MHT and SL schemes, the data structures depend on the size of outsourced data and range from 4 to 22GB. In our scheme, after executing all requests in a volume, the data structure size ranges from 1 to 150MB.

9. CONCLUSIONS

We propose a novel solution to provable data possession with support for dynamic operations, access to shared data by multiple users, and revision control. Our solution utilizes a new type of data structure that we term balanced update tree. Unique features of our scheme include orders of magnitude faster than in other schemes data verification and removing the need for the server to maintain data structures linear in the size of the outsourced data. The advantages come at the cost of requiring the client to maintain a data structure of modest, but non-constant size.

10. ACKNOWLEDGMENTS

This work was supported in part by grants FA9550-09-1-0223 from the Air Force Office of Scientific Research and CNS-1223699 from the National Science Foundation. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the funding agencies.

11. REFERENCES

[1] IT cloud services user survey, pt. 2: Top benefits & challenges. http://blogs.idc.com/ie/?p=210.

[2] G. Adelson-Velskii and E.M. Landis. An algorithm for the organization of information. In *Proceedings of the USSR Academy of Sciences*, pages 263–266, 1962.

[3] A. Anagnostopoulos, M. Goodrich, and R. Tamassia. Persistent authenticated dictionaries and their applications. In *International Conference on Information Security (ISC)*, pages 379–393, 2001.

[4] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *ACM Conference on Computer and Communications Security (CCS)*, pages 598–609, 2007.

[5] G. Ateniese, R. Di Pietro, L. Mancini, and G. Tsudik. Scalable and efficient provable data possession. In *Security and Privacy in Communication Networks (SecureComm)*, 2008.

[6] G. Ateniese, S. Kamara, and J. Katz. Proofs of storage from homomorphic identification protocols. In *Advances in Cryptology – ASIACRYPT*, pages 319–333, 2009.

[7] J. Bentley. Decomposable searching problems. *Information Processing Letters*, 8(5):244–251, 1979.

[8] K. Bowers, A. Juels, and A. Oprea. HAIL: A high-availability and integrity layer for cloud storage. In *ACM Conference on Computer and Communications Security (CCS)*, pages 187–198, 2009.

[9] K. Bowers, A. Juels, and A. Oprea. Proofs of retrievability: Theory and Implementation. In *ACM Workshop on Cloud Computing Security (CCSW)*, pages 43–54, 2009.

[10] E. Chang and J. Xu. Remote integrity check with dishonest storage server. In *European Symposium on Research in Computer Security (ESORICS)*, pages 223–237, 2008.

[11] R. Curtmola, O. Khan, R. Burns, and G. Ateniese. MR. PDP: Multiple-replica provable data possession. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 411–420, 2008.

[12] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. Interval trees. In *Computational Geometry*, chapter 10.1, pages 212–217. Springer-Verlag, second edition, 2000.

[13] Y. Dodis, S. Dadhan, and D. Wichs. Proofs of retrievability via hardness amplification. In *Theory of Cryptography Conference (TCC)*, pages 109–127, 2009.

[14] N. Dushyanth, D. Austin, and R. Antony. Write off-loading: Practical power management for enterprise storage. *Transactions on Storage*, 4(3):10:1–10:23, 2008.

[15] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive NFS tracing of email and research workloads. In *USENIX Conference on File and Storage Technologies (FAST)*, 2003.

[16] C. Erway, A. Kupcu, C. Papamanthou, and R. Tamassia. Dynamic provable data possession. In *ACM Conference on Computer and Communications Security (CCS)*, pages 213–222, 2009.

[17] M. Goodrich, C. Papamanthou, R. Tamassia, and N. Triandopoulos. Athos: Efficient authentication of outsourced file systems. In *International Conference on Information Security*, pages 80–96, 2008.

[18] M. Goodrich, R. Tamassia, and A. Schwerin. Implementation of an authenticated dictionary with skip lists and commutative hashing. In *DARPA Information Survivability Conference and Exposition*, pages 68–82, 2001.

[19] A. Heitzmann, B. Palazzi, C. Papamanthou, and R. Tamassia. Efficient integrity checking of untrusted network storage. In *ACM International Workshop on Storage Security and Survivability (StorageSS)*, pages 43–54, 2008.

[20] A. Juels and B. Kaliski. PORs: Proofs of retrievability for large files. In *ACM Conference on Computer and Communications Security (CCS)*, pages 584–597, 2007.

[21] A. Oprea and M. Reiter. Integrity checking in cryptographic file systems with constant trusted storage. In *USENIX Security Symposium*, pages 183–198, 2007.

[22] R. Popa, J. Lorch, D. Molnar, H. Wang, and L. Zhuang. Enabling security in cloud storage SLAs with CloudProof. In *USENIX Annual Technical Conference*, pages 355–368, 2011.

[23] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33:668–676, 1990.

[24] F. Sebe, J. Domingo-Ferrer, A. Martinez-Belleste, Y. Deswarte, and J.-J. Quisquater. Efficient remote data possession checking in critical information infrastructures. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 20(8):1034–1038, 2008.

[25] H. Shacham and B. Waters. Compact proofs of retrievability. In *Advances in Cryptology – ASIACRYPT*, pages 90–107, 2008.

[26] C. Wang, Q. Wang, K. Ren, and W. Lou. Ensuring data storage security in cloud computing. In *International Workshop on Quality of Service*, pages 1–9, 2009.

[27] Q. Wang, C. Wang, J. Li, K. Ren, and W. Lou. Enabling public verifiability and data dynamics for storage security in cloud computing. In *European Symposium on Research in Computer Security (ESORICS)*, pages 355–370, 2009.

[28] L. Wei, H. Zhu, Z. Cao, W. Jia, and A. Vasilakos. SecCloud: Bringing secure storage and computation in cloud. In *ICDCSW*, pages 52–61, 2010.

[29] K. Zeng. Publicly verifiable remote data integrity. In *International Conference on Information and Communications Security (ICICS)*, pages 419–434, 2008.

[30] Q. Zheng and S. Xu. Fair and dynamic proofs of retrievability. In *ACM Conference on Data and Application Security and Privacy (CODASPY)*, pages 237–248, 2011.