Efficient Dynamic Provable Possession of Remote Data via Update Trees

YIHUA ZHANG and MARINA BLANTON, University of Notre Dame

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 a not fully trusted remote 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 additionally 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 for interaction during update operations besides communicating the updates themselves.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems; E.1 [**Data Structures**]: Trees; H.3.4 [**Information Storage and Retrieval**]: Systems and Software

General Terms: Security, Verification, Algorithms.

Additional Key Words and Phrases: Provable data possession, outsourced storage, integrity verification, balanced tree.

1. INTRODUCTION

Today, cloud services enable convenient on-demand access to computing and storage resources, which makes them attractive and economically sensible for resource-constrained clients. Security and privacy, however, have been suggested to be the top impediment on the way of harnessing full benefits of these services (see, e.g., [IDC 2008]). 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 [Ateniese et al. 2007; Juels and Kaliski 2007] and consists of many results such as [Curtmola et al. 2008; Shacham and Waters 2008; Sebe et al. 2008; Chang and Xu 2008; Zeng 2008; Ateniese et al. 2009; Bowers et al. 2009a; 2009b; Dodis et al. 2009; Wei et al. 2010] that allow for integrity verification of large-scale remotely stored data. At a 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. It can often be achieved that the number of verified

Part of this work appeared in the Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS) 2013.
Authors' address: M. Blanton and Y. Zhang, Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN 46556.

DOI: http://dx.doi.org/10.1145/0000000.0000000

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 [Oprea and Reiter 2007; Ateniese et al. 2008; Heitzmann et al. 2008; Goodrich et al. 2008; Erway et al. 2009; Wang et al. 2009; Wang et al. 2009; Zheng and Xu 2011; Popa et al. 2011] additionally allow the client to issue modify, insert, and delete requests, after each of which the client and the server interactively ensure that their state was updated correctly.

The motivation for this work comes from (i) improving the performance of the existing schemes when modifications to the data are common, and (ii) simultaneously extending the available solutions with several additional 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. Furthermore, by issuing a flush command (which is described in detail later in the paper), the client will be able to keep the size of the maintained update tree below a desired constant threshold if necessary (at the cost of extra communication).

We view performance improvement as the most crucial aspect of this work. All prior work on PDP/POR with support for dynamic operations requires the client to retrieve some meta-data (the size of which is a function of the overall outsourced storage size) for each dynamic operation before the client can apply the operation and correctly update the verification data it locally maintains. Our scheme eliminates the need for such communication and the corresponding computation. In our scheme, besides sending the update itself, no additional rounds, communication, or expensive computation are needed. Instead, verification that the data that the server maintains is correct, which covers verification of correct execution of dynamic operations by the server, is performed only at the time of retrieving the data and through periodic audit queries. Both of these checks are also present in prior work. Thus, we eliminate the need to directly check correct application of each dynamic operation and achieve the same security guarantees as in prior work. This distinctive feature of our scheme results in substantial communication and computation savings over the course of its deployment (e.g., while communication savings may be on the order of a couple of KB per block update, over a period of time they will translate into actual monetary savings for the client). We also note that our scheme results in reduced storage size at the server, but the client's storage is increased.

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 we build support for.

In addition, our solution provides natural support for revision control which can be of value for certain applications as well. In the existing solutions the server typically maintains only the up-to-date values of each data block. Support for revision control can then be added by means of additional techniques (such as [Anagnostopoulos et al. 2001]), 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.

Lastly, achieving public verifiability, where the client can outsource the task of verifying the integrity and availability of remote storage to a third party, or aggregate verification, where multiple data blocks can be combined together during periodic challenge queries, are often desired features. Our scheme can be easily extended to support them.

To summarize, our solution enjoys the following features:

— *Improved efficiency in handling dynamic operations.* In our solution there is no work associated with updates besides communicating them to the server (e.g., when a data block is modified a large number of times and is consequently deleted), while all prior schemes we are aware of invest resources in client-server interaction after each dynamic operation to ensure that the client's state is correctly updated.
— *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 most 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 flush command. The flush command was specifically designed for reducing the size of the data structure and is expected to be called periodically to keep the size of the data structure below a desired threshold.
— *Support for multi-user access to outsourced data.* We add support for multiple users which also includes conflict resolution for dynamic operations performed simultaneously by different users on the same data block. We distinguish between centralized and distributed settings, in which the users access the outsourced untrusted storage through a central proxy (centralized environment) or directly communicate with each other and the storage server (distributed environment).
— *Support for revision control.* Our solution provides natural support for revision control which allows clients to retrieve previous versions of their data and efficiently verify its integrity. Enabling revision control does not increase complexity of the scheme.
— *Public verifiability.* Our scheme can be easily modified to support public verifiability, which allows the client to outsource periodic verification of storage integrity and availability to a third party auditor (who is different from the server).
— *Verification aggregation.* Our scheme supports aggregation of multiple blocks into a single block to reduce communication during periodic integrity audits.

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 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. A detailed performance comparison of our and other dynamic PDP (DPDP) schemes is given in section 8, which shows that both computation and communication overhead of our scheme is orders of magnitude lower than those of other solutions.

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 schemes that support dynamic operations on outsourced storage.

One line of research [Juels and Kaliski 2007; Ateniese et al. 2008; Wang et al. 2009] relies on so-called sentinels or verification tokens 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 pseudorandom 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 verifies 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 are 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 with support for dynamic operations utilizes specialized data structures such as Merkle hash trees (MHT) or chained hashes [Oprea and Reiter 2007; Wang et al. 2009; Popa et al. 2011] or skip lists [Heitzmann et al. 2008; Goodrich et al. 2008; Erway et al. 2009] to organize outsourced data blocks. When a MHT 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 MHT-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 distributed locations, e.g., inserting multiple blocks at the same position i will result in the height of the tree growing for each inserted data block. Because the tree can become extremely unbalanced over time (e.g., if a large number of blocks are inserted into the same location), there will be a large variance in the time needed to locate different blocks within the data structure. For that reason, recent solutions use balanced MHTs, which allows the worst-case communication and computation complexity of an operation to be reduced from $O(n)$ to $O(\log n)$ for storage consisting of n blocks. Such publications include [Mo et al. 2012] (a MHT based on a B+ tree) and [Stefanov et al. 2012] (the details are not specified).

To support dynamic operations, [Erway et al. 2009] also develops a scheme based on a skip list. It extends the original skip list [Pugh 1990] by incorporating *label* [Goodrich et al. 2001] and *rank* information to enable efficient authentication of client's updates. Recall that in a skip list [Pugh 1990] each node v contains two pointers $\mathsf{right}(v)$ and $\mathsf{down}(v)$ used for searching. In [Goodrich et al. 2001], each node is also assigned a label $f(v)$ computed by applying a commutative hash function to $\mathsf{right}(v)$ and $\mathsf{down}(v)$. 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 MHT by treating $\mathsf{right}(n)$ and $\mathsf{down}(n)$ as sibling nodes). To make verification efficient, [Erway et al. 2009] integrates *rank* information into each node v, defined as 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 treating 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 operations on it as before. While this approach guarantees 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 corresponds to a large number of blocks.

Another related work [Stefanov et al. 2012] builds an authenticated file system called Iris. The solution is to construct a mapping from file names to block numbers and outsource both the data and meta-data to the cloud, where balanced MHTs are used to authenticate information returned by the cloud. We view the major contributions of that work as (i) providing support for dynamic proofs of retrievability through sparse erasure codes that guarantee that all data can be recovered without loss and (ii) designing and building a proxy for centralized multi-user setting that caches data received from the cloud and provides parallel execution when possible. Both of these contributions are complementary to our work as we briefly explain later in the paper. Iris applies a range tree structure to compress the information about a consecutive range of blocks that have been updated the same number of times into a single node. This is the closest approach from PDP/POR literature to our solution, but there are nevertheless substantial differences. That is, as the scheme still uses MHT to authenticate the metadata, all updates still require additional rounds of interaction and a number of applications of a hash function which is logarithmic in the size of the metadata. This is in contrast to our solution which involves a single client-to-server transmission for dynamic operations and does not involve any hash computation for metadata verification. Furthermore, Iris deals with file updates and does not have provisions for inserting individual blocks in the middle of a file. This means that, to insert new data into a file, all consecutive blocks will have to be updated, which further increases the cost of dynamic operations.

The data structure that we build is unique and has distinct properties that make it favorably compare to other schemes. First, each node in our update tree corresponds to a range of block indices (instead of a single 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 on users' file access patterns [Ellard et al. 2003] 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 the MHT or skip list schemes), in our solution it is independent of the size of the stored data (but the server's storage is larger). 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 are condensed into a single node. Due to its moderate size, the client can maintain the data structure locally, which makes the updates and the verification process more efficient. In particular, only during read and audit requests the client receives data from the server and verifies its correctness by checking a constant number of blocks per request. Prior work, on the other hand, requires bidirectional communication and verification of information received as part of all of read, audit, and update operations for each data block. Furthermore, we can specify requirements that define when the data structure should be re-balanced. That is, once the requirement is violated, the tree is re-organized to satisfy the constraint. As an example, the constraint of AVL trees [Adelson-Velskii and Landis 1962] can be used that requires that the heights of a node's subtrees must differ by at most 1 or any other constant.

The schemes based on a MHT and skip list maintain only most recent data and do not provide the ability to retrieve old versions. Thus, such schemes have to be modified to incur additional overhead if support for revision control is desired. To the best of our knowledge, revision control in the context of PDP/POR schemes was mentioned only in the full version of [Erway et al. 2009] and in [Wang et al. 2013]. Then, [Erway et al. 2009] employs costly techniques from [Anagnostopoulos et al. 2001] and [Papamanthou and Tamassia 2007] that increase the server's storage cost and the communication and computation cost of each dynamic operation. Similar overhead has to be added to the solution from [Wang et al. 2013]. In general, to enable revision control, any MHT-based or skip-list-based solution can be extended with a persistent authenticated data structure from [Anagnostopoulos et al. 2001], which on each update copies the leaf-to-root path and the path is visited when searching for the updated node. The approach uses $O(\log n)$ extra space for each update. As mentioned earlier, our approach, on the other hand, has a natural support for revision control capability and only requires the server to maintain data blocks that correspond to previous versions (which is inevitable in any scheme) together with authentication information in the form of message authentication codes (MACs) on those blocks.

Multi-user access has been treated in the context of outsourced storage with a third-party auditor. A line of publications by B. Wang et al. [Wang et al. 2012a; 2012b; Liu et al. 2013; Wang et al. 2013] assume that users sign data blocks with their private keys in the public-private key setting. The publications incorporate features such as user revocation [Wang et al. 2013], which allows for efficient re-signing of shared data that was previously signed by a revoked user, and protection of the identity of the signer from the untrusted storage server [Wang et al. 2012a; 2012b; Liu et al. 2013], which allows any member of an access group to anonymously utilize shared resources using ring or group signatures. These features are complementary to our work, when our scheme is used in the multi-user setting with separate user keys, and can be incorporated into our solution as well. Also, 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, e.g., the approach employed in SUNDR [Li et al. 2004] where the users communicate through an untrusted server (with slightly weaker security guarantees), but which has different design goals. SUNDR and similar solutions are therefore also complementary to our work.

Prior to this work, the notions of a range tree [Bentley 1979] and an interval tree [de Berg et al. 2000] were used in the database domain to deal with range queries. A range tree built on a set of 1-dimensional points is a balanced binary tree with the leaves storing the points and the intermediate nodes storing the largest value stored in their left subtrees. When the tree is used to respond to a range query, e.g., searching for all points with values in the range $[L, U]$, the tree will be traversed to search for L and U respectively. As the result of the procedure, all nodes whose values lie within the range $[L, U]$ will be returned. The range tree data structure 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.

An interval tree [de Berg et al. 2000] is an ordered binary tree structure that holds intervals. All intervals that are completely to the left (resp., right) of the interval represented by a node are stored in its left (resp., right) subtree. Furthermore, all intervals that overlap with the interval represented by the node are stored in a separate data structure linked to the node. When an interval tree is used to respond to a range query, starting from the root, the query range will be compared with that of the current node. If the query range is completely to the left (resp., right) of the interval of the node, the search procedure will be recursively carried out in its left (resp., right) subtree. The procedure continues until the nodes whose intervals overlap with the query range are all found. Different from our update tree, interval trees cannot support insertion of block ranges which require partitioning of existing ranges/intervals in the tree or index changes. The operational details of update trees are therefore very different from those of interval trees. One of most significant challenges of this work was to design an update tree that can be re-balanced at low cost after arbitrary changes to it. A balanced update tree is therefore one of the novel aspects of this work.

3. PROBLEM DEFINITION

We consider the problem in which a resource-limited client is in possession of a large amount of data partitioned into blocks. Let K denote the initial number of blocks and m_i denote the data block at index i, where $1 \leq i \leq K$. 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 purpose 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. 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 flush command can be used for a certain range of data blocks to permanently erase previous versions of the data and deleted blocks.

We define a DPDP 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.

— $\mathsf{Init}(\langle \mathsf{sk}, m_1, \ldots, m_K \rangle, \langle \perp \rangle) \rightarrow \{\langle \mathsf{M}_C \rangle, \langle \mathsf{M}_S, D \rangle\}$ is a protocol run between the client and the server, during which the client uses sk to encode the initial data blocks $m_1, \ldots, m_K$ 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 protocol 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 op is either modification (0), insertion (1), or deletion (-1), and 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.

— $\mathsf{Challenge}(\langle \mathsf{sk}, \mathsf{M}_C \rangle, \langle \mathsf{M}_S, D \rangle) \rightarrow b$ is a protocol run between the client and the server, during which the client verifies integrity of all data blocks stored with the server and outputs a bit b such that $b = 1$ only if the verification was successful.[1]

— $\mathsf{Flush}(\langle \mathsf{sk}, \mathsf{M}_C, \mathsf{ind}, \mathsf{num}, m_{\mathsf{ind}}, \ldots, m_{\mathsf{ind}+\mathsf{num}-1} \rangle, \langle \mathsf{M}_S, D \rangle) \rightarrow \{\langle \mathsf{M}'_C \rangle, \langle \mathsf{M}'_S, D' \rangle\}$ 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 [Erway et al. 2009]. First, update and retrieve operations are defined as interactive protocols rather than several algorithms run by either the client or the server. Second, in the current formulation, we can use the Retrieve protocol for both reading data blocks and performing periodic Challenge audits (in prior work, Challenge was defined explicitly, while Retrieve could be derived from it). Verification of each Retrieve is necessary to ensure that correct blocks were received even if the integrity of the overall storage is assured through periodic audits, and the verification is performed similar to periodic audits. 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 c to guarantee the desired confidence level. This functionality can be easily adopted to implement periodic Challenge audits, but for completeness of this work we choose to explicitly describe the Challenge protocol as well.

The constant c is computed in our solution in the same way as in prior work (e.g., [Ateniese et al. 2007] and others): if num blocks are to be checked and the server tampers with t of them, the probability that at least one tampered block is among the verified 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, when the server tampers with $\geq 1\%$ of the total or retrieved content, to ensure that 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 DPDP 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 verifying that the most recent version was returned (or, when revision control is used, a specific previous version, including deleted content, as

[1]The algorithm can also be defined to output the indices of the blocks that did not pass verification (if any). Such modification will be trivial to realize with our (and other) solutions.

requested by the client). The server is considered fully untrusted and can modify the stored data in any way it wishes (including deleting the data). Our goal is to design a scheme in which any violations of data integrity or availability will be detected by the client. More precisely, in the single-user setting the security requirements are formulated as a game between a challenger (who acts as the client) and any probabilistic polynomial time (PPT) adversary $\mathcal{A}$ (who acts as the server):

— *Setup:* The challenger runs $\mathsf{sk} \leftarrow \mathsf{KeyGen}(1^\kappa)$. $\mathcal{A}$ specifies data blocks $m_1, \ldots, m_K$ and their number K for the initialization and obtains initial transmission from the challenger.
— *Queries:* $\mathcal{A}$ 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. $\mathcal{A}$ 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 $\mathcal{A}$ to the beginning of the $\mathsf{Challenge}$ query a polynomial number of times with the purpose of data extraction. The challenger's goal is to extract the challenged portions of the data from $\mathcal{A}$'s responses that pass verification.

Definition 3.1. A DPDP scheme is called secure if for any PPT adversary $\mathcal{A}$ who can win the above game with a non-negligible probability, there exists an extractor that allows the client to extract the challenged data in polynomial time.

The existence of an extractor in this definition means that the adversary that follows any strategy can win the game above with probability negligibly larger than the probability with which the client is able to extract correct data. In our scheme, the probability of catching a cheating server is the same as in prior literature (analyzed in section 6.2).

The above definition ensures that the server is unable to pass the verification when replaying old or invalid data blocks. It is also possible to carry out a different type of a replay attack, where an outsider replays user's messages to the server. If such messages are accepted and applied by the server, the data the server stores will become incorrect and client's integrity verification will fail. To the best of our knowledge, this type of replay attacks has not been discussed in prior literature (and has not been discussed so far in this work). It is, however, not difficult to defend against such attacks using standard security mechanisms. In particular, assuming that the client and the server use authenticated channels for their communication, this type of replay attacks will be detected (and consequently prevented) if each transmitted request is guaranteed to be unique. This can be easily achieved by maintaining the state in the form of a counter and using the current value of the counter as the request id. Then if the server receives a request with an id which does not exceed a previously observed id, the server will disregard the request.

When the setting is generalized to multiple users who would like to have access to a shared content, the users still do not trust the storage server, which implies that the server does not enforce any access control with respect to retrieving or modifying the outsourced data blocks by the users. We thus distinguish between *centralized* and *distributed* environments. In the former, access control is 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 data storage to a third party 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. Thus, the original security definition applies. In the distributed environment where all users trust each other, the security experiment can be defined analogously, where the challenger now represents all users. 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 single-user scheme that does not have full support for revision control. In Section 7, we extend it with features of public verifiability, verification aggregation, multi-user access, and support for revision control.

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 that given a security parameter 1^κ produces 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. We term the new data structure a *block update tree*. In the update tree, each node corresponds to a range of data blocks on which an update (insertion, deletion, or modification) has been performed. The challenge with constructing the tree is 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, all operations on the remote storage (i.e., retrieve, insert, delete, modify, flush, and audit) 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 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 reduce 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 [Adelson-Velskii and Landis 1962]. 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 v of the update tree which indicates how the ranges of the blocks stored in the subtree rooted at v 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 will allow the client to verify that the data she receives corresponds to the most recent version (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 using its MAC. To support dynamic operations, the update tree is first initialized to empty. To modify a range of existing blocks, we insert a node in the tree that indicates that the version of the blocks in the range has increased. To insert a range of blocks, the client creates a node in the tree with the new blocks and also indicates that the indices of the blocks that follow need to be increased by the number of inserted blocks. This offset is stored at a single node in the tree, which removes the need to touch many nodes. To delete a range of blocks, the deleted blocks are marked with operation type "-1" in the tree and the offset of blocks that follow is adjusted accordingly. Then to perform an update (insert, delete, or modify), the client first modifies the tree, computes the MACs of the blocks to be updated, and communicates the blocks (for insertion and modification only) and the MACs to the server. Upon receiving the request, the server also modifies the tree according to the request and stores the received data and MACs. If the server behaves honestly, the server's update tree will be identical to the client's update tree (i.e., all changes to the tree are deterministic). To retrieve a range of blocks, the client receives a number of data blocks and their corresponding MACs from the server and verifies their integrity by using information stored in the tree.

The purpose of the tree is thus to ensure authenticity and freshness of the block parameters used in the MACs returned by the server. In other words, our solution will prevent the server from returning old data that could pass verification (known as replay attacks), which is achieved by ensuring that a new MAC on a block is always computed on a unique set of parameters that have not been previously used for any data block in the outsourced storage. These parameters include the above mentioned block's index, version number, operation type, and identification number that may have different meanings depending on the operation type. Therefore, to carry out a replay attack, the server needs to forge a MAC, which can be successful only with a negligible probability.

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 used with the update tree, GID and FID, 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. FID is incremented for each flush operation and each flush is assigned a unique identifier. For a given data block, the combination of its version number (see below) and FID will uniquely identify a given revision of the block. In addition to having 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 flush operation for all affected data blocks (at which point they are combined into one 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 FID at the time of the last flush on the affected data blocks (if no flush 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 FID) by observing its value, we use non-overlapping value ranges 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, data blocks that precede the range of the node and are stored within the subtree rooted at the node. 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 child 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). In Table I, we summarize variables used in update tree operations.

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 are Section 5.

— $\mathsf{UTInsert}(\mathsf{T}, \mathsf{s}, \mathsf{e})$ inserts a range of $(\mathsf{e} - \mathsf{s} + 1)$ new blocks at index s into the update tree T. The function returns a node v that corresponds to the newly inserted block range.

Table I. List of symbols used in update tree operations.

T	update tree on which various operations will be triggered
u or w	node of an update tree
s	start index of a block range on which the client triggers an operation
e	end index of a block range on which the client triggers an operation
dir	binary value indicating left or right direction
op	type of operation performed by the client (i.e., insert, modify, or delete)
$\mathsf{u.Op}$	type of operation associated with node u
$\mathsf{u.L}, \mathsf{u.U}$	start and end indices of the data blocks associated with node u
$\mathsf{u.V}$	number of modifications performed on the data blocks associated with node u
$\mathsf{u.ID}$	identification number associated with node u (i.e., FID or GID)
$\mathsf{u.R}$	offset associated with node u
$\mathsf{u.P_l}, \mathsf{u.P_r}, \mathsf{u.P_p}$	left child, right child, and parent nodes of node u

— $\mathsf{UTDelete}(\mathsf{T}, \mathsf{s}, \mathsf{e})$ 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 a sequence of nodes 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 might have never been modified.
— $\mathsf{UTFlush}(\mathsf{T}, \mathsf{s}, \mathsf{e})$ 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 index s and FID.

The protocols that define our solution are as follows:

(1) $\mathsf{KeyGen}(1^\kappa) \rightarrow \{\mathsf{sk}\}$: the client executes $\mathsf{sk} \leftarrow \mathsf{Gen}(1^\kappa)$.
(2) $\mathsf{Init}(\langle \mathsf{sk}, m_1, \ldots, m_K\rangle, \langle \perp \rangle) \rightarrow \{\langle \mathsf{M}_C\rangle, \langle \mathsf{M}_S, D\rangle\}$: 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 $1 \le i \le K$, 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 FID, 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) $\mathsf{Update}(\langle \mathsf{sk}, \mathsf{M}_C, \mathsf{op}, \mathsf{ind}, \mathsf{num}, m_{\mathsf{ind}}, \ldots, m_{\mathsf{ind}+\mathsf{num}-1}\rangle, \langle \mathsf{M}_S, D\rangle) \rightarrow \{\langle \mathsf{M}'_C\rangle, \langle \mathsf{M}'_S, D'\rangle\}$: the functionality of this protocol is determined by the operation type op and is defined as:
   (a) *Insert* $\mathsf{op} = 1$. 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 among 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, num, and t_i's to the server. For insertion and modification operations, m_i's are also 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 there is no integrity verification for each dynamic operation, saving computation and a round of interaction. Instead, the server records the operation in its metadata, which will be used for proving the integrity of returned blocks at retrieval time.

(4) $\mathsf{Retrieve}(\langle \mathsf{sk}, \mathsf{M}_C, \mathsf{ind}, \mathsf{num}\rangle, \langle \mathsf{M}_S, D\rangle) \to \{\langle m_{\mathsf{ind}}, \ldots, m_{\mathsf{ind}+\mathsf{num}-1}\rangle, \langle \perp \rangle\}$:
  (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 attributes L, U and pointer to the data blocks from u , 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 $C \leftarrow \mathsf{UTRetrieve}(\mathsf{M}_C, \mathsf{ind}, \mathsf{ind} + \mathsf{num} - 1)$. For each received 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 $j \geq 0$ is the data block's position within the node's data blocks.
  (d) If $b_i = 1$ for each m_i, the client is assured of integrity of the returned data and outputs the blocks. Otherwise, the client indicates integrity violation by outputting $\perp$ (possibly together with some data blocks).

(5) $\mathsf{Challenge}(\langle \mathsf{sk}, \mathsf{M}_C\rangle, \langle \mathsf{M}_S, D\rangle) \to b$:
  (a) The client chooses c distinct indices $i_1, \ldots, i_c$ at random between 1 and the current number of outsourced data blocks and sends them to the server.
  (b) The server executes $U_j \leftarrow \mathsf{UTRetrieve}(\mathsf{M}_S, i_j, i_j)$ for $j \in [1, c]$ (if some of the indices are adjacent, they can be combined into a single UTRetrieve operation). For each U_j, the server retrieves the attributes L, U, and pointer to the data block from M_S, locates the blocks and their tags $\langle m_{i_j}, t_{i_j}\rangle$ in D, and sends them to the client.
  (c) Upon the receipt of c data blocks and their corresponding tags $\langle m_{i_1}, t_{i_1}\rangle, \ldots, \langle m_{i_c}, t_{i_c}\rangle$, the client executes $U_j \leftarrow \mathsf{UTRetrieve}(\mathsf{M}_C, i_j, i_j)$ for each j (or for each range when some indices are adjacent). For each data block m_{i_j}, the client verifies its tag using the same computation as in Retrieve.
  (d) If verification of all c blocks was successful, the client outputs 1; otherwise, it outputs 0.

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

Note that, as in prior work [Ateniese et al. 2007; Erway et al. 2009], the performance of Challenge and Retrieve operations can be slightly optimized by sending a pseudorandom number generator seed to the server instead of the indices $i_1, \ldots, i_c$. Given the seed, the server can generate c indices, which removes the need to communicate them to the server.

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 flush ID to be 0.

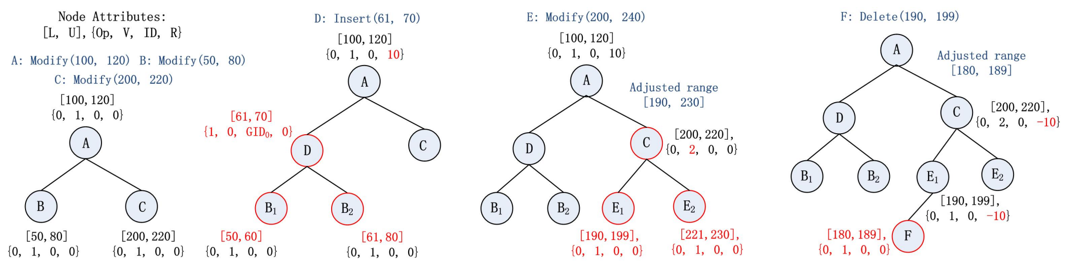


Fig. 1. Example of update tree operations.

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

The first operation is an insertion, the range of which falls on left side of node A's range and overlaps with the range of node B. To insert the blocks, we partition B's range into two (by creating two nodes) and make node D 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 offset stored at a node always applies to the blocks of the node itself and its right subtree only, and for this operation the offset of all ancestors of D for which D lies in the left subtree need to be updated.

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), causing it to overlap 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 falls on the right to A's range. This means that the indices in the original request are adjusted during traversal based on A's offset. 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 (i.e., ancestor nodes for which F is in the left subtree) is adjusted to reflect the change in block indices for these nodes themselves and their right children.

In what follows, we first present sub-routines called by the main algorithms followed by the algorithms for each operation.

5.1. Sub-routines

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

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

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

ALGORITHM 1: UTInsertNode(u, w, dir)

```
1:  if (dir = left) then
2:     while (1) do
3:        $u.R = u.R + w.Op \cdot (w.U - w.L + 1)$
4:        if ($u.P_l \neq$ 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 ($u.P_r \neq$ 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
```

lines 10–18 correspond to the the right subtree case. When w is inserted into the left subtree of 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 node u. When w is inserted into the right subtree of u, on the other hand, the range of w should be modified based on the offsets of its ancestors 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 purpose 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 (either a newly created or existing node) unless a delete node partitions the range.

When the range [s, e] 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.

In the algorithm, lines 3–11 corresponds to the case when the range we are searching for [s, e] is to the right of the current node's range. In that case, we first adjust the range [s, e] based on the information stored at the current node u (lines 4–5) and then either go down the node's right child or create a new node if the right subtree is not present. Lines 12–19 correspond to the case when the range [s, e] is to the left of the current node's range, in which case we first update the offset of the current node based on the operation (line 13) and then either proceed to the left child or create a new node if the left subtree is not present. Lines 20–32 then correspond to the case when the range of the current node overlaps [s, e]. In that case, we add the triple that consists of the current node u (the first found node the range of which overlaps with [s, e]) and the range [s, e] adjusted through traversal to the set returned to the calling routine. The tricky part here is to avoid returning nodes that correspond to deleted block ranges. If

ALGORITHM 2: UTFindNode(u, s, e, op)

```
1: $\mathsf{S} = \emptyset$
2: while (1) do
3:    if $(\mathsf{s} - \mathsf{u.R} > \mathsf{u.U})$ then
4:       $\mathsf{s} = \mathsf{s} - \mathsf{u.R} - \mathsf{u.Op} \cdot (\mathsf{u.U} - \mathsf{u.L} + 1)$
5:       $\mathsf{e} = \mathsf{e} - \mathsf{u.R} - \mathsf{u.Op} \cdot (\mathsf{u.U} - \mathsf{u.L} + 1)$
6:       if $\mathsf{u.P_r} \neq \mathrm{NULL}$ then
7:          $\mathsf{u} = \mathsf{u.P_r}$
8:       else
9:          $\mathsf{u}' = \mathsf{UTCreateNode}(\mathsf{s}, \mathsf{e}, 1 - |\mathsf{op}|, \mathsf{ID}, \mathsf{op})$
10:         insert $\mathsf{u}'$ as $\mathsf{u.P_r}$, and exit the loop;
11:      end if
12:   else if $((\mathsf{op} \leq 0$ and $\mathsf{e} - \mathsf{u.R} < \mathsf{u.L})$ or $(\mathsf{op} > 0$ and $\mathsf{s} - \mathsf{u.R} < \mathsf{u.L}))$ then
13:      $\mathsf{u.R} = \mathsf{u.R} + \mathsf{op}(\mathsf{e} - \mathsf{s} + 1)$
14:      if $\mathsf{u.P_l} \neq \mathrm{NULL}$ then
15:         $\mathsf{u} = \mathsf{u.P_l}$
16:      else
17:         $\mathsf{u}' = \mathsf{UTCreateNode}(\mathsf{s}, \mathsf{e}, 1 - |\mathsf{op}|, \mathsf{ID}, \mathsf{op})$
18:         insert $\mathsf{u}'$ as $\mathsf{u.P_l}$, and exit the loop;
19:      end if
20:   else
21:      if $(\mathsf{u.Op} \neq -1)$ then
22:         $\mathsf{S} = \mathsf{S} \cup \{\langle \mathsf{u}, \mathsf{s} - \mathsf{u.R}, \mathsf{e} - \mathsf{u.R} \rangle\}$
23:      else if $(\mathsf{u.Op} = -1$ and $\mathsf{op} = 1)$ then
24:         $\mathsf{S} = \mathsf{S} \cup \{\mathsf{UTFindNode}(\mathsf{u.P_r}, \mathsf{s} + \mathsf{u.U} - \mathsf{u.L} + 1 - \mathsf{u.R}, \mathsf{e} + \mathsf{u.U} - \mathsf{u.L} + 1 - \mathsf{u.R}, \mathsf{op})\}$
25:      else if $(\mathsf{u.Op} = -1$ and $(\mathsf{op} = 0$ or $\mathsf{op} = -1))$ then
26:         if $(\mathsf{s} - \mathsf{u.R} < \mathsf{u.L})$ then
27:            $\mathsf{S} = \mathsf{S} \cup \{\mathsf{UTFindNode}(\mathsf{u}, \mathsf{s}, \mathsf{u.L} - 1 + \mathsf{u.R}, \mathsf{op})\}$
28:            $\mathsf{S} = \mathsf{S} \cup \{\mathsf{UTFindNode}(\mathsf{u.P_r}, \mathsf{u.U} + 1, \mathsf{e} - \mathsf{u.R} + \mathsf{u.U} - \mathsf{u.L} + 1, \mathsf{op})\}$
29:         else if $(\mathsf{s} - \mathsf{u.R} \geq \mathsf{u.L})$ then
30:            $\mathsf{S} = \mathsf{S} \cup \{\mathsf{UTFindNode}(\mathsf{u.P_r}, \mathsf{s} - \mathsf{u.R} + \mathsf{u.U} - \mathsf{u.L} + 1, \mathsf{e} - \mathsf{u.R} + \mathsf{u.U} - \mathsf{u.L} + 1, \mathsf{op})\}$
31:         end if
32:      end if
33:      return S
34:   end if
35: end while
36: return $\{\langle \mathsf{u}', \mathrm{NULL}, \mathrm{NULL} \rangle\}$
```

such a node is found (lines 23–31), we need to ignore it and keep searching until we find a node that represents either an insertion or modification operation. This is the only situation when the set of size larger than 1 is returned. 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. 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 $[\mathsf{s}, \mathsf{e}]$ 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 $[\mathsf{s}, \mathsf{e}]$ and $[\mathsf{u.L}, \mathsf{u.U}]$. If the two ranges are identical (only lines 15–19 will be executed), several attributes of u (i.e., V, ID and Op) will be reset with values that depend on the operation type. If only the lower (only the upper) bound of the two ranges coincide (if statements on lines 1–3 and 4–7, resp.), we reset the range of the current node to $[\mathsf{s}, \mathsf{e}]$ (lines 15–19), fork a new node corresponding to the remaining range, and insert

ALGORITHM 3: UTUpdateNode(u, s, e, op)

```
if (u.L = s and e < u.U) then
    N′ = UTCreateNode(e + 1, u.U, u.V, ID, u.Op)
    UTInsertNode(u.P_r, N′, left)
else if (u.L < s and e = u.U) then
    N′ = UTCreateNode(u.L, s − 1, u.V, ID, u.Op)
    u.R = u.R + u.Op(s − u.L)
    UTInsertNode(u.P_l, N′, right)
else if (u.L < s and e < u.U) then
    N′ = UTCreateNode(u.L, s − 1, u.V, ID, u.Op)
    N″ = UTCreateNode(e + 1, u.U, u.V, ID, u.Op)
    u.R = u.R + u.Op(s − u.L)
    UTInsertNode(u.P_l, N′, right)
    UTInsertNode(u.P_r, N″, left)
end if
if (op = 0) then
    UTSetNode(u, s, e, u.V + 1, u.ID, u.Op)
else if (op = −1) then
    UTSetNode(u, s, e, u.V, u.ID, Op)
end if
return u
```

it into the right (resp., left) sub-tree of current node (lines 1–7). 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 to insert the new node in the tree. If neither the lower nor the upper bound match with each other, we fork two child nodes corresponding to the remaining head and tail ranges, and insert each of them into the left or right subtree of current node, respectively (lines 8–14). Similar to other cases, UTInsertNode is called to place the new nodes. The time complexity of this function is $O(\log n)$.

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.

For simplicity of exposition, we omit calls to UTBalance in the description of all main routines except UTFlush. In what follows, it is implicitly assumed that when a node is inserted into the tree (during UTInsert, UTModify, or UTDelete operation), the tree is checked whether re-balancing must take place and balanced if necessary.

Re-balancing involves node rotations, during which the offsets of some of the nodes directly involved in the rotation may need to be modified in order to correctly maintain information stored in the tree. In what follows, we detail such offset changes for all types of node rotations. Any unbalanced tree can be represented as one of the four trees in sub-figures 1, 2, 4, and 5 of Figure 2. Here N_1, N_2, and N_3 denote nodes, while A, B, C, and D denote subtrees with the same height. When the tree has the structure of sub-figure 1 (resp., 4), it requires one left and one right (resp., one right and one left) rotation to arrive at the balanced tree in sub-figure 3 (resp., 6). When the tree has the structure of sub-figure 2 (resp., 5), it requires a single right (resp., left) rotation to arrive at the balanced tree in sub-figure 3 (resp., 6). When performing any of the four rotations above (i.e., 1-to-2, 2-to-3, 4-to-5, or 5-to-6), there is always a single node whose left sub-tree will be restructured, which will require changes to the node's offset. For instance, when performing a left rotation to transform sub-figure 1 to 2, the only node whose left subtree gets restructured is N_2, which previously had a single subtree B as

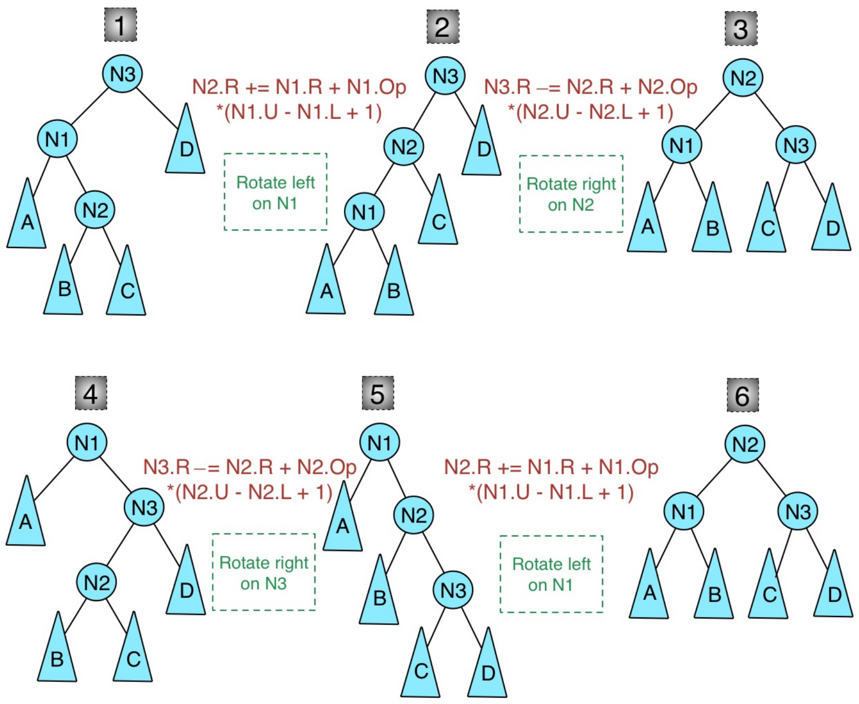


Fig. 2. Offset modification during tree balancing.

its left child and now has an additional subtree A and node N_1 added in. Therefore, to adjust the offset of N_2, the client needs to take into account the offset of node N_1 and its block range when the node's operation type is either insertion or a deletion. A similar pattern can be observed for the other left rotation (i.e., 5-to-6) and two right rotations (i.e., 2-to-3 and 4-to-5), and a similar approach can be applied to modify the offset of the corresponding node (N_2, N_3 and N_3) as well.

$\mathsf{UTFree}(\mathsf{u})$ frees the memory occupied by the subtree rooted at node u and its complexity is linear in the subtree size.

5.2. Main routines

$\mathsf{UTInsert}(\mathsf{T}, \mathsf{s}, \mathsf{e})$ updates the update tree T for an insert request with the block range $[\mathsf{s}, \mathsf{e}]$. This function creates and inserts a new node in the tree, and 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, $\mathsf{UTFindNode}$ 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 ($\mathsf{s}' \neq \mathrm{NULL}$) 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.

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

```
1:  $\mathsf{GID} = \mathsf{GID} + 1$
2:  $\mathsf{S} = \mathsf{UTFindNode}(\mathsf{T}, \mathsf{s}, \mathsf{e}, 1)$
3:  $(\mathsf{u}', \mathsf{s}', \mathsf{e}') = \mathsf{S}[0]$
4:  if $(\mathsf{s}' \neq \mathrm{NULL})$ then
5:     $\mathsf{u} = \mathsf{UTCreateNode}(\mathsf{u}'.\mathsf{L}, \mathsf{u}'.\mathsf{U}, \mathsf{u}'.\mathsf{V}, \mathsf{u}'.\mathsf{ID}, \mathsf{u}'.\mathsf{Op})$
6:     $\mathsf{UTSetNode}(\mathsf{u}', \mathsf{s}', \mathsf{e}', 0, \mathsf{GID}, 1)$
7:     if $(\mathsf{u}.\mathsf{L} = \mathsf{s}')$ then
8:        $\mathsf{UTInsertNode}(\mathsf{u}'.\mathsf{P_r}, \mathsf{u}, \mathsf{left})$
9:     else
10:       $\mathsf{u}'.\mathsf{R} = \mathsf{u}'.\mathsf{R} + \mathsf{u}.\mathsf{Op} \cdot (\mathsf{s}' - \mathsf{u}.\mathsf{L})$
11:       $\mathsf{w}_1 = \mathsf{UTCreateNode}(\mathsf{u}.\mathsf{L}, \mathsf{s}' - 1, \mathsf{u}.\mathsf{V}, \mathsf{u}.\mathsf{ID}, \mathsf{u}.\mathsf{Op})$
12:       $\mathsf{w}_2 = \mathsf{UTCreateNode}(\mathsf{s}', \mathsf{u}.\mathsf{U}, \mathsf{u}.\mathsf{V}, \mathsf{u}.\mathsf{ID}, \mathsf{u}.\mathsf{Op})$
13:       $\mathsf{UTInsertNode}(\mathsf{u}'.\mathsf{P_l}, \mathsf{w}_1, \mathsf{right})$
14:       $\mathsf{UTInsertNode}(\mathsf{u}'.\mathsf{P_r}, \mathsf{w}_2, \mathsf{left})$
15:       $\mathsf{UTFree}(\mathsf{u})$
16:    end if
17: end if
18: return $\mathsf{u}'$
```

$\mathsf{UTInsert}$ can add at most two nodes to the tree, which may increase the height differences of some node's child subtrees by 1 and require re-balancing. To address this, we update the heights of the subtrees at the point of node insertion and all of their ancestors in the tree. This involves traversing up the tree to the root node and re-balancing if necessary at some node on the path to the root. Note that because the height difference can increase by at most 1, $\mathsf{UTBalance}$ will need to be invoked at most once and will perform at most one AVL algorithm's node rotation using constant work.

$\mathsf{UTModify}(\mathsf{u}, \mathsf{s}, \mathsf{e})$, when called with $\mathsf{u} = \mathsf{T}$, updates the update tree T based on a modification request with block range $[\mathsf{s}, \mathsf{e}]$ and returns the set of nodes corresponding to the range. It is given in Algorithm 5. The algorithm creates a node for the range if T is empty (lines 2–4), and otherwise it invokes $\mathsf{UTFindNode}$ 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:

(1) If the adjusted range $[\mathsf{s}_i, \mathsf{e}_i]$ is contained in u_i's range $[\mathsf{u}_i.\mathsf{L}, \mathsf{u}_i.\mathsf{U}]$, u_i is the only node to be modified, and this is handled by $\mathsf{UTUpdateNode}$ (lines 10–11).
(2) If the adjusted range $[\mathsf{s}_i, \mathsf{e}_i]$ overlaps with the ranges of u_i and its left subtree (lines 12–14), the algorithm first updates the range of u_i by calling $\mathsf{UTUpdateNode}$ and then recursively calls another $\mathsf{UTModify}$ to update the remaining nodes in the left subtree. Exactly the same logic is used when $[\mathsf{s}_i, \mathsf{e}_i]$ overlaps with the ranges of u_i and its right subtree (lines 15–17).
(3) If the adjusted range $[\mathsf{s}_i, \mathsf{e}_i]$ overlaps with the range of u_i and both of its subtrees (lines 18–21), the algorithm first updates u_i using $\mathsf{UTUpdateNode}$, and then calls $\mathsf{UTModify}$ twice to handle the changes to the left and right subtrees of u_i, respectively.

Let us consider the example in Figure 3 to better understand how $\mathsf{UTModify}$ works. We start with the last update tree in Figure 1 that contains 8 nodes (sub-figure 1). Suppose that the client triggers a modify operation with the range $[120, 195]$. This range partially overlaps with the ranges of nodes A and E_1 in the tree. In particular, because node A has offset $R = 10$ due to a previously triggered insert operation, the up-to-date range of A is $[110, 130]$. The offset of node E_1, on the other hand, is 0 (which is the sum

ALGORITHM 5: UTModify(u, s, e)

```
1: $\mathsf{C} = \emptyset$
2: if ($\mathsf{u} = \mathrm{NULL}$) then
3:     $\mathsf{w} = \mathsf{UTCreateNode}(\mathsf{s}, \mathsf{e}, 1, \mathsf{FID}, 0)$
4:     $\mathsf{C} = \mathsf{C} \cup \{\mathsf{w}\}$
5: else
6:     $\mathsf{S} = \mathsf{UTFindNode}(\mathsf{u}, \mathsf{s}, \mathsf{e}, 0)$
7:     for $i = 0$ to $|\mathsf{S}| - 1$ do
8:         $(\mathsf{u}_i, \mathsf{s}_i, \mathsf{e}_i) = \mathsf{S}[i]$
9:         if ($\mathsf{s}_i \neq \mathrm{NULL}$) then
10:            if ($\mathsf{u}_i.\mathsf{L} \leq \mathsf{s}_i$ and $\mathsf{e}_i \leq \mathsf{u}_i.\mathsf{U}$) then
11:                $\mathsf{C} = \mathsf{C} \cup \{\mathsf{UTUpdateNode}(\mathsf{u}_i, \mathsf{s}_i, \mathsf{e}_i, 0)\}$
12:            else if ($\mathsf{s}_i < \mathsf{u}_i.\mathsf{L}$ and $\mathsf{e}_i \leq \mathsf{u}_i.\mathsf{U}$) then
13:                $\mathsf{C} = \mathsf{C} \cup \{\mathsf{UTUpdateNode}(\mathsf{u}_i, \mathsf{u}_i.\mathsf{L}, \mathsf{e}_i, 0)\}$
14:                $\mathsf{C} = \mathsf{C} \cup \{\mathsf{UTModify}(\mathsf{u}_i, \mathsf{s}_i + \mathsf{u}_i.\mathsf{R}, \mathsf{u}_i.\mathsf{L} - 1 + \mathsf{u}_i.\mathsf{R})\}$
15:            else if ($\mathsf{u}_i.\mathsf{L} \leq \mathsf{s}_i$ and $\mathsf{u}_i.\mathsf{U} < \mathsf{e}_i$) then
16:                $\mathsf{C} = \mathsf{C} \cup \{\mathsf{UTUpdateNode}(\mathsf{u}_i, \mathsf{s}_i, \mathsf{u}_i.\mathsf{U}, 0)\}$
17:                $\mathsf{C} = \mathsf{C} \cup \{\mathsf{UTModify}(\mathsf{u}_i, \mathsf{u}_i.\mathsf{U} + 1 + \mathsf{u}_i.\mathsf{R}, \mathsf{e}_i + \mathsf{u}_i.\mathsf{R})\}$
18:            else if ($\mathsf{s}_i < \mathsf{u}_i.\mathsf{L}$ and $\mathsf{e}_i > \mathsf{u}_i.\mathsf{U}$) then
19:                $\mathsf{C} = \mathsf{C} \cup \{\mathsf{UTUpdateNode}(\mathsf{u}_i, \mathsf{u}_i.\mathsf{L}, \mathsf{u}_i.\mathsf{U}, 0)\}$
20:                $\mathsf{C} = \mathsf{C} \cup \{\mathsf{UTModify}(\mathsf{u}_i, \mathsf{s}_i + \mathsf{u}_i.\mathsf{R}, \mathsf{u}_i.\mathsf{L} - 1 + \mathsf{u}_i.\mathsf{R})\}$
21:                $\mathsf{C} = \mathsf{C} \cup \{\mathsf{UTModify}(\mathsf{u}_i, \mathsf{u}_i.\mathsf{U} + 1 + \mathsf{u}_i.\mathsf{R}, \mathsf{e}_i + \mathsf{u}_i.\mathsf{R})\}$
22:            end if
23:        else
24:            $\mathsf{C} = \mathsf{C} \cup \{\mathsf{u}_i\}$
25:        end if
26:    end for
27: end if
28: return C
```

of $\mathsf{R} = 10$ at A and $\mathsf{R} = -10$ at E_1 that correspond to 10 inserted and 10 deleted blocks before the position of the blocks of E_1) and its up-to-date range is $[190, 199]$. Therefore, the modify operation will add three new nodes that correspond to (i) the range $[120, 130]$ that overlaps with A's range, (ii) the range $[190, 195]$ that overlaps with E_1's range and (iii) the "gap" range $[131, 189]$ that lies between the ranges of A and E_1. UTModify handles this by first invoking UTFindNode, which returns A as the first node on the path from the root the range of which overlaps with the range of the operation. A subsequent call to UTUpdateNode on the range $[120, 130]$ results in the range of A being updated and a new node G_1 being inserted into the tree. This operation is reflected in sub-figure 2 of Figure 3. After this insertion, the subtree rooted at E_1 becomes unbalanced and a call to UTBalance yields a right rotation and the tree depicted in sub-figure 3 of Figure 3.

Next, UTModify makes another call to UTModify on the remaining range, which in turn calls UTFindNode. UTFindNode returns E_1, which is now the only node in the tree the range of which overlaps with the unprocessed portion of the operation's range. The function subsequently invokes UTUpdateNode on the range $[190, 195]$, which results in the range of E_1 being updated and a new node G_2 being inserted into the tree. The result is shown in sub-figure 4 of Figure 3. In the resulting tree, the subtree rooted at node C is unbalanced and after rotating the nodes we obtain the tree in sub-figure 5 of Figure 3.

Now the function will recursively invoke UTModify one more time on the "gap" range $[131, 189]$. Because the range no longer overlaps any range of the nodes in the update tree, UTFindNode will insert a new node G_3 at the correct place in the tree. Sub-figure 6 in Figure 3 reflect the result of this operation (after which the tree needs to be re-

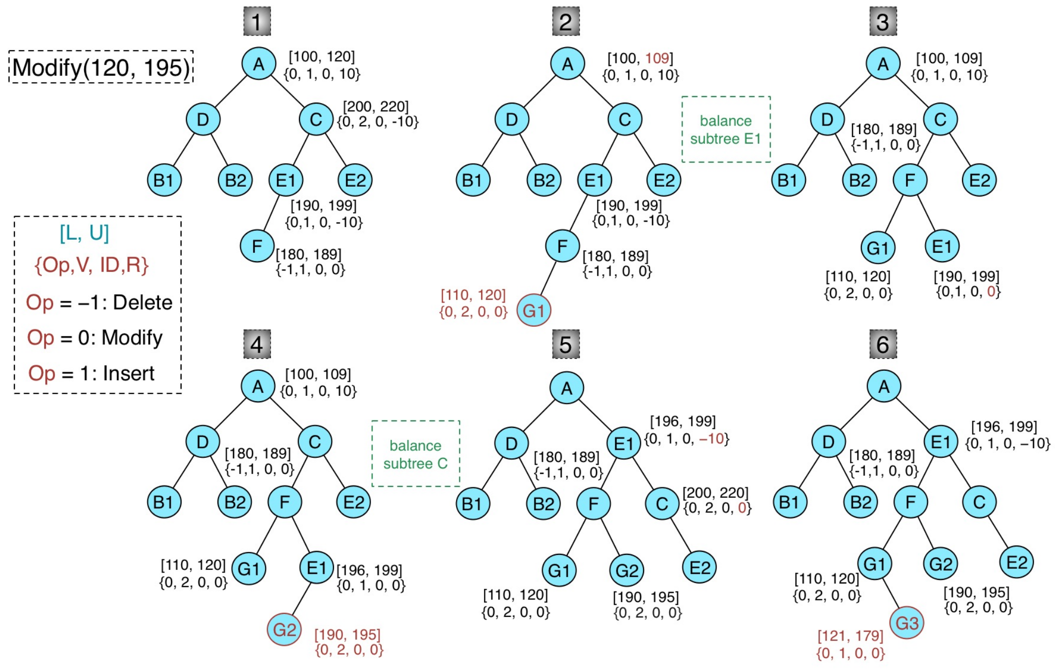


Fig. 3. Example of the Modify algorithm.

balanced again at node A). Note that node rotations performed during re-balancing may require updates to the offset values of some of the nodes whose position relative to the parent and/or child nodes changes, as was detailed in Figure 2.

In a single call to UTModify (and UTDelete below, which is similar to UTModify) multiple nodes may need to be created and added to the tree, which can make the update tree unbalanced. Each new node, however, still will require at most one call to UTBalance of constant work for some node on the path to the root. We note that finding the place of node rotation will not require traversing the entire path to the root for each inserted node because changes are localized. In particular, node rotations happen one level higher for each subsequent node insertion because nodes are inserted next to each other, and it can be observed from the example in Figure 3.

$\mathsf{UTDelete}(\mathsf{u}, \mathsf{s}, \mathsf{e})$, when called with $\mathsf{u} = \mathsf{T}$, updates the update tree T based on a deletion request with the block range $[\mathsf{s}, \mathsf{e}]$. It does not delete any node from T, but rather finds all nodes whose ranges fall into $[\mathsf{s}, \mathsf{e}]$, sets their operation types to -1, and returns them to the caller. UTDelete is very similar to UTModify and is given in Algorithm 8 in the appendix.

$\mathsf{UTRetrieve}(\mathsf{u}, \mathsf{s}, \mathsf{e})$, when called with $\mathsf{u} = \mathsf{T}$, returns the nodes whose ranges overlap with $[\mathsf{s}, \mathsf{e}]$. Algorithm 6 gives the details. Similar to UTModify, multiple cases can occur based on the overlap of the range $[\mathsf{s}, \mathsf{e}]$ with $[\mathsf{u.L}, \mathsf{u.U}]$. Unlike UTModify, the function does not call UTFindNode but rather traverses the tree itself, and thus there are additional cases to consider. In addition to when $[\mathsf{s}, \mathsf{e}]$ is contained in u's range (lines 6–11), overlaps with u's range, but starts before or ends after u's range (lines 16–21) or both (lines 22–25), we also have the case when $[\mathsf{s}, \mathsf{e}]$ does not overlap with u's range, i.e., $[\mathsf{s}, \mathsf{e}]$ is before or after u's range (lines 12–15). In all cases when $[\mathsf{s}, \mathsf{e}]$ or a part of it does not overlap with the range of the current node u, the algorithm is called recursively on one or both

ALGORITHM 6: $\mathsf{UTRetrieve}(\mathsf{u}, \mathsf{s}, \mathsf{e})$

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

subtrees until a node whose range overlaps with $[\mathsf{s}, \mathsf{e}]$ is found (lines 5–26) or not found (lines 2–4). The bottom of recursion is reached when the (partitioned) range on which the function is called is contained within the range of the current node (lines 6–9) or the range is not present in the tree. Care should be exercised when the returned node represents a deletion operation (lines 9–10). 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.

$\mathsf{UTFlush}(\mathsf{T}, \mathsf{s}, \mathsf{e})$ replaces all nodes in the tree T that correspond to blocks in the range $[\mathsf{s}, \mathsf{e}]$ with a single node with the range $[\mathsf{s}, \mathsf{e}]$. The goal of a flush 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 7 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 $[\mathsf{s}', \mathsf{e}']$, 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 $[\mathsf{s}, \mathsf{e}]$ and new FID (line 49). The routine returns adjusted lower bound s' and updated FID.

To illustrate how the tree is being traversed and balanced in the process, let us consider the example in Figure 4. In the figure, u_1 and u_2 correspond to the tree nodes that incorporate s and e, respectively, and T' is their lowest common ancestor. The

ALGORITHM 7: $\mathsf{UTFlush}(\mathsf{T}, \mathsf{s}, \mathsf{e})$

```
1: $\mathsf{FID} = \mathsf{FID} + 1$
2: $\mathsf{S} = \mathsf{UTRetrieve}(\mathsf{T}, \mathsf{s}, \mathsf{e})$
3: set $u_1$ and $u_2$ to the nodes in S from T with the smallest and largest indices, respectively
4: set $T'$ to the lowest common ancestor of $u_1$ and $u_2$ in S
5: set $s'$ to s adjusted through traversal
6: set $e'$ to e adjusted through traversal
7: if $s' \neq u_1.L$ then
8:    $\mathsf{UTSetNode}(u_1, u_1.L, s-1, u_1.V, u_1.ID, u_1.Op)$
9:    $w_1 = \mathsf{UTCreateNode}(s, u_1.U, u_1.V, u_1.ID, u_1.Op)$
10:   $\mathsf{UTInsertNode}(u_1.P_r, w_1, \mathsf{left})$
11: end if
12: if $e' \neq u_2.U$ then
13:   $\mathsf{UTSetNode}(u_2, e'+1, u_2.U, u_2.V, u_2.ID, u_2.Op)$
14:   $w_2 = \mathsf{UTCreateNode}(u_2.L, e', u_2.V, u_2.ID, u_2.Op)$
15:   $\mathsf{UTInsertNode}(u_2.P_l, w_2, \mathsf{right})$
16: end if
17: if $u_1 \neq u_2$ then
18:   if $(u_1.P_r \neq \text{NULL})$ and $(u_1$ is not $u_2$'s ancestor$)$ then
19:     $\mathsf{UTFree}(u_1.P_r)$
20:     $u_1 = \mathsf{UTBalance}(u_1)$
21:   end if
22:   if $(u_2.P_l \neq \text{NULL})$ and $(u_2$ is not $u_1$'s ancestor$)$ then
23:     $\mathsf{UTFree}(u_2.P_l)$
24:     $u_2 = \mathsf{UTBalance}(u_2)$
25:   end if
26:   while $(u_1.P_p \neq T')$ and $(u_1 \neq T')$ do
27:     if $u_1 = u_1.P_p.P_l$ then
28:       $\mathsf{UTFree}(u_1.P_p.P_r)$
29:       set $u_1$ to be a direct child of $u_1.P_p.P_p$
30:     else if $u_1 = u_1.P_p.P_r$ then
31:       $u_1 = \mathsf{UTBalance}(u_1.P_p)$
32:     end if
33:   end while
34:   while $(u_2.P_p \neq T')$ and $(u_2 \neq T')$ do
35:     if $u_2 = u_2.P_p.P_r$ then
36:       $\mathsf{UTFree}(u_2.P_p.P_l)$
37:       set $u_2$ to be a direct child of $u_2.P_p.P_p$
38:     else if $u_2 = u_2.P_p.P_l$ then
39:       $u_2 = \mathsf{UTBalance}(u_2.P_p)$
40:     end if
41:   end while
42:   while $(T' \neq \text{NULL})$ do
43:     $T' = \mathsf{UTBalance}(T')$
44:     $T' = T'.P_p$
45:   end while
46: else
47:   remove $u_1$ from the tree
48: end if
49: $\mathsf{UTModify}(\mathsf{T}, \mathsf{s}, \mathsf{e})$
50: return $(s', \mathsf{FID})$
```

nodes and their subtrees shown using dotted lines corresponds to the nodes whose entire subtrees are to be removed. We start by traversing the path from u_1 to T' and removing nodes in the left subtree of T' with block indices larger than adjusted s.

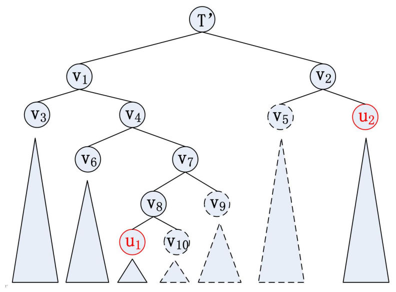


Fig. 4. Illustration of the flush algorithm.

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 together with their subtrees, nodes v_8 and v_7 are also removed, and u_1 takes the place of v_7. At this point u_1 becomes the right child of its parent v_4, and we balance the subtree rooted at u_1's parent and make u_1 point to v_4 (lines 30–32 of the algorithm). This re-balancing procedure is repeated for the child subtrees of v_1, until the left subtree of T' is completely balanced and 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 or re-balancing are performed when u_2 is the right child or the left child of its parent, respectively (lines 34–40 of the algorithm). For a example, in Figure 4 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 is to re-balance the subtree rooted at T' and the subtrees of all 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 re-balancing procedure. We obtain a balanced tree T with all nodes in the range $[\mathsf{s},\mathsf{e}]$ removed and insert one node corresponding to this range that indicates that the flush number FID of all blocks in the range $[\mathsf{s},\mathsf{e}]$ has been increased.

6. ANALYSIS OF THE SCHEME

6.1. 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 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.

We note that in the worst case $O(\mathsf{e}-\mathsf{s})$ nodes are inserted into the tree, but all nodes with which the block range specified in the operation overlaps must be consecutive nodes in the tree. Let t denote the number of nodes whose ranges overlap with the operation's range. The above means that on average $t/2$ overlapping nodes will be leaf nodes the tree, $t/4$ overlapping nodes will be at distance 1 from the leaf level, $t/8$ nodes will be at distance 2, etc. This is relevant because UTModify traverses the tree in the top-down fashion until the next unprocessed overlapping node is encountered in the tree. All new nodes, however, are inserted as leaf nodes, which means that their point of insertion might be at some distance from the place where insertion is triggered. The next thing to notice is that the extra work associated with tree traversal for the purpose of node creation is still $O(t)$ because the distance from the overlapping nodes to the bottom of the tree is on average small. In particular, the number of traversed nodes can be characterized by summation $t\sum_{i=1}^{\log t}\frac{i}{2^i}$, which approaches $2t$ from below as t approaches infinity. This means that we can still maintain $O(\log n+\min(n,\mathsf{s}-\mathsf{e}))$ complexity.

UTRetrieve does not add nodes to the tree and its complexity is similarly $O(\log n+\min(n,\mathsf{e}-\mathsf{s}))$. Lastly, UTFlush 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 UTFlush is less trivial to compute, we analyze it in more detail. Note that UTFlush calls the re-balancing function UTBalance, the worst case complexity of which is $O(\log n)$, at most $O(\log n)$ times. However, due to the careful construction of the tree and the flush function, the total number of operations that rearrange the tree is only $O(\log n)$ (plus node deallocation time that gives the overall complexity of the function).

THEOREM 6.1. *The time complexity of* UTFlush *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, re-balancing is triggered when the height difference is 2 or more. More generally, any constant $c\geq 2$ can be used as the criterion for re-balancing, 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 thus concentrate on showing that the re-balancing itself takes $O(\log n)$ time.

Recall that the complexity of UTBalance (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 UTFlush we first locate the nodes that contain the smallest and largest block indices falling within the flush range (u_1 and u_2, respectively), and then call either UTBalance or UTFree while 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 re-balancing 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 4, 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 re-balancing 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 4, 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 re-balanced 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 re-balancing the tree rooted at T' is linear in the sum of the number of calls to UTFree and UTBalance, or the height of the tree T, and is therefore $O(\log n)$. The flush 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 parent and moves up the tree (i.e., the initial re-balancing cost can be at most linear in the distance from T' to the leaf level of the tree, but the cost of each consecutive re-balancing operation is constant). We obtain that the overall re-balancing cost of UTFlush is $O(\log n)$ and its overall cost is $O(\log n + \min(n, \mathsf{e} - \mathsf{s})))$ $\square$

Next, we analyze the complexity of the protocols themselves. It is easy to see that Init has time and communication complexity of K, 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. Challenge has computation and communication complexities of $O(\log n + c)$ and $O(c)$, respectively, where constant c bounds success probability of a cheating server. Lastly, the complexities of Flush are $O(\log n + \mathsf{num})$ and $O(\mathsf{num})$, because the client needs to communicate num MACs to the server.

6.2. Security analysis

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

THEOREM 6.2. *The proposed update tree scheme is a secure DPDP scheme assuming the security of MAC construction.*

PROOF. The challenger runs $\mathsf{sk} \leftarrow \mathsf{KeyGen}(1^\kappa)$, initializes the data blocks according to the adversary $\mathcal{A}$'s request, and honestly performs and answers $\mathcal{A}$'s queries. Suppose that $\mathcal{A}$ wins the security 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 since the data blocks are part of the 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, FID, 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 FID are always different). In that case, we use $\mathcal{A}$'s advantage to break the security of the MAC scheme as follows: instead of having access to the MAC key, the challenger answers $\mathcal{A}$'s queries by querying $\mathsf{Mac}_{\mathsf{sk}}(\cdot)$ on the content generated according to the DPDP scheme and stores all MACs that it sends to $\mathcal{A}$. If $\mathcal{A}$ wins the security game and at least one of the returned MACs is not among the MACs that the challenger produced, but passes verification, the challenger outputs that MAC and the message on which it was produced as a successful MAC forgery. It is clear that if $\mathcal{A}$ is able to win the security game with a non-negligible probability such that the correct content cannot be recovered, the challenger can break the security of the MAC scheme with a non-negligible probability, which contradicts our assumption of security of the MAC scheme. □

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. Maintaining constant client storage

In our scheme, as described, the client maintains an update tree that grows over time. If the flush command is not used, the block ranges stored in the update tree may eventually become so partitioned that the tree has to store a node per outsourced data block (i.e., be linear in the size of the outsourced storage and comparable to the size of MHT and skip list in other solutions), losing its benefits. A growing amount of local storage is also more susceptible to corruption or other fault than a state of constant size. For that reason, we propose that the client specifies the bound on the local storage that it can maintain.

If at some point of the system operation the update tree is to exceed the constant threshold, we propose the following to reduce the local state below the desired threshold. The client can trigger the flush operation periodically to reduce the size of the update tree below the threshold. Note that the flush operation can be called on any node of the tree and collapses the subtree rooted at that node into a single node. This allows for gradual reduction of the size of the tree and intelligent strategies for choosing a portion of the tree to flush. For example, one possibility is to flush a range of blocks that have not been modified recently. The rationale for this choice is that if flush is applied to data block ranges that are actively being edited, the number of nodes in the update tree will likely rapidly increase after the flush if the blocks continue to be modified. Another possibility is to locate a subtree with nodes corresponding to small ranges of blocks. The rationale for this choice is that calling flush on nodes with small ranges will be most effective at reducing the tree size and minimizing the amount of computation and communication that the client has to do to perform this command (i.e., to update the MACs of the corresponding data blocks stored at the server).

While a client is free to use any desired threshold on the local storage, for a single user, we recommend setting the threshold to a least few MBs. This is feasible only for desktop computers, but even for constrained devices such as mobile phones. As we demonstrate in section 8, this amount of storage is very likely to be sufficient for the client to store the entire data structure locally (without a need to do a flush) for a long period of time because this size corresponds to maintaining information about hundreds of thousands of dynamic operations.

7.2. Resilience to metadata loss

In our scheme, as currently described, the client's ability to verify integrity and availability of the outsourced data relies on having the metadata, i.e., the update tree, intact. If the client's update tree is corrupted or lost for any reason including system's malfunction or compromise, the client can no longer use it to verify correctness of the returned blocks.[2]To improve resiliency of our solution to the loss of client's metadata,

[2]Note that even if the client loses all of its metadata, the server is still limited in the scope of the attack it can perform. The server can return an outdated version of a block or a different block if the block indices have changed, but the server will still have to return a block which the client previously stored with the server (because of MAC unforgeability).

we suggest that the client periodically computes and stores with the server information that can authenticate the update tree. In particular, the client computes a MAC of the update tree (using some specific representation of the tree) together with its version number and sends the MAC to the server. After receiving the MAC, the server takes a snapshot of the current update tree and separately stores it together with the MAC. In the event of metadata loss, the client can request from the server the most recently stored MAC of its tree together with the corresponding version of the server's update tree and use the MAC to verify the integrity of the update tree returned by the server. An attractive feature of this strategy (unlike the traditional data backup) is that the client does not need to send any portion of the update tree to the server, as the server always maintains an exactly the same copy of the update tree. This means that the client can store a MAC that authenticates the update tree with the server very frequently resulting in minimal loss of updates to the outsourced storage in the event of corruption or loss of client's storage. The only requirement is that the client should to be able to retrieve the latest version of the update tree backup to be able to carry out the recovery and thus the client will want to replicate the version number on different devices or storage media to guarantee its availability.

Notice that the above strategy is also resilient to replay attacks, in which the server attempts to return an old version of the update tree at the request of tree recovery. This is because at the time of verifying integrity of the retrieved update tree, the client first recomputes the MAC using the up-to-date version number (for which correctness is guaranteed) and the returned update tree, and then compares the result to the returned MAC. The client relies on the update tree only if the two MACs are equal. If the server returns an old update tree, in order to pass verification, it needs to forge a MAC using the old update tree and the up-to-date version number, for which the success probability is negligible.

7.3. Public verifiability

To enable outsourcing of periodic audits to a third party auditor, we replace the use of a MAC scheme for data authentication with a signature scheme. In more detail, let a signature scheme be defined by three algorithms $(\mathsf{SGen}, \mathsf{Sign}, \mathsf{SVerify})$, where SGen on input security parameter 1^κ outputs a public-private key pair (pk, sk), Sign on input private key sk and message $m \in \{0,1\}^*$ outputs signature σ, and $\mathsf{SVerify}$ on input a public key pk, message m, and signature σ outputs a bit b, which is set to 1 when the verification was successful. The security requirement is that after observing signatures on polynomially many messages of adversary's choice, an adversary without access to the private key is unable to forge a valid signature on a new message with more than a negligible probability. We then modify our DPDP scheme to have KeyGen algorithm execute $(pk, sk) \leftarrow \mathsf{SGen}(1^\kappa)$, after which the client publishes pk and privately stores sk, and replace every instance of $t \leftarrow \mathsf{Mac}_{\mathsf{sk}}(m)$ with $\sigma \leftarrow \mathsf{Sign}_{sk}(m)$ and every instance of $b \leftarrow \mathsf{Verify}_{\mathsf{sk}}(m, t)$ with $b \leftarrow \mathsf{SVerify}_{pk}(m, \sigma)$ in Init, Update, $\mathsf{Retrieve}$, $\mathsf{Challenge}$, and Flush algorithms. Now verification can be performed by any third party with possession of public key pk and an up-to-date copy of the update tree T while retaining all other features of the scheme.

Note that, similar to other publications, a third-party auditor in our scheme does not learn any data stored at the server during an audit because any data is stored at the server in encrypted form (assuming the data is sensitive and needs protection).

7.4. Verification aggregation

To improve efficiency of periodic $\mathsf{Challenge}$ queries, prior literature suggested aggregating the data blocks used in the verification to reduce communication overhead. That is, instead of transmitting c data blocks in response to a Challenge query, the server combines them into a single data block without weakening security guarantees. Note that all other operations (updates and retrieval) require transmission of the blocks themselves.

Our scheme can also be easily modified to support blockless Challenge queries. In particular, we can employ the same block aggregation mechanism as suggested in [Erway et al. 2009]. Now the content of a data block d used in computing its MAC (or signature) is replaced with a short tag $\mathcal{T}(d)$, which is computed as $g^d \bmod N$, where N is a product of two large primes and g is an element of high order in $\mathbb{Z}_N^*$. Then during Challenge, the server send tags $\mathcal{T}(m_{i_1}), \ldots, \mathcal{T}(m_{i_c})$ for blocks indices specified in the query instead of the blocks themselves, the corresponding MACs, and a combined block m. The block m is computed as $m = \sum_{j=1}^{c} a_j m_{i_j}$, where a_j's are random values sent by the client together with the challenge. Upon receiving this response, the client first verifies the authenticity of each MAC and then checks that $g^m \bmod N = \prod_{j=1}^{c} \mathcal{T}(m_{i_j})^{a_j} \bmod N$. If all checks succeed, the verification is successful. We obtain that the tags are normally significantly shorter than the data blocks themselves, and the communication cost of Challenge is substantially reduced.

This modification to the scheme affects how the client reconstructs the data for the purpose of the security definition. In particular, it is required that if an adversary wins the security game with a non-negligible probability, the challenger can extract the data blocks from the adversary's response. Now the data blocks are extracted by (i) interacting with the adversary c times, where each time a different set of a_j's is used with the same block indices and (ii) solving the system of linear equations $a_1 m_{i_1} + a_2 m_{i_2} + \ldots + a_c m_{i_c} = m$ formed by varying a_j's and m. We refer the reader to [Erway et al. 2009] for more information.

7.5. Enabling multi-user support

We next extend the 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.

7.5.1. Distributed setting. In this setting, the users trust each other; they locally maintain update trees and notify each other about updates to 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[3], 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.

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

Communication between the users can be synchronized via any suitable mechanism (e.g., using Paxos [Lamport 1978]). There, however, needs to be a mechanism for resolving conflicting updates. Two simultaneous updates are said to be conflicting if changing the order in which they are executed with the original parameters is not guaranteed to result in the same storage content. In our context, an operation is characterized by the operation type and its range of block indices. Then two updates with parameters $(\mathsf{op}_1, \mathsf{ind}_1, \mathsf{num}_1)$ and $(\mathsf{op}_2, \mathsf{ind}_2, \mathsf{num}_2)$, where $\mathsf{ind}_1 \leq \mathsf{ind}_2$, are non-conflicting if (i) op_1 is a modify operation and (ii) $\mathsf{ind}_1 + \mathsf{num}_1 \leq \mathsf{ind}_2$. All other updates are conflicting. We categorize all conflicts into *automatically resolvable* (locally resolved by each user) and *manually resolvable* (require user interaction), which are detailed below.

— *Automatically resolvable conflicts* occur when an insertion (resp., deletion) with $\mathsf{ind}_1, \mathsf{num}_1$ is triggered simultaneously with another operation on a succeeding, but not overlapping range $\mathsf{ind}_2, \mathsf{num}_2$, i.e., $\mathsf{ind}_1 < \mathsf{ind}_2$ (resp., $\mathsf{ind}_1 + \mathsf{num}_1 < \mathsf{ind}_2$). Such conflicts can be automatically resolved without requiring user intervention if the users agree on the strategy for resolving them.
— *Manually resolvable conflicts* occur in the remaining cases, namely, if an insertion is triggered simultaneously with another operation with the same start index or when neither operation is an insertion and the ranges overlap. Such conflicts would normally require user coordination to reconcile the differences.

In presence of many simultaneous requests, the users determine if they are conflicting by considering each pair of them. In Appendix B provide additional details regarding how multi-user support in the distributed setting can be implemented.

7.5.2. 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 authority (CA), which could be an organization to which the users belong or a service provider who outsourced its data 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 allows the CA to reduce both the size of the update tree that the CA and the storage server maintain and 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.

7.6. Enabling support for revision control

We next show how our scheme can be extended to support revision control. To do so, we need to specify how a user can retrieve (i) a specific version of data blocks and (ii) deleted data blocks (prior to a flush on them, which permanently removes them from the server). Adding these capabilities will give the client the ability to retrieve any existing block or any block that previously existed. Note that when revision control is enabled, there are no changes to the update tree that both the client and server maintain from the basic scheme. The only difference at the server side is that the server maintains old versions of data blocks together with client's MACs on those blocks (i.e., when the client modifies a block, its new version and the corresponding MAC are appended to the storage instead of replacing the previous version of the data block and its MAC). This also implies that when retrieving old blocks the indices of which shifted after consecutive insert or delete operations, the up-to-date block indices should be used during the retrieval to ensure that correct blocks are returned.

Specific version retrieval can be realized by modifying the Retrieve protocol to add version 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 MACs that correspond to version V. The client then verifies the response using the intended version V and other attributes obtained from UTRetrieve. This functionality requires no changes to UTRetrieve, but the server now retrieves the requested version of the blocks instead of their most recent versions using the tree nodes that the function returns. 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.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 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 requested deleted range.

We denote the modified UTRetrieve that retrieves deleted data as UTRetrieveDel$(\mathsf{u}, \mathsf{s}, \mathsf{e})$. Instead of ignoring nodes that represent deletions, it returns them as the output, which allows the server to locate the necessary blocks and their MACs. UTRetrieveDel uses the same interface and has similar functionality to that of UTRetrieve. In particular, the same cases based on the overlap of the range $[\mathsf{s}, \mathsf{e}]$ and $[\mathsf{u.L}, \mathsf{u.U}]$ can occur as in UTRetrieve, and the routine is called recursively to deal with each situation. 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, while UTRetrieve ignores them 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.

We note that the interface works at the level of the blocks. Thus, if a user would like to retrieve the content of the storage at a particular time in the past, deciding what blocks and what versions of the blocks to retrieve needs to be done based on additional information stored externally. In other words, our system allows for retrieval and verification of any block and any version of that block that existed in the storage at some point in the past, but the interface requires the user to specify the block index at the present time (which will be translated to the index that the block had at the time of MAC computation).

In the multi-user distributed setting with distinct keys and permissions, enforcing access control when permissions change may become more complex with revision control capabilities, when, for instance, new users are granted access only to current and future revisions of data. In this case, any suitable key management mechanism can be employed, which we do not further discuss here.

Also, with revision control enabled, the use of the flush command which reduces client's storage size becomes problematic if the revision history and old versions of data blocks are to be retained. Under these circumstances, we propose that the user(s) outsource portions of the update tree in the event that their local storage exceeds the desired threshold for local storage, and the part of the tree being outsourced needs to be authenticated using a traditional mechanism such as a MHT. Outsourcing portions of the update tree to the server increases communication and computation overhead of the solution, but we note that enabling revision control in other schemes is generally expensive and savings are still expected.

We see two possibilities for this strategy that minimize client's costs. The first possibility is to compute hashes of the update tree nodes in the same way as in a MHT, assume that the tree is to be outsourced, and use the local storage as a cache for storing as many nodes of the tree as the space permits (similar to [Stefanov et al. 2012]). In this case, a viable caching strategy is to evict the least recently accessed nodes from the local cache, which will result in nodes higher up in the tree residing in the cache more frequently than nodes at the bottom of the tree. The second possibility is to always store the nodes higher in the tree locally (which on average are accessed more frequently during tree traversal) and outsource only the nodes at the bottom levels of the update tree. In this case, the client stores the root node and as many other top levels of the update tree as the space permits and outsources the remaining levels to the server. Each locally stored leaf node is treated as the root of an outsourced subtree rooted at that node and contains a hash that authenticates its subtree (computed as the root of a MHT would be computed on that subtree). This possibility allows the client to store more nodes locally because there is no need to store hashes associated with the locally stored nodes above the leaf level. As a direction of future work, we plan to analyze in detail the cost of outsourcing portions of the update tree to the server and determine which caching strategy performs better in practice.

7.7. Verification of dynamic operations

In a real-world setting, it may be desirable for the client to verify that a dynamic operation was correctly executed by the server, which may fail to apply the update for non-malicious reasons (such as software error or interrupt). Our scheme can be extended to implement this feature as follows: When the server receives an update request from the client and processes it, the server additionally collects all nodes from the update tree that have been modified during the tree traversal. The server then arranges the nodes in an agreed-upon order according to their position in the tree (e.g., pre-order or in-order), computes the hash of the concatenation of all those nodes' attributes, and returns the hash to the client as a proof of correct modification of the tree. Upon receipt of the hash, the client compares it to its expected value that the client computed in the same manner at the time of forming its update request and updating the tree.

The above assumes that the client and the server balance their respective update trees in the same manner (i.e., agree on the constant difference in the heights of a node's subtrees, exceeding which triggers balancing procedure), which is not required otherwise. In this case, however, if the trees are balanced differently, the nodes' offsets in the trees are not guaranteed to match and the client will be unable to fully verify correctness of the tree at the server's side.

7.8. Turning PDP into POR

As mentioned earlier, [Stefanov et al. 2012] provides a mechanism for dynamic POR (as opposed to PDP), which ensures that all data is recoverable. The approach of [Stefanov et al. 2012] is designed for multi-user centralized setting, and when our scheme is used in the same context, the mechanism of [Stefanov et al. 2012] applies to this work as well. Recall that the solution of [Stefanov et al. 2012] requires that the proxy stores data blocks corresponding to erasure codes of updated blocks locally and periodically offloads them to the cloud. The blocks with erasure codes cannot be sent to the server after an update to hide correspondence between an updated block and the respective erasure code blocks. The solution may also be applicable to a single-user setting if the client can afford to locally store data blocks corresponding to erasure codes and only periodically offload them to the server.

Table II. Asymptotic complexities of DPDP schemes.

Scheme	Cost per operation (Server and Client)						Storage	
	Update or Insert : Delete		Retrieve		Challenge		Server	Client
	Comp.	Comm.	Comp.	Comm.	Comp.	Comm.		
MHT	$O(K)$	$O(K)$	$O(K)$	$O(K)$	$O(K)$	$O(K)$	$O(K)$	$O(1)$
SL	$O(\log K + t)$	$O(\log K + t)$	$O(\log K + t)$	$O(\log K + t)$	$O(c \log K)$	$O(c \log K)$	$O(K)$	$O(1)$
Ours	$O(\log M + t)$	$O(t) : O(1)$	$O(\log M + t)$	$O(t)$	$O(c \log M)$	$O(c)$	$O(K)$	$O(M)$

8. PERFORMANCE EVALUATION

To evaluate performance of our scheme and provide a comparison with prior solutions, we designed a number of experiments that measure the computation, communication, and storage requirements of three different schemes. The schemes that we compare are: (i) our basic update tree (UTree) solution, (ii) solutions based on (unbalanced) Merkle hash tree (MHT) [Wang et al. 2009] as well as the balanced version of MHT (such as [Mo et al. 2012]) and (iii) solutions based on (balanced) skip lists (SL) [Erway et al. 2009; Heitzmann et al. 2008]. The asymptotic complexities of these schemes are given in Table II. The standard MHT is not balanced, while the complexities of schemes based on balanced MHTs are the same as that of the (balanced) skip list schemes in the table (and thus balanced MHT is not listed separately in the table).

The table provides computation complexities per operation as well as storage complexities for both the server (in addition to the data blocks themselves, i.e., space for maintaining metadata) and the client. In the table, K denotes the number of data blocks stored at the server, and M denotes the number of dynamic operations on the stored blocks. For all operations except challenge, it is assumed that the operation is executed on a *consecutive* range consisting of t blocks. Complexity for a single block can be obtained by substituting t with 1. We assume that in a retrieve operation verification of all t blocks is performed in all schemes, while the challenge operation is for the *entire storage* and is probabilistic, during which c randomly chosen data blocks are retrieved and verified.

Although previous schemes were not necessarily designed to be executed on a range of data blocks, we optimize each operation for t consecutive blocks as much as possible and report the complexities of optimized operations. In particular, the worst-case complexity of an operation with the MHT-based scheme is linear in the size of the repository because after arbitrary insertions and deletions the height of the tree is $O(K)$ in the worst case. In the complexities that we report, we assume that the number of inserted blocks t per operation will not exceed $O(K)$ and thus the overall complexity is bounded by $O(K)$. For SL and balanced MHT operations, we can achieve $O(\log K + t)$ for an update operation by updating t nodes and at most two paths to the root. Insert operations are performed by creating a SL or balanced MHT from the t blocks to be inserted and integrating them into the main data structure (this, for instance, will require re-balancing of the MHT). The complexity of this operation for both SL and balanced MHT is $O(\log K + T)$; the same applies to deletion of t consecutive blocks. The complexities of update and insert operations are identical for all schemes and are listed together. The complexities of delete operation is similar to those of update/insert; for that reason, we combine them into a single column with update/insert operations and list both complexities only when they differ.

Note that the Iris file system [Stefanov et al. 2012] is one of the most relevant solutions, while its complexities are not listed in Table II. This is because the complexities of operations in Iris cannot be directly compared to the schemes we list in the table using the same metric. We therefore discuss that approach separately. In particular, Iris does not explicitly support block insert or delete operations, but operates at the level of files. This means that if a block is inserted in a middle of a file, all blocks that follow will need to be updated. The worst case computation and communication complexity of this operation is $O(K)$. Also, since Iris builds a (balanced) filename hierarchy and a separate MHT for the blocks in each file, let us denote the number of files in the system by F. Then the complexities of an update or retrieve operation are $O(\log K + \log F + t)$, the complexities of an insert or delete operation are $O(K + \log F)$, and the complexities of a challenge operation are $O(c(\log K + \log F))$ (recall that c controls the probability with which the client detects data corruption, as defined in section 3). When the client (or a proxy on behalf of clients) maintains cache, the overhead of an operation is reduced because some requests are fulfilled locally instead of issuing queries to the remote storage.

In our experiments, we evaluate the performance of the schemes in three different settings: (i) 1GB of outsourced storage with 4KB data blocks, (ii) 256GB of storage with 4KB blocks, and (iii) 256GB of storage with 64KB blocks. The first 1GB+4KB setting was chosen for consistency with experiments in a number of prior publications and the other two allow us to examine the systems' behavior when one parameter remains fixed while the other changes. 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. We note that this setup and the experiments we conduct put our scheme at disadvantage for the following reasons:

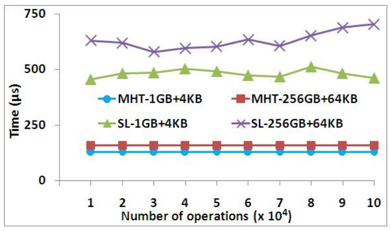
— For compatibility with other schemes, almost all experiments we conduct are on single blocks, while the advantages of our scheme are most pronounced when each operation corresponds to a consecutive range of blocks.
— The size of the outsourced storage will be larger in practice than 1GB or even 256GB in our experiments (i.e., on the order of TBs or even PBs). For instance, in the most recent work [Stefanov et al. 2012], the scheme is setup to support outsourced storage up to 1PB with 4KB blocks. For all schemes except ours this increases the size of the data structure (MHT or SL) and thus communication and computation per operation. For example, with 1PB storage, the overhead is logarithmic in 2^{38} for other schemes with balanced data structure.
— Opposite to other schemes, performance of our scheme may improve when it is used with a large outsourced storage because on average ranges of blocks corresponding to dynamic operations will be less partitioned in the update tree (which is caused by the overlap of ranges for different operations).

We also evaluate the performance of the schemes on real data gathered by Microsoft Research data centers over a course of a week. We believe the access patterns we observed in the traces are representative of a large number of small to medium size enterprise data centers.

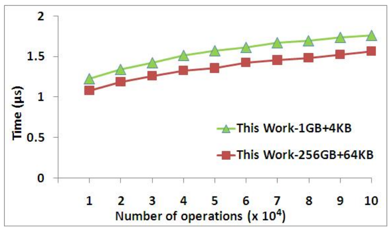
We implement our and MHT solutions in C, while the SL scheme was implemented in Java as in [Erway et al. 2009]. Despite the programming language difference, the time to compute a hash function is similar in both implementations. Then because the overall computation of the SL scheme is dominated by hash 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. The experiments were run on 2.4GHz Linux machines (both the client and the server).

8.1. Computation

To evaluate computation, 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

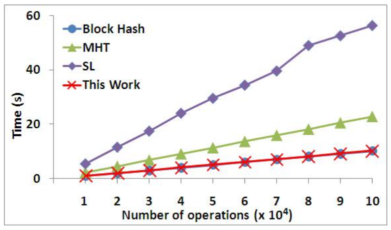


(a) MHT and SL

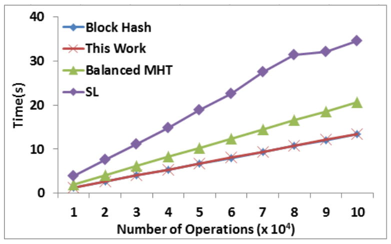


(b) UTree

Fig. 5. Average client's computation time after n single-block randomly chosen operations for 1GB+4KB setting (without block hash and MAC computation).



(a) single-block mixed operations



(b) same position insertions

Fig. 6. Aggregate client's computation time after n operations for 1GB+4KB setting.

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 insert, delete, modify, and retrieve operations at random and execute it on a randomly chosen data block. From the three schemes, only ours provides a natural support for querying ranges of blocks.[4] Then because in practice accesses are often consecutive (see, e.g., [Ellard et al. 2003]), the experiment's results for our scheme give the upper bound of what is expected in practice. For all operations except delete, the client is to compute a hash (or MAC) of the data block used in the request. Because this computation is common to all schemes, we separately measure it and the remaining time (note that computing MAC of a block takes slightly longer than a hash of the block, and we either list the differences explicitly or include the differences together with the remaining portions of our scheme).

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 5 plots average and Figure 6(a) plots the aggregate performance of all schemes for the 1GB+4KB setting, while Table III provides average computation for all three settings. Note that in Figure 6 the common overhead of block hash/MAC computation is included in the performance, while in the table the common and additional computation are shown separately. Moreover, in Figure 6, for clarity we plot the common overhead of block hash only, and include the difference between MAC and hash computation into the performance of our scheme. Lastly, we were unable to complete 256GB+4KB experiments for SL due to its extensive computation (primarily to build the data structure) and memory requirements.

[4]The SL solution in [Erway et al. 2009] can provide a limited support for block ranges as previously described.

Table III. Average client's computation time per operation after n operations in μsec.

Operation type	File size	Block size	Scheme	Total number of operations n					Block hash/MAC
				20,000	40,000	60,000	80,000	100,000	
Single-block mixed random operations	1GB	4KB	UTree	1.34	1.51	1.62	1.70	1.77	138
			MHT	127	127	127	127	127	134
			SL	481	502	473	512	462	134
	256GB	64KB	UTree	1.19	1.33	1.43	1.49	1.57	1976
			MHT	148	147	148	147	148	1972
			SL	619	595	633	652	703	1972
	256GB	4KB	UTree	1.20	1.32	1.43	1.50	1.56	138
			MHT	180	179	179	178	183	134
Single block same position insertions	1GB	4KB	UTree	1.13	1.17	1.21	1.26	1.76	138
			balanced MHT	72.7	72.9	72.8	73.0	72.5	134
			SL	250	235	242	258	212	134

For this experiment, we measured the performance of plain MHT (as opposed to balanced MHT, which bounds the overhead of each operation). This, however, does not place solutions based on balanced MHTs at disadvantage because in this experiment the tree does not become very unbalanced and the performance does not suffer. This is because the number of operations is smaller than the total number of blocks and the operations are spread out across all blocks. Compared to the performance of balanced MHT, the results reported in Table III and Figures 5 and 6(a) are slightly lower, but the difference is within 1%. This is primarily due to avoiding the cost of re-balancing the tree without seeing significant advantages to re-balancing (because the difference in the size of the path from the root to a node is small for the nodes in the tree).

As can be seen from the results, the overhead of UTree scheme is 2 to 3 orders of magnitude lower than in the other schemes. Almost all of the total work in the UTree scheme comes from MAC computation, while in the MHT and SL schemes 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 considered a new access pattern that inserts data blocks at the same position in a file. When balancing is not used, this access pattern makes the underlying data structure very unbalanced. Because the latest approaches use balanced data structures, we show the results of this experiment for balanced MHT, (balanced) SL, and our scheme in Table III and Figure 6(b). The performance in this experiment is rather consistent with the performance in other experiments.

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 operations except retrieve/challenge that return a response, while all operations in MHT and SL require bidirectional communication. For that reason, we measured the aggregate data exchanged for each operation, with-

Table IV. Aggregate communication size after n operations measured in MB.

File size	Block size	Scheme	Total number of operations n				
			20,000	40,000	60,000	80,000	100,000
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	1,920	2,880	3,840	4,800
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

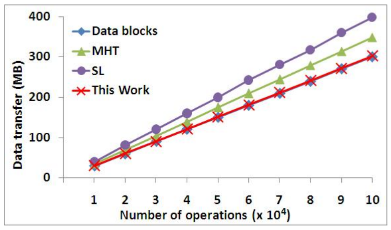


Fig. 7. Aggregate communication for 1GB+4KB.

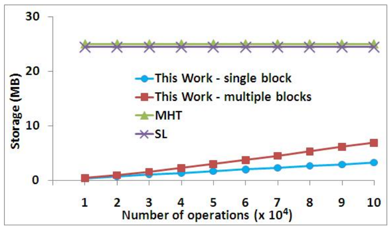


Fig. 8. Server's storage overhead for 1GB+4KB.

out considering the direction of data transfer. The results are given in Table IV and Figure 7, 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 III, UTree's communication is independent of the data structure size or the setting and is constant per operation. For MHT and SL schemes, however, 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 (14–30%), which could be a fairly large burden for users with a limited network bandwidth. The overhead of UTree scheme, on the other hand, is no more than 0.6%. Lastly, the difference in performance of MHT and SL schemes can be explained by the length of the proof and the size of elements within the proof (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. Recall that the data structures we consider do not include the initially uploaded data blocks, which is is common to any scheme. In both MHT and SL schemes, the server maintains a data structure with the number of nodes linear in the storage size, with each node storing several attributes and a hash value, while the client keeps only constant-sized data. In our scheme, both the server and the client maintain a data structure of moderate size (which can also be reduced using flush) with each node storing several attributes (but no hash), while the server additionally maintains a MAC for every block. The data structures can be viewed as consisting of a static portion that corresponds to the initially uploaded data and a dynamic portion that corresponds to dynamic operations issued afterwards. In MHT and SL schemes, the static component is linear in the number of outsourced blocks and is expected to be fairly large. In particular, for MHT, the total number of

Table V. The size of data structures after n single-block or range mixed operations measured in MB.

File size	Block size	MHT for any n	SL for any n	UTree for n operations					MACs for UTree
				20,000	40,000	60,000	80,000	100,000	
1GB	4KB	25	24	0.70	1.37	2.03	2.66	3.28	7
				0.97	2.26	3.74	5.32	6.94	
256GB	64KB	400	391	0.71	1.43	2.15	2.85	3.57	112
				0.71	1.46	2.15	2.89	3.60	
256GB	4KB	6400	6206	0.71	1.43	2.15	2.86	3.57	1792
				0.72	1.44	2.16	2.88	3.61	

nodes in the data structure is twice the number of outsourced blocks while for SL, the number is slightly less than twice the total number of blocks. It is also the reason for not storing the data structures at the client side. In our scheme, however, there is no static component in the update tree, but the server still needs to maintain information in the form of MACs linear in the number of outsourced blocks. The size of the dynamic component 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 reduces 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 [Anagnostopoulos et al. 2001]. The use of persistent data structures increases 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 are given in Table V and Figure 8. In the table, the first row of each UTree setting corresponds to single-block mixed dynamic operations at random locations and the second row corresponds to similar range operations (1–20 blocks per operation). We also list the server's storage for maintaining initially uploaded MACs for each outsourced block in our solution in the last column, while the third and fourth columns list the amount of server's storage necessary to hold data structures in MHT and SL schemes. The values are estimated based on the number of nodes in the data structures (measured using UTree, MHT, and SL implementations) and the approximate node size of 50 bytes for each scheme. They do not correspond to runtime memory measurements. Clearly, there is a large difference in the performance of our scheme and other approaches for the tested settings.

The results correspond to the original solutions, without support for revision control, which means that the number of data blocks remains constant after executing an equal number of different types of updates. As expected, the size of UTree grows linearly with the number of dynamic operations. Another observation that aligns with our experiments above is that the difference in the update tree size after an equal number of range and single-block operations significantly reduces as the number of outsourced blocks increases (compare, e.g., 112% and 1% overhead at 100,000 operations). As before, it is caused by fewer range overlaps, which results in fewer node partitioning and smaller tree size.

8.4. Real life data

While our evaluation so far was comprehensive, it was performed on synthetic randomly generated data. We thus also conduct experiments on real life data sets from [Dushyanth et al. 2008], which consist of file traces gathered from Microsoft data centers for a period of a week. We believe that the access patterns in the traces are repre-

Table VI. Client's aggregate computation time measured in seconds for each volume.

Volume	Max offset	Operations ($\times 10^6$)	MHT	SL	UTree	Block hash	Block MAC
Proj-0	170 GB	4.2	6,400	21,000	8.4	563	579
Proj-1	880 GB	24	29,000	95,000	1,200	3216	3312
Proj-2	880 GB	29	48,000	120,000	2,200	3886	4002
Proj-3	240 GB	2.2	670	2,300	3	295	304
Proj-4	240 GB	6.5	3,400	12,000	150	871	897

sentative of data usage seen in practice. The traces were collected below the file system cache and capture all block-level reads and writes performed on 36 volumes. For our experiments, we select five volumes that belong to a single server (a research project server) and contain 66 million events. Each event contains a timestamp, a disk number, the start logical block number (i.e., offset), the number of blocks transferred, and its type (i.e., read or write).

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

Because the communication overhead of our scheme is always smaller than in MHT and SL schemes, we concentrate here on computation and storage overhead. The runtime with blocks of size 4KB is presented in Table VI. We list the times for MHT, SL, and UTree without initialization time (for building data structures or computing MACs) in columns 4, 5, and 6, respectively. We note that we view initialization as originally storing the client's data together with building and storing verification information for all blocks at the server. The last two columns represent the times for computing a hash (for MHT and SL schemes) or MAC (for UTree scheme) of each data block, respectively, which are part of initialization costs. We note that initialization in our scheme is comprised of only MAC computation, while the initialization costs of MHT and SL schemes in addition to block hash computation include producing hashes of intermediate nodes in the data structure. The cost of data block hashes, however, dominates the initialization cost. As can be seen from the table, our solution (after initialization) is about two orders of magnitude faster than the other two schemes.

For the storage overhead, we measure the data structure size after executing all operations appeared in a file trace. In case of MHT and SL schemes, the data structures depend on the size of outsourced data and range from 4 to 22GB. In our scheme, after executing all requests in a volume, the data structure size ranges from 1 to 150MB at the client side, while the size of the storage at the server ranges from 1.2 to 6.2GB due to the need to additionally store MACs.

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 a balanced update tree. Unique features of our scheme include orders of magnitude faster than in other schemes computation 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.

Acknowledgments

This work was supported in part by grants FA9550-09-1-0223 and FA9550-13-1-0066 from the Air Force Office of Scientific Research and grants CNS-1223699 and CNS-1319090 from the National Science Foundation. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

ADELSON-VELSKII, G. AND LANDIS, E. 1962. An algorithm for the organization of information. In *Proceedings of the USSR Academy of Sciences*. 263–266.

ANAGNOSTOPOULOS, A., GOODRICH, M., AND TAMASSIA, R. 2001. Persistent authenticated dictionaries and their applications. In *International Conference on Information Security (ISC)*. 379–393.

ATENIESE, G., BURNS, R., CURTMOLA, R., HERRING, J., KISSNER, L., PETERSON, Z., AND SONG, D. 2007. Provable data possession at untrusted stores. In *CCS*. 598–609.

ATENIESE, G., DI PIETRO, R., MANCINI, L., AND TSUDIK, G. 2008. Scalable and efficient provable data possession. In *Security and Privacy in Communication Networks (SecureComm)*. 9:1–9:10.

ATENIESE, G., KAMARA, S., AND KATZ, J. 2009. Proofs of storage from homomorphic identification protocols. In *Advances in Cryptology – ASIACRYPT*. 319–333.

BENTLEY, J. 1979. Decomposable searching problems. *Information Processing Letters 8,* 5, 244–251.

BOWERS, K., JUELS, A., AND OPREA, A. 2009a. HAIL: A high-availability and integrity layer for cloud storage. In *ACM Conference on Computer and Communications Security (CCS)*. 187–198.

BOWERS, K., JUELS, A., AND OPREA, A. 2009b. Proofs of retrievability: Theory and Implementation. In *ACM Workshop on Cloud Computing Security (CCSW)*. 43–54.

CHANG, E. AND XU, J. 2008. Remote integrity check with dishonest storage server. In *ESORICS*. 223–237.

CURTMOLA, R., KHAN, O., BURNS, R., AND ATENIESE, G. 2008. MR. PDP: Multiple-replica provable data possession. In *International Conference on Distributed Computing Systems (ICDCS)*. 411–420.

DE BERG, M., VAN KREVELD, M., OVERMARS, M., AND SCHWARZKOPF, O. 2000. Interval trees. In *Computational Geometry* Second Ed. Springer-Verlag, Chapter 10.1, 212–217.

DODIS, Y., DADHAN, S., AND WICHS, D. 2009. Proofs of retrievability via hardness amplification. In *TCC*.

DUSHYANTH, N., AUSTIN, D., AND ANTONY, R. 2008. Write off-loading: Practical power management for enterprise storage. *Transactions on Storage 4,* 3, 10:1–10:23.

ELLARD, D., LEDLIE, J., MALKANI, P., AND SELTZER, M. 2003. Passive NFS tracing of email and research workloads. In *USENIX Conference on File and Storage Technologies (FAST)*.

ELLIS, C. AND GIBBS, S. 1989. Concurrency control in groupware systems. In *SIGMOD*. 399–407.

ERWAY, C., KUPCU, A., PAPAMANTHOU, C., AND TAMASSIA, R. 2009. Dynamic provable data possession. In *ACM Conference on Computer and Communications Security (CCS)*. 213–222.

GOODRICH, M., PAPAMANTHOU, C., TAMASSIA, R., AND TRIANDOPOULOS, N. 2008. Athos: Efficient authentication of outsourced file systems. In *International Conference on Information Security*. 80–96.

GOODRICH, M., TAMASSIA, R., AND SCHWERIN, A. 2001. Implementation of an authenticated dictionary with skip lists and commutative hashing. In *DARPA Information Survivability Conference and Exposition*.

HEITZMANN, A., PALAZZI, B., PAPAMANTHOU, C., AND TAMASSIA, R. 2008. Efficient integrity checking of untrusted network storage. In *StorageSS*. 43–54.

IDC. 2008. IT cloud services user survey, pt. 2: Top benefits & challenges. http://blogs.idc.com/ie/?p=210.

JUELS, A. AND KALISKI, B. 2007. PORs: Proofs of retrievability for large files. In *CCS*. 584–597.

LAMPORT, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 558–565.

LI, J., KROHN, M., MAZIERES, D., AND SHASHA, D. 2004. Secure untrusted data repository (SUNDR). In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 121–136.

LIU, X., ZHANG, Y., WANG, B., AND YAN, J. 2013. Mona: Secure multi-owner data sharing for dynamic groups in the cloud. *IEEE Transactions on Parallel and Distributed Systems 24,* 6, 1182–1191.

MO, Z., ZHOU, Y., AND CHEN, S. 2012. A dynamic proof of retrievability (PoR) scheme with $o(\log n)$ complexity. In *IEEE ICC, Communication and Information Systems Security Symposium*.

OPREA, A. AND REITER, M. 2007. Integrity checking in cryptographic file systems with constant trusted storage. In *USENIX Security Symposium*. 183–198.

PAPAMANTHOU, C. AND TAMASSIA, R. 2007. Time and space efficient algorithms for two-party authenticated data structures. In *International Conference on Information and Communications Security (ICICS)*. 1–15.

POPA, R., LORCH, J., MOLNAR, D., WANG, H., AND ZHUANG, L. 2011. Enabling security in cloud storage SLAs with CloudProof. In *USENIX Annual Technical Conference*. 355–368.

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

SEBE, F., DOMINGO-FERRER, J., MARTINEZ-BELLESTE, A., DESWARTE, Y., AND QUISQUATER, J.-J. 2008. Efficient remote data possession checking in critical information infrastructures. *TKDE 20*, 1034–1038.

SHACHAM, H. AND WATERS, B. 2008. Compact proofs of retrievability. In *ASIACRYPT*. 90–107.

STEFANOV, E., DIJK, M. V., JUELS, A., AND OPREA, A. 2012. Iris: A scalable cloud file system with efficient integrity checks. In *Annual Computer Security Applications Conference (ACSAC)*. 229–238.

WANG, B., LI, B., AND LI, H. 2012a. Knox: Privacy-preserving auditing for shared data with large groups in the cloud. In *International Conference on Applied Cryptography and Network Security (ACNS)*. 507–525.

WANG, B., LI, B., AND LI, H. 2012b. Oruta: Privacy-preserving public auditing for shared data in the cloud. In *IEEE CLOUD*. 295–302.

WANG, B., LI, B., AND LI, H. 2013. Public auditing for shared data with efficient user revocation in the cloud. In *IEEE International Conference on Computer Communications (INFOCOM)*. 2904–2912.

WANG, C., CHOW, S., WANG, Q., REN, K., AND LOU, W. 2013. Privacy-preserving public auditing for secure cloud storage. *IEEE Transactions on Computers 62,* 2, 362–375.

WANG, C., WANG, Q., REN, K., AND LOU, W. 2009. Ensuring data storage security in cloud computing. In *International Workshop on Quality of Service*. 1–9.

WANG, Q., WANG, C., LI, J., REN, K., AND LOU, W. 2009. Enabling public verifiability and data dynamics for storage security in cloud computing. In *ESORICS*. 355–370.

WEI, L., ZHU, H., CAO, Z., JIA, W., AND VASILAKOS, A. 2010. SecCloud: Bringing secure storage and computation in cloud. In *ICDCS Workshops*. 52–61.

ZENG, K. 2008. Publicly verifiable remote data integrity. In *ICICS*. 419–434.

ZHENG, Q. AND XU, S. 2011. Fair and dynamic proofs of retrievability. In *CODASPY*. 237–248.

A. ADDITIONAL ALGORITHMS

Here we give a complete specification of two algorithms:

— $\mathsf{UTDelete}(\mathsf{u}, \mathsf{s}, \mathsf{e})$, when called with $\mathsf{u} = \mathsf{T}$, updates the update tree T based on a deletion request with the block range $[\mathsf{s}, \mathsf{e}]$ and returns the set of updated nodes. This function is similar to $\mathsf{UTModify}$ and is given in Algorithm 8.

— $\mathsf{UTRetrieveDel}(\mathsf{u}, \mathsf{s}, \mathsf{e})$, 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.

B. MULTI-USER SUPPORT IN DISTRIBUTED SETTING

In what follows, we detail a mechanism for a possible realization of the multi-user support in the distributed setting. Recall that there are multiple update trees created according to the permission groups. Any two simultaneous updates can be conflicting, in which case we divide them into automatically-resolvable and manually-resolvable conflicts, and any update without conflicts is non-conflicting.

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 corresponding 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. After the end of each time slot the user determines if any conflicts

ALGORITHM 8: UTDelete(u, s, e)

```
1: $\mathsf{C} = \emptyset$
2: if ($\mathsf{u} = \mathrm{NULL}$) then
3:     $\mathsf{w} = \mathsf{UTCreateNode}(\mathsf{s}, \mathsf{e}, 0, -1)$
4:     $\mathsf{C} = \mathsf{C} \cup \{\mathsf{w}\}$
5: else
6:     $\mathsf{S} = \mathsf{UTFindNode}(\mathsf{u}, \mathsf{s}, \mathsf{e}, -1)$
7:     for $i = 0$ to $|\mathsf{S}| - 1$ do
8:         $(\mathsf{u}_i, \mathsf{s}_i, \mathsf{e}_i) = \mathsf{S}[i]$
9:         if ($\mathsf{s}_i \neq \mathrm{NULL}$) then
10:            if ($\mathsf{u}_i.\mathsf{L} \leq \mathsf{s}_i$ and $\mathsf{e}_i \leq \mathsf{u}_i.\mathsf{U}$) then
11:                $\mathsf{C} = \mathsf{C} \cup \{\mathsf{UTUpdateNode}(\mathsf{u}_i, \mathsf{s}_i, \mathsf{e}_i, -1)\}$
12:            else if ($\mathsf{s}_i < \mathsf{u}_i.\mathsf{L}$ and $\mathsf{e}_i \leq \mathsf{u}_i.\mathsf{U}$) then
13:                $\mathsf{C} = \mathsf{C} \cup \{\mathsf{UTUpdateNode}(\mathsf{u}_i, \mathsf{u}_i.\mathsf{L}, \mathsf{e}_i, -1)\}$
14:                $\mathsf{C} = \mathsf{C} \cup \{\mathsf{UTDelete}(\mathsf{u}_i, \mathsf{s}_i + \mathsf{u}_i.\mathsf{R}, \mathsf{u}_i.\mathsf{L} - 1 + \mathsf{u}_i.\mathsf{R})\}$
15:            else if ($\mathsf{u}_i.\mathsf{L} \leq \mathsf{s}_i$ and $\mathsf{u}_i.\mathsf{U} < \mathsf{e}_i$) then
16:                $\mathsf{C} = \mathsf{C} \cup \{\mathsf{UTUpdateNode}(\mathsf{u}_i, \mathsf{s}_i, \mathsf{u}_i.\mathsf{U}, -1)\}$
17:                $\mathsf{C} = \mathsf{C} \cup \{\mathsf{UTDelete}(\mathsf{u}_i, \mathsf{u}_i.\mathsf{U} + 1 + \mathsf{u}_i.\mathsf{R}, \mathsf{e}_i + \mathsf{u}_i.\mathsf{R})\}$
18:            else if ($\mathsf{s}_i < \mathsf{u}_i.\mathsf{L}$ and $\mathsf{e}_i > \mathsf{u}_i.\mathsf{U}$) then
19:                $\mathsf{C} = \mathsf{C} \cup \{\mathsf{UTUpdateNode}(\mathsf{u}_i, \mathsf{u}_i.\mathsf{L}, \mathsf{u}_i.\mathsf{U}, -1)\}$
20:                $\mathsf{C} = \mathsf{C} \cup \{\mathsf{UTDelete}(\mathsf{u}_i, \mathsf{s}_i + \mathsf{u}_i.\mathsf{R}, \mathsf{u}_i.\mathsf{L} - 1 + \mathsf{u}_i.\mathsf{R})\}$
21:                $\mathsf{C} = \mathsf{C} \cup \{\mathsf{UTDelete}(\mathsf{u}_i, \mathsf{u}_i.\mathsf{U} + 1 + \mathsf{u}_i.\mathsf{R}, \mathsf{e}_i + \mathsf{u}_i.\mathsf{R})\}$
22:            end if
23:        else
24:            $\mathsf{C} = \mathsf{C} \cup \{\mathsf{u}_i\}$
25:        end if
26:    end for
27: end if
28: return C
```

arise by checking each pair of announced operations for conflicts. If any conflicts are determined, conflict resolution takes place. 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:

— *Automatically resolvable conflicts:* To resolve this type of conflict, we propose to apply operational transformation (OT) [Ellis and Gibbs 1989] 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 (we assume that a 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 we can use OT as a black box.

— *Manually resolvable conflicts:* In general, it is not possible to resolve such conflicts without user coordination and the users will need to collaborate to resolve the conflict. For some applications, however, it may be feasible to resolve certain types of conflict from this category automatically (e.g., insert and delete operations with the same start index may be automatically resolvable).

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,

ALGORITHM 9: $\mathsf{UTRetrieveDel}(u, s, e)$

```
C = ∅
if (u ≠ NULL) then
    if (u.L ≤ s − u.R and e − u.R ≤ u.U) then
        if (u.Op = −1) then
            C = C ∪ {u}
            C = C ∪ {UTRetrieveDel(u.P_r, s + u.U − u.L + 1, e + u.U − u.L + 1)}
        end if
    else if (e − u.R < u.L) then
        C = C ∪ {UTRetrieveDel(u.P_l, s, e)}
    else if (s − u.R > u.U) then
        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))}
    else if (s − u.R < u.L and u.L ≤ e − u.R ≤ u.U) then
        C = C ∪ {UTRetrieveDel(u.P_l, s, u.L + u.R − 1)}
        C = C ∪ {UTRetrieveDel(u, u.L + u.R, e)}
    else if (u.L ≤ s − u.R ≤ u.U and e − u.R > u.U) then
        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))}
        C = C ∪ {UTRetrieveDel(u, s, u.U + u.R)}
    else if (s − u.R < u.L and e − u.R > u.U) then
        C = C ∪ {UTRetrieveDel(u.P_l, s, u.L + u.R − 1)}
        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))}
        C = C ∪ {UTRetrieveDel(u, u.L + u.R, u.U + u.R)}
    end if
end if
return C
```

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. If no conflicting requests are found, 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.

Permission changes in this setting are jointly handled by the users according to the fact that shared keys are used. To grant a new user access to a permission group, all that needs to be done is to communicate to the user the necessary key material and update tree. In the event that a user should be granted access to partial resources of what a permission group currently covers, the users change the key material for the storage that the new user was granted and perform flush on that portion of the storage. This operation updates all affected MACs and empties the update tree. When a user leaves a permission group, if the users are trusted not to abuse the system in this collaborative setting, no changes are necessary. Otherwise, the key material for the affected data blocks is replaced with freshly chosen keys and flush is applied to the affected storage.

When public verifiability is desirable and symmetric keys are replaced with public-private key pairs, users sign updated blocks using their respective private keys. Now when a user's permissions are reduced, the blocks to which the user no longer has access are to be re-signed with keys corresponding to the remaining users with access to those blocks. In that case, to avoid the complexity of downloading the corresponding portion of shared data and re-signing it, we can apply the same mechanism of proxy re-signatures as in [Wang et al. 2013] that provides a server with a re-sign key generated through an interaction between the revoked and the remaining users. The server is now able to re-sign all invalidated blocks on behalf of the authorized users using the re-sign key. Another interesting feature in this setting with public-private keys is to allow each user in a permission group to anonymously share data with others without revealing their identities (and, in particular, without revealing their identities to the storage server). We can achieve this using a group signature scheme that allows any member of the group to sign messages while keeping the identity secret from verifiers similar to, e.g., [Wang et al. 2012b; Liu et al. 2013].

A potential requirement for a distributed setting is to tolerate slow or disconnected networks when a user regularly gets offline or experiences poor connectivity and hence is unable to receive or process updates originated from other users. Recall that in our solution consistency of the user's view (and thus correctness of scheme's operation) depends on the user's ability to receive information about all updates performed on the outsourced storage so far. In order to enable each user to retrieve all missed updates after the user gets back online, without requiring any given user to be constantly connected, we propose that the users use an external publishing medium that securely logs all messages sent by the users and the times when the messages were received. Assuming that the medium guarantees high availability and global accessibility, and is either trusted by all users or authenticity and completeness of the logs can be verified, a user that recovers from a disconnect can retrieve from the medium all updates to the shared metadata occurred during user's absence and locally apply them to yield a consistent state with those of other peers. We leave the problem of dealing with manually resolvable conflicts in presence of slow or disconnected clients who lose connection after submitting a conflicting update as a direction of future work.

When the users are not reliable and are susceptible to a compromise, the model can be strengthened to allow the remaining users to recover from a compromise of an individual user (at the cost of more expensive operations). This would require a public-key setting, where each user maintains its own private signing key and signs all updates with its id. Then when a compromise of a user is detected, the remaining users will identify and if necessary remove changes to the shared data originated from that user following the compromise. This will be done with the help of the publishing medium that records all updates and allows the remaining user to search for updates originated from a specific user id. Note that in most cases, dealing with user compromise and recovery of shared data will require manual intervention of the users. We leave the detailed implementation of this idea as a direction of future work.