A Generic Dynamic Provable Data Possession Framework

Mohammad Etemad Alptekin Küpçü
{metemad, akupcu}@ku.edu.tr
Koç University, İstanbul, Turkey

Abstract

Ateniese et al. introduced the Provable Data Possession (PDP) model in 2007. Following that, Erway et al. adapted the model for dynamically updatable data, and called it the Dynamic Provable Data Possession (DPDP) model. The idea is that a client outsources her files to a server, and later on challenges the server to obtain a proof that her data is kept intact. During recent years, many schemes have been proposed for this purpose, all following a similar framework.

We analyze in detail the exact requirements of dynamic data outsourcing schemes regarding security and efficiency, and propose a general framework for constructing such schemes that encompasses existing DPDP-like schemes as different instantiations. We show that a dynamic data outsourcing scheme can be constructed given black-box access to an implicitly-ordered authenticated data structure (that we define). Moreover, for blockless verification efficiency, a homomorphic verifiable tag scheme is also needed. We investigate the requirements and conditions these building blocks should satisfy, using which one can easily check applicability of a given building block for dynamic data outsourcing. Finally, we provide a comparison among different building blocks.

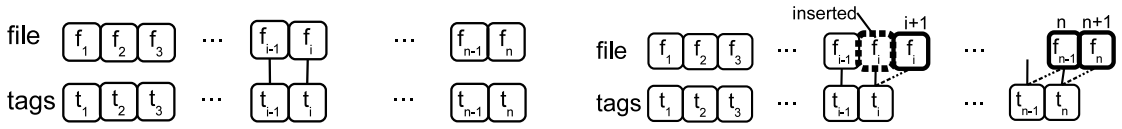
1 Introduction

Data outsourcing is a useful application that is getting more acceptance as the communication and networking technology advances. It brings many advantages such as huge amount of cheap storage, world-wide access to data, and reduced management overhead, while imposing security objections such as *integrity* and *confidentiality*. Lack of integrity protection and confidentiality are the most notable barriers toward full integration of data outsourcing at the cloud, since the data owner loses the direct control over her data.

PDP. Ateniese *et al.* [3] proposed the first static data outsourcing scheme called *provable data possession* (PDP) that provides probabilistic integrity guarantees. During the pre-computation phase, the client divides a file into a number of equal-size *blocks*, computes a cryptographic *tag* for each block, and sends the file together with the vector of tags to an untrusted server. Later, she repeatedly audits the outsourced data to check if the server stores her data intact. A successful audit ensures the client that her data is stored correctly at the server, with high probability. Similar schemes were proposed using different techniques [14] or supplying extra properties [15, 28, 44].

Generalization. Ateniese *et al.* [5] proposed a general framework for building public-key homomorphic linear authenticators (HLA) from any identification protocol with homomorphic properties. They used these HLAs to propose a general framework for constructing a publicly-verifiable proof of storage (PoS) scheme. The resultant PoS scheme supports only the *static* data with communication complexity independent of the data size, and unlimited number of audits. Note that the Ateniese *et al.* [3] PDP scheme is an instantiation of the Ateniese *et al.* [5] generalization.

Dynamic PDP. The main problem with the general framework of Ateniese *et al.* [5] is that it does not support dynamic operations on the outsourced data. Starting with Erway et al. DPDP scheme [18], many dynamic data outsourcing schemes have been proposed [45, 47, 20]. Almost all these schemes use a similar architecture: homomorphic verifiable tags combined with authenticated data structures (ADSs). The differences lie in the ADSs they use, or the cryptographic tags they benefit from. We study the requirements of DPDP-like schemes and give a general framework for constructing such schemes. Our framework enables plug-and-play use of the primitives.



(a) A file and its PDP tags.

(b) A new block is inserted at the i^{th} position.

Figure 1: Inserting a new block at the i^{th} position in a PDP. (Indices and tags are not updated yet in (b).)

1.1 Brief Context

Provable Data Possession is a memory-checking scheme that provides probabilistic guarantees of possession of the outsourced file using a challenge-response mechanism. The client divides a file f into n blocks, $f = (f_1||f_2||...||f_n)$, computes a *tag* t_i for each block f_i, as shown in Figure 1a, and finally transfers the file and the tags to the server, deleting her local copy.

A PDP tag for the i^{th} block, f_i, is computed as $t_i = (h(W_i).g^{f_i})^d \mod N$, where $N = pq$ is an RSA modulus with $p = 2p' + 1$ and $q = 2q' + 1$ as two safe primes, g is a generator of quadratic residues QR_N, $W_i = v||i$ where v is a unique file identifier, $pk = (N, g)$ is the public key and $sk = (e, d, v)$ such that $(e, d) \equiv 1 \ mod \ p'q'$ is the secret key [3]. W_i ties each tag to the current PDP instantiation (identified by v) and the block's position i. If a new block is inserted at (or deleted from) position i, each block $j > i$ needs to be updated with its new index as $j + 1$ ($j - 1$ for deletion) and their tags need to be re-computed accordingly. This is illustrated in Figure 1b, where a new block is inserted at position i (of Figure 1a) and the block indices and tags are not updated yet. The tags $t_i, ..., t_n$ must be re-computed, costing $O(n)$ public-key operations at the worst case.

To make sure the server keeps storing the file intact, the client regularly sends a *challenge* to the server. The challenge contains a subset of block indices selected randomly, and an associated random number per challenged block. Upon receipt, the server constructs a *proof* using the challenged blocks and their tags, and sends it back to the client for verification. Since the PDP tags are *homomorphic* (discussed later), instead of sending all challenged blocks and their tags, the server combines the challenged blocks into a single block and the tags into a single tag, reducing the proof size, while the client is still able to verify them as a whole. Essentially, this relies on polynomially-many challenges creating a system of linear equations that enables extracting the individual tags and blocks [3].

Authenticated Data Structures (ADS) contain the security information generated by a trusted client on a dataset and used by untrusted responders to provide cryptographic proofs showing the answers to the client's queries are valid [40, 41, 33, 26]. The ADS is constructed using, and depends on, all client data. On each update to the client data, the corresponding part(s) of the ADS should also be updated accordingly. Most ADSs support both membership and non-membership queries, and the proofs can be verified using only a small digest information.

There are different types of ADSs: accumulators [8], authenticated skip lists [25], authenticated hash tables [32], Merkle hash trees [30], 2-3 trees [31], and hierarchical combinations [21, 22]. The accumulator computes an accumulated value over all data items, and a witness for each item as the membership proof. Search operations are performed in constant time, but an update requires the accumulated value and all witnesses be recomputed that is a linear operation. In authenticated skip lists, Merkle hash trees, and 2-3 trees, both search and update are logarithmic (in the number of data items) operations. The authenticated hash table makes one operation constant-time while keeping the other one sublinear. The costs of a hierarchical ADS depends on the costs of the ADSs it is built on. We refer the reader to the related papers for details, but in Section 3 we discuss some of these ADSs and their security definitions.

1.2 Observations

We investigated the data outsourcing schemes, their security, advantages, and weaknesses. The results are given below as a set of observations to be used as a guideline for constructing a DPDP-like scheme either form scratch or using the existing building blocks.

- **Observation 1**. If a client (data owner) outsources *only* her data to an untrusted cloud server, the server can manipulate the data, while the client has no way of detecting such a misbehaviour. Therefore, before outsourcing her data, the client should accompany it with cryptographic techniques enabling later verifications. The client can divide the data into (equal-size) blocks, compute a tag for each one, and store them at the server, as most of the existing schemes [3, 15, 10]. Then, she can perform spot-checking that ensures the data integrity with high probability.

- **Observation 2**. There are two problems with this solution when it comes to dynamically updating the data blocks. The first problem is the possibility of *replay* attacks: The server can ignore an update and use the old authentic data and tags to pass the client verification successfully. Since the old tags were created correctly, they still pass the verification. The second problem is that the tags are bound to the exact locations of the respective blocks to prevent re-ordering. An update in an intermediate location of the outsourced data shifts all remaining blocks, requiring all the corresponding tags be re-computed.

- **Observation 3**. For the first problem, an ADS is employed to store the tags. The ADS protects the integrity of tags, which in turn, protect the integrity of data blocks. The most important thing here is that with each update, the client must also update some local information, such that the old tag is no longer valid. Remember that an ADS has a small digest. By storing it locally and updating it with every update, the client keeps a small local state about the latest version of data, and hence can ensure the proofs match this last version.

- **Observation 4**. For the second problem, one still needs to tie the tags/blocks to their locations, but needs to do it implicitly rather than explicitly. An implicitly-ordered ADS (such as rank-based authenticated skip list [18]) solves this problem. Since the tags are not explicitly bound to the exact locations of the corresponding blocks, the (blocks and the) tags can be updated efficiently. The locations can be also computed using the (rank) information in the ADS.

Since storing information about all file blocks or all updates at the client side is not efficient, any solution in the dynamic setting should have sublinear (preferably constant) metadata. The client stores digest of the ADS locally, and outsources the ADS along with the file.

However, if the server sends all challenged file blocks with their membership proofs in the ADS to the client, this is not communication-efficient. Using a homomorphic verifiable tag scheme, the client computes a tag for each block, and stores these tags (not the blocks themselves) in the ADS. To answer an audit query, the server computes an aggregation of the challenged blocks (that is about a block in length), the membership proofs of their respective tags (that are small in length compared to the blocks), and sends them to the client for verification.

A tag scheme used for dynamic data outsourcing should provide the following properties:

- **Homomorphicity**. Two tags T_{m_i} and T_{m_j} of two blocks m_i and m_j can be combined into a value $T_{m_i} \oplus T_{m_j}$ as the tag of $m_i \odot m_j$, where $\oplus$ and $\odot$ denote known operations. This enables the server send an aggregation of file blocks and an aggregation of their tags instead of sending them all to the client, for communication efficiency.

- **Unforgeability**. It should be computationally infeasible for the server to compute a tag for a block of his choice or find another block with the same tag.

The ADS, in addition to being secure, should satisfy the following conditions:

- **Efficiency**. The ADS should support efficient search and update. We investigate different kinds of ADSs and compare their operation costs. Our results reveal that the tree-based ADSs are efficient, supporting search and update with $O(\log n)$ cost, where n is the number of data blocks.

- **Ease of rebalance**. To have the logarithmic costs, the ADS should remain balanced after any number of updates. The binary search tree based solutions (e.g., Merkle tree) used in many schemes do not satisfy this condition (see Section 3.1 for details), and are not good candidates for dynamic data outsourcing. On the other hand, authenticated versions of skip list, AVL tree, red-black tree, and 2-3 tree can be efficiently rebalanced.

- **Implicit order**. The ADS must arrange the data blocks or their respective tags according to their location in the outsourced data in a way that updating one tag does not affect (all) the remaining tags. One way is to divide the set of file blocks into subsets, each storing the number of blocks it contains (known as the *rank*). It is obvious that an update in a subset will not affect the other subsets with whom there is nothing in common.

We show the general framework for providing data authenticity in dynamic data outsourcing contexts that consists of a homomorphic verifiable tag scheme and an implicitly-ordered ADS to keep the tags.

1.3 Our contributions

Under the observations above, our contributions are:

- We propose a generic framework for construction of efficient DPDP schemes, describing the required components and their properties. We show that efficient DPDP schemes can be constructed given **black-box access** to an *implicitly-ordered ADS* and a *homomorphic verifiable tag* scheme. The existing schemes [18, 45, 47, 20] are all specific cases of our general model.
- We investigate the requirements of ADSs to be used in DPDP schemes: the ADS should be *implicitly-ordered*. We show that a relative indexing mechanism like *rank* satisfies the requirements. We also show how to convert an explicitly-ordered ADS into an implicitly-ordered ADS and apply this on several popular ADSs.
- The tag scheme used should be homomorphic and verifiable. Homomorphism improves *performance* by auditing without downloading all challenged blocks, and verifiability provides *security*.
- We investigate two kinds of DPDP schemes: *basic* and *blockless*. The basic scheme does not use a tag scheme, but is secure. The blockless scheme employs homomorphic verifiable tags, and is more communication-efficient.

Section 2 presents our model and the related work. Section 3 defines authenticated data structures and how to make them implicitly-ordered. In Section 4, we show our basic construction using an implicitly-ordered ADS and proves its security. Section 5 defines the necessary tag schemes and incorporates them to our basic construction to achieve our blockless DPDP framework, and proves its security. Finally, in Section 6 we discuss some extensions and compare the known solutions within our framework.

2 Background

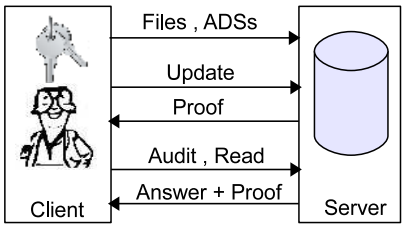


Figure 2: Our model.

Model. There are two parties in our DPDP model. The *data owner* (also known as the *client*) performs the required pre-computations, e.g., she computes the tags and builds the ADS, and transfers them along with her files to the *server*. Later, she asks the server to perform audit and update (deletion, modification, or insertion) operations on the outsourced data, giving him the necessary information about the operation. The server performs the operation and prepares a proof that the operation is performed properly, and sends it back to the client. We expect that the client executes the audit occasionally to make sure the server keeps storing her data intact. This is depicted in Figure 2.

Adversarial model. The server is untrusted, i.e., he can act maliciously or made by others to act so. He may cheat by attacking the integrity of the outsourced data through adding/deleting files or updating the file contents, while trying to be undetected, or hiding data loss due to viruses or hardware crashes. In this paper, we consider confidentiality as an orthogonal issue, where it can be satisfied by encrypting each data block individually. Thus, we only focus on integrity.

Notation. We use $x \leftarrow X$ to denote that x is sampled uniformly from the set X, $|X|$ to represent the number of elements of X, and $||$ to show concatenation. PPT denotes probabilistic polynomial time, and λ is the security parameter. We represent a file f with n blocks as $f = (f_1||f_2||...||f_n)$, and use f_i to refer to the i^{th} block.

A function $\nu : Z^+ \rightarrow [0, 1]$ is called *negligible* if $\forall$ *positive polynomials* p, $\exists$ *constant* c *such that* $\forall\ k > c$, $\nu(k) < 1/p(k)$. *Overwhelming* probability is greater than or equal to $1 - \nu(k)$ for some negligible function $\nu(k)$. By *efficient algorithms*, we mean those with expected running time polynomial in λ.

2.1 Related Work

Static PDP. The PDP scheme [3] provides probabilistic guarantees of possession of the outsourced file using a challenge-response mechanism. It uses homomorphic verifiable tags, using which the server aggregates the challenged blocks and tags, and sends only *one aggregated block* and *one aggregated tag*, reducing the communication.

Wang *et al.* [43] used BLS signatures [9] in a similar manner to provide *privacy preserving public auditing* for static data. The scheme is publicly verifiable to allow a *third party auditor* (TPA) to perform the auditing on behalf of the client, while preventing the TPA from accessing the client data. The idea is to blind the (PDP-like) proofs coming from the server. Similarly, Wang [44] proposed the concept of *proxy* provable data possession (PPDP). The proxy is an entity who is involved on behalf of the client in a publicly verifiable PDP protocol with the server and checks the integrity of the client's outsourced data regularly. Shah *et al.* [36, 37] encrypts the data before outsourcing, and uses privacy-preserving third-party auditing.

Zhu *et al.* [48] proposed a cooperative PDP (CPDP) scheme using homomorphic verifiable response and hash index hierarchy for a cooperation environment involving multiple cloud service providers that store and maintain the client's data. This is different from the multi-copy schemes in the sense that the client is only communicating with one entity that is known as the *organizer*. Th organizer is responsible for communication between the client and the servers.

Dynamic PDP. The above schemes do not support dynamic operations on the outsourced data, making them suitable only for archival purposes.

Ateniese *et al.* [4] proposed SPDP as a semi-dynamic PDP scheme. They pre-compute and store at the server a number of random challenges with the corresponding answers, i.e., the number of challenges a client can perform is limited and fixed a priori. Moreover, an update invalidates all remaining (unused) challenge answers, and the client should compute and send to the server a new set of challenge answers to be used by the challenges following the update.

Erway *et al.* [18] introduced the rank-based authenticated skip list to give the first fully dynamic PDP scheme. The rank-based authenticated skip list supports block updates with $O(\log n)$ cost, where n is the number of file blocks. In their first scheme, they used data blocks in the rank-based authenticated skip list, and hence, all challenged blocks are sent to the client. In their second scheme using the RSA-based tags, all challenged blocks are aggregated into one and sent to the client as part of the proof. Esiner *et al.* [20] extended this work using FlexList [19] that uses the size of data in bytes (not the number of blocks) accessible from each node. This allows the data blocks to be variable in size as well as the variable-size updates.

Wang *et al.* [45] used the Merkle hash tree [30] with BLS signatures, which are publicly verifiable homomorphic authenticators, to propose a publicly verifiable DPDP scheme. However, there is an important issue: In cases the insertion (or deletion) operations are not distributed uniformly, and most of them are concentrated on a specific part of the file, the tree will be imbalanced after applying these operations, losing its logarithmic benefits (see Section 3.1 for details).

Barsoum and Hasan [6] proposed a DPDP scheme performing full block-level dynamic operations on the outsourced data. It uses a *block status table* to keep track of the updates on the outsourced data.

Etemad and Küpçü [21] extended DPDP to provide reliability and scalability using multiple servers without the need to modify anything at the client side. It uses a hierarchical rank-based authenticated skip list that supports distribution and replication of any part of the outsourced data, arbitrarily.

Generic PDP. Ateniese *et al.* [5] proposed a framework to use any identification protocol satisfying certain homomorphic properties for constructing public-key homomorphic linear authenticators (HLAs). Then, they showed how a publicly-verifiable proof of storage scheme satisfying the following properties can be built using any public-key HLA: 1) The data is static, 2) the communication complexity is independent of the file size, and 3) it supports an unbounded number of verifications. Until our work, no work has extended this generic framework to the dynamic setting.

Static Proofs of Retrievability (PoR). As these are closely related to PDP schemes, we will briefly mention PoR type schemes as well. First proposed by Juels and Kaliski [29], PoR is an integrity checking scheme for *static* outsourced data. It provides strong retrievability guarantees using erasure-correcting codes, i.e., in case of any unauthorized data manipulation, the erasure-correcting code will help recover the original data. Juels and Kaliski's PoR supports only a limited number of challenges. Compact PoR [35] combines erasure-correcting codes together with PDP-like techniques. It supports public verifiability and removes the upper bound on the number of audits. Dodis *et al.* [16] identified different variants of PoR and gave optimal PoR schemes for each one, with improved static PoR schemes.

Dynamic PoR. Stefanov *et al.* [39] proposed Iris as a dynamic PoR scheme inside a cloud file system. Iris is a semi-dynamic PoR scheme since the erasure-coding data are stored locally (on a trusted party called *portal*). The first dynamic PoR scheme with full security definition and proof is proposed by Cash *et al.* [11, 12]. Recently, Chandran *et al.* [13] and Shi *et al.* [38] proposed other dynamic PoR schemes.

Generic PoR. Shacham and Waters [35] showed the relationship among static PDP and PoR schemes: A static PoR scheme can be constructed employing a PDP scheme together with erasure-correcting codes. Etemad and Küpçü [23] proposed a general framework for constructing dynamic PoR schemes, using black-box access to a static PoR scheme and a dynamic PDP scheme. They also discussed different architectures of the outsourced memory and their effect on the scheme efficiency. The focus of this work is on DPDP, and the PoR discussion is included for completeness only. But, our DPDP generalization will also benefit the DPoR generalization of Etemad and Küpçü [23] as a building block.

3 Ordered Authenticated Data Structures

Authenticated data structures are employed by an untrusted server answering clients' queries on behalf of a trusted data owner for proving authenticity of the answer [40, 41, 33, 26]. Since the data is no more in direct control of the owner, these ADSs ensure the client about originality of the answer. An ADS scheme is defined as follows [33, 22]:

Definition 3.1 (ADS scheme) *An **ADS scheme** consists of the following polynomial-time algorithms:*

- $(sk, pk) \leftarrow \texttt{KeyGen}(1^\lambda)$ *is a probabilistic algorithm run by the client. Given the security parameter λ as input, it generate a secret and public key pair (sk, pk). She sends the public key to the server.*
- $(ans, \pi) \leftarrow \texttt{Certify}(pk, cmd)$ *is run by the server upon receipt of a command cmd from the client, taking the public key pk as input. If cmd is a query command, the client uses the proof π to verify authenticity of the answer ans. If cmd is an update command, the ans is null, and π conveys information for the client to update her local metadata.*
- $(\{\texttt{accept}, \texttt{reject}\}, st') \leftarrow \texttt{Verify}(sk, cmd, ans, \pi, st)$ *is run by the client to verify the server responses. The secret key sk, the answer ans, the proof π, and the current local metadata st are given as input. It generates an* `accept` *or a* `reject` *signal based on the verification result. The client updates her local metadata (to st') according to update commands whose proofs are accepted.*

3.1 The Update Problem

At a high level, the ADS computes and keeps a function (e.g., hash) of each data item to be protected. It can be stored locally, or outsourced along with the data. During verification, the client checks the received data items against their respective values in the ADS, and decides on authenticity of the answer.

Assume the dataset to be outsourced is $D = (d_1, d_2, ..., d_n)$, and a function $g()$ is used to compute a value $v_j = g(d_j, j)$[1] for each $d_j \in D$. The index j of each data item d_j is given as input to $g()$ to bind the computed value v_j to the location of d_j. This is because the order of data items is important; e.g., changing the order of blocks of a text file generates a different file.

When a new data item d_i is being inserted, the set D is modified to $D = (d_1, d_2, ..., d_{i-1}, d_i, d_{i+1}, ..., d_{n+1})$. Each item located already at position j with $i \leq j \leq n$ is now at position $j + 1$, and its v_j should now be updated as $v_{j+1} = g(d_{j+1}, j + 1)$. A new value $v_i = g(d_i, i)$ is computed and stored for the recently-added d_i. Similar steps are taken when an item d_i is deleted, making all values v_{i+1} to v_n be re-computed.

Therefore, the worst case cost of an update operation is linear in the number of data items. This is because each data item d_j is bound to its dedicated position j that is changed with any insertion/deletion in the interval $[1, j]$. We call these ADSs the *explicitly-ordered ADSs*. Several examples of different explicitly-ordered ADSs used for data outsourcing are provided in Section 3.3. In contrast, the ADSs that do not explicitly bind each item to its dedicated position are called *implicitly-ordered ADSs*. Section 3.2 formalizes this concept and Section 3.4 gives examples of different implicitly-ordered ADSs.

The property an implicitly-ordered ADS uses to arrange the data items does not directly depend on their positions, but can be used to locate each item. The actual requirement is that whenever an item is added or removed, no other items (or a very limited number of them) are affected. This is possible by recursively dividing a dataset into subsets of its successive items and computing a property on each subset. This informally means that each subset keeps the order of its elements without regarding other subsets, ensuring that a change in one subset would not affect the other subsets.

[1]The function may take other inputs, but these two inputs are sufficient for describing the problem simply.

Table 1: ADS comparison based on operation complexities.

Complexity class	Example	Search	Update
Linear	(Ordered) Accumulator [8]	$O(1)$	$O(n)$
Sublinear	Authenticated hash table [32]	$O(1)$	$O(n^\epsilon)$
Logarithmic	Rank-based authenticated skip list [18]	$O(\log n)$	$O(\log n)$

The property can be the *rank* (the number of items or the size of data) of each subset. However, not all similar properties can be used for constructing implicitly-ordered ADSs. For example, the maximum (minimum) of the item positions or values in the subset cannot be used for building such ADSs. Since the maximum (minimum) of the *item positions* depends directly on the item positions, an insertion or deletion will again affect all remaining nodes. The maximum (minimum) of the *item values* arranges the items in the ADS differently from their order in the dataset, and hence, is not suitable for searching.

To support dynamic operations *efficiently*, an ADS should not store (any function of the) explicit positions of its data items. An implicit property such as the rank helps find and locate all items recursively, while making efficient updates possible.

3.2 From Explicitly-ordered ADSs to Implicitly-ordered ADSs

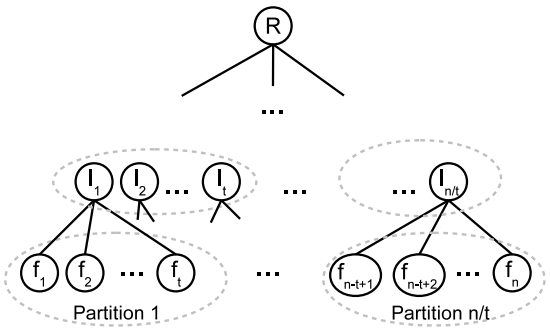


Figure 3: General form of partitioning.

Converting a linear explicitly-ordered ADS to an implicitly-ordered ADS essentially means reducing the update cost from linear to sublinear. For this, we can divide the dataset into partitions, and store each partition in a separate explicitly-ordered ADS. Hence, updating an item will only affect one partition. To keep unity of the entire dataset, we need to tie these partitions together in a secure manner. The general form is shown in Figure 3. The n items at the lowest level are divided into n/t partitions, each of size t. All items in a partition are children of a node at one level above. These internal nodes are also divided into partitions of size t in the same way. This process goes on until the root is constituted, and generates $\lceil \log_t n \rceil + 1$ levels.

Letting $t = n$ produces one partition of size n at level zero, and a root node at level one. The update cost is $O(n)$ and the search cost is $O(1)$. Accumulator is an example of this type. Setting $t = \sqrt{n}$ leads to $\sqrt{n}$ partitions at level zero, each of size $\sqrt{n}$, organized in a three-level architecture. Hence, the search cost is 3, which is $O(1)$, and the update cost is $\sqrt{n}$ at each level, leading to $3\sqrt{n}$ in total, that is $O(\sqrt{n})$. When $t = n^\epsilon$, the search cost is $O(\log_t n) = O(1/\epsilon)$ and the update cost is $O(t \log_t n) = O(n^\epsilon/\epsilon)$. For constant ϵ, these are $O(1)$ and $O(n^\epsilon)$ as in the authenticated hash table [32]. The minimum possible value is $t = 2$ that generates partitions of size two at the lowest level, and $\log n$ levels, leading to $2 \log n = O(\log n)$ search cost and $O(\log n)$ update cost. In fact, this is a binary tree. A comparison among these ADS types regarding their operation complexities is given in Table 1.

Therefore, an implicitly-ordered ADS is tree-based in general. The rank-based authenticated skip list [18] and Flexlist [19] are two examples. Flexlist is built on the length of data, and handles variable-length data blocks. Formal definitions follow.

Notation. A tree T has some leaf nodes on which the internal nodes are built. We denote the set of leaves and internal nodes of T as $L(T)$ and $I(T)$, respectively. Each internal node has some (two for binary tree) children. We show the parent node of each node $u \in T$ as $P(u)$. By definition, the parent node of the root is null. For each non-leaf internal node $u \in I(T)$, we show its children as $C(u) = \{v \in T : P(v) = u\}$.

Definition 3.2 (Rank-based tree) *In a rank-based tree: 1) Each node $u \in T$ has a property called **rank**, denoted as $R(u)$. 2) For each leaf node $u \in L(T)$, $R(u)$ = the number (size) of data block(s) u stores*[2]*. 3) For each internal node $u \in I(T)$, $R(u) = \Sigma_{v \in C(u)} R(v)$.*

[2]A leaf node storing a vector can count it as one element, or as many as its size. In either case, the order of elements of the vector is preserved separately, e.g., by storing hash of the whole vector or binding each element to its position.

By rank-based ADS, we mean a rank-based tree. Figures 7a and 10a present rank-based ADSs where the rank of each node is the total number and size of blocks accessible through that node, respectively.

Definition 3.3 (Authenticated tree) *In an authenticated tree: 1) Each node $u \in T$ stores some authentication information denoted as $A(u)$. 2) For each leaf node $u \in L(T)$, $A(u)$ is computed as a function $g(d_u, inf_u)$, where d_u is the data item stored at u and inf_u shows the other required information (e.g., the position of u in the tree, or its label). 3) For each internal node $u \in I(T)$, $A(u)$ is computed as a function $g(A(v_1), ..., A(v_t), inf_u)$, where $A(v_i)$ is the authentication information of the i^{th} children of u and inf_u represents other required information (e.g., the number of u's children, or its label).*

Lemma 3.1 *An authenticated tree can be rank-based.*

Proof 3.1 *Storing rank $R(u)$ of each node u in the extra information inf_u of the function $g()$ in Definition 3.3 makes an authenticated tree rank-based. This does not affect security of the authenticated tree.*

Definition 3.4 (Implicitly-ordered tree) *An implicitly-orde- red tree is an authenticated tree in which: 1) the authentication information $A(u)$ of a node u does not directly depend on the position of u in the tree, while 2) all leaf nodes can be located starting from the root, given their position in the tree.*

Lemma 3.2 *A rank-based authenticated tree is implicitly-ordered.*

Proof 3.2 *The first condition is easy to verify. The following search algorithm shows how to locate a leaf node using its position in the tree, starting from root. (It uses a binary tree for simplicity and familiarity, but it is applicable on other trees with slight modification.)*

Search. To search for the i^{th} node in a rank-based ADS, starting from the root, we compare i with the rank of the left child (*below* in an skip list). If $i \leq leftchild.rank$, the node we are looking for is in the left subtree and we continue with the left child, otherwise we set $i = i - leftchild.rank$ to compute the proper relative index, and continue with the right child. This process goes on until the i^{th} node is met at the leaf level, as illustrated in Algorithm 3.1. This algorithm works for a tree that stores data items only at the leaf nodes. If the internal nodes also store data items, we need to check for the data in all visited nodes. It requires small changes to support this case. Search algorithms of the existing schemes [18, 20] are special cases of our generic implicitly-ordered ADS search algorithm.

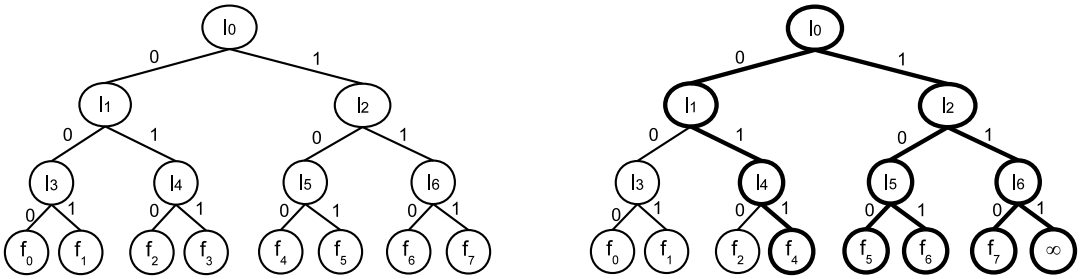
3.3 Explicitly-ordered Authenticated Data Structures

Merkle hash tree [30] is an explicitly-ordered ADS. Figure 4a presents a Merkle hash tree storing a file f divided into 8 blocks as $f = (f_0||f_1||...||f_7)$. The file blocks are stored only in leaf nodes, and each internal node stores a value computed as a function of values of it children. We do not show the functions in figures for simplicity. To find the i^{th} block, the search algorithm follows the corresponding path on the tree based on the binary representation of i. Since each block occupies its dedicated position, inserting a value in an intermediate position requires all existing values from that position up to the end be moved one place ahead and re-bound to their new positions. After deleting a block, e.g., the fourth

Algorithm 3.1: `Search`

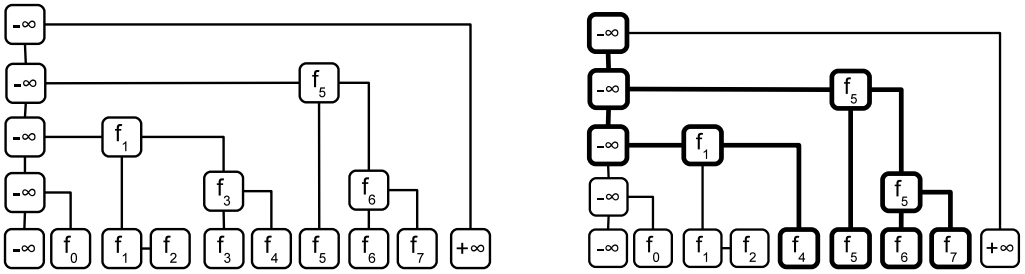
Input: *Node*: Root node of the tree,
i: The relative position of the block being searched.
Output: π: The membership proof.

```
1  if Node.leftchild == NULL then
2  |   return Node.value;
3  else
4  |   if i ≤ Node.leftchild.rank then
5  |   |   return Search(Node.leftchild, i) || Node.rightchild.value;
6  |   else
   |   |   // If interior nodes store data, first compare with the current node itself.
7  |   |   return Node.leftchild.value ||
8  |   |   Search(Node.rightchild, i − Node.leftchild.rank);
```



(a) An explicitly-ordered Merkle hash tree. (b) f_3 is deleted.

Figure 4: Deleting a block (f_3) form an explicitly-ordered Merkle hash tree.



(a) An explicitly-ordered authenticated skip list. (b) f_3 is deleted.

Figure 5: Deleting a block (f_3) form an explicitly-ordered authenticated skip list.

block (f_3) in Figure 4a, all remaining blocks should be shifted one place back, and re-bound to the new positions, as illustrated in Figure 4b. The nodes updated due to this deletion are drawn bold. Note that the internal nodes also need to be updated, since the authentication information changes. This operation requires $O(n)$ re-computations not only on the data structure but also on the authentication information, requiring cryptographic re-computation as well.

Authenticated skip list is another tree-based explicitly-ordered ADS as used in Figure 5a for storing the same file f=(f_0|| f_1||...||f_7). Deleting f_3 leads to the ADS in Figure 5b that shows changes in all remaining nodes up to the end in bold lines.

There is an important difference between a Merkle hash tree and an authenticated skip list: while the former is probabilistically balanced, the latter may become imbalanced. Even if the position-based complexity issues are somehow resolved, in case of many insertions to the same position, a Merkle tree loses the logarithmic cost benefits and becomes a linear ADS.

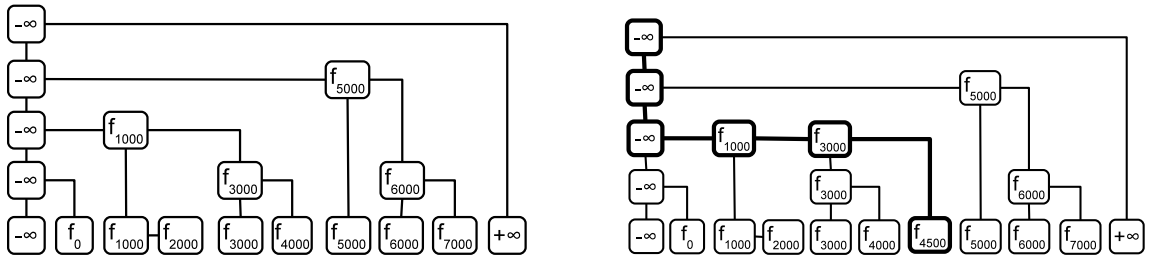
Since the main problem with these two ADSs is the position update problem, one may choose not to assign the block positions consecutively, and leave some positions empty between any two successive blocks. This allows some limited dynamism: When a block is inserted, it is assigned an unused position in the range it belongs to. Figure 6a represents an authenticated skip list who stores the blocks at positions that are multiples of their position in the file, i.e., f_0 is stored at position 0, f_1 at position 1000, f_2 at position 2000,.... Hence, 999 new blocks can be inserted between each two successive blocks. Adding a new block between f_4 and f_5 in Figure 6a leads to the ADS in Figure 6b.

When all empty positions are occupied, the original problem will again be encountered, though it can be solved for another limited time with a *rebuild* that assigns new positions to all blocks and creates empty positions in between. The rebuild is a linear operation.

There is another important problem with this ADS. The client has no way to find the position each block is outsourced at. For example, she cannot realize that the 4^{th} and 6^{th} blocks are stored at positions 4000 and 5000, respectively, but the 5^{th} block is kept at position 4500. Hence, she cannot perform block search on the outsourced data that is the base for other operations.

3.4 Implicitly-ordered Authenticated Data Structures

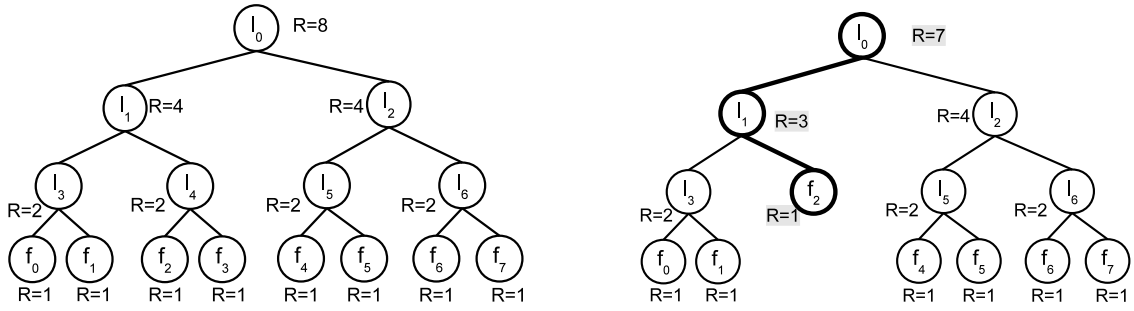
An implicitly-ordered Merkle hash tree and an implicitly-ordered authenticated skip list are shown in Figures 7a and 9a, respectively. Each node has a *rank* specifying the number of blocks accessible through that node, and is used during search, as described in Algorithm 3.1. File blocks are stored only at leaf nodes. Each internal node stores a value computed based on the values of its children and its own rank.



(a) An explicitly-ordered authenticated skip list with non-contiguous block indices.

(b) A block is added to the position 4500.

Figure 6: Adding a block into an explicitly-ordered authenticated skip list with non-contiguous block indices.



(a) An implicitly-ordered Merkle hash tree.

(b) f_3 is deleted.

Figure 7: Deleting a block (f_3) form an implicitly-ordered Merkle hash tree.

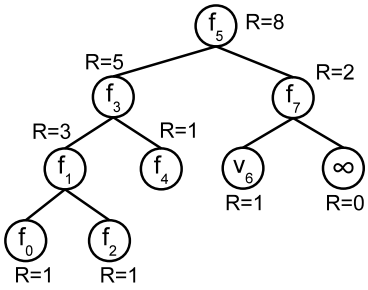


Figure 8: Storing data at internal nodes as well.

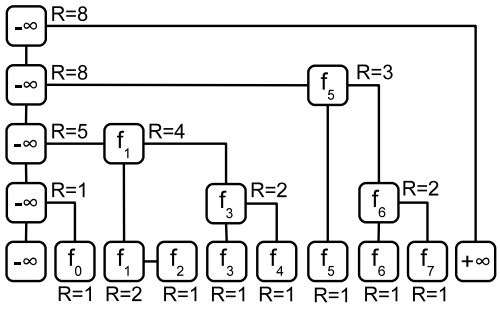
An update operation affects the nodes on the path from the updated node up to the root, as shown in Figures 7b and 9b, that is an $O(\log n)$ operation [34, 26]. No update is required at the other parts of the ADS.

Alternatively, one can store the blocks at both leaf and internal nodes. This way, Figure 7a is converted into Figure 8. The search algorithm needs to look for file blocks at the internal nodes as well.

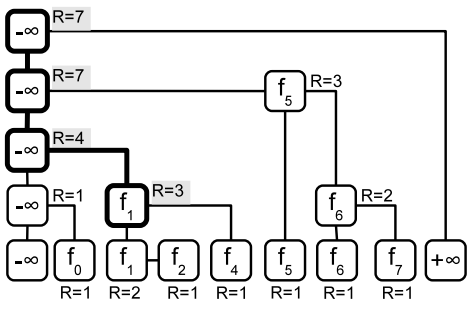
FlexList [19] is another rank-based authenticated skip list, where the rank of a node is the length of data accessible through that node, as represented in Figure 10a. This figure stores a file of size 15 KB divided into blocks of different lengths. The operation complexities are logarithmic. Figure 10b illustrates insertion of a new block (f_3) of size 2 KB into Figure 10a.

Since the blocks are not bound to their positions, and the rank of each node is independent of the rank of other nodes that do not have anything in common (e.g., a rank update in the left subtrees of Figures 7a and 8 will not affect their right subtrees), update operations will *not* affect other nodes. Deleting a block, e.g., the fourth block (f_3) in Figures 7a and 9a, affects only the ranks of $O(\log n)$ nodes on the path from the node storing f_3 up to the root, as illustrated in Figures 7b and 9b. The positions of the remaining blocks will be adapted automatically. The same holds when a new block is inserted, as shown in Figure 10b.

Other ADSs. We can add the rank property into other authenticated balanced trees, such as AVL [1], red-black [7, 27], or 2-3 trees [2] to make them implicitly-ordered. In these ADSs, after an update, only the nodes on the path from the update point up to the root may need to update their ranks. In the AVL tree, for example, there are two kinds of rebalancing operations to eliminate any imbalance in the tree that might happen after an update [46]: single rotation and double rotation. Figure 11a shows an imbalanced AVL tree after an insertion in the left subtree of R_i. The single rotation changes the ADS into the one shown in Figure 11b, which shows that this operation changes only the ranks of the nodes R_i and R_{i+t}. The ranks of all subtrees and the path from the parent of R_i up to the root will not change, but their authentication information require re-computation done in $O(\log n)$ time.

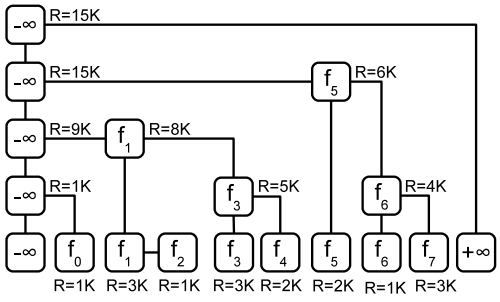


(a) An implicitly-ordered authenticated skip list.

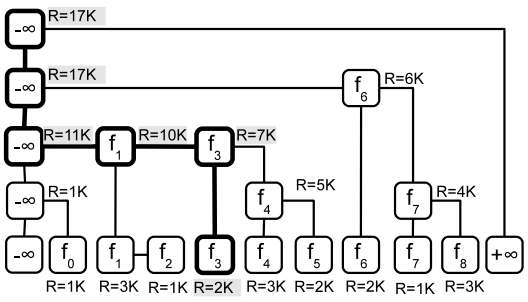


(b) f_3 is deleted.

Figure 9: Deleting a block (f_3) form an implicitly-ordered authenticated skip list.



(a) A FlexList storing a 15KB file.



(b) A 2KB block f_3 is inserted.

Figure 10: Inserting a new block (f_3) into a FlexList.

3.5 ADS Security Definitions

Definition 3.5 (Correctness of ADS) *For all valid proofs π and answers ans the server generates for the client commands,* `Verify` *algorithm accepts with overwhelming probability.*

Definition 3.6 (The ADS security game) *is played between two stateful parties: the challenger acting as client, and the adversary playing the role of server:*

Key generation *The challenger generates the private and public keys (sk, pk) by running* $\texttt{KeyGen}(1^\lambda)$*, and shares the public key pk with the adversary.*

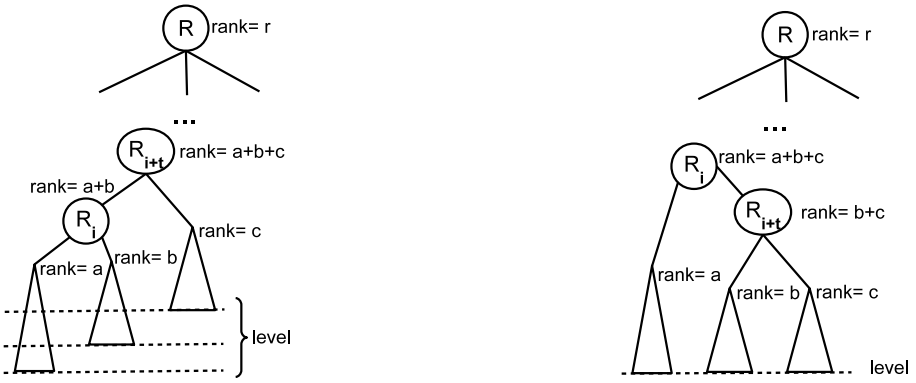
Setup *The adversary sends a command cmd with an answer ans and a proof π to the challenger. The challenger verifies them using* `Verify`*, and shares the result with the adversary. The challenger applies the update commands whose proofs are accepted on her local data. These interactions are repeated polynomially-many times. Call the latest version of date stored at the challenger's ADS, D.*

Challenge *The adversary sends to the challenger a command cmd, an answer ans', and a proof π'. He wins if his answer ans' does not match the result of running cmd on D, while cmd, ans', and π' are accepted by the challenger.*

Definition 3.7 (Security of ADS) *The ADS is secure if no PPT adversary can win the ADS game with probability better than negligible in the security parameter.*

Proof 3.3 *This is proven for different ADSs separately. Merkle [30] introduced and showed the security of Merkle hash tree. Goodrich et al. [24] investigated the security of the ADSs based on RSA one-way accumulator. Noar and Nissim [31] did the job for 2-3 trees. Papamanthou et al. proved the security of authenticated hash tables [32] and the ADSs based on the authenticated skip list or red black tree [33]. Etemad and Küpçü [21, 22] proved security of the HADS.*

Adding rank affects only the function A(.) computing authentication information of the nodes. It does not influence security of the ADS; the same security definitions and proofs are applicable. For example, the process of adding rank to an authenticated skip list and proving its security is discussed in [18].



(a) An imbalanced AVL tree.

(b) Single rotation and rank update.

Figure 11: Single rotation and rank update on .

4 Basic DPDP Construction

PDP [3] and many following schemes [14, 43, 44, 36, 37, 48] support only static data, making them suitable for archival storage. However, to make the cloud storage functional, we need to support dynamic operations on the outsourced data.

Two problems with dynamism, as stated in Observation 2, are the complexity of update operations and the possibility of *replay attacks*, i.e., a malicious server can ignore updates on the outsourced data and generate proofs using an old (correct) version of data that passes the client verification. One simple way to prevent the replay attacks is to store some (small) information (e.g., a hash value) about each block at the client side (as metadata), which entails $O(n)$ local storage, where n is the number of file blocks. But, this local metadata cannot be easily outsourced as it will be either insecure or inefficient, and it does not solve the complexity problem. On the other hand, keeping nothing about the outsourced data locally, the client has no way of checking data integrity.

There are different ways of generating constant-size metadata for a large file divided into n blocks: Aggre- gating or hash-chaining all blocks, storing the blocks in an accumulator or in a tree. The aggregation and hash-chaining require all blocks for verification and are inefficient communication-wise. The accumulator is not efficient computationally since an update has cost $O(n)$. We need an ADS sampling all file blocks with a constant-size metadata, where a block update does not affect other blocks. The ADS should associate each block to its specific position in the file to prevent re-ordering them maliciously, and efficiently preserve their order even in the presence of update operations.

Definition 4.1 (DPDP scheme) *A DPDP scheme is composed of the following interactive protocols between a stateful client and a stateful server*[3]*:*

- $\texttt{Setup}(1^\lambda)$*: The client starts up this protocol to generate the secret and public keys, given the security parameter* λ*. She gives the public key to server.*
- $\texttt{Update}(\overrightarrow{op}, \overrightarrow{ind}, \overrightarrow{val})$*: The client uses this protocol to ask the server perform each* $op_i \in \{\texttt{Insert}, \texttt{Delete}, \texttt{Modify}\}$ *on the position* ind_i*, given the value* val_i *(empty for deletion). There should be a value in the* ind_i^{th} *position for modification.*
- $\texttt{Read}(\overrightarrow{ind})$*: is used by the client to read the blocks specified by an index vector* $\overrightarrow{ind}$*. She specifies* $\overrightarrow{ind}$ *as input, and outputs the respective values* $\overrightarrow{val}$ *with a proof* π *showing authenticity of* $\overrightarrow{val}$*.*
- $\texttt{Audit}()$*: The client specifies a challenge vector and uses this protocol to check if the server keeps storing the outsourced data correctly. She finally outputs an acceptance or a rejection notification.*

During the execution of `Setup`, both the client and the server create their own local states. The protocols following `Setup` will use these local states, and the following `Update` protocols will update them. The client can either send the whole file f in one execution of the `Update` protocol, or send it

[3]We define the scheme as a set of interactive protocols, different from the original definition of Erway *et al.* [18], and similar to the dynamic PoR definition of Cash *et al.* [11].

block-by-block by running the `Update` protocol multiple times. Both methods result in the same final configuration on the server.

We defined two similar protocols `Read` and `Audit`. For now, they do the same job[4], i.e., the `Audit` can be used to read the selected parts of the outsourced data. Later on, when we talk about the efficiency, it will be clear that a separate read protocol is needed.

4.1 DPDP from Implicitly-ordered ADSs (Basic Construction)

Constructing the ADS. An implicitly-ordered ADS, Figure 7a, 9a, or 10a for instance, is used to construct a dynamic provable data possession scheme. Each block of a large file $f = (f_1||f_2||...||f_n)$ is assigned to the corresponding node of the ADS. Leaf nodes store a function (e.g., hash) of the assigned blocks, and each internal node is assigned a value computed as a function of the values of its children and its own rank, according to the construction rules of the ADS.

The client constructs the ADS and outsources it together with the file to the server, and keeps only the digest stored at the root of the ADS. The `Update` command builds the ADS at the outset, and performs the following updates on it. It requires the operation $op \in \{\texttt{Insert}, \texttt{Delete}, \texttt{Modify}\}$, the block index i, and the new value of the block for `Insert` and `Modify`. For each update, the client prepares the corresponding command according to the ADS format, and sends it to the server. On receipt, the server performs the update on both the file and the respective ADS, and sends the proof of operation to the client. The client verifies the proof, and if accepted, updates the locally-stored digest to the new value. Later audits will be checked against this new value.

To read the i^{th} block, the client uses `Read` protocol with the input i that returns the i^{th} block and its membership proof. The client accepts the block if the proof is verified. In an `Audit` protocol, the client chooses a vector of block indices at random, $\vec{c}$, and sends the corresponding command to the server, who generates the membership proof for all challenged blocks and sends it back. The client reconstructs the required parts of the ADS according to the proof, and compares the computed root digest with the one stored locally. Any mismatch leads to a rejection, and is an indication of the server's misbehavior. Figure 12 represents our basic DPDP construction using an implicitly-ordered ADS.

4.2 Dynamic Provable Data Possession Security Definitions

We define security of a DPDP scheme through the following game, inspired by the definitions from [3, 18]:

Definition 4.2 (The data possession game of DPDP) *The game is played between a stateful challenger $\mathcal{S}$ acting as the client and a stateful adversary $\mathcal{A}$ playing the role of the server.*

Setup. *The challenger runs* $\texttt{Setup}(1^\lambda)$ *to generate the secret and public key pair (sk, pk), and shares the public key with the server.*

Adaptive queries. *The adversary $\mathcal{A}$ specifies a file F and asks the challenger $\mathcal{S}$ to outsource it. $\mathcal{S}$ starts the corresponding* `Update` *protocol with $\mathcal{A}$. Then, $\mathcal{A}$ gives the required information to $\mathcal{S}$ and asks her to start a protocol* `Update`, `Read` *or* `Audit`. *$\mathcal{S}$ starts the requested protocol and notifies $\mathcal{A}$ about the result, whether an acceptance or a rejection. $\mathcal{S}$ updates her local metadata according to the* `Update` *protocols that are verified. This process is repeated polynomially-many times. Call the last version of file generated according to the accepted* `Update` *runs, F.*

Challenge. *Finally, $\mathcal{A}$ asks $\mathcal{S}$ to start an* `Audit` *protocol. $\mathcal{S}$ specifies a challenge vector $\vec{c}$ and runs the* `Audit` *protocol with $\mathcal{A}$. $\mathcal{A}$ wins if his answer is accepted by $\mathcal{S}$. For extraction, $\mathcal{S}$ can reset $\mathcal{A}$ to the point before the challenge phase started, and repeat the challenge polynomially-many times. ($\mathcal{S}$ is about to extract the challenged parts of F through the $\mathcal{A}$'s responses that are accepted by* `Audit`*.)*

Definition 4.3 (Security of DPDP) *A DPDP scheme is secure if for any PPT adversary $\mathcal{A}$ who wins the above game with non-negligible probability, there exists a PPT extractor who can extract the challenged parts of the file with non-negligible probability by rewinding $\mathcal{A}$ polynomially many times.*

[4] An inefficient way to read t blocks is to use the `Audit` protocol t times; one block is read in each run.

Let $\Lambda = (\texttt{KeyGen}, \texttt{Certify}, \texttt{Verify})$ be a secure ADS scheme.
Construct a DPDP scheme $D = (\texttt{Setup}, \texttt{Update}, \texttt{Read}, \texttt{Audit})$ as:

- $\texttt{Setup}(1^\lambda)$:
  - The client runs $(sk, pk) = \Lambda.\texttt{KeyGen}(1^\lambda)$,
  - outputs (sk, pk), and sends pk to the server.
- $\texttt{Update}(\overrightarrow{op, ind, val})$:
  - The client prepares the respective update command cmd, and
  - sends the command cmd and $(\overrightarrow{op, ind, val})$ to the server.
  - The server applies $(\overrightarrow{op, ind, val})$ on the outsourced data,
  - runs $\pi = \Lambda.\texttt{Certify}(pk, cmd)$, and sends π to the client.
  - The client runs $\Lambda.\texttt{Verify}(sk, cmd, null, \pi, st_C)$.
  - If it is accepted, she updates her local state accordingly.
  - Otherwise, the server's misbehaviour is detected.
- $\texttt{Read}(\overrightarrow{ind})$:
  - The client send the read command cmd to the server.
  - The server runs $(ans, \pi) = \Lambda.\texttt{Certify}(pk, cmd)$, and
  - sends back the answer ans and proof π.
  - The client runs $\Lambda.\texttt{Verify}(sk, cmd, ans, \pi, st_C)$.
  - If it is accepted, she consumes the read data.
  - Otherwise, the server's misbehaviour is detected.
- $\texttt{Audit}()$:
  - The client generates a challenge vector $\vec{c}$, and
  - sends the respective audit command cmd to the server.
  - The server runs $(ans, \pi) = \Lambda.\texttt{Certify}(pk, cmd)$, and
  - sends back the answer ans and proof π.
  - The client runs $\Lambda.\texttt{Verify}(sk, cmd, ans, \pi, st_C)$.
  - If it is accepted, she will be convinced with high probability that the server is storing her data intact.
  - Otherwise, the server's misbehaviour is detected.

Figure 12: The basic DPDP construction.

4.3 Security of Basic DPDP Construction

Theorem 4.1 *If $\Lambda = (\texttt{KeyGen}, \texttt{Certify}, \texttt{Verify})$ is a secure implicitly-ordered ADS scheme, our construction, given in Figure 12, is a secure DPDP scheme.*

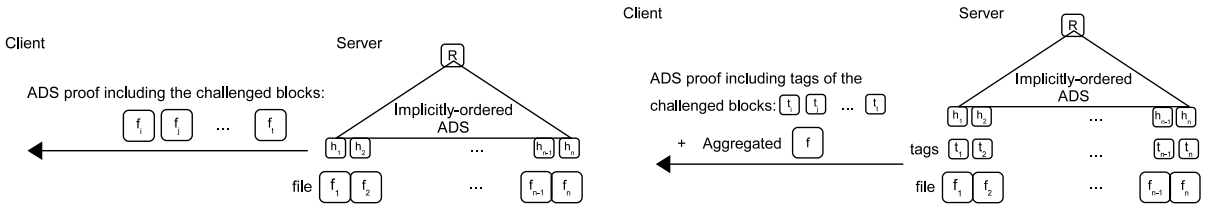
Proof 4.1 *According to Definition 4.3, a DPDP scheme is secure if the challenger can extract the actual challenged blocks whenever the adversary wins the data possession game. That is, if the adversary's answer is accepted by the challenger, he should have knowledge of the challenged blocks.*

We reduce security of the DPDP scheme to that of the underlying implicitly-ordered ADS. If a PPT adversary $\mathcal{A}$ wins the data possession game with non-negligible probability, we use it to construct a PPT algorithm $\mathcal{B}$ who breaks security of the ADS with non-negligible probability or we can extract the original blocks from the challenge. $\mathcal{B}$ acts as the challenger in the data possession game with $\mathcal{A}$ and, in parallel, plays the role of server in the ADS game with ADS challenger $\mathcal{C}_\mathcal{A}$. $\mathcal{B}$ internally runs an honest server.

$\mathcal{C}_\mathcal{A}$ generates the ADS public and private keys, and sends the public key to $\mathcal{B}$ who forwards a copy to $\mathcal{A}$ and the honest server $\mathcal{S}$. The parties run the adaptive queries phase during which $\mathcal{B}$ sends $\mathcal{A}$'s requests to $\mathcal{C}_\mathcal{A}$ and sends its answer to $\mathcal{A}$ and $\mathcal{S}$.

Once the adaptive queries phase is done, $\mathcal{A}$ asks $\mathcal{B}$ to start an `Audit` *protocol. $\mathcal{B}$ generates a random challenge vector $\vec{c}$ and sends it to $\mathcal{A}$ and $\mathcal{S}$. On receipt, $\mathcal{A}$ prepares and returns the answer and proof to $\mathcal{B}$. If the proof verifies, then the adversary wins the game. There are two options:*

1. *The adversary's answer (the challenged blocks returned) is different from that of $\mathcal{S}$. In this case, $\mathcal{B}$ forwards adversary's answer and proof to $\mathcal{C}_\mathcal{A}$ as a forgery for the ADS.*



(a) Basic scheme.

(b) Blockless scheme.

Figure 13: Proof generation using the basic and blockless DPDP schemes.

2. *The adversary's answer (the challenged blocks returned) matches that of $\mathcal{S}$. In this case, $\mathcal{B}$ outputs adversary's answer as the result of extraction. (Note that the challenged data blocks are themselves included in the proof, and the extractor does not need to put any extra effort.)*

Whenever $\mathcal{A}$ wins with some probability p, one of the above options must have happened with the same probability. Since the ADS is secure, either p must be negligible, or we can extract the challenged blocks correctly, with non-negligible probability.

5 Blockless Dynamic Provable Data Possession

In the basic scheme given in previous section, the server sends all challenged blocks as part of the proof to the client. This is required by the ADS to enable the client verify authenticity of the challenged blocks. Obviously, block length can be a system parameter, and can be made small. But remember that smaller blocks would mean more blocks for the same file size, increasing n, and hence increasing the complexity of the ADS operations. There is indeed a way to ensure the server's response to the client is independent of the block size. This is called *blockless verification* in the literature.

The observations mentioned in Section 1.2 state that the main problems with dynamic operations over the outsourced data are related to managing the tags. Storing the tags, instead of file blocks, in an implicitly-ordered ADS enables the client to update the tags according to the updates on the outsourced file. The ADS guarantees the authenticity of tags, which in turn, guarantee integrity of the respective file. Moreover, this reduces the communication (not asymptotically) since a tag is smaller than the respective block, and the tag size is a constant independent of the block size.

Using the *homomorphic verifiable tags* (HVT) that are introduced first by Ateniese *et al.* [3] and later used by others in diverse forms [18, 48, 35], brings another advantage: *blockless verification* [3]. It means the client can verify that the server possesses the challenged blocks, even without access to the actual blocks. Using this important property, in response to a challenge, the server aggregates all challenged blocks into one block and sends it together with the ADS proof that includes the respective tags to the client. In essence, instead of putting all challenged blocks in the proof, the corresponding tags are included. This reduces the proof size considerably. Figures 13a and 13b visualize proof generation using the basic and efficient DPDP schemes, respectively.

Let b be the size of a block, t be the size of a tag value ($t < b$), and a be the size of a the authentication information $A(.)$. A challenge vector of size c generates a proof of size (at most) $c(b + a \log n)$ in the basic scheme and $b + c(t + a \log n)$ using HVTs. For a 1 GB file divided into $n = 500,000$ blocks each of size $b = 2$ KB, $c = 460$ blocks in a challenge, and employing a 32 B hash function as $A(.)$, the ADS proof is $\sim$ 273 KB. Hence, the basic scheme generates proofs of size 273 KB + 460∗2 KB $\simeq$ 1193 KB. Using HVTs of size $t = 128$ B, all challenged blocks are aggregated into one block and sent to the client along with the ADS proof including the respective tags. This leads to a proof of size 273 KB + 2 KB + 460∗128 B $\simeq$ 332 KB in our example, showing $\sim$ 4X improvement.

5.1 Tag Schemes

Tag schemes are used to provide message integrity that aims checking whether the received message is exactly the one sent by the origin. Data outsourcing schemes proposed for cloud use the same concept to enable the client check the authenticity of retrieved data.

To catch a misbehaving server with a high probability, the client regularly audits her data. Each audit checks authenticity of c randomly-selected blocks that is required to provide satisfactory probabilistic authenticity guarantees. A regular verifiable tag scheme is not efficient enough in this regard.

Using a homomorphic verifiable tag scheme, all challenged blocks can be aggregated into one block, in a way that the authenticity of all of them (and hence, the whole outsourced file with high probability) can be inferred. We assume the aggregation of data blocks are performed by the server, but either the client or the server can aggregate the tags. This, however, will not affect the communication efficiency.

Definition 5.1 (Homomorphic verifiable tag scheme) *A homomorphic verifiable tag scheme consists of the following polynomial-time algorithms:*

- $(sk, pk) \leftarrow \texttt{KeyGen}(1^\lambda)$ *is a probabilistic algorithm run by the client to set up the scheme and generate a secret and public key pair* (sk, pk)*, given the security parameter* λ*. The client then sends the public key* pk *to the server.*
- $\vec{t} \leftarrow \texttt{TagGen}(sk, f)$ *is a probabilistic algorithm run by the client to generate the tags of file blocks. Given the secret key* sk *and the file* f *as input, it generates a vector of homomorphic tags* $\vec{t}$*.*
- $\mu \leftarrow \texttt{CombineData}(pk, f, \vec{c})$ *is a deterministic algorithm run by the server upon receipt a challenge vector* $\vec{c}$*. It also takes as input the public key, and the file* f*, and generates a data block* μ*.*
- $\tau \leftarrow \texttt{CombineTag}(pk, \vec{t}, \vec{c})$ *is a deterministic algorithm run by either the client or server to generate an aggregated tag* τ*, given the public key* pk*, the vector of tags* $\vec{t}$*, and a challenge vector* $\vec{c}$ *as input.*
- $(\mu, \tau) \leftarrow \texttt{Prove}(pk, f, \vec{t}, \vec{c})$ *is a deterministic algorithm run by the server to compute and return an (aggregated) answer* μ *along with the aggregated tag* τ *to a query given as a challenge vector* $\vec{c}$*.*
- $\{\texttt{accept}, \texttt{reject}\} \leftarrow \texttt{Verify}(sk, \vec{c}, \mu, \tau)$ *is a deterministic algorithm run by the client to verify the received block(s) and tag(s). The secret key* sk*, the challenge vector* $\vec{c}$*, and the aggregated block* μ *and tag* τ *are given as input. It outputs an acceptance or a rejection notification.*

We now give the security definitions of an HVT scheme inspired by definitions from [5].

Correctness. An HVT scheme is correct if $\forall\ \lambda \in \mathbb{N}$, $\forall\ (pk, sk) \leftarrow \texttt{KeyGen}(1^\lambda)$, $\forall\ f$, $\forall\ \vec{t} \leftarrow \texttt{TagGen}(sk, f)$, and $\forall\ \vec{c} \in \mathbb{Z}_n^c$ we have: $\texttt{Verify}(sk, \vec{c}, \texttt{CombineData}(pk, f, \vec{c}), \texttt{CombineTag}(pk, \vec{t}, \vec{c})) = \texttt{accept}$.

Security. Informally, an HVT scheme is secure if no PPT adversary can generate a valid aggregated tag (or a set of tags to be aggregated) for a new set of data blocks were not already provided by the client. Next, we define the unforgeability game for the HVT scheme.

Definition 5.2 (Unforgeability game of an HVT scheme) *Let* $\Pi = (\texttt{KeyGen}, \texttt{Tag}, \texttt{CombineData}, \texttt{CombineTag}, \texttt{Prove}, \texttt{Verify})$ *be an HVT scheme. The Unforgeability game* $\texttt{ForgeGame}_{\mathcal{A}}(\lambda)$ *between a challenger* $\mathcal{S}$ *acting as the client and an adversary* $\mathcal{A}$ *playing the role of server is defined as follows. The adversary has oracle access to* $\texttt{TagGen}(sk, .)$ *for any message of his choice.*

- ***Initialization***. $\mathcal{S}$ *runs* $\texttt{KeyGen}(1^\lambda)$ *to generate the key pair* (pk, sk)*, and shares* sk *with* $\mathcal{A}$*.*
- ***Setup***. *The adversary* $\mathcal{A}$*, given* 1^λ *and oracle access to* $\texttt{TagGen}(sk, .)$*, outputs a file* f*. The challenger runs* $\vec{t} \leftarrow \texttt{TagGen}(sk, f)$ *and sends* $\vec{t}$ *to* $\mathcal{A}$*.* $\mathcal{A}$ *continues to have oracle access to* $\texttt{TagGen}(sk, .)$*.*
- ***Challenge***. $\mathcal{A}$ *outputs a challenge vector* $\vec{c}$*, a data block* μ'*, and a tag* t'*.*

$\mathcal{A}$ wins the game if his answer (μ' and t') to the selected challenge vector $\vec{c}$ gets accepted by the challenger while $\mu' \neq \texttt{CombineData}(pk, f, \vec{c})$. The output of the experiment is 1 in this case, and 0 otherwise.

Definition 5.3 (Unforgeability of an HVT scheme) *A homomorphic verifiable tag scheme* $\Pi = (\texttt{KeyGen}, \texttt{TagGen}, \texttt{CombineData}, \texttt{CombineTag}, \texttt{Prove}, \texttt{Verify})$ *is unforgeable under an adaptive chosen-message attack if* $\forall$ *PPT adversaries* $\mathcal{A}$*,* $\exists$ *a negligible function* $\nu()$ *such that:* $\Pr[\texttt{ForgeGame}_{\mathcal{A}}(\lambda) = 1] \leq \nu(\lambda)$*.*

5.2 Blockless DPDP Construction

Constructing the ADS. The client computes HVTs of all file blocks, puts them in an implicitly-ordered ADS, and outsources the ADS along with the file and tags at the server, while keeping the digest of the ADS locally as metadata. The ADS guarantees the server stores the tags intact, and uses the correct tags in responses to the client queries. The HVTs, in turn, are used to verify that the server keeps genuine outsourced file blocks. This combination results in an efficient DPDP scheme.

Later on, the client challenges a random subset of the blocks. In response, the server gives a proof including two parts. The first part is generated using the ADS and, if accepted, indicates the tags coming from the server are authentic. The second part is the aggregation of the challenged blocks, which

Let $\Lambda = (\texttt{KeyGen}, \texttt{Certify}, \texttt{Verify})$ be a secure ADS scheme, and $\Pi = (\texttt{KeyGen}, \texttt{TagGen}, \texttt{CombineTag}, \texttt{CombineData}, \texttt{Prove}, \texttt{Verify})$ be a secure HVT scheme. Construct a DPDP scheme $D = (\texttt{Setup}, \texttt{Update}, \texttt{Read}, \texttt{Audit})$ as:

- $\texttt{Setup}(1^\lambda)$:
  - The client runs $(sk_\Lambda, pk_\Lambda) = \Lambda.\texttt{KeyGen}(1^\lambda)$, and $(sk_\Pi, pk_\Pi) = \Pi.\texttt{KeyGen}(1^\lambda)$,
  - outputs $sk = \{sk_\Lambda, sk_\Pi\}$ and $pk = \{pk_\Lambda, pk_\Pi\}$, and
  - sends pk to the server.
- $\texttt{Update}(\overrightarrow{op, ind, val})$:
  - The client runs $t_i = \Pi.\texttt{TagGen}_{sk_\Pi}(ind_i, val_i)\ \forall\ op_i \in \{\texttt{Insert}, \texttt{Modify}\}$,
  - prepares cmd according to the ADS format, and
  - sends $[\vec{t},]$ $(\overrightarrow{op, ind, val})$, and cmd to the server.
  - The server [stores $\vec{t},$] processes $(\overrightarrow{op, ind, val})$,
  - runs $\pi = \Lambda.\texttt{Certify}(pk_\Lambda, cmd)$, and sends back π.
  - The client runs $\Lambda.\texttt{Verify}(sk_\Lambda, cmd, null, \pi, st_C)$.
  - If it is accepted, she updates her local state accordingly.
  - Otherwise, the server's misbehaviour is detected.
- $\texttt{Read}(\overrightarrow{ind})$:
  - The client sends the read command cmd to the server.
  - The server runs $(ans_t, \pi) = \Lambda.\texttt{Certify}(pk_\Lambda, cmd)$,
  - sets $ans_b = \{requested\ blocks\}$ and $ans = \{ans_b, ans_t\}$, and
  - sends ans and π to the client.
  - The client runs $\Lambda.\texttt{Verify}(sk_\Lambda, cmd, ans_t, \pi, st_C)$ and $\Pi.\texttt{Verify}(sk_\Pi, \overrightarrow{ind}, \Pi\texttt{CombineData}(pk_\Pi, ans_b, \overrightarrow{ind}), \Pi.\texttt{CombineTag}(pk_\Pi, ans_t, \overrightarrow{ind}))$.
  - If both are accepted, she consumes the data, ans_b.
  - Otherwise, the server's misbehavior is detected.
- $\texttt{Audit}()$:
  - The client generates a challenge vector $\vec{c}$, and
  - sends the respective audit command cmd to the server.
  - The server runs $(ans_t, \pi) = \Lambda.\texttt{Certify}(pk_\Lambda, cmd)$ and $(\mu, \tau) \leftarrow \texttt{Prove}(sk_\Pi, f, \vec{t}, \vec{c})$, and sends ans_t, π, μ and τ to the client.
  - The client runs $\Lambda.\texttt{Verify}(sk_\Lambda, cmd, ans_t, \pi, st_C)$ and $\Pi.\texttt{Verify}(sk_\Pi, \vec{c}, \mu, \tau)$.
  - If both verifications were accepted, she will be convinced that the server is storing her data intact, with high probability.
  - Otherwise, the server's misbehavior is detected.

Figure 14: Construction of a blockless dynamic provable data possession scheme.

(together with a similar aggregation of the tags[5]) shows that the server keeps the challenged file blocks (and hence, the outsourced file with high probability) intact.

Figure 14 gives the general form of constructing an efficient dynamic provable data possession scheme given black-box access to an HVT scheme and an implicitly-ordered ADS scheme.

5.3 Security of the Blockless DPDP

Theorem 5.1 *If* Π=($\texttt{KeyGen}, \texttt{TagGen}, \texttt{CombineTag}, \texttt{CombineData}, \texttt{Prove}, \texttt{Verify}$) *is a secure HVT scheme, and* Λ=($\texttt{KeyGen}, \texttt{Certify}, \texttt{Verify}$) *is a secure implicitly-ordered ADS, our construction in Figure 14 is a secure DPDP scheme according to Definition 4.3.*

Proof 5.1 *According to Definition 4.3, a DPDP scheme is secure if the adversary possesses sufficient knowledge of the challenged blocks in order to win the data possession game. This, in turn, helps the challenger extract the actual challenged blocks.*

[5]This aggregation can be done by either the server or the client that already has the required tags. To lighten the burden of the client, we can ask the server to compute and send it as part of the proof.

We reduce security of the DPDP scheme to that of its building blocks: the implicitly-ordered ADS and the homomorphic verifiable tag scheme. If a PPT adversary $\mathcal{A}$ wins the data possession game with non-negligible probability, we use it to construct a PPT algorithm $\mathcal{B}$ who breaks security of either of the two schemes, with non-negligible probability. $\mathcal{B}$ plays the role of the challenger in the data possession game with $\mathcal{A}$, and simultaneously, acts as the server in the ADS game played with the ADS challenger $\mathcal{C}_\mathcal{A}$ and in the tag game played with the tag challenger $\mathcal{C}_\mathcal{T}$. $\mathcal{B}$ internally runs an honest server $\mathcal{S}$.

$\mathcal{C}_\mathcal{A}$ and $\mathcal{C}_\mathcal{T}$ generate their public and private key pairs, and share their public keys with $\mathcal{B}$ who relays a copy to $\mathcal{A}$ and the honest server $\mathcal{S}$. The parties execute the adaptive queries phase during which $\mathcal{A}$ asks $\mathcal{B}$ to start a protocol $\in \{\texttt{Update}, \texttt{Read}, \texttt{Audit}\}$. For `Read` *and* `Audit`*, $\mathcal{B}$ generates a challenge vector, sends it to both $\mathcal{A}$ and $\mathcal{S}$, and forwards the answer parts from $\mathcal{A}$ to the respective challengers for verification. $\mathcal{B}$ notifies $\mathcal{A}$ about the verification result. For* `Update`*, $\mathcal{B}$ first asks $\mathcal{C}_\mathcal{T}$ to generate and return the respective tags (if it is insertion or modification), and then asks $\mathcal{C}_\mathcal{A}$ to prepare the respective command. Finally, $\mathcal{B}$ relays to both $\mathcal{A}$ and $\mathcal{S}$ what he receives from $\mathcal{C}_\mathcal{A}$ and $\mathcal{C}_\mathcal{T}$.*

Once the adaptive queries phase is done, $\mathcal{A}$ asks $\mathcal{B}$ to start an `Audit` *protocol. $\mathcal{B}$ specifies a random challenge vector $\vec{c}$, and sends it to both $\mathcal{A}$ and $\mathcal{S}$. On receipt, $\mathcal{A}$ and $\mathcal{S}$ generate the answer and proof and send them back. The adversary wins the game if the proof verifies. Two possible cases are:*

1. *The adversary's answer (the challenged blocks returned) is different from that of $\mathcal{S}$. This case represents a forgery for the ADS. $\mathcal{B}$ separates the respective parts of the adversary's answer and proof, relays them to $\mathcal{C}_\mathcal{A}$, and wins the ADS game with $\mathcal{C}_\mathcal{A}$.*
2. *The adversary's answer (the challenged blocks returned) matches that of $\mathcal{S}$. In this case, $\mathcal{B}$ rewinds the adversary to the state before the* `Audit` *protocol ran, and repeats the* `Audit` *protocol polynomially-many times to extract the challenged blocks. (We give a high level proof idea as we did not give a concrete construction. While existing HVT schemes all combine the data and tags linearly, any combining formula is applicable subject to existence of a proper extraction method. Currently, only the extraction of linear combinations is known. Since the adversary sends the aggregated data blocks to the challenger, we should show the challenger is able to extract the original file blocks. Ateniese et al. [5] showed the extraction process for the homomorphic linear authenticators where the aggregated answer is a linear combination of the challenged blocks. The extractor rewinds the adversary polynomially-many times. The process stops when c-many (c is the size of challenge vector) accepted linearly independent answers are found, and the challenged blocks can be extracted by solving the system of linear equation on the challenged blocks and the respective answers.)*

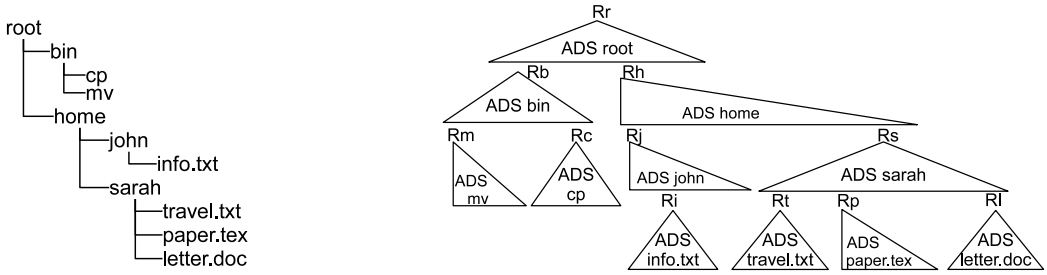
One of the above cases must have happened with some probability p for the adversary to win with the same probability p. Since both the ADS and tag schemes are secure, either p must be negligible, or we can extract the challenged blocks correctly, with non-negligible probability. Putting all together, our general blockless DPDP construction given in Figure 14 is secure supposed the underlying tag and implicitly-ordered ADS schemes are secure.

6 Extensions and Comparison

6.1 Hierarchical DPDP

Both our DPDP schemes provide integrity of a single file outsourced to an untrusted cloud server. However, sometimes a hierarchy of files is being outsourced [18], or some problems require extra properties such as availability [21]. Both problems can be handled in the same way employing hierarchical schemes. A useful property of implicitly-ordered ADSs is that they can be organized in a hierarchical manner, where the real data items are stored at leaf ADSs and each level serves as data for a level above. Making a hierarchical ADS (HADS) using (potentially different) regular ADSs is analyzed in detail by Etemad and Küpçü [21, 22].

We briefly describe building an HADS for a hierarchy of files and folders using the example in Figures 15a and 15b. An ADS is built for each file in the innermost folders, e.g., info.txt in folder tom and travel.txt, paper.tex, and letter.doc in folder rose, in Figure 15a. Roots of these ADSs in each folder are used as data to create an ADS for the folder. This process goes on until the ADS of root folder is built whose digest is stored in the client's local metadata. Figure 15b shows the HADS constructed for the hierarchy of files in Figure 15a.



(a) A small part of a directory structure.

(b) An HADS storing the directory.

Figure 15: A small part of a directory structure stored in an HADS.

An important property of the HADS is that it does not affect the DPDP scheme. It only affects the server proof generation and the client verification mechanism. Moreover, using the HADS, the server can employ any degree of replication and distribution. He can create any number of the same ADS and/or even distribute them among multiple servers, while the whole process is transparent to the client [21].

Employing the HADS or storing the whole directory hierarchy in a flat form, i.e., building a separate ADS for each file and storing them all in the same place without regarding their location in the directory hierarchy is a storage/computation tradeoff for the client. Using HADS, she stores a constant metadata while she needs to verify proofs of all level in the respective hierarchy. On the other hand, a flat architecture requires her to verify only one proof while increasing the metadata linearly in the number of all outsourced files.

6.2 Comparison

Our scheme proposes a general framework for constructing efficient DPDP schemes, given access to a secure implicitly-ordered ADS scheme and a secure HVT scheme. All existing DPDP schemes are specific cases of this general model.

By ignoring data dynamism (hence, throwing away the implicitly-ordered ADS) and using RSA-based HVTs, we achieve the PDP [3]. Also, ignoring data dynamism and using algebraic signature based HVTs gives the data possession checking scheme by Chen [14]. PPDP [44] and CPDP [48] are also static schemes based on BLS signatures.

Zhang and Blanton [47] store a range of data blocks on which an update is performed for each node in the *block update tree*. The integrity of each block is protected using MAC. Instantiating the ADS as the block update tree, and the HVT as the MAC in our general model generates this scheme.

Esiner *et al.* [20] proposed FlexList, which supports variable-size blocks, and used it to construct a DPDP scheme, FlexDPDP. FlexList, as an implicitly-ordered ADS, together with the RSA-based HVTs make a special case of our general scheme.

Using an implicitly-ordered Merkle hash tree together with BLS signatures for blocks reduces our scheme to that proposed by Wang *et al.* [45].

A comparison among these schemes is given in Table 2.

7 Conclusion

A general framework for constructing secure DPDP schemes is presented in this paper. Also, requirements of the building blocks are discussed. We argued that a secure DPDP scheme can be constructed with black-box access to a homomorphic verifiable tag scheme and an implicitly-ordered ADS scheme.

The implicitly-ordered ADS is based on a property that does not depend directly on the block positions, making later updates possible. However, to be efficient enough, tree-like structures such as authenticated skip list, or 2-3 tree are good candidates. These structures pose a good rebalance property after updates.

The homomorphicity of tag schemes helps aggregate all challenged blocks into one, hence, enhances the communication while still providing verifiability.

We showed that almost all existing DPDP schemes use the same architecture. We analyzed this architecture in detail and developed the general framework for constructing (PDP and) DPDP schemes,

Table 2: A comparison of DPDP schemes. ($\checkmark^\circ$ means the scheme is not efficiently or fully dynamic.)

Scheme	Dynamic?	**ADS**		**Tag**	
		Type	Rank-based?	Type	Homomorphic?
PDP [3]	×	Vector	×	RSA-based	✓
AS-RDPC [14]	×	Vector	×	Algebraic signature	✓
PPDP [44]	×	Vector	×	BLS signature	✓
CPDP [48]	×	Vector	×	BLS signature	✓
Wang *et al.* [42]	$\checkmark^\circ$	Matrix	×	Blinded token	×
[6]	$\checkmark^\circ$	Block status table	×	Incremental numbers	×
SPDP [4]	$\checkmark^\circ$	Vector	×	Hash-based token	×
EDPDP [47]	$\checkmark^\circ$	Block update tree	×	MAC	×
PV-DPDP [45]	$\checkmark^\circ$	Merkle hash tree	×	BLS signature	✓
DPDP [18] (scheme I)	✓	Auth. skip list	✓	-	×
DPDP [18] (scheme II)				Factoring-based	✓
DPDP [17] (RSA tree)		RSA tree			
FlexDPDP [20]	✓	FlexList	✓	Factoring-based	✓

encompassing existing ones. Newer proposals can now build the two parts (the ADS and the tags) separately. Once the ADS is proven secure, it can be converted to an implicitly-ordered version via our framework and can be used to construct a basic DPDP scheme. If unforgeability of the homomorphic tags is also proven, they can be combined to obtain a blockless DPDP scheme using our framework. Thus, we expect our framework to yield to novel proposals improving several aspects of the existing schemes, and also to serve as a comparison metric.

Acknowledgement

We acknowledge the support of TÜBİTAK, the Scientific and Technological Research Council of Turkey, under project number 114E487, as well as European Union COST Action IC1306.

References

[1] M AdelsonVelskii and Evgenii Mikhailovich Landis. An algorithm for the organization of information. Technical report, DTIC Document, 1963.

[2] Alfred V Aho and John E Hopcroft. *Design & Analysis of Computer Algorithms*. Pearson Education India, 1974.

[3] Giuseppe Ateniese, Randal Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary Peterson, and Dawn Song. Provable data possession at untrusted stores. In *CCS'07*. ACM, 2007.

[4] Giuseppe Ateniese, Roberto Di Pietro, Luigi V Mancini, and Gene Tsudik. Scalable and efficient provable data possession. In *SecureComm*, page 9. ACM, 2008.

[5] Giuseppe Ateniese, Seny Kamara, and Jonathan Katz. Proofs of storage from homomorphic identification protocols. In *ASIACRYPT*, pages 319–333. Springer, 2009.

[6] Ayad Barsoum and Anwar Hasan. Enabling dynamic data and indirect mutual trust for cloud computing storage systems. *IEEE Parallel and Distributed Systems*, 24(12), 2013.

[7] Rudolf Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta informatica*, 1:290–306, 1972.

[8] J. Benaloh and M. De Mare. One-way accumulators: A decentralized alternative to digital signatures. In *EUROCRYPT'93*, pages 274–285. Springer, 1994.

[9] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *ASIACRYPT*, pages 514–532. Springer, 2001.

[10] K.D. Bowers, A. Juels, and A. Oprea. Hail: A high-availability and integrity layer for cloud storage. In *CCS'09*, pages 187–198. ACM, 2009.

[11] David Cash, Alptekin Küpçü, and Daniel Wichs. Dynamic proofs of retrievability via oblivious ram. In *EUROCRYPT'13*, pages 279–295. Springer, 2013.

[12] David Cash, Alptekin Küpçü, and Daniel Wichs. Dynamic proofs of retrievability via oblivious ram. *Journal of Cryptology*, pages 1–26, 2015.

[13] Nishanth Chandran, Bhavana Kanukurthi, and Rafail Ostrovsky. Locally updatable and locally decodable codes. In *Theory of Cryptography*, pages 489–514. Springer, 2014.

[14] Lanxiang Chen. Using algebraic signatures to check data possession in cloud storage. *Future Generation Computer Systems*, 29(7):1709–1715, 2013.

[15] Reza Curtmola, Osama Khan, and Randal Burns. Robust remote data checking. In *4th ACM international workshop on Storage security and survivability*, pages 63–68. ACM, 2008.

[16] Yevgeniy Dodis, Salil Vadhan, and Daniel Wichs. Proofs of retrievability via hardness amplification. In *TCC*, pages 109–127. Springer, 2009.

[17] C. Chris Erway, Alptekin Kupcu, Charalampos Papamanthou, and Roberto Tamassia. Dynamic provable data possession. Cryptology ePrint Archive, Report 2008/432, 2008.

[18] Chris Erway, Alptekin Küpçü, Charalampos Papamanthou, and Roberto Tamassia. Dynamic provable data possession. *ACM Trans. on Information and System Security*, 17(4), 2015.

[19] Ertem Esiner, Adilet Kachkeev, A Küpçü, and Ö Özkasap. Flexlist: optimized skip list for secure cloud storage. Technical report, Technical Report, Koç University, 2013.

[20] Ertem Esiner, Alptekin Küpçü, and Öznur Özkasap. Analysis and optimization on flexdpdp: A practical solution for dynamic provable data possession. *Intelligent Cloud Computing (ICC14)*, 2014.

[21] Mohammad Etemad and Alptekin Küpçü. Transparent, distributed, and replicated dynamic provable data possession. In *Applied Cryptography and Network Security*, pages 1–18. Springer, 2013.

[22] Mohammad Etemad and Alptekin Küpçü. Database outsourcing with hierarchical authenticated data structures. In *Information Security and Cryptology–ICISC 2013*, pages 381–399. Springer, 2014.

[23] Mohammad Etemad and Alptekin Küpçü. Generic efficient dynamic proofs of retrievability. Cryptology ePrint Archive, Report 2015/880, 2015. `http://eprint.iacr.org/`.

[24] Michael Goodrich, Roberto Tamassia, and Jasminka Hasić. An efficient dynamic and distributed cryptographic accumulator. *Information Security*, pages 372–388, 2002.

[25] M.T. Goodrich, R. Tamassia, and A. Schwerin. Implementation of an authenticated dictionary with skip lists and commutative hashing. 2:68–82, 2001.

[26] M.T. Goodrich, R. Tamassia, and N. Triandopoulos. Efficient authenticated data structures for graph connectivity and geometric search problems. *Algorithmica*, 60(3):505–552, 2011.

[27] Leo J Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science*, pages 8–21. IEEE, 1978.

[28] Christian Hanser and Daniel Slamanig. Efficient simultaneous privately and publicly verifiable robust provable data possession from elliptic curves. *ePrint Archive*, 2013:392, 2013.

[29] Ari Juels and Burton S. Kaliski, Jr. Pors: proofs of retrievability for large files. In *CCS'07*, pages 584–597, New York, NY, USA, 2007. ACM.

[30] Ralph Merkle. A certified digital signature. In *CRYPTO'89*, pages 218–238. Springer, 1990.

[31] Moni Naor and Kobbi Nissim. Certificate revocation and certificate update. *Selected Areas in Communications, IEEE Journal on*, 18(4):561–570, 2000.

[32] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Authenticated hash tables. In *CCS'08*, pages 437–448. ACM, 2008.

[33] Charalampos Papamanthou and Roberto Tamassia. Time and space efficient algorithms for two-party authenticated datastructures. *Information and Communications Security*, pages 1–15, 2007.

[34] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[35] Hovav Shacham and Brent Waters. Compact proofs of retrievability. *Journal of cryptology*, 26(3), 2013.

[36] Mehul A Shah, Mary Baker, Jeffrey C Mogul, Ram Swaminathan, et al. Auditing to keep online storage services honest. In *HotOS*, 2007.

[37] Mehul A Shah, Ram Swaminathan, and Mary Baker. Privacy-preserving audit and extraction of digital contents. *IACR Cryptology ePrint Archive*, 2008:186, 2008.

[38] Elaine Shi, Emil Stefanov, and Charalampos Papamanthou. Practical dynamic proofs of retrievability. In *ACM CCS*, pages 325–336. ACM, 2013.

[39] Emil Stefanov, Marten van Dijk, Ari Juels, and Alina Oprea. Iris: A scalable cloud file system with efficient integrity checks. In *ACSAC*, pages 229–238. ACM, 2012.

[40] R. Tamassia. Authenticated data structures. *Algorithms-ESA 2003*, pages 2–5, 2003.

[41] Roberto Tamassia and Nikos Triandopoulos. On the cost of authenticated data structures. Technical report, Center for Geometric Computing, Brown University, 2003.

[42] Cong Wang, Qian Wang, Kui Ren, Ning Cao, and Wenjing Lou. Toward secure and dependable storage services in cloud computing. *IEEE Transactions on Services Computing*, 5(2):220–232, 2012.

[43] Cong Wang, Qian Wang, Kui Ren, and Wenjing Lou. Privacy-preserving public auditing for data storage security in cloud computing. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9. IEEE, 2010.

[44] Huaqun Wang. Proxy provable data possession in public clouds. *IEEE Trans. on Services Computing*, 6:551–559, 2013.

[45] Qian Wang, Cong Wang, Jin Li, Kui Ren, and Wenjing Lou. Enabling public verifiability and data dynamics for storage security in cloud computing. In *Computer Security–ESORICS 2009*, pages 355–370. Springer, 2009.

[46] Mark Allen Weiss. *Data structures and problem solving using Java*. Pearson Education Inc., 2009.

[47] Yihua Zhang and Marina Blanton. Efficient dynamic provable possession of remote data via balanced update trees. In *ACM SIGSAC*, pages 183–194. ACM, 2013.

[48] Yan Zhu, Hongxin Hu, Gail-Joon Ahn, and Mengyang Yu. Cooperative provable data possession for integrity verification in multicloud storage. *Parallel and Distributed Systems, IEEE Transactions on*, 23(12):2231–2244, 2012.