Dynamic Provable Data Possession*

Chris Erway, Alptekin Küpçü, Charalampos Papamanthou, Roberto Tamassia
Brown University, Providence, RI, USA
{cce,kupcu,cpap,rt}@cs.brown.edu

Abstract

As online storage-outsourcing services (*e.g.*, Amazon's S3) and resource-sharing networks (*e.g.*, peer-to-peer and grid networks) became popular, the problem of efficiently proving the integrity of data stored at untrusted servers has received increased attention. Ateniese *et al.* [2] formalized this problem with a model called *provable data possession* (PDP). In this model, data is preprocessed by the client and then sent to an untrusted server for storage. The client can then challenge the server to prove that the data has not been tampered with or deleted (without sending the actual data).

However, their PDP scheme applies only to static (or append-only) files. In reality, many outsourced storage applications (including network file systems and outsourced databases) need to handle dynamic data. This paper presents a definitional framework and an efficient construction for *dynamic provable data possession* (DPDP), which extends the PDP model to support provable *updates* to stored data (modifications to a block or insertion/deletion of a block). To achieve efficient DPDP, we use a new version of authenticated dictionaries based on rank information. The price of dynamic updates is a performance change from $O(1)$ to $O(\log n)$, for a file consisting of n blocks, while maintaining the same probability of detection. Yet, our experiments show that this price is very low in practice, and hence our system is applicable to real scenarios. Our contributions can be summarized as defining the DPDP framework formally, and constructing the first fully dynamic PDP solution, which performs verification without downloading actual data and is very efficient. We also show how our DPDP scheme can be extended to construct complete file systems and version control systems (*e.g.*, CVS) at untrusted servers, so that it can be used in complex outsourcing applications.

1 Introduction

Consider online storage-outsourcing services (*e.g.*, Amazon's S3), outsourced database services [13], peer-to-peer storage [11, 15] and network file systems [10, 12]. The common concern in all these systems is the fact that the server (or peer) who stores the client's data is not necessarily trusted. Therefore, users would like to check if their data has been tampered with or deleted. However, outsourcing the storage of very large files (or whole file systems) to remote servers presents an additional constraint: the client should not be required to download all stored data in order to validate it. This can be prohibitive in terms of bandwidth and time, especially if the client wishes to perform this check frequently.

Ateniese *et al.* [2] formalized this problem with a model called *provable data possession* (PDP). In this model, data (often represented as a file F) is preprocessed by the client, producing metadata that is used for verification purposes. The file and metadata is then sent to an untrusted server for storage, and the client may delete its copy of the file. The client keeps some (possibly secret) information himself to check server's responses later. The server proves the data has not been

*Research supported in part by the National Science Foundation under grants IIS–0713403 and OCI–0724806.

tampered with by responding to challenges sent by the client. The authors present several variations of their scheme under different cryptographic assumptions. These schemes provide probabilistic guarantees of data possession, where the client checks a random subset of stored data blocks with each challenge.

However, the PDP model and schemes presented before [2, 9, 22] apply only to the case of static, archival storage (*i.e.* a file that is outsourced and never changes). While this model fits some application scenarios (*e.g.*, libraries and scientific datasets), it is crucial to consider a dynamic case, where the users of untrusted storage also wish to modify their outsourced data—by inserting, modifying, or deleting stored blocks or files—while maintaining data possession guarantees. A dynamic PDP scheme could be used to provide real applications we are considering in this paper (outsourced online storage (*e.g.*, Amazon's S3), outsourced database services [13], peer-to-peer storage [11, 15] and network file systems [10, 12]).

Being aware of the importance of supporting dynamic security solutions for data possession applications, in this paper we provide a definitional framework and an efficient construction for *dynamic provable data possession* (DPDP), which extends the PDP model to support provable *updates* on the stored data. Given a file F consisting of n blocks, we define an update as either insertion of a new block (anywhere in the file, not only append), or modification of an existing block, or deletion of any block. Therefore our update operation describes the most general form of modifications a client may wish to perform on a file.

The solution we provide for DPDP is based on a new variant of authenticated dictionaries, based on [18], where we use *rank* information to organize dictionary entries, rather than using search keys. Thus we are able to support efficient authenticated operations on files at the block level, enabling file operations such as authenticated insert and delete. We prove the security of these updates using collision-resistant hash functions. We also show how to extend these data structures to support authenticating file system directory structure as well as file data itself, and show how our work can be extended to represent a whole file system. To the best of our knowledge, this is the first construction of a provable storage system that enables efficient proofs of a whole file system, enabling verification at different levels for different users (*e.g.*, every user can verify her own home directory).

The main contributions of this work can be summarized as follows:

- We provide a general definitional framework for *dynamic provable data possession* (DPDP) for the first time.
- To the best of our knowledge, we provide the first *fully dynamic* PDP solution, which performs verification without downloading actual data and is very efficient (with $O(\log n)$ communication, server computation and client computation, and $O(1)$ client storage requirements).
- We develop a new version of authenticated dictionaries, implemented with authenticated skip lists [18] based on rank information, rather than search keys, so that we can support efficient operations on data blocks. This contribution might be of independent interest.
- We show how our construction can be used in many realistic scenarios such as implementing provable outsourced file systems and versioning systems, and provide evaluation on the practicality of our scheme. Our evaluations suggest that using our schemes, the price of dynamism is very low in practice.

In order to compare and assess the efficiency of our work, we use the four main complexity measures from the PDP model [2] (n is the number of the blocks of the stored file):

1. *Server Computation*: This is the time needed by the server to process an update or to compute a proof for a certain block (or a set of blocks). In our solution, server computation is $O(\log n)$.

2. *Client Computation*: This is the time needed by the client to verify an answer to his query (*i.e.* to process the proof returned by the server). This time is also $O(\log n)$ in our construction.
3. *Communication Complexity*: This is the size of the proof returned by the untrusted server to the client, which for our solution is $O(\log n)$.
4. *Client Storage*: This is the amount of data that the client needs to keep to perform verification. In our solution, it is $O(1)$.

As said before and discussed in Section 7, our evaluations suggest that using our schemes, the price of dynamism is very low in practice.

Related work. Ateniese *et al.* [2] present an overview of previous work on protocols fitting their model, but find these approaches lacking; either they require expensive server computation or communication over the entire file [7, 17], linear storage complexity for the client [21], or do not provide secure guarantees for data possession [20].[1]

The PDP scheme of [2] provides an optimal protocol for the static case that achieves $O(1)$ server/client computation, $O(1)$ communication, and $O(1)$ client storage. Note that we have a change in performance, from $O(1)$ to $O(\log n)$. This is due to the fact that we support updates. It is an open problem to provide a fully capable DPDP with complexities better than $O(\log n)$. Nevertheless, our scheme achieves optimal client storage by requiring $O(1)$ space.

Juels and Kaliski [9] present proofs of retrievability (PORs) and, like the PDP model, focus on static archival storage of large files. Their scheme's effectiveness rests largely on preprocessing steps the client conducts before sending a file F to the server: "sentinel" blocks are randomly inserted to detect corruption, F is encrypted to hide these sentinels, and error-correcting codes are used to recover from corruption. As expected, the error-correcting codes improve the error-resiliency of their system. Unfortunately, these operations prevent any efficient extension to support updates, beyond simply replacing F with a new file F'. Furthermore, the number of queries a client can perform is limited, and fixed a priori. Shacham and Waters have an improved version of this protocol called Compact POR [22], but their solution also only supports static files.

In our solution, we regard error-correcting codes or encryption as external to our system. If the user wants to have more error-resiliency, she can provide us with a file that has error-correcting codes integrated. Furthermore, encrypted files can be provided to our constructions if the secrecy of the file is desired. All such modifications to the file are regarded as external to our system and are treated equally. Since in our constructions *we do not modify the file or assume any property on the file given*, our system will work in perfect compliance.

Simultaneously with our work, Ateniese *et al.* have developed a dynamic PDP solution [3]. Their idea is to come up with all future challenges during setup, and store pre-computed answers as metadata (at the client, or at the server in an authenticated and encrypted manner). Because of this, the number of updates and challenges a client can perform is limited and fixed a priori. Moreover, *it is not fully dynamic*: it cannot perform block insertions anywhere (only append-type insertions are possible). Under these limitations, they provide a protocol with optimal asymptotic complexity ($O(1)$ in all complexity measures giving the same probabilistic guarantees as our scheme). Yet, their work is in the random oracle model whereas our scheme is provably secure in the standard model. We also formally define a DPDP framework, and prove security accordingly.

The probability guarantee in [3] is fixed during setup since all challenges are prepared a priori. In our scheme, the client can decide on the probability guarantee for each challenge independently and on the fly. Unfortunately, *each update in their scheme requires re-creating all the remaining*

[1] The E-PDP construction in [2] does not provide data possession guarantees under cryptographic assumptions either, but the argument on its security relies on the assumption that the server does not have sufficient storage to let him answer challenges correctly without the data itself.

Scheme	SC	CC	Comm	CS	model	capabilities
PDP [2]	$O(1)$	$O(1)$	$O(1)$	$O(1)$	random oracle	append only
Scalable PDP [3]	$O(1)$	$O(1)$	$O(1)$	$O(1)$	random oracle	limited updates
Our work	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(1)$	standard	unlimited updates

Table 1: Comparison chart for state-of-the-art PDP schemes. SC denotes server computation, CC denotes client computation, Comm denotes communication overhead, and CS denotes client storage, as our four performance measures. The comparison of capabilities of these schemes in detail can be found in the related work section or in Section 7.

challenges. This can be a big problem especially when the whole file system is outsourced. Furthermore, our scheme can easily be applied to situations where blocks can be of variable size. This is a very useful feature for versioning systems, which we discuss in Section 6. Lastly, due to the use of symmetric key cryptography, their work is not suitable for public verification, as the authors point out. We have also developed an efficient public verifiability protocol (usable for official arbitration purposes), but in this paper we will focus only on supporting dynamic updates with our DPDP protocol. For more comparison, see Section 7.

Our work is also related to memory checking. Memory checking focuses on ensuring that the contents of memory are kept intact, and the problem has many similarities to our DPDP model. Unlike DPDP, there has been much theoretical work on memory checking, with lower bounds presented on performance [6, 16]. Clarke *et al.* [5] show that by using hashing, one can do no better than $O(\log n)$ for online memory checking. Similar bounds for many different primitives like hash functions, pseudo-random functions, and universal one-way hash functions have been shown by Blum *et al.* [4]. These results suggest that our DPDP scheme's performance may be the best possible. We admit that there is no proof of this, and a novel idea using a different primitive might perform better. Nevertheless, our construction can also be regarded as an optimal memory checking construction when the "file" is the memory.

2 Model

First we must define a DPDP scheme properly. We will build on the PDP definitions from [2, 9]. Unfortunately, those definitions are not general enough to cover as broad a range of even PDP protocols as we do. We start by defining a general DPDP scheme, and then show how the definition can be restricted to give a PDP scheme which is more general than the previous definitions.

Definition 1 (DPDP Scheme) *In a DPDP scheme, there are two parties. The* ***client*** *wants to off-load her files to the untrusted* ***server****. A complete definition of a DPDP scheme should describe the following (possibly randomized) efficient procedures for the client and the server, possibly with the order in which they should be executed.*

- $\mathsf{KeyGen}(1^k) \rightarrow \{\mathsf{sk}, \mathsf{pk}\}$ *is a probabilistic algorithm run by the* ***client****. It takes as input a security parameter, and outputs a secret key* sk *and a public key* pk*. The client stores the secret and public keys, and sends the public key to the server.*
- $\mathsf{PrepareUpdate}(\mathsf{sk}, \mathsf{pk}, F, \mathsf{info}, M_c) \rightarrow \{F', \mathsf{info}', M'\}$ *is an algorithm run by the* ***client*** *to prepare (a part of) the file for untrusted storage. As input, it takes secret and public keys, (a part of) the file* F *with the definition* info *of the update to be performed (*e.g.*, full re-write, modify block* i*, delete block* i*, add block* b *after block* i*, etc.), and the previous metadata* M_c*. The output is an "encoded" version of (a part of) the file* F' *(*e.g.*, by adding randomness, adding sentinels,*

encrypting for confidentiality, etc.), along with the information info' *about the update (changed to fit the encoded version), and the new metadata* M'*. The client sends* F', info', M' *to the server.*

- $\mathsf{PerformUpdate}(\mathsf{pk}, F_{i-1}, M_{i-1}, \mathsf{info}', F', M') \rightarrow \{F_i, M_i, M_c{}', P_{M_c'}\}$ *is run by the* ***server*** *in response to an update request from the client. The input contains the public key* pk*, the previous version of the file* F_{i-1}*, the metadata* M_{i-1} *and the client-provided values* F', info', M'*. The output is the new version of the file* F_i *and the metadata* M_i*, along with the metadata to be sent to the client* $M_c{}'$ *and its proof* $P_{M_c'}$*. The server sends* $M_c{}', P_{M_c'}$ *to the client.*
- $\mathsf{VerifyUpdate}(\mathsf{sk}, \mathsf{pk}, F, \mathsf{info}, M_c, M_c{}', P_{M_c'}) \rightarrow \{\mathtt{ACCEPT}, \mathtt{REJECT}\}$ *is run by the* ***client*** *to verify the server's behavior during the update. It takes all the inputs the* $\mathsf{PrepareUpdate}$ *algorithm did, plus the* $M_c{}', P_{M_c'}$ *sent by the server. It outputs acceptance or rejection signals. If the verification accepts, the client may delete* F*.*
- $\mathsf{Challenge}(\mathsf{sk}, \mathsf{pk}, M_c) \rightarrow \{c\}$ *is a probabilistic procedure run by the* ***client*** *to create a challenge for the server. It takes the secret and public keys, along with the latest client metadata* M_c *as input, and outputs a challenge* c *that is then sent to the server.*
- $\mathsf{Prove}(\mathsf{pk}, F_i, M_i, c) \rightarrow \{P\}$ *is the procedure run by the* ***server*** *upon receipt of a challenge from the client. It takes as input the public key, the latest version of the file and the metadata, and the challenge* c*. It outputs a proof* P *that is sent to the client.*
- $\mathsf{Verify}(\mathsf{sk}, \mathsf{pk}, M_c, c, P) \rightarrow \{\mathtt{ACCEPT}, \mathtt{REJECT}\}$ *is the procedure run by the* ***client*** *upon receipt of the proof* P *from the server. It takes as input the secret and public keys, the client metadata* M_c*, the challenge* c*, and the proof* P *sent by the server. It outputting accept ideally means that the server still has the file intact. We will define the security requirement of a DPDP scheme later.*

We assume there is a hidden input and output *clientstate* in all functions run by the client, and *serverstate* in all functions run by the server. Some inputs and outputs may be empty in some schemes. For example, the PDP scheme of [2] does not store any metadata at the client side. Also sk, pk can be used for storing multiple files, possibly on different servers. All these functions can be assumed to take some public parameters as an extra input if operating in the public parameters model, although our construction does not require such modifications. Besides outputting $\{\mathtt{ACCEPT}, \mathtt{REJECT}\}$, $\mathsf{VerifyUpdate}$ can also output a new client metadata M_c. In most of the scenarios, we imagine that this new metadata will be set as $M_c = M_c{}'$.

Retrieval of a (part of a) file is very similar to the challenge-response protocol above, composed of $\mathsf{Challenge}, \mathsf{Verify}, \mathsf{Prove}$ algorithms, except that along with the proof, the server also sends the requested (part of the) file, and the verification algorithm must use this (part of the) file in the verification process. Otherwise, even though the server proves the possession of the file, it may not necessarily send the client's file back. We strongly believe that a retrieval protocol must be part of any DPDP (or PDP) system specification, but don't define it formally, as it is very similar to the challenge-response protocol.

Definition 2 (PDP Scheme) *A PDP scheme is a restricted DPDP scheme, with the restriction that the algorithms* $\mathsf{PrepareUpdate}, \mathsf{PerformUpdate}, \mathsf{VerifyUpdate}$ *can only specify an update that is a full re-write (or append).*

As defined above, PDP is a restricted case of DPDP. We now show how our DPDP definition (when restricted in this way) fits some previous schemes. The PDP scheme of [2] has the same KeyGen algorithm definition, defines a restricted version of $\mathsf{PrepareUpdate}$ that can create the metadata for only one block at a time, and defines Prove and Verify algorithms similar to our definition. It lacks an explicit definition of $\mathsf{Challenge}$. The POR scheme of [9] defines KeyGen which does not return any public key, their *encode* algorithm is the PDP version of our $\mathsf{PrepareUpdate}$, and

they have *challenge*, *respond* and *verify* algorithms similarly defined as our $\mathsf{Challenge}, \mathsf{Prove}, \mathsf{Verify}$. In both papers, $\mathsf{PerformUpdate}$ is simply storing files (and tags), and $\mathsf{VerifyUpdate}$ always returns true (actually, nothing is sent back from the server to the client). It is clear that our definition allows a broader range of DPDP (and PDP) schemes.

Having defined what a DPDP scheme is composed of, we now define the security of such a scheme. While reading the security definition, note that the restriction to PDP case gives us a security definition for PDP schemes, compatible with previous definitions [2, 9, 22].

Definition 3 (Security of DPDP) *We say that a DPDP scheme is secure if for any probabilistic polynomial time (PPT) adversary who can win the following data possession game with non-negligible probability, there exists an extractor that can extract the challenged data by resetting and challenging the adversary polynomially many times.*

DATA POSSESSION GAME*: Played between the challenger who plays the role of the client and the adversary who acts as a server.*

1. KEYGEN*: The challenger runs* $\mathsf{KeyGen}(1^k) \rightarrow \{\mathsf{sk}, \mathsf{pk}\}$ *and sends the public key* pk *to the adversary.*
2. ACF QUERIES*: The adversary is very powerful. He can mount adaptive chosen file (ACF) queries as follows. The adversary specifies a message F and the related information* info *specifying what kind of update to perform (see Definition 1) and sends these to the challenger. The challenger runs* $\mathsf{PrepareUpdate}$ *on these inputs and sends the resulting* F', info', M' *to the adversary. Then the adversary replies with* $M_c{}', P_{M_c{}'}$ *which are verified by the challenger using the algorithm* $\mathsf{VerifyUpdate}$*. The result of the verification is told to the adversary. The adversary can repeat the interaction defined above polynomially many times.*
3. SETUP*: Finally, the adversary decides on messages F_i^* and related information* info_i^* *for all $i = 1, \ldots, N$ of adversary's choice of polynomially large $N \geq 1$. The ACF interaction is performed again, with the following rules. The first* info_1^* *must specify a full re-write (this corresponds to the first time the client sends a file to the server). Also, all the tags and proofs* $M_{c_i}^{*\prime}, P_{M_{c_i}^{*\prime}}$ *sent by the adversary must be accepted by the challenger's verification algorithm* Verify*. This is normal, because the client will not delete (parts of) her file if the server did not perform the update correctly.*
4. CHALLENGE*: Call the final version of the file F, which is created according to all the updates the adversary requested in the previous step. The challenger holds the latest metadata $M_c = M_{c_N}^{*\prime}$ sent by the adversary, which verified as accepting. Now the challenger creates a challenge using* $\mathsf{Challenge}(\mathsf{sk}, \mathsf{pk}, M_c) \rightarrow \{c\}$ *and sends it to the adversary. The adversary returns a proof P. If* $\mathsf{Verify}(\mathsf{sk}, \mathsf{pk}, M_c, c, P)$ *accepts, then the adversary wins. The challenger has the ability to reset the adversary to the beginning of the challenge phase and repeat this step polynomially many times for the purpose of extraction. Overall, the goal is to extract the challenged parts of F from the adversary's responses which are accepting.*

The PDP definition of [9] includes an extractor as part of the scheme, instead of the security definition. This results in a limited definition of an extractor, which works against only stateless adversaries (see [9]). To prevent introducing such artificial limitations, we prefer formalizing it as above (similar to [2]). Note that our definition coincides with standard extractor definitions in proofs of knowledge.

3 Authenticated Skip Lists

In order to implement our DPDP scheme, we will be using the authenticated skip list data structure which was introduced by Goodrich and Tamassia [8]. We could have used a Merkle tree instead, something that would probably make the presentation easier. Even though this would perfectly work for the static case, in the dynamic case, no exact algorithms have been presented for rebalancing a Merkle tree (basically we would need an authenticated red-black tree) and efficiently maintaining and updating authentication information, which, however, has been extensively studied for the case of the authenticated skip list [18].

The authenticated skip list is a skip list [19] (see Figure 1) with the difference that every internal node v of the skip list (which has two pointers, namely $\mathsf{right}(v)$ and $\mathsf{down}(v)$) also stores a label $f(v)$ that is a cryptographic hash and is computed using some collision resistant hash function h (*e.g.*, SHA-1 in practice) as a function of $f(\mathsf{right}(v))$ and $f(\mathsf{down}(v))$. Using this data structure, one can answer queries like "does 21 belong to the set represented with this skip list?" and also provide a proof that the given answer is correct. To be able to verify the proofs to his answers, the client must always hold the label $f(s)$ of the top leftmost node of the skip list (node w_7 in Figure 1). We call $f(s)$ the *basis* (or *root*), and it corresponds to the client's metadata in our DPDP construction ($M_c = f(s)$).In our construction, the leaves of the skip list represent the blocks of the file. When the client asks for a block, the server needs to send that block, along with a proof that the block is intact.

As we said before, we use the authenticated skip list data structure [8] to check the integrity of the file blocks. However, the updates we want to support in our DPDP scenario are insertions of a new block after the i-th block and deletion/modification of the i-th block (there is no search key in our case, in contrast with [8], which basically implements an authenticated dictionary). If we use indices of blocks as search keys in an authenticated dictionary, we have the following problem. Suppose we have a file consisting of 100 blocks $m_1, m_2, \ldots, m_{100}$ and we want to insert a block after the 40-th block. This means that the indices of all the blocks $m_{41}, m_{42}, \ldots, m_{100}$ should be incremented, and therefore an update becomes extremely inefficient. To overcome this difficulty, we define a novel hashing scheme and a way to search an authenticated skip list that is completely independent of the search keys—actually we are not going to use search keys at all.

We have a file F consisting of n blocks $m_1, m_2, \ldots, m_n$. The leaves of the skip list will contain some representation of the blocks, namely leaf i will store $\mathcal{T}(m_i)$. For now, let $\mathcal{T}(m_i) = m_i$ (we will define $\mathcal{T}(m_i)$ later). The actual block m_i will be stored somewhere in the hard drive of the untrusted server. Every internal node v of the skip list stores the size of the subtree rooted on this node, namely how many leaves of the skip list can be reached from this node, instead of storing a search key. We call this number *rank* of an internal node v and denote it with $r(v)$.

3.1 Rank-based authenticated skip lists

Suppose now we have stored n blocks $m_1, m_2, \ldots, m_n$ in our rank-based skip list. In order to find block i we use the following method. Obviously, the rank of the top leftmost node v of the skip list is n (all blocks can be reached from that node). Naturally we can define $\mathsf{low}(v) = 1$ and $\mathsf{high}(v) = n$. Let $\alpha = \mathsf{right}(v)$ and $\beta = \mathsf{down}(v)$ be the nodes that can be reached from v by following the right or the down pointer respectively. Suppose we search for block i. We can now compute the intervals $[\mathsf{low}(\alpha), \mathsf{high}(\alpha)]$ and $[\mathsf{low}(\beta), \mathsf{high}(\beta)]$ by setting $\mathsf{high}(\alpha) = \mathsf{high}(v)$, $\mathsf{low}(\alpha) = \mathsf{high}(v) - r(\alpha) + 1$, $\mathsf{high}(\beta) = \mathsf{low}(v) + r(\beta) - 1$ and $\mathsf{low}(\beta) = \mathsf{low}(v)$. If now $i \in [\mathsf{low}(\alpha), \mathsf{high}(\alpha)]$ we follow the right pointer that leads to α, else we follow the down pointer that leads to β. We continue in this way until we reach a node x that is at the zero-level with $\mathsf{low}(x) = i$. It is easy to see that there always exists such a node which corresponds to the i-th block. Finally note that we do not have to store

in the skip list nodes the numbers high and low. We just compute them on the fly using the stored ranks.

If we want to update a block, we traverse the search path (see Pugh [19]) for the element we want and update all the affected values while the recursion returns (*e.g.*, in case of an insertion, we increase the ranks along the traversed path and recompute the relevant hashes according to Definition 4).

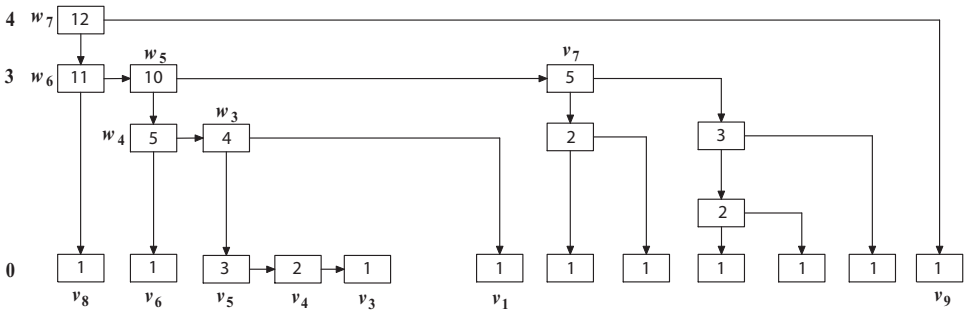


Figure 1: A skip list used to store a file of 12 blocks using ranks.

In order to extend authenticated skip lists with ranks, we must authenticate the ranks as well. Hence the hashing scheme defined in [8] can no longer be used. Let $I(v)$ denote the interval that corresponds to node v. For example, in Figure 1, we have $I(w_4) = [2, 6]$ while $I(w_3) = [3, 6]$. Let $l(v)$ be the level of node v in the skip list. Our new hashing scheme is as follows:

Definition 4 (Hashing scheme with ranks) *The label $f(v)$ of a node v in a skip list with ranks is computed as follows:*

- *if* $\mathsf{level}(v) > 0$ *then* $f(v) = h(\mathcal{A}||I(\mathsf{down}(v))||f(\mathsf{down}(v)), \mathcal{A}||I(\mathsf{right}(v))||f(\mathsf{right}(v)))$.
- *if* $\mathsf{level}(v) = 0$ *and* $\mathsf{right}(v) = \mathsf{null}$ *then* $f(v) = h(\mathcal{A}||I(v)||\mathcal{T}(\mathsf{data}(v)), \mathcal{A}||I(\mathsf{succ}(v))||\mathcal{T}(\mathsf{data}(\mathsf{succ}(v))))$.
- *if* $\mathsf{level}(v) = 0$ *and* $\mathsf{right}(v) \neq \mathsf{null}$ *then* $f(v) = h(\mathcal{A}||I(v)||\mathcal{T}(\mathsf{data}(v)), \mathcal{A}||I(\mathsf{right}(v))||f(\mathsf{right}(v)))$.

where $\mathcal{A} = l(v)||r(v)$, $||$ denotes the concatenation operator and $\mathsf{succ}(v)$ is the successor node of v.

Note that, before inserting any block (*i.e.* if initially the skip list was empty), the basis can easily be computed by hashing the sentinel values of the skip list; —the file consists of only two "fictitious" blocks— block 0 and block $+\infty$.

3.2 Queries

Suppose now the file F and a skip list on the file have been stored at the untrusted server. The client wants to verify the integrity of block i, and therefore queries for block i. The server constructs the proof $\Pi(i)$ for block i as follows (see Algorithm 3.1). Let $v_1, v_2, \ldots, v_m$ be the search path in the skip list for block i (note that node v_1 corresponds to block $i + 1$ and node v_2 corresponds to block i and therefore we are talking about the *reverse* path). For every node v_j $1 \leq j \leq m$, a 4-tuple $A(v_j)$ is added to the proof. The 4-tuple $A(v_j)$ contains the level $l(v_j)$, the rank $r(v_j)$, an interval $I(v_j)$ and a hash value (label) $f(v_j)$. For those nodes v_j that lie in the zero level, we have that the interval value is $I(v_j)$ and the hash value is $\mathcal{T}(\mathsf{data}(v_j))$. For those nodes that lie above the zero level, $I(v_j)$ equals the $I(v')$ and $f(v_j)$ equals $f(v')$, where v' is either $\mathsf{right}(v_j)$ or $\mathsf{down}(v_j)$ according from where v_j gets its hash value. For example, the proof for the 5-th block of the skip list of Figure 2 is depicted in Table 2. Finally, note that since any search path in the skip list is

expected to be of logarithmic length (in the number of blocks) with high probability, the size of the proof is logarithmic too.

Algorithm 3.1: $\Pi(i) = \mathsf{query}(i)$

1: Let $v_1, v_2, \ldots, v_m$ be the search path for block i;
2: return $\Pi(i) = \{A(v_1), A(v_2), \ldots, A(v_m)\}$;

node v	v_1	v_3	v_4	v_5	w_3	w_4	w_5	w_6	w_7
$l(v)$	0	0	0	0	2	2	3	3	4
$r(v)$	1	1	2	3	4	5	10	11	12
$I(v)$	$[6,6]$	$[5,5]$	$[4,5]$	$[3,5]$	$[6,6]$	$[2,2]$	$[7,12]$	$[1,1]$	$[12,12]$
$f(v)$	$\mathcal{T}(m_6)$	$\mathcal{T}(m_5)$	$\mathcal{T}(m_4)$	$\mathcal{T}(m_3)$	$f(v_1)$	$f(v_6)$	$f(v_7)$	$f(v_8)$	$f(v_9)$

Table 2: The proof for the 5-th block of the file F stored in the skip list of Figure 1.

3.3 Verification

After receiving the 4-tuples $A(v_j)$ which are the proof for a block m_i, the client has to process them and compute a value f'. If f' is equal to the locally stored metadata M_c, then the verification algorithm outputs `ACCEPT`, else it outputs `REJECT` (see Algorithm 3.2). If it outputs `ACCEPT`, then with high probability, the server indeed stores $\mathcal{T}(m_i)$ intact [18] (we recall that $\mathcal{T}(m_i)$ is a representation of the data of the actual block m_i —which can be viewed as m_i itself for the sake of presentation— and this is what is stored at the leaves of the skip list). In order to show how to process the given proof, we need to define hashing between 4-tuples:

Definition 5 (Hashing with ranks) *Given two 4-tuples $A(u) = (l(u), r(u), I(u), f(u))$, $A(v) = (l(v), r(v), I(v), f(v))$ that correspond to two consecutive nodes u, v of a path of the skip list such that $u = \mathsf{right}(v)$ or $u = \mathsf{down}(v)$ we define $\mathcal{S}(A(u), A(v))$ to be the 4-tuple*

$$(l(v), r(v), I(v), h(\mathcal{A}||I(v)||f(v), \mathcal{A}||I(u)||f(u)))$$

where $\mathcal{A} = l(v)||r(v)$.

Note that operation $\mathcal{S}$ is associative, namely for every three 4-tuples $A(v_i), A(v_j), A(v_k)$ such that v_i, v_j and v_k form an upward path in the skip list, it is $\mathcal{S}(A(v_i), A(v_j), A(v_k)) = \mathcal{S}(\mathcal{S}(A(v_i), A(v_j)), A(v_k))$. We now have the following result:

Lemma 1 *Let $v_1, v_2, \ldots, v_m$ be a reverse search path for a node x in a skip list where the hashing scheme with ranks is used. Let L be the maximum level, n be the number of items stored in the skip list and $A(v_i)$ $(i = 1, \ldots, m)$ be the 4-tuple that corresponds to each node v_i. Then*

$$\mathcal{S}(A(v_1), A(v_2), \ldots, A(v_m)) = (L, n, [1, n], f(s))$$

where $f(s)$ is the label of the top-leftmost node.

From now on, we denote with $\lambda(A(v_1), A(v_2), \ldots, A(v_m))$ the last element of the 4-tuple $\mathcal{S}(A(v_1), A(v_2), \ldots, A(v_m))$ (the value $f(s)$ in Lemma 1). Now, in order to perform verification, the client applies $\mathcal{S}(.,.)$ sequentially to the proof $\Pi(i)$ sent by the server, according to Definitions 5 and 4 and outputs `ACCEPT` or `REJECT` according to whether $\lambda(\Pi(i)) = M_c$ or not.

Note that we must authenticate all the intervals, the ranks and the levels since they are critical in verification. Suppose we query for the i-th block. Then the server sends us a proof that consists of the 4-tuples $\{A(v_1), A(v_2), \ldots, A(v_m)\}$. As before, we denote with $l(v_i), r(v_i), I(v_i), f(v_i)$

Algorithm 3.2: $\{\texttt{ACCEPT}, \texttt{REJECT}\} = \mathsf{verify}(\Pi(i), M_c)$

1: Let $\Pi(i)$ be the proof returned by Algorithm 3.1;
2: **if** $\lambda(\Pi(i)) = M_c$ **then** return ACCEPT;
3: return REJECT;

the level, the rank, the interval and the label that corresponds to node v_i for $i = 1, \ldots, m$, respectively. Note that for v_t such that $l(v_t) = 0$ we have that $f(v_t) = \mathcal{T}(\mathsf{data}(v_t))$. We process all zero-level nodes $v_1, \ldots, v_{k'}$ until we reach a node u such that $\mathsf{low}(u) = i$. If we apply $\mathcal{S}(,.,)$ to $\{A(v_1), A(v_2), \ldots, A(v_m)\}$ and the results equals M_c, then we are assured that the data corresponding to block i is $f(u)$.

3.4 Updates

The possible update types in our DPDP scheme are insertions of a new block after the i-th block, deletions of the i-th block, and modifications of the i-th block for $1 \leq i \leq n$. Suppose the client wants to insert a new block after the i-th block. He sends an "insert" query to the server. While performing the insertion (see Algorithm 3.3), the server also computes the proof $\Pi(i)$ (using Algorithm 3.1). Then he sends the proof $\Pi(i)$ along with the new metadata $M_c{}'$ to the client ($M_c{}'$ is the new basis). Note that when the server performs an insertion or deletion, it must update (and also include in the hashing scheme) the ranks and the intervals as well (see line 5 of Algorithm 3.3). This can be easily done in $O(\log n)$ time: it is easy to see that only ranks and intervals along the search path of the skip list are affected.

Algorithm 3.3: $\{M_c', \Pi(i)\} = \mathsf{performUpdate}(i, \mathcal{T}, \mathsf{upd})$

1: set $\Pi(i) = \mathsf{query}(i)$;
2: **if** upd = ins **then** insert $\mathcal{T}$ after block i in the skip list;
3: **if** upd = del **then** delete block i in the skip list;
4: **if** upd = modify **then** set $\mathcal{T}(m_i) = \mathcal{T}$ in the skip list;
5: **forall** affected nodes v, update $A(v)$,$f(v)$ to $A'(v)$,$f'(v)$;
6: return $\{f'(s), \Pi(i)\}$; (s is the basis of the skip list)

After the client has received the proof $\Pi(i)$ and the metadata $M_c{}'$, he has to process $\Pi(i)$ and produce another updated proof $\Pi'(i)$ using Algorithm 3.4. This is done by using a variation of the algorithm update presented in [18], the presentation of which is deferred to the full version of the paper, due to space limitations.

Algorithm 3.4:
$\{\texttt{ACCEPT}, \texttt{REJECT}\} = \mathsf{verifyUpdate}(M_c', \Pi(i), i, \mathsf{upd})$

1: process $\Pi(i)$ according to [18] and upd to produce $\Pi'(i)$;
2: **if** $\lambda(\Pi(i)) = M_c'$ **then** return ACCEPT;
3: return REJECT;

To give some intuition of how Algorithm 3.4 produces proof $\Pi'(i)$, the reader can verify that Table 3 corresponds to $\Pi'(5)$, the proof that the client produces from Table 2 in order to verify the update "insert a new block with data $\mathcal{T}$ after block 5 at level 1 of the skip list of Figure 1". Note that this update causes the creation of two new nodes in the skip list, namely the node that holds

node v	v_2	v_3	v_4	v_5	w	w_3	w_4	w_5	w_6	w_7
$l(v)$	0	0	0	0	1	2	2	3	3	4
$r(v)$	1	1	2	3	4	5	6	11	12	13
$I(v)$	$[6,6]$	$[5,5]$	$[4,5]$	$[3,5]$	$[3,6]$	$[7,7]$	$[2,2]$	$[8,13]$	$[1,1]$	$[13,13]$
$f(v)$	$\mathcal{T}$	$\mathcal{T}(m_5)$	$\mathcal{T}(m_4)$	$\mathcal{T}(m_3)$	$f(v_2)$	$f(v_1)$	$f(v_6)$	$f(v_7)$	$f(v_8)$	$f(v_9)$

Table 3: The proof $\Pi'(5)$ as produced by Algorithm 3.4 for the update "insert a new block with data $\mathcal{T}$ after block 5 at level 1". Note that two new nodes are inserted, namely v_2, w, where $f(v_2) = h(0||1||[6,6]||\mathcal{T}, 0||1||[7,7]||\mathcal{T}(\mathsf{data}(v_1)))$ as defined in Definition 4 and that the ranks and the intervals along the search path are increased in order to accommodate the addition of one more block.

the data for the 6-th block, v_2, and node w (5-th line of Table 3) that needs to be inserted in the skip list at level 1.

4 DPDP scheme construction

4.1 Core construction

In this section, we present the main idea behind our DPDP construction, keeping the presentation simple. Later, in Section 4.2, we discuss techniques that can be used in practice to improve efficiency. We now present each of our algorithms as defined in Definition 1 in Section 2. In the following, n is the current number of blocks that the client's file has.

- $\mathsf{KeyGen}(1^k) \rightarrow \{\mathsf{sk}, \mathsf{pk}\}$: In our basic construction, we do not require any keys to be generated. So, this procedure's output is empty, and hence none of the other procedures make use of these keys.
- $\mathsf{PrepareUpdate}(\mathsf{sk}, \mathsf{pk}, F, \mathsf{info}, M_c) \rightarrow \{F', \mathsf{info}', M'\}$: This is a dummy procedure that outputs the file F and information info it receives as input. The metadata M_c as an input and M' as an output are empty (not used).
- $\mathsf{PerformUpdate}(\mathsf{pk}, F_{i-1}, M_{i-1}, \mathsf{info}', F', M') \rightarrow \{F_i, M_i, {M_c}', P_{{M_c}'}\}$: Inputs F_{i-1}, M_{i-1} are the previously stored file and metadata on the server (empty if this is the first run). F', info', M', which are output by $\mathsf{PrepareUpdate}$, are sent by the client (M' being empty). The file is stored as is, and the metadata stored at the server is a skip list (where for block b, $\mathcal{T}(b)$ is the block itself). The procedure updates the file according to info', outputting F_i, runs the skip list update procedure on the previous skip list M_{i-1} (or builds the skip list from scratch if this is the first run), outputs the resulting skip list as M_i, the new skip list root as ${M_c}'$, and the proof returned by the skip list update as $P_{{M_c}'}$. This corresponds to calling Algorithm 3.3 on inputs the new data $\mathcal{T}$ (in case of an insertion or a modification), a block index j and the kind of the update upd. Note that the index j and the type of the update upd is taken from info' and the new data $\mathcal{T}$ is simply F'. Finally, Algorithm 3.3 outputs ${M_c}'$ and $P_{{M_c}'} = \Pi(j)$ which are output by $\mathsf{PerformUpdate}$. The runtime is $O(\log n)$.
- $\mathsf{VerifyUpdate}(\mathsf{sk}, \mathsf{pk}, F, \mathsf{info}, M_c, {M_c}', P_{{M_c}'}) \rightarrow \{\mathtt{ACCEPT}, \mathtt{REJECT}\}$: Client metadata M_c is the previous skip list root the client has (empty for the first time), whereas ${M_c}'$ is the new root sent by the server. The client runs Algorithm 3.4 using the file F and the proof sent by the server $P_{{M_c}'}$, and compares the root output by that procedure with ${M_c}'$. If they are the same, the client sets $M_c = {M_c}'$ and accepts. The client may now delete the new block from its local storage. This procedure is a direct call of Algorithm 3.4. It runs in time $O(\log n)$.

- $\mathsf{Challenge}(\mathsf{sk}, \mathsf{pk}, M_c) \rightarrow \{c\}$: This procedure does not need any input apart from knowing the number of blocks, n, in the file on the outsourced storage. It might additionally take a security parameter C which is the number of blocks to challenge. This value will be analyzed when we talk about the evaluation of our protocol. The procedure creates C random block numbers between $1, \ldots, n$. This set of C random block numbers are sent to the server and is denoted with c. It runs in time $O(C)$. Note that C will be a constant.
- $\mathsf{Prove}(\mathsf{pk}, F_i, M_i, c) \rightarrow \{P\}$: This procedure uses the last version of the file F_i and the skip list M_i, and the challenge c sent by the client. It runs the skip list prover to create a proof on the challenged blocks. Namely, let $i_1, i_2, \ldots, i_C$ (C being the number of challenged blocks) be the indices of the challenged blocks. Prove calls Algorithm 3.1 C times (with arguments $i_1, i_2, \ldots, i_C$) and sends back C proofs. All these C skip list proofs form the output P. The runtime is $O(C \log n)$.
- $\mathsf{Verify}(\mathsf{sk}, \mathsf{pk}, M_c, c, P) \rightarrow \{\texttt{ACCEPT}, \texttt{REJECT}\}$: This procedure takes the last skip list root M_c the client has as input, as well as the challenge c sent to the server, and the proof P sent by the server. It then runs the skip list verification using the proof sent by the server, including the challenged blocks, which returns a skip list root. If this root matches M_c then the client accepts. All these are achieved by calling Algorithm 3.2 C times. This procedure takes $O(C \log n)$ time.

As presented above, the core DPDP construction does not provide blockless verification. Namely for each block b, we have $\mathcal{T}(b) = b$. In the next section, we show how to prevent downloading of the blocks by the client, and have a very efficient DPDP protocol.

4.2 Blockless verification using tags

In our construction above, we used the skip list leaves as the data blocks themselves. This requires the client to download all the challenged blocks for verification purposes, since the skip list proof includes leaves. To improve the efficiency and accomplish blockless verification, we can employ tags; we use homomorphic tags as in [2], but our tags are simpler and more efficient to compute.

We now set each leaf $\mathcal{T}(m_i)$ of our skip list to be the tag of block m_i. The tags (explained later) are small in size compared to data blocks. Therefore, making the skip list leaves to be tags have two main advantages. Firstly, the skip list can be kept in memory. Secondly, instead of downloading the data, the client can just download the tags. The integrity of the tags themselves are protected by the skip list, and the tags protect the integrity of the data. We now present modifications to our protocols to accommodate tags.

Before talking about the tags, we need to modify our KeyGen algorithm to output $\mathsf{pk} = (N, g)$, where $N = pq$ is a product of two primes, and g is an element of high order in Z_N^*. pk is sent to the server. There is no secret key.

In the skip list, tags $\mathcal{T}(m_i) = g^{m_i} \mod N$ will be used as the skip list leaves instead of the blocks. Therefore, the skip list proof will contain these tags instead of the blocks themselves. This computation can be carried out easily with the knowledge of the public key and the block. Alternatively, the server can store the tags for faster proof computation.

The Prove procedure now sends the skip list proof for the challenged blocks m_{i_j} ($1 \leq i_1, \ldots, i_C \leq n$ denote the challenged indices, where C is the number of challenged blocks, and n the total number of blocks), with the tags as leaves. The server also sends a combined block $M = \sum_{j=1}^{C} a_j m_{i_j}$, where a_j are random values sent by the client as part of the challenge. The size of this combined block is roughly the size of a single block, and thus imposes much smaller overhead than sending C blocks. This achieves blockless verification.

The Verify algorithm verifies the skip list proof as before, except now with tags as leaves. It

also computes $T = \prod_{j=1}^{C} \mathcal{T}(m_{i_j})^{a_j} \mod N$, and accepts if and only if $T = g^M \mod N$ and the skip list proof verifies.

The Challenge procedure can also be made more efficient by using the ideas in [2]. First, instead of sending random values a_j separately, the client can simply send a random key to a pseudo-random function that will generate those values. Second, a key to a pseudo-random permutation can be sent to select the challenged blocks: if $1 \leq i_j \leq n$ $(j = 1, \ldots, C)$ are pseudo-random block numbers generated using that key, the challenged blocks will be m_{i_j} for each $j = 1, \ldots, C$. Definitions of these pseudo-random families can be put into the public key. See [2] for more details on this efficient challenge procedure.

Theorem 1 *Assume the existence of a collision-resistant hash function and that the factoring assumption holds. The dynamic provable data possession scheme (DPDP) presented in Section 4.2 for a file consisting of n blocks has the following properties:*

1. *It is secure (according to Definition 3), as proven in Section 5;*
2. *The client uses $O(1)$ space;*
3. *The server uses expected $O(n)$ space w.h.p.;*
4. *The probability of detecting a tampered block when the server tampers with T random blocks is $1 - \left(\frac{n-T}{n}\right)^C$, where C is the number of the blocks probed;*
5. *The expected running time (at the server and client side) and communication complexity when updating a number of consecutive blocks is $O(\log n)$ w.h.p.;*
6. *The expected running time (at the server and client side) and communication complexity for challenging C random blocks is $O(C \log n)$ w.h.p..*

5 Security

In this section we prove the security of our DPDP scheme. From the two-party authenticated skip list (Theorem 1 in [18]), and our discussion in Section 3, we use the following result:

Lemma 2 *Assuming the existence of a collision-resistant hash function, the proofs generated using our rank-based authenticated skip list structure guarantees the integrity of its leaves $\mathcal{T}(m_i)$ with high probability.*

Theorem 2 (Security of core DPDP protocol) *The DPDP protocol presented in Section 4.1 is secure in the standard model according to Definition 3 assuming the existence of a collision-resistant hash function.*

Proof: Assume that the adversary wins the data possession game in Definition 3. Then, we show that the challenger can either extract the challenged blocks, or break the collision-resistance of the hash function used.

As input, the challenger is given a hash function. The challenger plays the data possession game with the adversary using this hash function, honestly answering every query of the adversary. As the only difference from the real game, the challenger does not remove the blocks sent by the adversary from its storage. This difference is invisible to the adversary, and so he will behave in the same manner as he would to an honest challenger. At the end, the adversary replies to the challenge sent by the challenger. If this proof verifies, and hence the adversary wins, it must be the case that either all the blocks are intact (and so the extractor can just output the blocks sent in the proof) or the challenger breaks the collision-resistance as follows. By Lemma 2, if we have a skip list proof that verifies, but at least one block that is different from the original block, the challenger can output the original block he stored and the different block sent in the proof as a

collision. Therefore, if the adversary has a non-negligible probability of winning the data possession game, the challenger can either extract or break the collision-resistance of the hash function with non-negligible probability. □

Next, we analyze our improved DPDP construction which uses tags. In this construction, we need an extra assumption stated below.

Definition 6 (Factoring Assumption) *For all PPT adversaries A and large-enough number $N = pq$ which is a product of two primes p and q, the probability that A can output p or q given N is negligible in the size of p and q.*

Theorem 3 (Security of tagged DPDP protocol) *The DPDP protocol presented in Section 4.2 is secure in the standard model according to Definition 3 assuming the existence of a collision-resistant hash function and that the factoring assumption holds.*

We would like to mention that our scheme relies on neither the RSA assumption nor the knowledge of exponent assumption.

Proof: Assume that the adversary wins the data possession game in Definition 3. Then, we show that the challenger can either extract the challenged blocks, or break the collision-resistance of the hash function used, or break the factoring assumption by interacting with the adversary.

The challenger is given a hash function and an integer $N = pq$ but not p or q. The challenger then samples a high-order element g (a random integer between 1 and N will have non-negligible probability of being of high order in Z_N^*, which suffices for the sake of reduction argument—a tighter analysis can also be performed). He interacts with the adversary in the data possession game honestly, using the given hash function, and creates the tags while using N as the modulus and g as the base. As the only difference from the real game, the challenger does not remove the blocks sent by the adversary from its storage. This difference is invisible to the adversary, and so he will behave in the same manner he would to an honest challenger. At the end, the adversary replies to the challenge sent by the challenger.

First, consider the case where only one block is challenged. If the adversary wins, and thus the proof verifies, then the challenger can either extract the block correctly, or break the factoring assumption, or break the collision-resistance of the hash function, as follows.

Call the block sent in the proof by the adversary x, and the original challenged block stored at the challenger b. If $x \neq b$ (and thus the extraction fails, because otherwise we can easily extract), then the challenger checks if $g^x = g^b \mod N$. If that is the case, the challenger can break the factoring assumption; otherwise, he breaks the collision-resistance. If $g^x = g^b \mod N$, this means $x = b \mod \phi(N)$, which means $x - b = k\phi(N)$ for some integer $k \neq 0$. Hence, $x - b$ can be used in Miller's lemma [14], which leads to factoring N.

Otherwise $g^x \neq g^b \mod N$. This means, there are two different tags that can provide a verifying skip list proof. By Lemma 2, the challenger can break the collision-resistance of the hash function by outputting $g^x \mod N$ and $g^b \mod N$.

Now consider challenging multiple blocks (C blocks). Let $i_1, i_2, \ldots, i_C$ be the C challenged indices. Recall that each block is not sent individually. Instead, the adversary sends a linear combination of blocks $M = \sum_{j=1}^{C} a_j m_{i_j}$ for random a_j sent by the challenger. We can easily plug in the extractor at the last paragraph of the proof of Theorem 4.3 in [2]. The idea is to reset and challenge with independent a_j (which means independent pseudo-random function keys) and get enough independent linear equations from the adversary to solve for each m_{i_j}. Note that the reduction to the factoring assumption is very easy, since, after computing $T = \prod_{j=1}^{C} \mathcal{T}(m_{i_j})^{a_j} \mod N$, if this value can be obtained by a different linear combination M' of blocks, it leads to factoring

using Miller's lemma [14] (we get $g^M = g^{M'} \mod N$ with $M \neq M'$). The tags for each block are sent separately, and so the reduction to collision-resistance works the same way as before. This concludes the proof of Theorem 3. □

Probability of detection. As we mentioned before, the client probes C blocks by calling the Challenge procedure. Obviously, if the server tampers with a block different than the probed ones, the server can cheat. Assume now that the server tampers with T blocks, while the client probes C blocks. We can easily prove that, if the total number of blocks is n, the probability that at least one of the probed blocks matches at least one of the tampered blocks is $1 - \left(\frac{n-T}{n}\right)^C$ (since there are $n - T$ non-tampered blocks, and choosing all C of them has probability $\left(\frac{n-T}{n}\right)^C$).

As mentioned before, error-correcting codes can be applied external to our system to increase error-resiliency of the file. We do not take into account such modifications when we consider the probability of detection. Furthermore, many usages of a DPDP system over many files/contents can tolerate small number of errors. As an example, consider movie files, music files, most (unofficial) text files, image files, *etc.* Thus, there are many real scenarios where several flipped bits will not cause real problems. More importantly, the probability of getting caught is so high that no respectable DPDP server will take the risks under proper business management.

6 Extensions and Applications

Our DPDP model as presented considers only a single large file F, but our scheme may be extended to support multiple files and directories, as in a network file system or version control system. We can also support the use of variable-sized blocks by modifying our rank mechanism, which may be useful in conserving bandwidth and reducing the number of updates necessary for some applications.

Hierarchical File System. We can extend our DPDP scheme to allow for authenticated storage and updates to multiple files existing within a directory hierarchy. The key idea is to use the root of each file's rank-based skip list (from our single-file scheme) as leaves in a parent dictionary structure which is used to map file names to files. Using an authenticated dictionary based on skip lists [18] allows us to combine and chain our proofs of possession and update operations with queries and proofs through the entire hierarchy of authenticated structures. Each directory is represented as a key-ordered skip list containing the set of files and subdirectories within it. Thus we can use skip lists in a nested manner to represent a hierarchical file system: one can think of the basis of the topmost skip list as the root of the file system, and at the bottom, leaves for the tags associated with blocks for each file (as depicted in Figure 2(a)).

This extension scheme affords certain flexibility in multi-user environments. Consider a case where a system administrator employs an untrusted storage provider. The system administrator can keep the skip list basis corresponding to the topmost directory, and use it to periodically check the integrity of the whole file system, learning only the sum $M = \sum a_i m_i$ of random blocks of users' data in the process. Each user can keep the skip list basis corresponding to her home directory, and use it to independently check the integrity of the directory hierarchy rooted at that basis, at any time and without need for cooperation from the administrator. Proofs of possession can be performed at any level of the hierarchy.

Since the basis of a skip list at level i is a leaf of another skip list at level $i-1$ in the hierarchy, one drawback to this construction is that upper levels of the hierarchy must be updated with each update to the lower levels. Still, the proof complexity stays relatively low; if n is the maximum number of leaves in each skip list, and the depth of the directory structure is d, then proofs on the whole hierarchy have size and computation time of $O(d \log n)$.

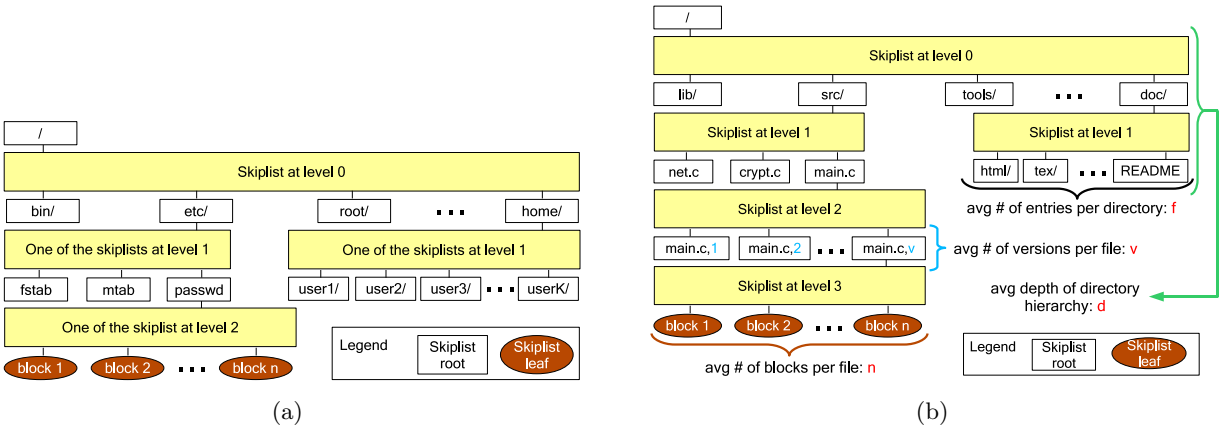


Figure 2: (a) A file system skip list with blocks as leaves, directories and files as roots of nested skip lists. (b) A version control file system. Notice the additional level of skiplists for holding versions of a file. For better performance, persistent authenticated skip lists [1] should be used for versions. The complexity of our proofs will then be $O(\log n + \log v + d \log f)$.

Variable-sized blocks. Although our DPDP scheme allows for updates that insert, modify and delete blocks within a file, most practical applications that operate above the file system layer do not perform updates that cleanly map to fixed-size block boundaries. An application that wishes to insert or delete an odd-sized chunk of data in a file (*e.g.*, adding or removing a line in a text file) would have to perform modifications to each of the blocks in the file following the change, in order to realign the new version of the file with the partition of fixed-size blocks. This renders all but block-sized updates at boundary points inefficient, since new tags and proofs must be generated for each changed block.

However, we can easily augment our ranking scheme to support updates that do not require changes to unaffected blocks. Recall that our ranking scheme assigns each internal skip list node v a rank $r(v)$ equivalent to the number of leaf nodes (blocks) that can be reached from the subtree rooted at v, while leaves are implicitly assigned a rank value of 1. We can support variable-sized blocks by defining each leaf node's rank to be equal to the size of its associated block (*i.e.* in bytes). Each internal node, in turn, is assigned a rank equivalent to the number of bytes that can be reached below it. Queries and proofs proceed the same as before, except that ranks and intervals associated with the search path refer to bytes, not blocks; update operations are phrased as, *e.g.*, "insert m bytes at byte index i". Such an insertion would require changing only the block containing the data at byte index i. Similarly, modifications and deletions affect only those blocks spanned by the range of bytes specified in the update.

Version control. We can build on these extensions to efficiently support versioning systems, such as a CVS repository. Consider a skip list for a file in the CVS repository. The server stores each version of the file, maintaining a skip list for each. The obvious way to achieve a versioning system is to keep one basis for each version of the file, which requires $O(v)$ client storage and $O(\log n)$ proof complexity, where v is the number of versions of the file.

To authenticate version numbers, each different version's basis can serve as leaves in yet another authenticated skip list, this one keyed by version number; as above, the client needs only store the topmost basis. Therefore, we can achieve a versioning system with only $O(1)$ storage at the client and $O(\log n + \log v)$ proof complexity.

As an efficiency improvement, we can use *persistent* authenticated skip lists [1] along with

our rank mechanism. These persistent data structures handle updates by adding new nodes for those affected by an update (nodes appearing along the search path), while preserving old internal nodes and roots corresponding to previous versions of the structure before each update. Therefore, instead of replicating the entire skip list for each update, the server needs to store only those nodes corresponding to blocks affected by it.

We can combine this versioning system for each file with the file system extension described above to support authenticating versions of entire directory hierarchies. If even the directories have v versions, and d is the depth of the directory hierarchy, the proof complexity for the versioning file system will be $O(d(\log n + \log v))$. See Figure 2(b) for details.

7 Performance evaluation

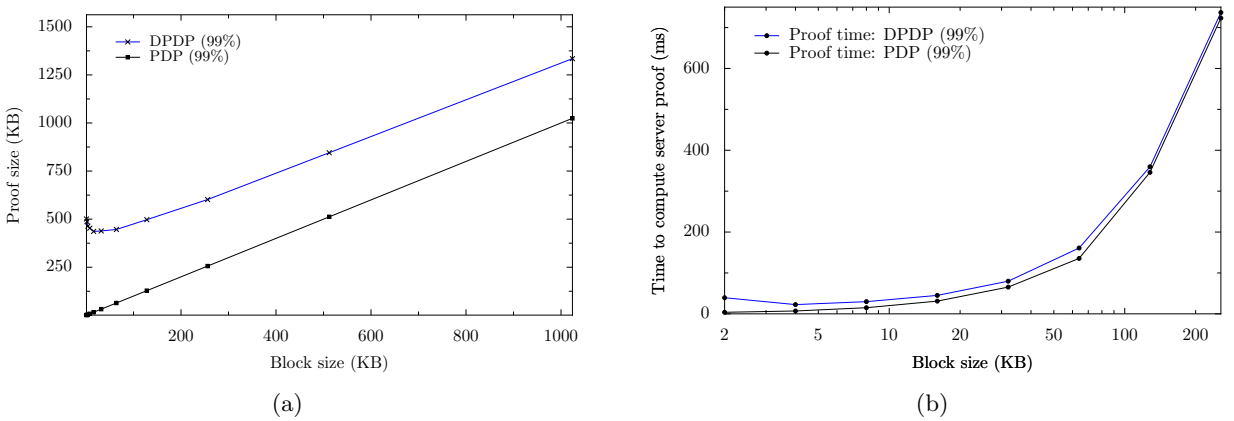
We evaluate of the performance of our DPDP scheme (as in Section 4.2) in terms of communication and computational overhead: here, our goal is to determine the *price of dynamism* over previous work on static PDP schemes. For ease of comparison, our evaluation uses the same scenario as in [2], where a server wishes to prove possession of a 1GB file. As observed in [2], the number of blocks that must be queried by a challenger to detect with some probability that a server has deleted or corrupted T blocks of the file is independent of the total number of blocks n, when T is defined as a fraction of n. Thus when $T = 1\%$ of n, detecting this fraction of incorrect data with 95% and 99% confidence requires challenging 300 and 460 blocks, respectively.

The expected size of proofs of possession under our scheme for a 1GB file under different block sizes is illustrated in Figure 3(a). Here, a DPDP proof consists of responses to 460 authenticated skip list queries, combined with a single verification block $M = \Sigma a_i m_i$ which grows linearly with the block size. The size of this block M is the same as that used by the PDP scheme in [2], and is thus represented by the line labeled PDP in Figure 3(a). The distance between this line and those for our DPDP scheme represents our communication overhead—the price of dynamism.

This overhead comes from our scheme's skip list query responses (illustrated in Table 2), each of which contains on average $1.5 \log n$ rows. Their total size decreases exponentially (but slowly) as the block size grows, providing near-constant overhead except at very small block sizes. This points to an optimal choice of block size that best minimizes total communication cost for a 1GB file: a block size of 16KB is best for both 95% and 99% confidence, resulting in a proof size of 289KB and 436KB, respectively, compared with PDP's verification block size of 16KB (for an overhead of 420KB). Selecting larger block sizes narrows the overhead of our scheme against that of PDP, reducing it to 318KB when using a 1MB block size. Nonetheless, all the proof sizes shown in Figure 3(a) are only when proving huge files (1GB).

Next we measure the computational performance of our scheme in answering challenges. Figure 3(b) presents the results of these experiments, which were performed on an AMD 2GHz Athlon X2 3800+ system with 2GB of RAM; results were averaged from 5 trials. As above, we compute the time required by our scheme for a 1GB file under varying block sizes, providing 99% confidence. As shown, our performance is dominated by computing M, even when not I/O-bound, and increases linearly with the block size. Note that the PDP scheme of [2] must also compute this M in response to challenges. Thus the computational overhead of our scheme due to dynamism—time spent traversing the skip list and building proofs—while logarithmic in the number of blocks, is extremely low in practice: even for a million 1KB blocks, 460 queries with proofs require less than 40ms to compute in total (as small as 13ms with larger blocks).

When comparing our work with the simultaneous work of Ateniese *et al.* [3], we immediately notice that their work is not suitable to use with variable sized blocks, and therefore a versioning system atop their work will be much less efficient. Furthermore, in some scenarios, their solution



(a) (b)

Figure 3: (a) Size of proofs of possession and their component pieces on a 1GB file, for 99% probability of detecting misbehavior. (b) Computation time required by the server in response to a challenge for a 1GB file, with 99% probability of detecting misbehavior.

might be less efficient than ours in practice, especially if the metadata is kept at the server. To give an idea, with each update, $2 \times 16 = 32$MB of metadata needs to be transferred between the server and the client [3] (this is much larger than our logarithmic proof size, which is a few hundred kilobytes in practice, even for very large files, as seen above). If the metadata is kept at the client, this is not a problem of communication, but of computation and storage.

7.1 Version control extension

We also assess the performance of a version control system built atop our scheme, as described in Section 6. Using public CVS repositories for the Samba and Tcl projects, we retrieved the sequence of updates from the RCS source of each file in each repository's main branch. These RCS updates come in two types: "insert m lines at line n" or "delete m lines starting at line n". For this evaluation, we consider a scenario where queries and proofs descend a search path through hierarchical skip list dictionaries corresponding to the directory structure, history of versions for each file, and finally to the source-controlled lines of each file (depicted in Figure 2(b)). For simplicity, we assume a naive scheme where each line of a file is assigned its own block; a smarter block-allocation scheme that collects contiguous lines during updates would yield fewer blocks, further reducing the overhead of our scheme.

In Table 4 we present performance characteristics of three public CVS repositories under this system. Here, "commits" refer to individual CVS checkins, each of which establish a new version and adds a new node in the version dictionary for that file. "Updates" describe the number of insertion or deletion operations required to support each commit. Total statistics sum the number of lines and kilobytes required to store all inserted lines across all versions, even after they have been removed from the file by later deletions.

We use these figures to estimate the performance of a proof of possession under this scheme: as described in Section 6, the cost of authenticating different versions of files within a directory hierarchy requires time and space complexity corresponding to the depth of the skip list hierarchy, and the width of each skip list encountered during the Prove procedure. (Further details on the complexity of these proofs are provided in Figure 2(b).) Adding an extra level of hierarchy to the chain of skip lists representing the versioned file system adds only an additive logarithmic overhead to the query proof size.

	Rsync	Samba	Tcl
Dates of activity	1996-2007	1996-2004	1998-2008
# of Files	371	1538	1757
# of Commits	11413	27534	24054
# of Updates	159027	275254	367105
Total Lines	238052	589829	1212729
Total KBytes	8331 KB	18525 KB	44585 KB
Avg. # of updates per commit	13.9	10	15.3
Avg. # of commits per file	30.7	17.9	13.7
Avg. # of entries per directory	12.8	7	19.8
Expected proof size, 99%	700 KB	1241 KB	1520 KB
Fraction of total size	8.4%	6.7%	3.4%

Table 4: CVS repository characteristics.

These figures, combined with the sizes of rank-based and regular skip list proofs as described in Section 3, provide an estimate of the average proof size required to validate a chain of skip lists which together verify an entire versioning file system. Similar to the calculations used in the first part of Section 7, this size includes the response to 460 challenge queries, along with it the size of the largest block, in order to estimate the size of the verification block M. This results in proofs of possession that require only a small fraction of that required to download the entire repository.

References

[1] A. Anagnostopoulos, M. Goodrich, and R. Tamassia. Persistent Authenticated Dictionaries and Their Applications. *ISC*, pages 379–393, 2001.

[2] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *ACM CCS*, pages 598–609, 2007. `http://eprint.iacr.org/2007/202`.

[3] G. Ateniese, R. D. Pietro, L. V. Mancini, and G. Tsudik. Scalable and efficient provable data possession. SecureComm, 2008. `http://eprint.iacr.org/2008/114`.

[4] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the Correctness of Memories. *Algorithmica*, 12(2):225–244, 1994.

[5] D. E. Clarke, S. Devadas, M. van Dijk, B. Gassend, and G. E. Suh. Incremental multiset hash functions and their application to memory integrity checking. In *ASIACRYPT*, pages 188–207, 2003.

[6] C. Dwork, M. Naor, G. N. Rothblum, and V. Vaikuntanathan. How efficient can memory checking be?, 2008. Manuscript.

[7] D. L. G. Filho and P. S. L. M. Barreto. Demonstrating data possession and uncheatable data transfer. Cryptology ePrint Archive, Report 2006/150, 2006. `http://eprint.iacr.org/2006/150`.

[8] M. T. Goodrich, R. Tamassia, and A. Schwerin. Implementation of an authenticated dictionary with skip lists and commutative hashing. In *DARPA Information Survivability Conference and Exposition II (DISCEX II)*, pages 68–82. IEEE Press, 2001.

[9] A. Juels and B. S. K. Jr. PORs: Proofs of retrievability for large files. In *ACM CCS*, pages 584–597, 2007.

[10] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. *FAST*, pages 29–42, 2003.

[11] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: an architecture for global-scale persistent storage. *SIGPLAN Not.*, 35(11):190–201, 2000.

[12] J. Li, M. Krohn, D. Mazieres, and D. Shasha. Secure Untrusted Data Repository (SUNDR). *OSDI*, pages 121–136, 2004.

[13] U. Maheshwari, R. Vingralek, and W. Shapiro. How to build a trusted database system on untrusted storage. In *OSDI'00: Proceedings of the 4th conference on Symposium on Operating System Design & Implementation*, pages 10–26, Berkeley, CA, USA, 2000. USENIX Association.

[14] G. Miller. Riemann's hypothesis and tests for primality. In *STOC*, pages 234–239, 1975.

[15] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A Read/Write Peer-to-Peer File System. *OSDI*, pages 31–44, 2002.

[16] M. Naor and G. N. Rothblum. The complexity of online memory checking. In *FOCS '05: Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science*, pages 573–584, 2005.

[17] A. Oprea, M. Reiter, and K. Yang. Space-Efficient Block Storage Integrity. *NDSS*, 2005.

[18] C. Papamanthou and R. Tamassia. Time and space efficient algorithms for two-party authenticated data structures. In *ICICS*, pages 1–15, 2007.

[19] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

[20] T. Schwarz and E. Miller. Store, Forget, and Check: Using Algebraic Signatures to Check Remotely Administered Storage. *ICDCS*, page 12, 2006.

[21] F. Sebe, A. Martinez-Balleste, Y. Deswarte, J. Domingo-Ferre, and J.-J. Quisquater. Time-bounded remote file integrity checking. Technical Report 04429, LAAS, July 2004.

[22] H. Shacham and B. Waters. Compact Proofs of Retrievability, 2008. `http://eprint.iacr.org/2008/073`.

A Skip lists

The *skip list* data structure (see Figure 4) is an efficient means for storing a set S of elements from an ordered universe. It supports the operations $\mathsf{find}(x)$ (determine whether element x is in S), $\mathsf{insert}(x)$ (insert element x in S) and $\mathsf{delete}(x)$ (remove element x from S). It stores a set S of elements in a series of linked lists $S_0, S_1, S_2, \ldots, S_t$. The base list, S_0, stores all the elements of S in order, as well as sentinels associated with the special elements $-\infty$ and $+\infty$. Each successive list S_i, for $i \geq 1$, stores a sample of the elements from S_{i-1}. To define the sample from one level to the next, we choose each element of S_{i-1} at random with probability $\frac{1}{2}$ to be in the list S_i.

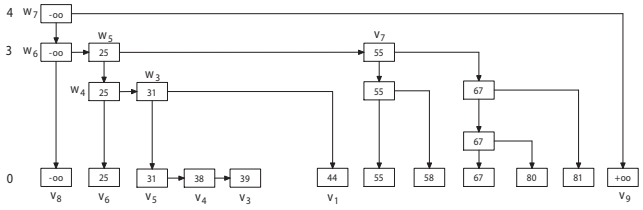


Figure 4: A skip list used to store the ordered set $\{25, 31, 38, 39, 44, 55, 58, 67, 80, 81\}$. The proof for the existence of element 39 (and for the absence of element 40) as proposed in [8] is the set $\{44, 39, 38, 31, f(v_1), f(v_6), f(v_7), f(v_8), f(v_9)\}$. The recomputation of $f(w_7)$ is performed by sequentially applying $h(\cdot, \cdot)$ to this set.