Efficient Dynamic Provable Possession of Remote Data via 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 wide availability of remote storage service providers prompted work in the security community that allows a client to verify integrity and availability of the data that 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 (i.e., insert, modify, or delete) stored data blocks while trying to lower the overhead associated with verifying the integrity of the stored data. In this work we develop a novel scheme, performance of which favorably compares with the existing solutions. Our solution enjoys a number of new features such as a natural support for operations on ranges of blocks, revision control, and support for multiple user access to shared content. The performance guarantees that we achieve stem from a novel data structure termed a *balanced update tree* and removing the need to verify update operations.

1 Introduction

Cloud computing and storage services are commonplace today and enable on-demand access to computing and data storage resources, which can be configured to meet unique constraints of the clients and utilized with minimal management overhead. The recent rapid growth in availability of cloud services makes such services attractive and economically sensible for clients with limited computing or storage resources who are unwilling or unable to procure and maintain their own computing infrastructure. It has been suggested, however, that the top impediment on the way of harnessing the benefits of cloud computing to the fullest extent is security and privacy considerations that prevent clients from placing their data or computations on the cloud (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, 18] and consists of many results such as [11, 22, 5, 21, 17, 10, 26, 6, 8, 9, 15, 23, 24, 12, 25, 27] that allow for integrity verification of large-scale remotely stored data. At high level, the idea consists of partitioning a large collection of data into data blocks and storing the blocks together with the meta-data 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 using the meta-data such that the number of verified data blocks is independent of the overall number of outsourced blocks, but allows with high probability to ensure that all stored blocks are intact and available. Schemes that support dynamic operations [5, 17, 15, 23, 24, 27] 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. 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 (modifications, insertions, and deletions) 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. A distinctive feature of our scheme is that all dynamic operations are not followed by integrity verification, which results in substantial communication and computation savings. Instead, verification can be performed at the time of retrieving the data or through periodic challenge queries as in prior work. Verifying a subset of the stored blocks periodically can incur larger overheads than verifying only the data which is being used, but can result in problems being detected earlier.

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 consistency and integrity, which current schemes do not provide. In addition to supporting this feature, our solutions 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 constant 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 consequently 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 of our scheme 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 multi-user access to the shared data.* Unlike prior work, we add support for multiple users which also includes conflict resolution for dynamic operations performed simultaneously on the same data block.

- *Support for revision control.* We provide natural support for revision control which allows clients to retrieve previous versions of their data and efficiently verify their integrity.

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 size of the data structure is not large (and is independent of the size of the outsourced data), 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 will be able to store the data structure locally.

As another tradeoff, our solution does not provide public verifiability that some of the solutions in the prior literature achieve. Public verifiability allows the client to outsource periodic verification of storage integrity to a third party auditor (who is different from the server). The need to verify correct execution of dynamic operations by the client in such schemes, however, still remains (unless the auditor is replaced by a constantly available intermediary who verifies all operations on behalf of the client). Thus, in most circumstances the client's overhead in our scheme is still lower than the client's overhead in other solutions with outsourced periodic integrity verification. We leave public verifiability of our scheme as a direction of future work.

2 Related Work

In this section we 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 [18, 5, 23] relies on so-called sentinels which are outsourced together with the client's data and are used to verify remotely stored blocks. In the setup phase, the client generates a predefined number of sentinels, each of which could be a function of κ data blocks (where κ is a security parameter) chosen according to a pseudo-random function. The client encrypts all sentinels and stores them together with the data blocks at a remote server. To invoke verification the ith time, the client executes a challenge-response protocol with the server, as a result of which the client is able to verify integrity and availability of the blocks contained in the ith sentinel. Besides having a limited number of audits, a disadvantage of this scheme is in poor performance when blocks need to be updated. In particular, when the client updates the jth data block, he needs to retrieve all remaining sentinels which have not yet been consumed by the previous audit queries. This is because the unused sentinels that cover the jth data block need to be modified based on the new jth data block to prevent them from being invalidated. Furthermore, in order to prevent the cloud service provider from learning any mapping between data blocks and sentinels, the client has to retrieve all unused sentinels (regardless of whether they cover the data block being updated or not). This is performed for every update operation and incurs a significant communication and computation overhead.

Another line of research [15, 24, 17, 27] with support for dynamic operations utilizes specialized data structures such as Merkle hash trees [24, 17] or skip lists [15] to organize the data blocks outsourced to a server. When a Merkle hash tree is used, each leaf node corresponds to the hash of an individual data block, and each internal node is assigned a value that hashes the concatenation of its children's values. The client locally keeps the root value of the tree in order to verify the correctness of various operations. For example, if the server receives a read request on the ith block, it sends to the client the block itself together with the sibling nodes that lie on the path from the block to the root. The client then recomputes the root value based on the received information

and compares it with the one locally stored. For an update request on the ith block, the client retrieves the same information as that of the read request on the ith data block. After verifying the correctness of the received information, the client 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. In particular, a data block insert request at position i is handled by locating the $(i-1)$th block, replacing its node with a newly created one 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 span across the stored data at uniformly random locations, for instance, inserting multiple blocks at the same position i will result in the height of the tree grow for each inserted data block. Because the tree can become extremely unbalanced over time, there will be a large variance in the time needed to locate different blocks within the data structure.

To mitigate the performance drawback of Merkle hash tree based solutions, [15] develops a scheme based on a skip list. It extends the original skip list [20] by incorporating *label* [16] and *rank* information to enable efficient authentication of client's updates. Recall that in a skip list [20] each node v contains two pointers $\mathsf{rgt}(v)$ and $\mathsf{dwn}(v)$ used for searching. In [16], each node is also assigned a label $f(v)$ computed by applying a commutative hash function to $f(\mathsf{rgt}(v))$ and $f(\mathsf{dwn}(v))$ in a specific order to maintain the commutative property. Maintaining only the label of the skip list's start node is sufficient for the client to verify the integrity of various operations (the verification process is the same as that of Merkle hash tree by considering $\mathsf{rgt}(n)$ and $\mathsf{dwn}(n)$ as sibling nodes). To make verification efficient, the authors of [15] integrate *rank* information into each node v which is the number of bottom-level nodes reachable from it. 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 (in this case, a *rank* of a node denotes the number of bytes reachable from it) 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 a large number of blocks in it.

The original Merkle hash tree and skip list schemes maintain only the most recent copy of data and do not provide the ability to retrieve its previous versions. To incorporate the feature, [3] used a persistent authenticated data structure, which copies the leaf-to-root path for each update and the path is visited when searching for the updated node. The approach uses $O(\log n)$ 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) which is determined by a dynamic operation performed on a range of consecutive blocks. The reason for assigning a range of block indices to a node is motivated by a study [13] 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 (as in Merkle hash tree or skip list schemes), 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 dynamic operation, and additionally

multiple updates issued on the same range of blocks can 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. That is, 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.

As we support multi-user access, in a distributed setting we assume that the users can communicate directly and notify each other of the changes that they make to the repository. There are alternatives to this solution, for instance, the approach employed in SUNDR [19] where the users communicate through an untrusted server (with slightly weaker security guarantees). SUNDR and similar solutions are therefore complimentary to our work.

Prior to this work, the notion of a range tree was used in the database domain 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. One of most significant challenges of this work was to design a update tree that can be rebalanced at low cost after an 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 outsources her data to a storage or cloud server 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 that the client wishes to outsource 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* as a desirable feature to have. This allows the client, in addition to retrieving the most recent data, to access and verify previous versions of its data, including deleted content. A commit command can be used for a certain range of data blocks to permanently erase previous versions of the data and deleted blocks. Our scheme achieves this property at no extra cost beyond maintaining previous versions by the server, but it is not strictly necessary for a PDP scheme (i.e., the client can run her own revision control system on top of this scheme, but enabling this feature in our scheme lowers the overall overhead).

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

- $\mathsf{KeyGen}(1^\kappa) \rightarrow \{\mathsf{sk}\}$ is a probabilistic algorithm run by the client that on input a security parameter 1^κ produces client's private 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 the client and the server, during which the client uses sk to encode the initial data blocks $m_1, \ldots, m_N$ and store them at the server who maintains all data blocks outsourced by the client in D. Client's and server's metadata are maintained in M_C and M_S, respectively.
- $\mathsf{Update}(\langle \mathsf{sk}, \mathsf{M}_C, \mathsf{op}, \mathsf{ind}, \mathsf{num}, m_{\mathsf{ind}}, \ldots, m_{\mathsf{ind}+\mathsf{num}-1} \rangle, \langle \mathsf{M}_S, D \rangle) \rightarrow \{\langle \mathsf{M}'_C \rangle, \langle \mathsf{M}'_S, D' \rangle\}$ is a proto-

col run between the client and the server, during which the client prepares num blocks starting at index ind and updates them at the server. The operation type op is either modification (0), insertion (1), or deletion (−1), where no data blocks are used 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 the client and the server, during which the client requests num data blocks starting from index ind, obtains them from the server 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 the client and the server, during which the client re-stores num data blocks starting from index ind at the server. The server erases all previous copies of the data blocks in the range and as well as previously deleted by the client blocks that fall into the range if they were kept as part of versioning control.

Our formulation of the scheme has minor differences with prior definitions of DPDP, e.g., as given in [15]. 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 requested blocks are below the constant, all of them are verified.) This functionality can be easily adopted 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 call Retrieve c (or fewer if the indices are adjacent) times on indices $i_1, \ldots, i_c$. 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 can be computed in our and prior work as follows: if num blocks need to be checked and the server tampers with t of them, the probability that at least one of the verified blocks matches at least one of the tampered blocks (i.e., the client can detect the problem) is $1 - ((\mathsf{num} - t)/\mathsf{num})^c$. This gives us a mechanism for computing c as to achieve the desired probability of detection for a chosen level of data corruption. For instance, to ensure that if the server tampers 1% or more of the total or retrieved content, the client can detect this with 99% probability, we need to set $c = 460$ regardless of the total number of blocks. 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 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 PPT adversary $\mathcal{A}$ (who acts as the server):

Setup: the challenger runs $\mathsf{sk} \leftarrow \mathsf{KeyGen}(1^\kappa)$. The adversary 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 the adversary 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 of the storage $m_1, \ldots, m_R$ 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 only (non-verifying updates are considered not to have taken place), and the challenger and adversary 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 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.

When the setting is generalized to multiple users who would like to have access to a shared content, we assume that the users trust each other, but not the server. This implies that the server does not enforce any access control with respect to retrieving or modifying the outsourced data blocks by the users. In environments where access control is to be enforced by a central entity (which could be an organization to which the users belong or a service provider who lets its subscribers to retrieve or modify certain data, while outsourcing the storage to a third party storage provider), the central entity will assume the role of a proxy: it will receive access requests from the users, enforce access control to the storage, submit queries to and verify responses from the storage server, and forward the data to the appropriate user. In the multi-user environment, the security experiment can be defined analogously, where the challenger now represents all users (who mutually trust each other). In the event that the users choose to use different keys for different portions of the outsourced storage (for access control or other purposes), the integrity and retrievability of the data can still be shown using the same security definition, which is now invoked separately for each key.

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.

In what follows, we first present our basic scheme that works for a single user and does not have full support for revision control. Later we extend it with features of multi-user access and support for revision control in Section 7.

4 Proposed Scheme

4.1 Building blocks

In this work we rely on standard building blocks such as message authentication codes (MAC). A MAC scheme is 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-length 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 $m \notin Q$.

4.2 Overview of the scheme

To mitigate the need for performing verifications for each dynamic operation 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. The size of the tree is independent of the number of data blocks that the client outsourced to the server, but grows linearly with the number of updates for ranges of blocks.

We term the new data structure that we build for the purposes of this work 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 when the need arises 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 she has 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 nodes, the client creates a nodes in the tree with the new blocks and also indicates that the indices of the blocks that follow need to be increased by the number of inserted blocks. This offset is stored at a single node in the tree, which removes the need to touch many nodes or data blocks. 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.

4.3 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 to global parameters, each node in the update tree stores several attributes:

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

Data block range L, U represents 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 will be 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 to the values from which IDs for the two different types will be 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 pointer $\mathsf{P_p}$ points to the parent of the node.

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).

4.4 Construction

In this section, we provide the details of our construction. Because the solution relies on our update tree algorithms, we outline them first, while their detailed description and explanation 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. The function 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. The function 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 $[\mathsf{s}, \mathsf{e}]$. Note that the returned nodes are not guaranteed to cover the entire range as some data blocks from the range have never been modified.
- $\mathsf{UTCommit}(\mathsf{T}, \mathsf{s}, \mathsf{e})$ removes the nodes in T that correspond to the data blocks in the range $[\mathsf{s}, \mathsf{e}]$, balances the remaining tree, and returns an adjusted lower bound and CID.

The protocols that define our solution are as follows:

1. $\mathsf{KeyGen}(1^\kappa) \to \{\mathsf{sk}\}$: the client executes $\mathsf{sk} \leftarrow \mathsf{Gen}(1^\kappa)$.

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

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

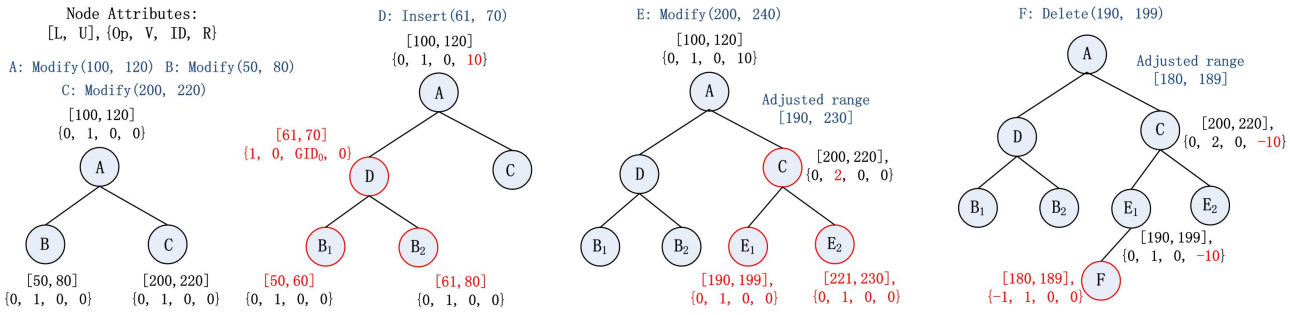


Figure 1: Example of update tree operations.

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) The client sends parameters op, ind and num to the server.
   (b) The server executes $C \leftarrow \mathsf{UTRetrieve}(\mathsf{M}_s, \mathsf{ind}, \mathsf{ind}+\mathsf{num}-1)$. For each $u \in C$, the server retrieves the attribute values from u (i.e., L, U and pointer to the data blocks), locates the data blocks and their tags $\langle m_i, t_i\rangle$ in D, and sends them to the client.
   (c) Upon receiving the $\langle m_i, t_i\rangle$, the client executes $C \leftarrow \mathsf{UTRetrieve}(\mathsf{M}_C, \mathsf{ind}, \mathsf{ind}+\mathsf{num}-1)$. The client chooses a random subset of data blocks of size $\min(c, \mathsf{num})$ for verification purposes. For each chosen data block m_i, the client locates the corresponding $u \in C$ 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 is the data block position within the node's data blocks. If $b_i = 1$ for each verified m_i, the client is assured of integrity of returned data.

5. $\mathsf{Commit}(\langle \mathsf{sk}, \mathsf{M}_C, \mathsf{ind}, \mathsf{num}, m_{\mathsf{ind}}, \ldots, m_{\mathsf{ind}+\mathsf{num}-1}\rangle, \langle \mathsf{M}_S, D\rangle) \rightarrow \{\langle \mathsf{M}'_C\rangle, \langle \mathsf{M}'_S, D'\rangle\}$:
   (a) The client executes $u \leftarrow \mathsf{UTCommit}(\mathsf{M}_C, \mathsf{ind}, \mathsf{ind}+\mathsf{num}-1)$ and stores updated metadata in M'_C. The client next computes $t_{\mathsf{ind}+i} \leftarrow \mathsf{Mac}_{\mathsf{sk}}(m_{\mathsf{ind}+i}||\mathsf{L}+i||0||\mathsf{CID}||0)$ for $0 \le i \le \mathsf{num}-1$, and sends the tags together with parameters ind and num to the server.
   (b) The server executes $u \leftarrow \mathsf{UTCommit}(\mathsf{M}_S, \mathsf{ind}, \mathsf{ind}+\mathsf{num}-1)$ and updates its metadata to M'_S. The server updates the tags of the affected blocks in D to obtain D'.

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 keep track of 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

Algorithm 1 UTInsertNode(u, w, dir)

```
 1: if (dir = left) then
 2:    while (1) do
 3:       u.R = u.R + w.Op(w.U − w.L + 1)
 4:       if (u.P_l ≠ NULL) then
 5:          u = u.P_l
 6:       else
 7:          insert w as u.P_l and exit;
 8:       end if
 9:    end while
10: else if (dir = right) then
11:    while (1) do
12:       if (u.P_r ≠ NULL) then
13:          u = u.P_r
14:       else
15:          insert w as u.P_r and exit;
16:       end if
17:    end while
18: end if
```

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 to 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. 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 present sub-routines called by the main algorithms followed by the algorithms for each operation.

5.1 Sub-routines

UTInsertNode(u, w, 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 (dir = left) or the rightmost (dir = right) node of the subtree. Its pseudo-code is given in Algorithm 1.

In the algorithm, lines 1–9 correspond to the case when the inserted node w belongs in the left subtree of u and further becomes the leftmost node of the subtree; similarly, lines 10–18 correspond to the the right subtree case. 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 (line 3) 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

Algorithm 2 UTFindNode(u, s, e, op)

```
1: S = ∅
2: while (1) do
3:   if (s − u.R > u.U) then
4:     s = s − u.R − u.Op(u.U − u.L + 1)
5:     e = e − u.R − u.Op(u.U − u.L + 1)
6:     if u.P_r ≠ NULL then
7:       u = u.P_r
8:     else
9:       u′ = UTCreateNode(s, e, 1 − |op|, ID, op)
10:      insert u′ as u.P_r, and exit the loop;
11:    end if
12:  else if ((op ≤ 0 and e − u.R < u.L) or (op > 0 and s − u.R < u.L)) then
13:    u.R = u.R + op(e − s + 1)
14:    if u.P_l ≠ NULL then
15:      u = u.P_l
16:    else
17:      u′ = UTCreateNode(s, e, 1 − |op|, ID, op)
18:      insert u′ as u.P_l, and exit the loop;
19:    end if
20:  else
21:    if (u.Op ≠ −1) then
22:      S = S ∪ {⟨u, s − u.R, e − u.R⟩}
23:    else if (u.Op = −1 and op = 1) then
24:      S = S ∪ {UTFindNode(u.P_r, s + u.U − u.L + 1 − u.R, e + u.U − u.L + 1 − u.R, op)}
25:    else if (u.Op = −1 and (op = 0 or op = −1)) then
26:      if (s − u.R < u.L) then
27:        S = S ∪ {UTFindNode(u, s, u.L − 1 + u.R, op)}
28:        S = S ∪ {UTFindNode(u.P_r, u.U + 1, e − u.R + u.U − u.L + 1, op)}
29:      else if (s − u.R ≥ u.L) then
30:        S = S ∪ {UTFindNode(u.P_r, s − u.R + u.U − u.L + 1, e − u.R + u.U − u.L + 1, op)}
31:      end if
32:    end if
33:    return S
34:  end if
35: end while
36: return {⟨u′, NULL, NULL⟩}
```

be modified as it traverses the tree. However, since the routine that calls this sub-routine will pass w with its range already updated, there is no need to further modify it. The time complexity of the sub-routine is $O(\log n)$, where n is the number of nodes in the update tree.

UTFindNode(u, s, e, op) searches the tree rooted at node u for a block range [s, e] for the purposes of executing operation op on that range. This is a recursive function that returns a set consisting of one or more nodes. The returned set normally consists of a single node (newly created or the first found node the range of which overlaps with [s, e]) unless a delete node partitions the range.

After the function is invoked on range [s, 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 [s, e] overlaps with the range of at least one existing tree node. The details are given in Algorithm 2.

Because this function can be called for any operation, similar to UTInsertNode, we need to

Algorithm 3 UTUpdateNode(u, s, e, op)

```
1: if (u.L = s and e < u.U) then
2:    N′ = UTCreateNode(e + 1, u.U, u.V, ID, u.Op)
3:    UTInsertNode(u.P_r, N′, left)
4: else if (u.L < s and e = u.U) then
5:    N′ = UTCreateNode(u.L, s − 1, u.V, ID, u.Op)
6:    u.R = u.R + u.Op(s − u.L)
7:    UTInsertNode(u.P_l, N′, right)
8: else if (u.L < s and e < u.U) then
9:    N′ = UTCreateNode(u.L, s − 1, u.V, ID, u.Op)
10:   N″ = UTCreateNode(e + 1, u.U, u.V, ID, u.Op)
11:   u.R = u.R + u.Op(s − u.L)
12:   UTInsertNode(u.P_l, N′, right)
13:   UTInsertNode(u.P_r, N″, left)
14: end if
15: if (op = 0) then
16:   UTSetNode(u, s, e, u.V + 1, u.ID, u.Op)
17: else if (op = −1) then
18:   UTSetNode(u, s, e, u.V, u.ID, Op)
19: end if
20: return u
```

update the offset R of the nodes lying on the traversal path (line 13) and additionally modify the range of block indices [s, e] during the traversal (lines 4–5). In the algorithm, lines 9–10 and 17–18 describe the case when the range being searched does not overlap with the ranges of any existing nodes. In this case, a new node u′ is created and returned with NULL flags (line 36). If the range overlaps with the ranges of one or more of the existing nodes, we add the triple $\langle u, s', e' \rangle$ to the set returned to the calling routine, where u is the first found node the range of which overlaps with (adjusted) [s, e] and s′ and e′ are adjusted s and e. Note that UTFindNode does not make any changes to the tree in case of range overlap, but rather lets the calling function perform all necessary changes. 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 (lines 21–32). This is the only situation when the set of size larger than 1 is returned. UTFindNode can be invoked for any dynamic operation, and its time complexity is $O(\log n)$.

UTUpdateNode(u, s, e, op) is called by a modification or deletion routine on a sub-tree rooted at node u when the range [s, e] of data blocks needs to be updated and falls into the range of u. Algorithm 3 details its functionality.

The function handles four different situations based on the type of intersection of ranges [s, e] and [u.L, u.U]. If the two ranges are identical (only lines 15–19 will be executed), several attribute of u (i.e., V, ID and Op) u 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 [s, e] (lines 15–19), fork a new node corresponding to the remaining range, and insert it into the right (resp., left) sub-tree of current root node (lines 1–7). 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 $O(\log n)$.

Algorithm 4 UTInsert($\mathsf{T}, \mathsf{s}, \mathsf{e}$)

```
1: GID = GID + 1
2: S = UTFindNode(T, s, e, 1)
3: (u′, s′, e′) = S[0]
4: if (s′ ≠ NULL) then
5:    u = UTCreateNode(u′.L, u′.U, u′.V, u′.ID, u′.Op)
6:    UTSetNode(u′, s′, e′, 0, GID, 1)
7:    if (u.L = s′) then
8:       UTInsertNode(u′.P_r, u, left)
9:    else
10:      u′.R = u′.R + u.Op · (s′ − u.L)
11:      w_1 = UTCreateNode(u.L, s′ − 1, u.V, u.ID, u.Op)
12:      w_2 = UTCreateNode(s′, u.U, u.V, u.ID, u.Op)
13:      UTInsertNode(u′.P_l, w_1, right)
14:      UTInsertNode(u′.P_r, w_2, left)
15:      UTFree(u)
16:   end if
17: end if
18: return u′
```

UTCreateNode($\mathsf{L}, \mathsf{U}, \mathsf{V}, \mathsf{ID}, \mathsf{Op}$) creates a new node with attribute values specified in the parameters.

UTSetNode($\mathsf{u}, \mathsf{L}, \mathsf{U}, \mathsf{V}, \mathsf{ID}, \mathsf{Op}$) sets the attributes of node u to the values specified in the parameters.

UTBalance(u) balances the tree rooted at node u 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.

UTFree(u) frees the memory occupied by the subtree rooted at node u.

5.2 Main routines

UTInsert($\mathsf{T}, \mathsf{s}, \mathsf{e}$) updates the update tree T for an insert request with the block range $[\mathsf{s}, \mathsf{e}]$ by inserting a node in the tree. Its pseudo-code is given in Algorithm 4. The main functionality of the routine is (i) to find a position for node insertion (line 2), and (ii) to insert a new node into the tree (lines 3–18). At line 1, the global variable GID is incremented and its current value is used for the newly created node. When the range $[\mathsf{s}, \mathsf{e}]$ does not overlap with any existing nodes, UTFindNode on line 2 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 ($\mathsf{s}' \neq$ NULL) and determines the number of nodes that need to be created. In particular, if the (adjusted) insertion position s' equals to the lower bound of u', u' is substituted with a new node (line 6) and is inserted into the right subtree of the new node (line 8). Otherwise, u' is split into two nodes, which are inserted into the left and right subtrees of u', respectively (lines 13–14), while u' itself is set to correspond to the insertion.

UTModify($\mathsf{u}, \mathsf{s}, \mathsf{e}$), when called with $\mathsf{u} = \mathsf{T}$, updates the update 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. It is given in Algorithm 5. The algorithm creates a node for the range if T is empty, and otherwise it invokes UTFindNode to locate the positions of nodes that need to be modified. After finding the nodes, the algorithm distinguishes between three cases based on how the (adjusted) range $[\mathsf{s}, \mathsf{e}]$ (i.e., $[\mathsf{s}_i, \mathsf{e}_i]$) overlaps with the range of a found node u_i:

Algorithm 5 UTModify(u, s, e)

```
1: C = ∅
2: if (u = NULL) then
3:    w = UTCreateNode(s, e, 1, CID, 0)
4:    C = C ∪ {w}
5: else
6:    S = UTFindNode(u, s, e, 0)
7:    for i = 0 to |S| − 1 do
8:       (u_i, s_i, e_i) = S[i]
9:       if (s_i ≠ NULL) then
10:         if (u_i.L ≤ s_i and e_i ≤ u_i.U) then
11:            C = C ∪ {UTUpdateNode(u_i, s_i, e_i, 0)}
12:         else if (s_i < u_i.L and e_i ≤ u_i.U) then
13:            C = C ∪ {UTUpdateNode(u_i, u_i.L, e_i, 0)}
14:            C = C ∪ {UTModify(u_i, s_i + u_i.R, u_i.L − 1 + u_i.R)}
15:         else if (u_i.L ≤ s_i and u_i.U < e_i) then
16:            C = C ∪ {UTUpdateNode(u_i, s_i, u_i.U, 0)}
17:            C = C ∪ {UTModify(u_i, u_i.U + 1 + u_i.R, e_i + u_i.R)}
18:         else if (s_i < u_i.L and e_i > u_i.U) then
19:            C = C ∪ {UTUpdateNode(u_i, u_i.L, u_i.U, 0)}
20:            C = C ∪ {UTModify(u_i, s_i + u_i.R, u_i.L − 1 + u_i.R)}
21:            C = C ∪ {UTModify(u_i, u_i.U + 1 + u_i.R, e_i + u_i.R)}
22:         end if
23:      else
24:         C = C ∪ {u_i}
25:      end if
26:   end for
27: end if
28: return C
```

1. If $[s_i, e_i]$ is contained in $[u_i.L, u_i.U]$, u_i is the only node to be modified, and this is handled by UTUpdateNode (line 11).
2. If $[s_i, e_i]$ overlaps with the ranges of u_i and either its left or right subtree, the algorithm first updates the range of u_i by calling UTUpdateNode sub-routine (line 13 or 16), and then recursively calls another UTModify to update the remaining nodes (line 14 or 17).
3. If $[s_i, e_i]$ overlaps with the range of u_i and both of its subtrees, the algorithm first updates u_i using UTUpdateNode (line 19), and then calls UTModify twice to handle the changes to the left and right subtrees of u_i, respectively (lines 20–21).

UTDelete(u, s, e), when called with u = T, updates the update tree T based on a deletion request with the block range [s, e]. It does not delete any node from T, but rather finds all nodes whose ranges fall into [s, e], sets their operation types to −1, and returns them to the caller. UTDelete is very similar to UTModify and is given in Algorithm 6.

UTRetrieve(u, s, e), when called with u = T, returns the nodes whose ranges fall into [s, e]. Algorithm 7 gives the details. Similar to UTModify, five cases can occur based on the overlap of the range [s, e] and [u.L, u.U]. The additional two cases are when the two ranges do not overlap (lines 12–15), and the algorithm is called recursively on a subtree until a node whose range falls into [s, e] (lines 6–26) is found, or not (lines 2–4). Care should be exercised when the returned node represents a deletion operation (line 10). In this case, we skip the node and keep searching within its right sub-tree until a node that represents either an insertion or modification is found.

Algorithm 6 UTDelete(u, s, e)

```
1: C = ∅
2: if (u = NULL) then
3:    w = UTCreateNode(s, e, 0, −1)
4:    C = C ∪ {w}
5: else
6:    S = UTFindNode(u, s, e, −1)
7:    for i = 0 to |S| − 1 do
8:       (u_i, s_i, e_i) = S[i]
9:       if (s_i ≠ NULL) then
10:         if (u_i.L ≤ s_i and e_i ≤ u_i.U) then
11:            C = C ∪ {UTUpdateNode(u_i, s_i, e_i, −1)}
12:         else if (s_i < u_i.L and e_i ≤ u_i.U) then
13:            C = C ∪ {UTUpdateNode(u_i, u_i.L, e_i, −1)}
14:            C = C ∪ {UTDelete(u_i, s_i + u_i.R, u_i.L − 1 + u_i.R)}
15:         else if (u_i.L ≤ s_i and u_i.U < e_i) then
16:            C = C ∪ {UTUpdateNode(u_i, s_i, u_i.U, −1)}
17:            C = C ∪ {UTDelete(u_i, u_i.U + 1 + u_i.R, e_i + u_i.R)}
18:         else if (s_i < u_i.L and e_i > u_i.U) then
19:            C = C ∪ {UTUpdateNode(u_i, u_i.L, u_i.U, −1)}
20:            C = C ∪ {UTDelete(u_i, s_i + u_i.R, u_i.L − 1 + u_i.R)}
21:            C = C ∪ {UTDelete(u_i, u_i.U + 1 + u_i.R, e_i + u_i.R)}
22:         end if
23:      else
24:         C = C ∪ {u_i}
25:      end if
26:   end for
27: end if
28: return C
```

UTCommit(T, s, e) replaces all nodes in tree T whose range falls into the range [s, e] with a single node with the range [s, 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. Algorithm 8 gives the details. In the algorithm, 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 by s′ and e′, respectively) become the left or right bound of the nodes that contain them (lines 1–16). Second, we traverse T from the two nodes to their least common ancestor T′, remove the nodes with ranges falling into the range [s′, e′], and balance the tree if necessary (lines 18–41). Third, we traverse T from T′ to the root, and balance the tree if necessary (line 42–45). Lastly, we add a node with [s, e] and new CID (line 49). The routine returns adjusted lower bound s′ and updated CID.

To better illustrate how the algorithm works, 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 (as defined on lines 3–4 of Algorithm 8). 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′. Every 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 (lines 27–29 of the algorithm). For the example in the figure, it means that nodes v_{10} and v_9 are removed

Algorithm 7 UTRetrieve$(\mathsf{u}, \mathsf{s}, \mathsf{e})$

```
1: C = ∅
2: if (u = NULL) then
3:    w = UTCreateNode(s, e, 0, 0, 0)
4:    C = C ∪ {w}
5: else
6:    if (u.L ≤ s − u.R and e − u.R ≤ u.U) then
7:       if (u.Op ≠ −1) then
8:          C = C ∪ {u}
9:       else
10:         C = C ∪ {UTRetrieve(u.P_r, s + u.U − u.L + 1, e + u.U − u.L + 1)}
11:      end if
12:   else if (e − u.R < u.L) then
13:      C = C ∪ {UTRetrieve(u.P_l, s, e)}
14:   else if (s − u.R > u.U) then
15:      C = C ∪ {UTRetrieve(u.P_r, s − u.R − u.Op(u.U − u.L + 1), e − u.R − u.Op(u.U − u.L + 1))}
16:   else if (s − u.R < u.L and u.L ≤ e − u.R ≤ u.U) then
17:      C = C ∪ {UTRetrieve(u.P_l, s, u.L + u.R − 1)}
18:      C = C ∪ {UTRetrieve(u, u.L + u.R, e)}
19:   else if (u.L ≤ s − u.R ≤ u.U and e − u.R > u.U) then
20:      C = C ∪ {UTRetrieve(u.P_r, u.U + 1 − u.Op(u.U − u.L + 1), e − u.R − u.Op(u.U − u.L + 1))}
21:      C = C ∪ {UTRetrieve(u, s, u.U + u.R)}
22:   else if (s − u.R < u.L and e − u.R > u.U) then
23:      C = C ∪ {UTRetrieve(u.P_l, s, u.L + u.R − 1)}
24:      C = C ∪ {UTRetrieve(u.P_r, u.U + 1 − u.Op(u.U − u.L + 1), e − u.R − u.Op(u.U − u.L + 1))}
25:      C = C ∪ {UTRetrieve(u, u.L + u.R, u.U + u.R)}
26:   end if
27: end if
28: return C
```

together with their subtrees, nodes v_8 and v_7 are also removed, and u_1 takes te 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 (lines 30–32 of the algorithm). This means that rebalancing is executed (if necessary) after u_1 takes place of v_7, v_4, and v_1. In the general case, the process of removing nodes and subtrees and rebalancing repeats until u_1 becomes the direct child of T'.

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 (lines 34–40 of the algorithm). 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 on lines 42–45 of the algorithm 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 UTInsert adds one or two nodes to the tree, and all

Algorithm 8 UTCommit($\mathsf{T}, \mathsf{s}, \mathsf{e}$)

1: $\mathsf{CID} = \mathsf{CID} + 1$
2: $\mathsf{S} = \mathsf{UTRetrieve}(\mathsf{T}, \mathsf{s}, \mathsf{e})$
3: set u_1 and u_2 to be the nodes in S, that are part of the tree, and has the smallest and largest indices respectively
4: set T' to be the lowest common ancestor of u_1 and u_2 in S
5: set s' to be s adjusted through traversal
6: set e' to be e adjusted through traversal
7: **if** $\mathsf{s}' \neq \mathsf{u}_1.\mathsf{L}$ **then**
8: $\quad \mathsf{UTSetNode}(\mathsf{u}_1, \mathsf{u}_1.\mathsf{L}, \mathsf{s} - 1, \mathsf{u}_1.\mathsf{V}, \mathsf{u}_1.\mathsf{ID}, \mathsf{u}_1.\mathsf{Op})$
9: $\quad \mathsf{w}_1 = \mathsf{UTCreateNode}(\mathsf{s}, \mathsf{u}_1.\mathsf{U}, \mathsf{u}_1.\mathsf{V}, \mathsf{u}_1.\mathsf{ID}, \mathsf{u}_1.\mathsf{Op})$
10: $\quad \mathsf{UTInsertNode}(\mathsf{u}_1.\mathsf{P}_\mathsf{r}, \mathsf{w}_1, \mathsf{left})$
11: **end if**
12: **if** $\mathsf{e}' \neq \mathsf{u}_2.\mathsf{U}$ **then**
13: $\quad \mathsf{UTSetNode}(\mathsf{u}_2, \mathsf{e}' + 1, \mathsf{u}_2.\mathsf{U}, \mathsf{u}_2.\mathsf{V}, \mathsf{u}_2.\mathsf{ID}, \mathsf{u}_2.\mathsf{Op})$
14: $\quad \mathsf{w}_2 = \mathsf{UTCreateNode}(\mathsf{u}_2.\mathsf{L}, \mathsf{e}', \mathsf{u}_2.\mathsf{V}, \mathsf{u}_2.\mathsf{ID}, \mathsf{u}_2.\mathsf{Op})$
15: $\quad \mathsf{UTInsertNode}(\mathsf{u}_2.\mathsf{P}_\mathsf{l}, \mathsf{w}_2, \mathsf{right})$
16: **end if**
17: **if** $\mathsf{u}_1 \neq \mathsf{u}_2$ **then**
18: $\quad$ **if** ($\mathsf{u}_1.\mathsf{P}_\mathsf{r} \neq$ NULL) **and** (u_1 is not u_2's ancestor) **then**
19: $\quad\quad \mathsf{UTFree}(\mathsf{u}_1.\mathsf{P}_\mathsf{r})$
20: $\quad\quad \mathsf{u}_1 = \mathsf{UTBalance}(\mathsf{u}_1)$
21: $\quad$ **end if**
22: $\quad$ **if** ($\mathsf{u}_2.\mathsf{P}_\mathsf{l} \neq$ NULL) **and** (u_2 is not u_1's ancestor) **then**
23: $\quad\quad \mathsf{UTFree}(\mathsf{u}_2.\mathsf{P}_\mathsf{l})$
24: $\quad\quad \mathsf{u}_2 = \mathsf{UTBalance}(\mathsf{u}_2)$
25: $\quad$ **end if**
26: $\quad$ **while** ($\mathsf{u}_1.\mathsf{P}_\mathsf{p} \neq \mathsf{T}'$) **and** ($\mathsf{u}_1 \neq \mathsf{T}'$) **do**
27: $\quad\quad$ **if** $\mathsf{u}_1 = \mathsf{u}_1.\mathsf{P}_\mathsf{p}.\mathsf{P}_\mathsf{l}$ **then**
28: $\quad\quad\quad \mathsf{UTFree}(\mathsf{u}_1.\mathsf{P}_\mathsf{p}.\mathsf{P}_\mathsf{r})$
29: $\quad\quad\quad$ set u_1 to be a direct child of $\mathsf{u}_1.\mathsf{P}_\mathsf{p}.\mathsf{P}_\mathsf{p}$
30: $\quad\quad$ **else if** $\mathsf{u}_1 = \mathsf{u}_1.\mathsf{P}_\mathsf{p}.\mathsf{P}_\mathsf{r}$ **then**
31: $\quad\quad\quad \mathsf{u}_1 = \mathsf{UTBalance}(\mathsf{u}_1.\mathsf{P}_\mathsf{p})$
32: $\quad\quad$ **end if**
33: $\quad$ **end while**
34: $\quad$ **while** ($\mathsf{u}_2.\mathsf{P}_\mathsf{p} \neq \mathsf{T}'$) **and** ($\mathsf{u}_2 \neq \mathsf{T}'$) **do**
35: $\quad\quad$ **if** $\mathsf{u}_2 = \mathsf{u}_2.\mathsf{P}_\mathsf{p}.\mathsf{P}_\mathsf{r}$ **then**
36: $\quad\quad\quad \mathsf{UTFree}(\mathsf{u}_2.\mathsf{P}_\mathsf{p}.\mathsf{P}_\mathsf{l})$
37: $\quad\quad\quad$ set u_2 to be a direct child of $\mathsf{u}_2.\mathsf{P}_\mathsf{p}.\mathsf{P}_\mathsf{p}$
38: $\quad\quad$ **else if** $\mathsf{u}_2 = \mathsf{u}_2.\mathsf{P}_\mathsf{p}.\mathsf{P}_\mathsf{l}$ **then**
39: $\quad\quad\quad \mathsf{u}_2 = \mathsf{UTBalance}(\mathsf{u}_2.\mathsf{P}_\mathsf{p})$
40: $\quad\quad$ **end if**
41: $\quad$ **end while**
42: $\quad$ **while** ($\mathsf{T}' \neq$ NULL) **do**
43: $\quad\quad \mathsf{T}' = \mathsf{UTBalance}(\mathsf{T}')$
44: $\quad\quad \mathsf{T}' = \mathsf{T}'.\mathsf{P}_\mathsf{p}$
45: $\quad$ **end while**
46: **else**
47: $\quad$ remove u_1 from the tree
48: **end if**
49: $\mathsf{UTModify}(\mathsf{T}, \mathsf{s}, \mathsf{e})$
50: **return** $(\mathsf{s}', \mathsf{CID})$

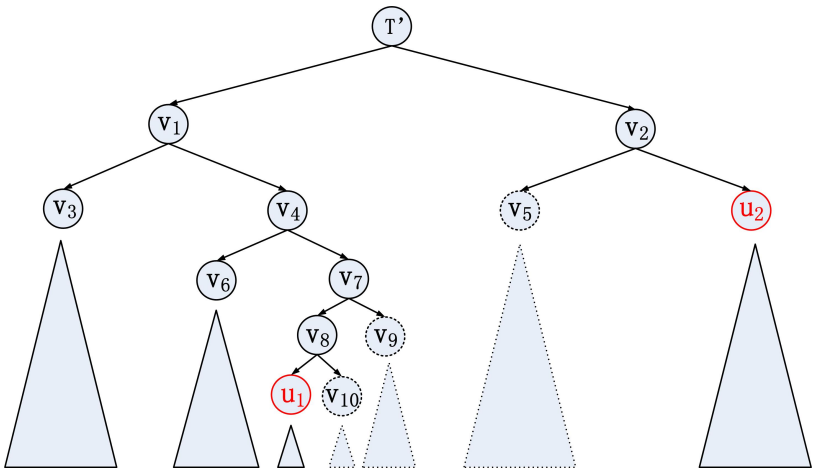


Figure 2: Illustration of the commit algorithm.

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 UTModify and 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 on average. Their time complexity is $O(\log n + \min(n, \mathsf{e} - \mathsf{s}))$, and both the size of the range $\mathsf{e} - \mathsf{s} + 1$ and the number of nodes in the tree form the upper bound on the number of returned nodes. UTRetrieve does not add nodes to the tree and its complexity is similarly $O(\log n + \min(n, \mathsf{e} - \mathsf{s}))$. Lastly, UTCommit removes between 0 and $O(\min(n, \mathsf{e} - \mathsf{s}))$ nodes from the tree and its time complexity is also $O(\log n + \min(n, \mathsf{e} - \mathsf{s}))$.

Because the complexity of UTCommit is less trivial to compute, we analyze it in more detail. Note that UTCommit calls the tree rebalancing function UTBalance, the worst case complexity of which is $O(\log n)$, at most $O(\log n)$ number of times. However, due to the careful construction of the tree and the commit function, the total number of operations that rearrange the tree per UTCommit is only $O(\log n)$ (plus, node deallocation time which gives the total $O(\log n + \min(n, \mathsf{e} - \mathsf{s}))$ time).

Theorem 1 *The time complexity of* UTCommit *is* $O(\log n + \min(n, \mathsf{e} - \mathsf{s})))$.

Proof In what follows, we assume that a tree is balanced if for each node in the tree the heights of subtrees rooted at the node's children differ by at most 1. In other words, rebalancing is triggered when the height difference is 2 or more. More generally, any constant $c \geq 2$ can be used as the criterion for rebalancing, and the claimed complexity will still hold.

It is clear that memory deallocation time associated with the nodes whose subtrees are being removed from the tree, i.e., the aggregate complexity of all UTFree calls, is $O(\min(n, \mathsf{e} - \mathsf{s}))$ and we therefore concentrate on showing that the rebalancing itself takes $O(\log n)$ time.

Recall that the complexity of UTBalance, when executed on a tree both child subtrees of which are balanced is linear in the height difference of the child subtrees. Also recall that during UTCommit we first locate the nodes that contain the smallest and largest block indices falling within the commit range (u_1 and u_2, respectively), and then call either UTBalance or UTFree during the process of moving up toward their lowest common ancestor T'. As was shown earlier, when u_1 is the left child of its parent only UTFree is called and no rebalancing takes place, but this can increment the difference in the height of u_1's subtree and that of its new sibling by 1 or 2 (the latter happens only if u_1's original sibling had a subtree with the larger height and the new sibling has the subtree of larger height than u_1's original parent node). After performing this step multiple times (where each time u_1 is still the left child of its parent), the difference can increase linearly in the total

number of times u_1 is moved up the tree. Referring back to the example in Figure 2, when u_1 takes the place of v_7, the difference between the heights of trees rooted at sibling nodes v_6 and u_1 can be larger by at most $2 \cdot 2$ than the original difference between the heights of the trees rooted at v_6 and v_7.

When u_1 is the right child of its parent, we call the rebalancing procedure on u_1's parent. In this case, the maximum height difference of its two subtrees is equal to twice the number of UTFree operations issued since the occurrence of the most recent UTBalance plus 1. Going back to the example in Figure 2, when UTBalance is called on v_4 (i.e., after replacing v_7 with u_1), the maximum height difference between the subtrees rooted at v_6 and u_1 is 5. The height difference then defines the runtime of the balancing procedure, which is linear in that difference. For the consecutive operations in the figure while u_1 remains the right child of it parent and moves up the tree, the difference between the heights of children subtrees of the nodes being rebalanced can be at most 2 and thus balancing takes constant time. The same analysis applies to node u_2 with the procedures invoked when u_2 is the left or right child of its parents reversed from those for u_1.

We obtain that the aggregate time for rebalancing the tree rooted at T' is linear in the number of calls to UTFree and UTBalance or the height of the tree T and is therefore $O(\log n)$. The commit function also balances the overall tree T as T' moves up the tree one node at a time. The complexity of this process can be shown similar to the complexity of balancing the subtrees while u_1 remains to be the right child of its parents and moves up the tree (i.e., the initial rebalancing cost can be at most linear in the distance from T' to the leaf level of the tree, but the cost of each consecutive rebalancing operation is constant). We obtain that the overall rebalancing cost of UTCommit is $O(\log n)$ and its overall cost is $O(\log n + \min(n, \mathsf{e} - \mathsf{s})))$ □

Next, we analyze the complexity of the protocols themselves. It is easy to see that Init has time and communication complexity of N, i.e., the number of transmitted blocks. Update for any operation has time complexity of $O(\log n + \mathsf{num})$ and communication complexity of $O(\mathsf{num})$. Retrieve has the same complexities as Update, unless it is used for integrity verification only 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 transmitted 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 2 *The proposed update tree scheme is a secure DPDP scheme assuming the security of MAC construction.*

Proof sketch Suppose that the adversary $\mathcal{A}$ wins the data possession game. 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 Retrieve or Challenge request of size $\mathsf{num} < c$ is always 1, and the probability that a cheating server is caught on a request of size $\mathsf{num} \geq c$ is $1 - ((\mathsf{num} - t)/\mathsf{num})^c$, where t is the number of tampered blocks among the challenged blocks.

7 Extending the Basic Scheme

7.1 Enabling multi-user support

In this section, we extend our scheme with support for multiple users who would like to jointly access outsourced data. Two additional considerations now come in play and affect how a DPDP scheme operates: *access control* and *conflict resolution*. That is, in a generic multi-user environment, access to an object is permitted according to a predefined access control policy, and simultaneous updates by multiple users of the same shared content require additional provisions. In what follows, we base our description on user trust relationships and distinguish between *distributed* and *centralized* settings.

Distributed setting. In this setting, the users trust each other; they locally maintain update trees and notify each other about updates on the shared data. First, the users identify the set of unique permissions that exist within the entire storage. Each unique set of access rights can be specified as a group of users who are granted access to the data objects with the respective permissions. In this collaborative environment, user groups can also be formed based on user preferences. The outsourced data is then divided based on the set of permission groups, each portion is assigned a unique key[1], and each user maintains a key and separate update tree for the portion of the storage associated with each group to which the user belongs. By maintaining trees only for the data blocks to which the user has access, the user's storage is reduced to the necessary minimum. Also, all users within the same permission group will announce their updates and synchronize them with other members of the group, which makes an exclusive tree for each group attractive, as all users will maintain identical copies of the tree and locally balance them in exactly the same way.

Each time user $\mathcal{U}$ performs an update on shared objects, the user notifies the remaining members of the group about the update, which allows them to consistently modify their copies of the respective update tree. To enable users to maintain consistent views in presence of conflicting updates, we propose for the users to maintain loosely synchronized clocks and in each time slot first announce their intended requests to the group, resolve any conflicts that arise, and only then submit the requests themselves. In detail, when $\mathcal{U}$ would like to submit a request to the server, it announces its intent (the operation type, index range, and the time slot) to the members of the permission group, and after the end of the time slot determines if any conflicts arise. We categorize all conflicts into *automatically resolvable* (performed locally by each user) and *manually resolvable* (require user interaction), which are detailed below. After conflict resolution, $\mathcal{U}$ submits her request to the server together with the proper ordering of the request and other simultaneous requests with which it conflicts (which, for instance, could be indexed by user id and sequence number). This will ensure that, even if the server receives users' requests out of order, their proper order will be enforced when it has impact on consistency and data verifiability. In presence of many simultaneous requests, the users determine if they are conflicting by considering each pair of them. In what follows, we exemplify our strategy for two conflicting requests, but it can be easily extended to resolve conflicts between a larger number of requests:

[1]One key is required for scheme operation (i.e., MAC computation), but if data confidentiality is also desired, the users can also agree on an encryption key.

- *Manually resolvable conflicts* occur when the ranges of two simultaneous dynamic operations overlap with each other (for instance, when two users attempt to update the same data block). In general, it is not possible to resolve such conflicts without user coordination of the updates, but for specific applications it may be feasible to resolve certain types of conflict from this category automatically (e.g., in some cases overlapping insert and delete operations might be automatically resolvable).
- *Automatically resolvable conflicts* occur when an insertion or deletion with $\mathsf{ind}_1, \mathsf{num}_1$ is triggered simultaneously with another operation on a succeeding, but not overlapping range $\mathsf{ind}_2, \mathsf{num}_2$. That is, for insertion $\mathsf{ind}_1 < \mathsf{ind}_2$ and for deletion $\mathsf{ind}_1 + \mathsf{num}_1 < \mathsf{ind}_2$. To resolve this type of conflict, we propose to apply operational transformation (OT) [14] that allows each owner of a conflicting request to locally resolve the conflict. Using OT, the users need to agree on a set of rules based on which modifications to the requests will be performed. In our setting, one possibility for such rules is to execute the conflicting requests based on the numerical order of their owners' ids (it is assumed that a single user will not announce requests that already conflict with each other). This means that if the insertion or deletion operation with the lower range is executed first, the indices of the second request will be adjusted by num_1. Note that OT can be used as a black box in our solution.

Other types of requests are considered non-conflicting, and the users merely submit them to the server. Because all users trust each other and synchronize their requests, security of this setting can be shown analogously to the single-user case.

Centralized setting. In this setting, the users are not trusted and should be prevented from issuing requests that may invalidate outsourced data. To enforce proper access control, there exists a central trusted authority (CA), which could be an organization to which the users belong or a service provider who lets its subscribers retrieve and update data outsourced to a third party storage server. The CA serves the role of a proxy, and users do not maintain any metadata themselves. All user requests go through the CA, who examines them with respect to users' privileges, detects any conflicts, resolves any automatically resolvable and ask the senders to resolve manually resolvable conflicts. The CA is also in a good position to perform query optimization, e.g., by merging two read requests on overlapping or consecutive ranges. This will allow the CA to reduce the size of the update tree that the CA and the storage server maintain as well as reduce query processing time. Lastly, the CA submits the queries to the storage server. For retrieve requests, the CA verifies the response and forwards the data to the appropriate users.

7.2 Enabling support for revision control

In this section, we show how our scheme can be extended to support revision control. 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 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 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 UTRetrieve. This functionality requires no changes to UTRetrieve, but using the tree nodes that the function returns the server will retrieve the requested the requested version of the blocks instead of their most recent versions. All steps of the Retrieve protocol proceed unmodified with the exception that the client uses its requested version V during MAC verification in step (c) instead of node versions $u.\mathsf{V}$ returned by UTRetrieve.

Deleted data retrieval can be realized by extending the Retrieve protocol's interface with a flag that indicates that deleted data is to be returned and modifying 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 $[\mathsf{s}, \mathsf{e}]$ that contains the desired deleted range and includes one or more non-deleted blocks before and after the deleted range being requested.

We denote the modified UTRetrieve that retrieves deleted data as $\mathsf{UTRetrieveDel}(\mathsf{u}, \mathsf{s}, \mathsf{e})$. Instead of ignoring nodes that represent deletions, it returns them as the output, which will also allow the server to locate the necessary blocks and their tags. UTRetrieveDel then uses the same interface as UTRetrieve and has similar functionality to that of UTRetrieve. In particular, similar to UTRetrieve, five cases can occur based on the overlap of the range $[\mathsf{s}, \mathsf{e}]$ and $[\mathsf{u.L}, \mathsf{u.U}]$, and the routine is called recursively to deal with each of the situations. Unlike UTRetrieve, however, since all deleted ranges can be found within the tree, the function does not create any new nodes corresponding to the blocks that have not been updated (i.e., lines 2–4 of UTRetrieve are omitted). Furthermore, the routine needs to collect only nodes that correspond to deleted blocks, UTRetrieve ignores such nodes and returns all other nodes in the range. This means that we need to change the condition ($\mathsf{u.Op} \neq -1$) on line 7 of UTRetrieve to the opposite ($\mathsf{u.Op} = -1$). For completeness the algorithm for UTRetrieveDel is given in Appendix A.

8 Performance Evaluation

To evaluate the performance of our scheme and provide a comparison with performance of prior solutions, we designed a number of 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) [17], and (iii) the solution based on skip lists (SL) [15].

In our experiments, 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. Lastly, while the overall amount of outsourced storage in our experiments is not very large (i.e., in practice clients might want to outsource TBs of data), we believe that the experiments represent realistic scenarios. That is, 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.

We implement our and MHT solutions in C, while the SL scheme was implemented as in [15] in Java. Despite the programming language difference, the time to compute a hash function is similar in both implementations. Then 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.

8.1 Computation

For the purposes of computation evaluation, we measure the client's time after executing a number n of 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.

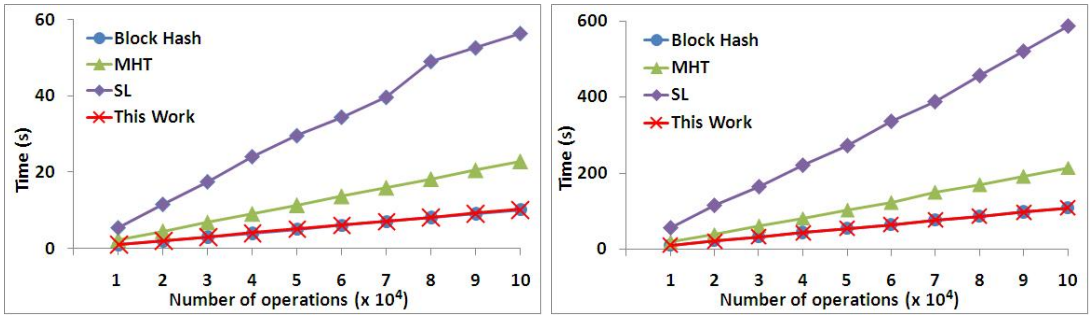
In the first experiment, we choose one of four operations (insert, delete, modify, and retrieve) at random and execute it on a randomly selected single data block. From the schemes that we compare, only our solution provides a natural support for querying ranges of blocks (the SL solution in [15] can provide a limited support for block ranges as previously described). Then because in practice accesses are often consecutive in their nature (see, e.g., [13]), the performance of our scheme in this experiment can be viewed as the upper bound on what is expected 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 1 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 is 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.

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 [15], 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 1 and Figure 3(b).

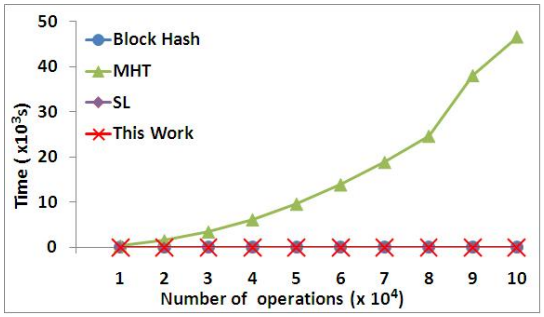
Compared to the single-block operations, performance of MHT and SL schemes deteriorates by a factor of 9–10, while for our UTree, which was designed to work on ranges, there is only a slight (20%–70%) increase in the performance. The increase in the update tree size can be explained by using more nodes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

[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 computation alone, we include the additional overhead associated with the MAC into the performance of our scheme.



(a) single-block mixed operations

(b) multiple-block mixed operations



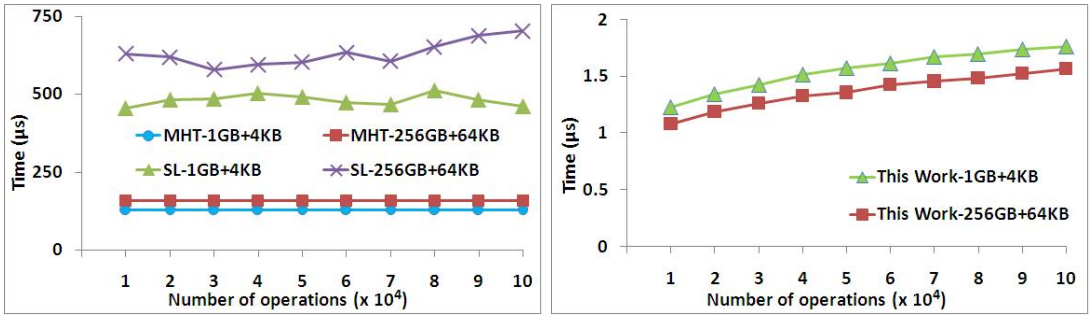
(c) same position insertions

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

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

For the third experiment, we considered 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 1 and Figure 3(c). As can be seen from the table, 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



(a) MHT and SL (b) UTree

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

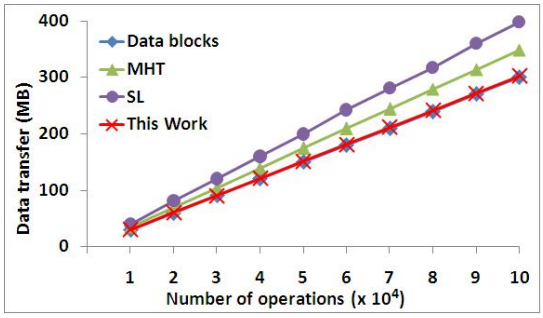


Figure 5: Aggregate communication overhead with 1GB outsourced storage and 4KB block.

situation by balancing MHT in a similar way to our solution, as the client does not have complete information about the MHT.

8.2 Communication

To evaluate communication, we measure the volume of data exchanged between the client and the server after executing a different number of client's single-block requests. The data transferred in each operation consists of a data block (except for deletion) and corresponding auxiliary data. The data block cost is common to all three schemes, while the auxiliary data 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 2 and Figure 5, where data block communication is added to the performance of each scheme in the figure, and it is listed separately in the table.

Because deletion does not involve data block transfer in all three schemes, the average size of data block communication per operation is 3/4 of the block size. As can be observed from Table 1, 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

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 blocks	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 blocks	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 blocks	60	120	180	240	300

Table 2: Aggregate communication size after n operations for various schemes measured in MB.

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 (SL scheme stores more attributes per node than MHT scheme).

8.3 Storage

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

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. It is also the reason for outsourcing the data structures together with the data blocks to the server. In our scheme, however, 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 3 and Figure 6. 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.

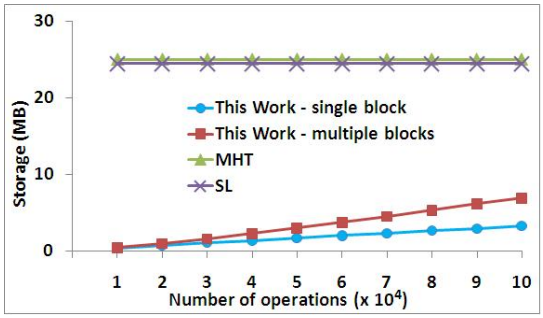


Figure 6: Server's storage overhead with 1GB outsourced storage and 4KB block.

File size	Block size	MHT for any n	SL for 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 3: The size of data structures for various schemes after n single-block or range mixed operations measured in MB.

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. And 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.

9 Conclusions

In this work, 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 storage 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 and no support for public verifiability.

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)*, pages 9:1–9:10, 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] Y. Dodis, S. Dadhan, and D. Wichs. Proofs of retrievability via hardness amplification. In *Theory of Cryptography Conference (TCC)*, pages 109–127, 2009.

[13] 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.

[14] C. Ellis and S. Gibbs. Concurrency control in groupware systems. In *SIGMOD International Conference on Management of Data*, pages 399–407, 1989.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] J. Li, M. Krohn, D. Mazieres, and D. Shasha. Secure untrusted data repository (SUNDR). In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 121–136, 2004.

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

[21] 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.

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

[23] 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.

[24] 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.

[25] L. Wei, H. Zhu, Z. Cao, W. Jia, and A. Vasilakos. SecCloud: Bringing secure storage and computation in cloud. In *IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 52–61, 2010.

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

[27] 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.

A Additional Algorithms

Here we give a complete specification of $\mathsf{UTRetrieveDel}(\mathsf{u}, \mathsf{s}, \mathsf{e})$, which when called on the update tree T and block range $[\mathsf{s}, \mathsf{s}+\mathsf{e}-1]$ returns all previously deleted blocks that fall within the range. The description is given in Algorithm 9.

Algorithm 9 UTRetrieveDel(u, s, e)

```
1: C = ∅
2: if (u ≠ NULL) then
3:    if (u.L ≤ s − u.R and e − u.R ≤ u.U) then
4:       if (u.Op = −1) then
5:          C = C ∪ {u}
6:          C = C ∪ {UTRetrieveDel(u.P_r, s + u.U − u.L + 1, e + u.U − u.L + 1)}
7:       end if
8:    else if (e − u.R < u.L) then
9:       C = C ∪ {UTRetrieveDel(u.P_l, s, e)}
10:   else if (s − u.R > u.U) then
11:      C = C ∪ {UTRetrieveDel(u.P_r, s − u.R − u.Op(u.U − u.L + 1), e − u.R − u.Op(u.U − u.L + 1))}
12:   else if (s − u.R < u.L and u.L ≤ e − u.R ≤ u.U) then
13:      C = C ∪ {UTRetrieveDel(u.P_l, s, u.L + u.R − 1)}
14:      C = C ∪ {UTRetrieveDel(u, u.L + u.R, e)}
15:   else if (u.L ≤ s − u.R ≤ u.U and e − u.R > u.U) then
16:      C = C ∪ {UTRetrieveDel(u.P_r, u.U + 1 − u.Op(u.U − u.L + 1), e − u.R − u.Op(u.U − u.L + 1))}
17:      C = C ∪ {UTRetrieveDel(u, s, u.U + u.R)}
18:   else if (s − u.R < u.L and e − u.R > u.U) then
19:      C = C ∪ {UTRetrieveDel(u.P_l, s, u.L + u.R − 1)}
20:      C = C ∪ {UTRetrieveDel(u.P_r, u.U + 1 − u.Op(u.U − u.L + 1), e − u.R − u.Op(u.U − u.L + 1))}
21:      C = C ∪ {UTRetrieveDel(u, u.L + u.R, u.U + u.R)}
22:   end if
23: end if
24: return C
```