Dynamic Provable Data Possession

C. Chris Erway   Alptekin Küpçü   Charalampos Papamanthou   Roberto Tamassia

Brown University, Providence RI
{cce,kupcu,cpap,rt}@cs.brown.edu

September 21, 2009

Abstract

As storage-outsourcing services and resource-sharing networks have become popular, the problem of efficiently proving the integrity of data stored at untrusted servers has received increased attention. In the provable data possession (PDP) model, the client preprocesses the data and then sends it to an untrusted server for storage, while keeping a small amount of meta-data. The client later asks the server to prove that the stored data has not been tampered with or deleted (without downloading the actual data). However, the original PDP scheme applies only to static (or append-only) files.

We present a definitional framework and efficient constructions for dynamic provable data possession (DPDP), which extends the PDP model to support provable updates to stored data. 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 $O(\log n)$ (or $O(n^\epsilon \log n)$), for a file consisting of n blocks, while maintaining the same (or better, respectively) probability of misbehavior detection. Our experiments show that this slowdown is very low in practice (e.g., 415KB proof size and 30ms computational overhead for a 1GB file). We also show how to apply our DPDP scheme to outsourced file systems and version control systems (e.g., CVS).

1 Introduction

In cloud storage systems, the server (or peer) that 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 download all stored data in order to validate it since this may be prohibitive in terms of bandwidth and time, especially if the client performs this check frequently (therefore *authenticated data structure* solutions [32] cannot be directly applied in this scenario).

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

However, PDP and related schemes [2, 7, 12, 31] apply only to the case of static, archival storage, i.e., a file that is outsourced and never changes (simultaneously with our work, Ateniese et al. [3] present a scheme

Scheme	**Server comp.**	**Client comp.**	**Comm.**	**Model**	**Block operations**				**Probability of detection**
					append	modify	insert	delete	
PDP [2]	$O(1)$	$O(1)$	$O(1)$	RO	✓				$1-(1-f)^C$
Scalable PDP [3]	$O(1)$	$O(1)$	$O(1)$	RO	✓*	✓*		✓*	$1-(1-f)^C$
DPDP I	$O(\log n)$	$O(\log n)$	$O(\log n)$	standard	✓	✓	✓	✓	$1-(1-f)^C$
DPDP II	$O(n^\epsilon \log n)$	$O(\log n)$	$O(\log n)$	standard	✓	✓	✓	✓	$1-(1-f)^{\Omega(\log n)}$

Table 1: Comparison of PDP schemes: original PDP scheme [2]; Scalable PDP [3]; our scheme based on authenticated skip lists (DPDP I); and our scheme based on RSA trees (DPDP II). A star (*) indicates that a certain operation can be performed only a limited (pre-determined) number of times. We denote with n the number of the blocks of the file, with f the fraction of the corrupted blocks, and with C a constant, i.e., independent of n. In all constructions, the storage space is $O(1)$ at the client and $O(n)$ at the server.

with somewhat limited dynamism, which is discussed in detail in the related work section). While the static model fits some application scenarios (e.g., libraries and scientific datasets), it is crucial to consider the dynamic case, where the client updates the outsourced data—by inserting, modifying, or deleting stored blocks or files—while maintaining data possession guarantees. Such a dynamic PDP scheme is essential in practical cloud computing systems for file storage [13, 16], database services [17], and peer-to-peer storage [14, 20].

In this paper, we provide a definitional framework and efficient constructions 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.

Our DPDP solution is based on a new variant of authenticated dictionaries, where we use *rank* information to organize dictionary entries. Thus we are able to support efficient authenticated operations on files at the block level, such as authenticated insert and delete. We prove the security of our constructions using standard assumptions.

We also show how to extend our construction to support data possession guarantees of a hierarchical file system as well as file data itself. 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) and at the same time not having to download the whole data (as opposed to [10]). Our scheme yields a provable outsourced versioning system (e.g., CVS), which is evaluated in Section 8 by using traces of CVS repositories of three well-known projects.

1.1 Contributions

The main contributions of this work are summarized as follows:

1. We introduce a formal framework for *dynamic provable data possession* (DPDP);
2. We provide the first efficient *fully dynamic* PDP solution;
3. We present a rank-based authenticated dictionary built over a skip list. This construction yields a DPDP scheme with logarithmic computation and communication and the same detection probability as the original PDP scheme (DPDP I in Table 1);
4. We give an alternative construction (Section 6) of a rank-based authenticated dictionary using an RSA tree [26]. This construction results in a DPDP scheme with improved detection probability but higher server computation (see DPDP II in Table 1);

5. We present practical applications of our DPDP constructions to outsourced file systems and versioning systems (e.g., CVS, with variable block size support); **(6)** We perform an experimental evaluation of our skip list-based scheme.

Now, we outline the performance of our schemes. Denote with n the number of blocks. The *server computation*, i.e., the time taken by the server to process an update or to compute a proof for a block, is $O(\log n)$ for DPDP I and $O(n^{\epsilon} \log n)$ for DPDP II; the *client computation*, i.e., the time taken by the client to verify a proof returned by the server, is $O(\log n)$ for both schemes; the *communication complexity*, i.e., the size of the proof returned by the server to the client, is $O(\log n)$ for both schemes; the *client storage*, i.e., the size of the meta-data stored locally by the client, is $O(1)$ for both schemes; finally, the *probability of detection*, i.e., the probability of detecting server misbehavior, is $1-(1-f)^C$ for DPDP I and $1-(1-f)^{\Omega(\log n)}$ for DPDP II, for fixed logarithmic communication complexity, where f is the ratio of corrupted blocks and C is a constant, i.e., independent of n.

We observe that for DPDP I, we could use a dynamic Merkle tree (e.g., [15, 21]) instead of a skip list to achieve the same asymptotic performance. We have chosen the skip list due to its simple implementation and the fact that algorithms for updates in the two-party model (where clients can access only a logarithmic-sized portion of the data structure) have been previously studied in detail for authenticated skip lists [25] but not for Merkle trees.

1.2 Related work

The PDP scheme by Ateniese et al. [2] provides an optimal protocol for the *static* case that achieves $O(1)$ costs for all the complexity measures listed above. They review previous work on protocols fitting their model, but find these approaches lacking: either they require expensive server computation or communication over the entire file [9, 23], linear storage for the client [30], or do not provide security guarantees for data possession [29]. Note that using [2] in a dynamic scenario is insecure due to replay attacks. As observed in [8], in order to avoid replay attacks, an authenticated tree structure that incurs logarithmic costs must be employed and thus constant costs are not feasible in a dynamic scenario.

Juels and Kaliski present proofs of retrievability (PORs) [12], focusing 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 [31], but their solution is also static (see [7] for a summary of POR schemes and related trade-offs).

Simultaneously with our work, Ateniese et al. have developed a dynamic PDP solution called Scalable PDP [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 approach, the number of updates and challenges a client can perform is limited and fixed a priori. Also, one cannot perform block insertions anywhere (only append-type insertions are possible). Furthermore, each update requires re-creating all the remaining challenges, which is problematic for large files. Under these limitations (otherwise the lower bound of [8] would be violated), they provide a protocol with optimal asymptotic complexity $O(1)$ in all complexity measures giving the same probabilistic guarantees as our scheme. Lastly, their work is in the random oracle model whereas our scheme is provably secure in the standard model (see Table 1 for full comparison).

Finally, our work is closely related to memory checking, for which lower bounds are presented in [8, 22]. Specifically, in [8] it is proved that all non-adaptive and deterministic checkers have read and write query complexity summing up to $\Omega(\log n/\log\log n)$ (necessary for sublinear client storage), justifying the $O(\log n)$ cost in our scheme. Note that for schemes based on cryptographic hashing, an $\Omega(\log n)$ lower bound on the proof size has been shown [6, 33]. Related bounds for other primitives have been shown by Blum et al. [4].

2 Model

We build on the PDP definitions from [2]. We begin by introducing a general DPDP scheme and then show how the original PDP model is consistent with this definition.

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:*

- $\mathsf{KeyGen}(1^k) \rightarrow \{\mathsf{sk}, \mathsf{pk}\}$ *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;*
- $\mathsf{PrepareUpdate}(\mathsf{sk}, \mathsf{pk}, F, \mathsf{info}, M_c) \rightarrow \{\mathsf{e}(F), \mathsf{e}(\mathsf{info}), \mathsf{e}(M)\}$ *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 a block after block i, etc.), and the previous metadata M_c. The output is an "encoded" version of (a part of) the file* $\mathsf{e}(F)$ *(e.g., by adding randomness, adding sentinels, encrypting for confidentiality, etc.), along with the information* $\mathsf{e}(\mathsf{info})$ *about the update (changed to fit the encoded version), and the new metadata* $\mathsf{e}(M)$*. The client sends* $\mathsf{e}(F), \mathsf{e}(\mathsf{info}), \mathsf{e}(M)$ *to the server;*
- $\mathsf{PerformUpdate}(\mathsf{pk}, F_{i-1}, M_{i-1}, \mathsf{e}(F), \mathsf{e}(\mathsf{info}), \mathsf{e}(M)) \rightarrow \{F_i, M_i, M_c', P_{M_c'}\}$ *is an algorithm 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* $\mathsf{e}(F), \mathsf{e}(\mathsf{info}), \mathsf{e}(M)$*. Note that the values* $\mathsf{e}(F), \mathsf{e}(\mathsf{info}), \mathsf{e}(M)$ *are the values produced by* $\mathsf{PrepareUpdate}$*. 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;*
- $\mathsf{VerifyUpdate}(\mathsf{sk}, \mathsf{pk}, F, \mathsf{info}, M_c, M_c', P_{M_c'}) \rightarrow \{\mathsf{accept}, \mathsf{reject}\}$ *is run by the **client** to verify the server's behavior during the update. It takes all inputs of the* $\mathsf{PrepareUpdate}$ *algorithm,*[1] *plus the $M_c', P_{M_c'}$ sent by the server. It outputs acceptance (F can be deleted in that case) or rejection signals;*
- $\mathsf{Challenge}(\mathsf{sk}, \mathsf{pk}, M_c) \rightarrow \{c\}$ *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;*
- $\mathsf{Verify}(\mathsf{sk}, \mathsf{pk}, M_c, c, P) \rightarrow \{\mathsf{accept}, \mathsf{reject}\}$ *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. An output of accept ideally means that the server still has the file intact. We will define the security requirements of a DPDP scheme later.*

[1] However, in our model F denotes part of some encoded version of the file and not part of the actual data (though for generality purposes we do not make it explicit).

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. Apart from $\{\mathsf{accept}, \mathsf{reject}\}$, algorithm $\mathsf{VerifyUpdate}$ can also output a new client metadata M_c. In most scenarios, this new metadata will be set as $M_c = M_c'$.

Retrieval of a (part of a) file is 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. We also note that a PDP scheme is consistent with the DPDP scheme definition, with algorithms $\mathsf{PrepareUpdate}$, $\mathsf{PerformUpdate}$ and $\mathsf{VerifyUpdate}$ specifying an update that is a full re-write (or append).

As stated above, PDP is a restricted case of DPDP. The PDP scheme of [2] has the same algorithm definition for key generation, 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 $\mathsf{Challenge}$ (though one is very easy to infer). $\mathsf{PerformUpdate}$ consists of performing a full re-write or an append (so that *replay* attacks can be avoided), and $\mathsf{VerifyUpdate}$ is used accordingly, i.e., it always accepts in case of a full re-write or it is run as in DPDP in case of an append. It is clear that our definition allows a broad range of DPDP (and PDP) schemes.

We now define the security of a DPDP scheme, inspired by the security definitions of [2, 7]. Note that the restriction to the PDP scheme gives a security definition for PDP schemes compatible with the ones in [2, 3].

Definition 2 (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 (at least) the challenged parts of the file 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* $\mathsf{KeyGen}(1^k) \to \{\mathsf{sk}, \mathsf{pk}\}$ *and sends the public key* pk *to the adversary;*

2. ACF QUERIES*: The adversary is very powerful. The adversary 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* $\mathsf{e}(F), \mathsf{e}(\mathsf{info}), \mathsf{e}(M)$ *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 further request challenges, return proofs, and be told about the verification results. 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* $i = 1, \ldots, R$ *of adversary's choice of polynomially-large (in the security parameter* k*)* $R \geq 1$. *The ACF interaction is performed again, with the first* info_1^* *specifying a full re-write (this corresponds to the first time the client sends a file to the server). The challenger updates his local metadata only for the verifying updates (hence, non-verifying updates are considered not to have taken place—data has not changed);*

4. CHALLENGE*: Call the final version of the file* F*, which is created according to the verifying updates the adversary requested in the previous step. The challenger holds the latest metadata* M_c *sent by*

the adversary and verified as accepting. Now the challenger creates a challenge using the algorithm $\mathsf{Challenge}(\mathsf{sk}, \mathsf{pk}, M_c) \rightarrow \{c\}$ *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 (at least) the challenged parts of* F *from the adversary's responses which are accepting.*

Note that our definition coincides with extractor definitions in *proofs of knowledge*. For an adversary that answers a non-negligible fraction of the challenges, a polynomial-time extractor must exist. Furthermore, this definition can be applied to the POR case [7, 12, 31], in which by repeating the challenge-response process, the extractor can extract the whole file with the help of error-correcting codes. The probability of catching a cheating server is analyzed in Section 5.

Finally, if a DPDP scheme is to be truly publicly verifiable, the Verify algorithm should not make use of the secret key. Since that is the case for our construction (see Section 4), we can derive a public verifiability protocol usable for official arbitration purposes; this work is currently under development.

3 Rank-based authenticated skip lists

In order to implement our first DPDP construction, we use a modified authenticated skip list data structure [11]. This new data structure, which we call a *rank-based authenticated skip list*, is based on authenticated skip lists but indexes data in a different way. Note that we could have based the construction on any authenticated search data structure, e.g., Merkle tree [18] instead. This would perfectly work for the static case. But in the dynamic case, we would need an authenticated red-black tree, and unfortunately no algorithms have been previously presented for rebalancing a Merkle tree while efficiently maintaining and updating authentication information (except for the three-party model, e.g., [15]). Yet, such algorithms have been extensively studied for the case of the authenticated skip list data structure [25]. Before presenting the new data structure, we briefly introduce authenticated skip lists.

The authenticated skip list is a skip list [27] (see Figure 1) with the difference that every node v above the bottom level (which has two pointers, namely $\mathsf{rgt}(v)$ and $\mathsf{dwn}(v)$) 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 $f(\mathsf{rgt}(v))$ and $f(\mathsf{dwn}(v))$. 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 these 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.

We can use an authenticated skip list to check the integrity of the file blocks. However, this data structure does not support efficient verification of the indices of the blocks, which are used as query and update parameters in our DPDP scenario. The updates we want to support in our DPDP scenario are insertions of a new block after the i-th block and deletion or modification of the i-th block (there is no search key in our case, in contrast to [11], 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 new hashing scheme that takes into account rank information.

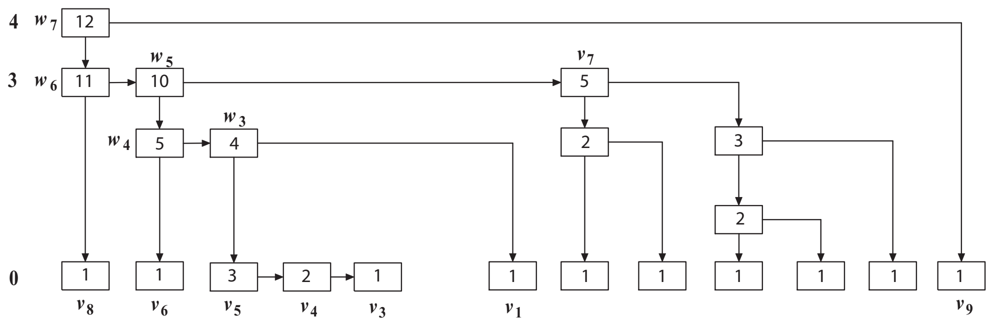


Figure 1: Example of rank-based skip list.

3.1 Authenticating ranks

Let F be a file consisting of n blocks $m_1, m_2, \ldots, m_n$. We store at the i-th bottom-level node of the skip list a representation $\mathcal{T}(m_i)$ of block m_i (we will define $\mathcal{T}(m_i)$ later). Block m_i will be stored elsewhere by the untrusted server. Each node v of the skip list stores the number of nodes at the bottom level that can be reached from v. We call this value the *rank* of v and denote it with $r(v)$. In Figure 1, we show the ranks of the nodes of a skip list. An insertion, deletion, or modification of a file block affects only the nodes of the skip list along a search path. We can recompute bottom-up the ranks of the affected nodes in constant time per node.

The top leftmost node of a skip list will be referred to as the *start node*. For example, w_7 is the start node of the skip list in Figure 1. For a node v, denote with $\mathsf{low}(v)$ and $\mathsf{high}(v)$ the indices of the leftmost and rightmost nodes at the bottom level reachable from v, respectively. Clearly, for the start node s of the skip list, we have $r(s) = n$, $\mathsf{low}(s) = 1$ and $\mathsf{high}(s) = n$be the nodes that can be reached from v by following the right or the down pointer respectively. Using the ranks stored at the nodes, we can reach the i-th node of the bottom level by traversing a path that begins at the start node, as follows. For the current node v, assume we know $\mathsf{low}(v)$ and $\mathsf{high}(v)$. Let $w = \mathsf{rgt}(v)$ and $z = \mathsf{dwn}(v)$. We set

$$\begin{aligned}
\mathsf{high}(w) &= \mathsf{high}(v)\,,\\
\mathsf{low}(w) &= \mathsf{high}(v) - r(w) + 1\,,\\
\mathsf{high}(z) &= \mathsf{low}(v) + r(z) - 1\,,\\
\mathsf{low}(z) &= \mathsf{low}(v)\,.
\end{aligned}$$

If $i \in [\mathsf{low}(w), \mathsf{high}(w)]$, we follow the right pointer and set $v = w$, else we follow the down pointer and set $v = z$. We continue until we reach the i-th bottom node. Note that we do not have to store high and low. We compute them on the fly using the ranks.

In order to authenticate skip lists with ranks, we extend the hashing scheme defined in [11]. We consider a skip list that stores data items at the bottom-level nodes. In our application, the node v associated with the i-th block m_i stores item $x(v) = \mathcal{T}(m_i)$. Let $l(v)$ be the level (height) of node v in the skip list ($l(v) = 0$ for the nodes at the bottom level).

Let $||$ denote concatenation. We extend a hash function h to support multiple arguments by defining

$$h(x_1, \ldots, x_k) = h(h(x_1)|| \ldots ||h(x_k))\,.$$

We are now ready to define our new hashing scheme:

Definition 3 (Hashing scheme with ranks) *Given a collision resistant hash function h, the label $f(v)$ of a node v of a rank-based authenticated skip list is defined as follows.*

Case 0: $v = \mathsf{null}$

$$f(v) = 0\,;$$

Case 1: $l(v) > 0$

$$f(v) = h(l(v), r(v), f(\mathsf{dwn}(v)), f(\mathsf{rgt}(v)))\,;$$

Case 2: $l(v) = 0$

$$f(v) = h(l(v), r(v), x(v), f(\mathsf{rgt}(v)))\,.$$

Before inserting any block (i.e., if initially the skip list was empty), the basis, i.e., the label $f(s)$ of the top leftmost node s of the skip list, 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 $+\infty$.

node v	v_3	v_4	v_5	w_3	w_4	w_5	w_6	w_7
$l(v)$	0	0	0	2	2	3	3	4
$q(v)$	0	1	1	1	1	5	1	1
$g(v)$	0	$\mathcal{T}(m_4)$	$\mathcal{T}(m_5)$	$f(v_1)$	$f(v_6)$	$f(v_7)$	$f(v_8)$	$f(v_9)$

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

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 issues query $\mathsf{atRank}(i)$ to the server. The server executes Algorithm 1, described below, to compute $\mathcal{T}(i)$ and a proof for $\mathcal{T}(i)$ (for convenience we use $\mathcal{T}(i)$ to denote $\mathcal{T}(m_i)$).

Let $v_k, \ldots, v_1$ be the path from the start node, v_k, to the node associated with block i, v_1. The reverse path $v_1, \ldots, v_k$ is called the *verification path* of block i. For each node v_j, $j = 1, \ldots, k$, we define boolean $d(v_j)$ and values $q(v_j)$ and $g(v_j)$ as follows, where we conventionally set $r(\mathsf{null}) = 0$:

$$d(v_j) = \begin{cases} \mathsf{rgt} & j = 1 \text{ or } j > 1 \text{ and } v_{j-1} = \mathsf{rgt}(v_j) \\ \mathsf{dwn} & j > 1 \text{ and } v_{j-1} = \mathsf{dwn}(v_j) \end{cases},$$

$$q(v_j) = \begin{cases} r(\mathsf{rgt}(v_j)) & \text{if } j = 1 \\ 1 & \text{if } j > 1 \text{ and } l(v_j) = 0 \\ r(\mathsf{dwn}(v_j)) & \text{if } j > 1,\ l(v_j) > 0 \text{ and } d(v_j) = \mathsf{rgt} \\ r(\mathsf{rgt}(v_j)) & \text{if } j > 1,\ l(v_j) > 0 \text{ and } d(v_j) = \mathsf{dwn} \end{cases},$$

$$g(v_j) = \begin{cases} f(\mathsf{rgt}(v_j)) & \text{if } j = 1 \\ x(v_j) & \text{if } j > 1 \text{ and } l(v_j) = 0 \\ f(\mathsf{dwn}(v_j)) & \text{if } j > 1,\ l(v_j) > 0 \text{ and } d(v_j) = \mathsf{rgt} \\ f(\mathsf{rgt}(v_j)) & \text{if } j > 1,\ l(v_j) > 0 \text{ and } d(v_j) = \mathsf{dwn} \end{cases}.$$

The proof for block i with data $\mathcal{T}(i)$ is the sequence $\Pi(i) = (A(v_1), \ldots, A(v_k))$ where $A(v) = (l(v), q(v), d(v), g(v))$. So the proof consists of tuples associated with the nodes of the verification path. Boolean $d(v)$ indicates whether the previous node is to the right or below v. For nodes above the bottom level, $q(v)$ and $g(v)$ are the rank and label of the successor of v that is not on the path. The proof $\Pi(5)$ for the skip list of Figure 1 is shown in Table 2. Due to the properties of skip lists, a proof has expected size $O(\log n)$ with high probability (whp).

Algorithm 1: $(\mathcal{T}, \Pi) = \mathsf{atRank}(i)$

```
1: Let $v_1, v_2, \ldots, v_k$ be the verification path for block $i$;
2: return representation $\mathcal{T}$ of block $i$ and proof $\Pi = (A(v_1), A(v_2), \ldots, A(v_k))$ for $\mathcal{T}$;
```

3.3 Verification

After receiving from the server the representation $\mathcal{T}$ of block i and a proof Π for it, the client executes Algorithm 2 to verify the proof using the stored metadata M_c.

Algorithm 2: $\{\mathsf{accept}, \mathsf{reject}\} = \mathsf{verify}(i, M_c, \mathcal{T}, \Pi)$

```
1: Let $\Pi = (A_1, \ldots, A_k)$, where $A_j = (l_j, q_j, d_j, g_j)$ for $j = 1, \ldots, k$;
2: $\lambda_0 = 0$; $\rho_0 = 1$; $\gamma_0 = \mathcal{T}$; $\xi_0 = 0$;
3: for $j = 1, \ldots, k$ do
4:     $\lambda_j = l_j$; $\rho_j = \rho_{j-1} + q_j$; $\delta_j = d_j$;
5:     if $\delta_j = \mathsf{rgt}$ then
6:         $\gamma_j = h(\lambda_j, \rho_j, \gamma_{j-1}, g_j)$;
7:         $\xi_j = \xi_{j-1}$;
8:     else $\{\delta_j = \mathsf{dwn}\}$
9:         $\gamma_j = h(\lambda_j, \rho_j, g_j, \gamma_{j-1})$;
10:        $\xi_j = \xi_{j-1} + q_j$;
11:    end if
12: end for
13: if $\gamma_k \neq M_c$ then
14:    return reject;
15: else if $\rho_k - \xi_k \neq i$ then
16:    return reject;
17: else $\{\gamma_k = M_c$ and $\rho_k - \xi_k = i\}$
18:    return accept;
19: end if
```

Algorithm 2 iteratively computes tuples $(\lambda_j, \rho_j, \delta_j, \gamma_j)$ for each node v_j on the verification path plus a sequence of integers ξ_j. If the returned block representation $\mathcal{T}$ and proof Π are correct, at each iteration of the for-loop, the algorithm computes the following values associated with a node v_j of the verification path:

- integer $\lambda_j = l(v_j)$, i.e., the level of v_j;
- integer $\rho_j = r(v_j)$, i.e., the rank of v_j;
- boolean δ_j, which indicates whether the previous node v_{j-1} is to the right or below v_j;
- hash value $\gamma_j = f(v_j)$, i.e., the label of v_j;
- integer ξ_j, which is equal to the sum of the ranks of all the nodes that are to the right of the nodes of the path seen so far, but are not on the path.

Lemma 1 *If $\mathcal{T}$ is the correct representation of block i and sequence Π of length k is the correct proof for $\mathcal{T}$, then the following properties hold for the values computed in iteration k of the for-loop of Algorithm 2:*

1. *Value ρ_k is equal to the number of nodes at the bottom level of the skip list, i.e., the number n of blocks of the file;*
2. *Value ξ_k is equal to $n - i$; and*
3. *Value γ_k is equal to the label of the start node of the skip list.*

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
$f(v)$	$\mathcal{T}$	$\mathcal{T}(m_5)$	$\mathcal{T}(m_4)$	$\mathcal{T}(m_3)$	$f(v_2)$	$f(v_1)$	$f(v_6)$	$f(v_7)$	$f(v_8)$	$f(v_9)$

Table 3: The proof $\Pi'(5)$ as produced by Algorithm 4 for the update "insert a new block with data $\mathcal{T}$ after block 5 at level 1".

3.4 Updates

The possible updates in our DPDP scheme are insertions of a new block after a given block i, deletion of a block i, and modification of a block i.

To perform an update, the client issues first query $\mathsf{atRank}(i)$ (for an insertion or modification) or $\mathsf{atRank}(i-1)$ (for a deletion), which returns the representation $\mathcal{T}$ of block i or $i-1$ and its proof Π'. Also, for an insertion, the client decides the height of the tower of the skip list associated with the new block. Next, the client verifies proof Π' and computes what would be the label of the start node of the skip list after the update, using a variation of the technique of [25]. Finally, the client asks the server to perform the update on the skip list by sending to the server the parameters of the update (for an insertion, the parameters include the tower height).

We outline in Algorithm 3 the update algorithm performed by the server (performUpdate) and in Algorithm 4 the update algorithm performed by the client (verUpdate). Input parameters $\mathcal{T}'$ and Π' of verUpdate are provided by the server, as computed by performUpdate.

Since updates affect only nodes along a verification path, these algorithms run in expected $O(\log n)$ time whp and the expected size of the proof returned by performUpdate is $O(\log n)$ whp.

Algorithm 3: $(\mathcal{T}', \Pi') = \mathsf{performUpdate}(i, \mathcal{T}, \mathsf{upd})$

```
 1: if upd is a deletion then
 2:     set $j = i - 1$;
 3: else {upd is an insertion or modification}
 4:     set $j = i$;
 5: end if
 6: set $(\mathcal{T}', \Pi') = \mathsf{atRank}(j)$;
 7: if upd is an insertion then
 8:     insert element $\mathcal{T}$ in the skip after the $i$-th element;
 9: else if upd is a modification then
10:     replace with $\mathcal{T}$ the $i$-th element of the skip list;
11: else {upd is a deletion}
12:     delete the $i$-th element of the skip list;
13: end if
14: update the labels, levels and ranks of the affected nodes;
15: return $(\mathcal{T}', \Pi')$;
```

To give some intuition of how Algorithm 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". This update causes the creation of two new nodes in the skip list, namely the node that holds 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. Note that $f(v_2) = h(0||1||\mathcal{T}, 0||1||\mathcal{T}(\mathsf{data}(v_1)))$ is computed as defined in Definition 3 and that the ranks along the search path are increased due to the addition of one more block.

Algorithm 4:
$\{\mathsf{accept}, \mathsf{reject}\} = \mathsf{verUpdate}(i, M_c, \mathcal{T}, \mathsf{upd}, \mathcal{T}', \Pi')$

```
1: if upd is a deletion then
2:     set $j = i - 1$;
3: else {upd is an insertion or modification}
4:     set $j = i$;
5: end if
6: if verify$(j, M_c, \mathcal{T}', \Pi')$ = reject then
7:     return reject;
8: else {verify$(j, M_c, \mathcal{T}', \Pi')$ = accept}
9:     from $i$, $\mathcal{T}$, $\mathcal{T}'$, and $\Pi'$, compute and store the updated label $M_c'$ of the start node;
10:    return accept;
11: end if
```

4 DPDP scheme construction

In this section, we present our DPDP I construction. First, we describe our algorithms for the procedures introduced in Definition 1. Next, we develop compact representatives for the blocks to improve efficiency (blockless verification). In the following, n is the current number of blocks of the file. The logarithmic complexity for most of the operations are due to well-known results about authenticated skip lists [11, 26]. Most of the material of this section also applies to the DPDP II scheme presented in Section 6.

4.1 Core construction

The server maintains the file and the metadata, consisting of an authenticated skip list with ranks storing the blocks. Thus, in this preliminary construction, we have $\mathcal{T}(b) = b$ for each block b. The client keeps a single hash value, called *basis*, which is the label of the start node of the skip list. We implement the DPDP algorithms as follows.

- $\mathsf{KeyGen}(1^k) \rightarrow \{\mathsf{sk}, \mathsf{pk}\}$: Our scheme does 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;

- $\mathsf{PrepareUpdate}(\mathsf{sk}, \mathsf{pk}, F, \mathsf{info}, M_c) \rightarrow \{\mathsf{e}(F), \mathsf{e}(\mathsf{info}), \mathsf{e}(M)\}$: This is a dummy procedure that outputs the file F and information info it receives as input. M_c and $\mathsf{e}(M)$ are empty (not used);

- $\mathsf{PerformUpdate}(\mathsf{pk}, F_{i-1}, M_{i-1}, \mathsf{e}(F), \mathsf{e}(\mathsf{info}), \mathsf{e}(M)) \rightarrow \{F_i, M_i, M_c', P_{M_c'}\}$: Inputs F_{i-1}, M_{i-1} are the previously stored file and metadata on the server (empty if this is the first run). $\mathsf{e}(F)$, $\mathsf{e}(\mathsf{info})$, $\mathsf{e}(M)$, which are output by $\mathsf{PrepareUpdate}$, are sent by the client ($\mathsf{e}(M)$ being empty). The procedure updates the file according to $\mathsf{e}(\mathsf{info})$, 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 basis as M_c', and the proof returned by the skip list update as $P_{M_c'}$. This corresponds to calling Algorithm 3 on inputs a block index j, the new data $\mathcal{T}$ (in case of an insertion or a modification) and the type of the update upd (all this information is included in $\mathsf{e}(\mathsf{info})$). Note that the index j and the type of the update upd is taken from $\mathsf{e}(\mathsf{info})$ and the new data $\mathcal{T}$ is $\mathsf{e}(F)$. Finally, Algorithm 3 outputs M_c' and $P_{M_c'} = \Pi(j)$, which are output by $\mathsf{PerformUpdate}$. The expected runtime is $O(\log n)$ whp;

- $\mathsf{VerifyUpdate}(\mathsf{sk}, \mathsf{pk}, F, \mathsf{info}, M_c, M_c', P_{M_c'}) \rightarrow \{\mathsf{accept}, \mathsf{reject}\}$: Client metadata M_c is the label of the start node of the previous skip list (empty for the first time), whereas M_c' is empty. The client runs Algorithm 4 using the index j of the update, M_c, previous data $\mathcal{T}$, the update type upd, the new data $\mathcal{T}'$ of the update and the proof $P_{M_c'}$ sent by the server as input (most of the inputs are included in info). If the procedure accepts, the client sets $M_c = M'_c$ (new and correct metadata has been computed). The client may now delete the new block from its local storage. This procedure is a direct call of Algorithm 4. It runs in expected time $O(\log n)$ whp;

- $\mathsf{Challenge}(\mathsf{sk}, \mathsf{pk}, M_c) \rightarrow \{c\}$: This procedure does not need any input apart from knowing the number of blocks in the file (n). It might additionally take a parameter C which is the number of blocks to challenge. The procedure creates C random block IDs between $1, \ldots, n$. This set of C random block IDs are sent to the server and is denoted with c. The runtime is $O(C)$;

- $\mathsf{Prove}(\mathsf{pk}, F_i, M_i, c) \rightarrow \{P\}$: 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 $i_1, i_2, \ldots, i_C$ be the indices of the challenged blocks. Prove calls Algorithm 1 C times (with arguments $i_1, i_2, \ldots, i_C$) and sends back C proofs. All these C proofs form the output P. The runtime is $O(C \log n)$ whp;

- $\mathsf{Verify}(\mathsf{sk}, \mathsf{pk}, M_c, c, P) \rightarrow \{\mathsf{accept}, \mathsf{reject}\}$: This function takes the last basis M_c the client has as input, the challenge c sent to the server, and the proof P received from the server. It then runs Algorithm 2 using as inputs the indices in c, the metadata M_c, the data $\mathcal{T}$ and the proof sent by the server (note that $\mathcal{T}$ and the proof are contained in P). This outputs a new basis. If this basis matches M_c then the client accepts. Since this is performed for all the indices in c, this procedure takes $O(C \log n)$ expected time whp.

The above construction requires the client to download all the challenged blocks for the verification. A more efficient method for representing blocks is discussed in the next section.

4.2 Blockless verification

We can improve the efficiency of the core construction by employing homomorphic tags, as in [2]. However, the tags described here are simpler and more efficient to compute. Note that it is possible to use other homomorphic tags like BLS signatures [5] as in Compact POR [31].

We represent a block b with its *tag* $\mathcal{T}(b)$. Tags are small in size compared to data blocks, which provides two main advantages. First, the skip list can be kept in memory. Second, instead of downloading the blocks, the client can just download the tags. The integrity of the tags themselves is protected by the skip list, while the tags protect the integrity of the blocks.

In order to use tags, we 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 $\mathbb{Z}_N^*$. The public key pk is sent to the server; there is no secret key.

The tag $\mathcal{T}(b)$ of a block b is defined by

$$\mathcal{T}(b) = g^b \mod N\,.$$

The skip list now stores the tags of the blocks at the bottom-level nodes. Therefore, the proofs provided by the server certify the tags instead of the blocks themselves. Note that instead of storing the tags explicitly, the server can alternatively compute them as needed from the public key and the blocks.

The Prove procedure computes a proof for the tags of 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 is the total number of blocks). 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. Thus, we have a much smaller overhead than for sending C blocks. Also, the Verify algorithm computes the value

$$T = \prod_{j=1}^{C} \mathcal{T}(m_{i_j})^{a_j} \mod N\,,$$

and accepts 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 indices of the challenged blocks $1 \leq i_j \leq n$ ($j = 1, \dots, C$). The definitions of these pseudo-random families can be put into the public key. See [2] for more details on this challenge procedure. We can now outline our main result (for the proof of security see Section 5):

Theorem 1 *Assume the existence of a collision-resistant hash function and that the factoring assumption holds. The dynamic provable data possession scheme presented in this section (DPDP I) has the following properties, where n is the current number of blocks of the file, f is the fraction of tampered blocks, and $C = O(1)$ is the number of blocks challenged in a query:*

1. *The scheme is secure according to Definition 2;*
2. *The probability of detecting a tampered block is $1 - (1 - f)^C$;*
3. *The expected update time is $O(\log n)$ at both the server and the client whp;*
4. *The expected query time at the server, the expected verification time at the client and the expected communication complexity are each $O(\log n)$ whp;*
5. *The client space is $O(1)$ and the expected server space is $O(n)$ whp.*

Note that the above results hold in expectation and with high probability due to the properties of skip lists [27].

5 Security

In this section we, prove the security of our DPDP scheme. While our proof refers specifically to the DPDP I scheme, it also applies to the DPDP II scheme discussed in the next section. Indeed, the only difference between the two schemes is the authenticated structure used for protecting the integrity of the tags.

We begin with the following lemma, which follows from the two-party authenticated skip list construction (Theorem 1 of [25]) and our discussion in Section 3.

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

Theorem 2 (Security of core DPDP protocol) *The DPDP protocol without tags is secure in the standard model according to Definition 2 and assuming the existence of a collision-resistant hash function.*

Proof: As input, the challenger is given a hash function, which he also passes on to the reductor. 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 provides the reductor the blocks (together with their ids) whose update proofs have verified, so that the reductor can keep them in its storage. Note that *the extractor does not know the original blocks*, only the reductor does. Also note that the reductor keeps updating the blocks in its storage when the adversary performs updates. Therefore, the reductor always keeps the latest version of each block. This difference is invisible to the adversary, and so he will behave in the same way as he would to an honest challenger. At the end, the adversary replies to the challenge sent by the challenger. The extractor just outputs the blocks contained in the proof sent by the adversary. 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 outputs the original blocks) or the reductor breaks collision-resistance as follows.

The challenger passes all the blocks (together with their ids) in the proof to the reductor. By Lemma 2, if we have a skip list proof that verifies, but at least one block that is different from the original block (thus the extractor failed), the reductor can output the original block (the –latest verifying version of the– block he stored that has the same block id) and the 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 (using the extractor) or break the collision-resistance of the hash function (using the reductor) with non-negligible probability. □

Next, we analyze our improved DPDP construction that uses tags. In this case, we need also the following standard assumption:

Definition 4 (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 DPDP protocol with tags) *The DPDP protocol with tags is secure in the standard model according to Definition 2, assuming the existence of a collision-resistant hash function and that the factoring assumption holds.*

Proof: 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 $N - 1$ will have non-negligible probability of being of high order in $\mathbb{Z}_N^*$, 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 in the previous proof, our challenger will have two sub-entities: An *extractor* who extracts the challenged blocks from the adversary's proof, and a *reductor* who breaks the collision-resistance of the hash function or factors N, if the extractor fails to extract the original blocks. The challenger acts the same as in the previous proof.

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 (using the extractor), or break the factoring assumption or the collision-resistance of the hash function (using the reductor), as follows.

Call the block sent in the proof by the adversary x, and the original challenged block stored at the reductor b. The extractor just outputs x. If the extractor succeeds in extracting the correct block, then we are done. Now suppose the extractor fails, which means $x \neq b$. The challenger provides the reductor with the block x in the proof, its block id, the hash function, and g, N. Then the reductor retrieves the original block b from its storage, and checks if $g^x = g^b \mod N$. If this is the case, the reductor can break the factoring assumption; otherwise, he breaks the collision-resistance of the hash function. If $g^x = g^b \mod N$, this means $x = b \mod \phi(N)$ (where $\phi(N)$ denotes the order of $\mathbb{Z}_N^*$, which is $(p-1)(q-1)$), which means $x - b = k\phi(N)$ for some integer $k \neq 0$ (since the extractor failed to extract the original block). Hence, $x - b$ can be used in Miller's Lemma [19], 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 reductor can break the collision-resistance of the hash function by outputting $(g^x \mod N)$ and $(g^b \mod N)$.

Now consider challenging 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 is supposed to send 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 of the extraction is to reset and challenge with independent a_j and get enough independent linear equations that verifies from the adversary to solve for each m_{i_j} (thus, the extractor is just an algebraic linear solver). In the equation $M = \sum_{j=1}^{C} a_j m_{i_j}$, we have C unknowns. Therefore, we can solve for individual blocks m_{i_j} if we get C verifying linearly independent equations on the same blocks. Therefore, if the adversary can respond to a non-negligible fraction of challenges, since the extractor needs only polynomially-many equations, by rewinding polynomially-many times, the extractor can extract the original blocks. If the extractor fails to extract the original blocks, we can employ the reductor as follows.

With each rewind, if the proof given by the adversary verifies, the challenger passes on the M value and the tags in the proof to the reductor, along with the challenge. Call each original blocks b_{i_j}. The reductor first checks to see if there is any tag mismatch: $\mathcal{T}(m_{i_j}) \neq g^{b_{i_j}} \mod N$, for some $1 \leq j \leq C$. If this is the case, the reductor can output $\mathcal{T}(m_{i_j})$ and $g^{b_{i_j}} \mod N$ for that particular j as a collision, using Lemma 2. If all the tags match the original block, the reductor uses the challenge and the ids of the challenged blocks to compute linear combination $B = \sum_{j=1}^{C} a_j b_{i_j}$ of the original blocks he stored. Since the proof sent by the adversary verified, we have $T = \prod_{j=1}^{C} \mathcal{T}(m_{i_j})^{a_j} \mod N = g^M \mod N$. Since all the tags were matching, we have $\mathcal{T}(m_{i_j}) = g^{b_{i_j}} \mod N$ for all $1 \leq j \leq C$. Replacing the tags in the previous equation, we obtain $T = g^B \mod N$. Now, if $M \neq B$, then it leads to factoring using Miller's Lemma [19] as before (we have $g^M = g^B \mod N$ with $M \neq B$). Otherwise, if $M = B$ for all the rewinds, then the reductor fails, but this means the extractor was successful. □

Concerning the probability of detection, the client probes C blocks by calling the Challenge procedure. Clearly, if the server tampers with a block other than those probed, the server will not be caught. Assume now that the server tampers with t blocks. 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 $1 - ((n-t)/n)^C$, since choosing C of $n - t$ non-tampered blocks has probability $((n-t)/n)^C$.

6 Rank-based RSA trees

We now describe how we can use ideas from [26] to implement the DPDP II scheme (see Table 1), which has a higher probability of detection, maintains logarithmic communication complexity but has increased update time.

In [26], a dynamic authenticated data structure called *RSA tree* is presented that achieves constant expected query time (i.e., time to construct the proof), constant proof size, and $O(n^\epsilon \log n)$ expected amortized update time, for a given $0 < \epsilon < 1$. We can add rank information to the RSA tree by explicitly storing ranks at the internal nodes. Using this data structure allows the server to answer $O(\log n)$ challenges with $O(\log n)$ communication cost since the proof for a block tag has $O(1)$ size.

The reason for sending additional challenges is the fact that the probability p of detection increases with number C of challenges, since $p = 1 - (1-f)^C$, where f is the fraction of tampered blocks. Therefore, by using an RSA tree with ranks to implement DPDP, we obtain the same complexity measures as DPDP I, except for the update time, which increases from $O(\log n)$ to $O(n^\epsilon \log n)$ (expected amortized), and achieve an improved probability of detection equal to $1 - (1-f)^{\Omega(\log n)}$.

We now describe how we can use the tree structure from [26] to support rank information. In [26], an ϵ is chosen between 0 and 1 and a tree structure[2] is built that has $O(1/\epsilon)$ levels, each node having degree $O(n^\epsilon)$. However, there is no notion of order in [26]. To introduce a notion of order we assume that the

[2]The use of such a tree is dictated by the specific cryptographic primitive used.

elements lie at the leaves of the tree and we view it as a B-tree with lower bound on the degree $t = 3n^\epsilon/4$ and therefore upper bound equal to $2t = 3n^\epsilon/2$, which are both viewed as constants. Therefore we can use known B-tree algorithms to do the updates with the difference that we rebuild the tree whenever the number of the blocks of the file increases from n to $2n$ or decreases from n to $n/4$. When we rebuild, we set the new constants for the degree of the tree. By the properties of the B-tree (all leaves lie at the same level), we can prove that it is not possible to change the number of the levels of the tree before a new rebuilt takes place. To see that, suppose our file initially consists of n blocks. Suppose now, for contradiction that the number of the levels of the tree changes before a new rebuilt takes place. Note that a new rebuilt takes place when at least $3n/4$ operations (insertions/deletions) take place. We distinguish two cases:

1. If the number of the levels of the tree increases, that means that the number b of the added blocks is at least $n^{1+\epsilon} - n$. Since there is no rebuilt it should be the case that $b \leq 3n/4$ and therefore that $n^{1+\epsilon} - n \leq 3n/4$, which is a contradiction for large n;

2. If the number of the levels of the tree decreases, that means that the number b of the deleted blocks is at least $n - n^{1-\epsilon}$. Since there is no rebuilt it should be the case that $b \leq 3n/4$, and therefore that $n - n^{1-\epsilon} \leq 3n/4$, which is again a contradiction for large n.

Therefore before a big change happens in the tree, we can rebuild (by using the same ϵ and by changing the node degree) the tree and amortize. This is important, because the RSA tree structure works for trees that do not change their depth during updates, since the constant proof complexity comes from the fact that the depth is not a function of the elements in the structure (unlike B-trees), but is always maintained to be a constant.

Using the above provably secure authenticated data structure based on [26] to secure the tags (where security is based on the *strong RSA assumption*), we obtain the following result:

Theorem 4 *Assume the strong RSA assumption and the factoring assumption hold. The dynamic provable data possession scheme presented in this section (DPDP II) has the following properties, where n is the current number of blocks of the file, f is the fraction of tampered blocks, and ϵ is a given constant such that $0 < \epsilon < 1$:*

1. *The scheme is secure according to Definition 2;*

2. *The probability of detecting a tampered block is $1 - (1 - f)^{\Omega(\log n)}$;*

3. *The update time is $O(n^\epsilon \log n)$ (expected amortized) at the server and $O(1)$ (expected) at the client;*

4. *The expected query time at the server, the expected verification time at the client and the worst-case communication complexity are each $O(\log n)$;*

5. *The client space is $O(1)$ and the server space is $O(n)$.*

Note that sending $O(\log n)$ challenges in [2, 3] or DPDP I would increase the communication complexity from $O(1)$ to $O(\log n)$ and from $O(\log n)$ to $O(\log^2 n)$, respectively.

7 Extensions and applications

Our DPDP scheme supports a variety of distributed data outsourcing applications where the data is subject to dynamic updates. In this section, we describe extensions of our basic scheme that employ additional layers of rank-based authenticated dictionaries to store hierarchical, application-specific metadata for use in networked storage and version control.

7.1 Variable-sized blocks

We now show how we can augment our hashing scheme to support variable-sized blocks (e.g., when we want to update a byte of a certain block). Recall that our ranking scheme assigns each internal node u a rank $r(u)$ equivalent to the number of bottom-level nodes (data blocks) reachable from the subtree rooted at u; these nodes (blocks) are conventionally assigned a rank equal to 1. We support variable-sized blocks by defining the rank of a node at the bottom level to be the size of its associated block (i.e., in bytes). Each internal node, in turn, is assigned a rank equivalent to the amount of bytes reachable from it. Queries and proofs proceed the same as before, except that ranks and intervals associated with the search path refer to byte offsets, not block indices, with updates phrased as, e.g., "insert m bytes at byte offset i". Such an update 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.

7.2 Directory hierarchies

We can also extend our DPDP scheme for use in storage systems consisting of multiple files within a directory hierarchy. The key idea is to place the start node of each file's rank-based authenticated structure (from our single-file scheme) at the bottom node of a parent dictionary used to map file names to files. Using key-based authenticated dictionaries [25], we can chain our proofs and update operations through the entire directory hierarchy, where each directory is represented as an authenticated dictionary storing its files and subdirectories. Thus, we can use these authenticated dictionaries in a nested manner, with the start node of the topmost dictionary representing the root of the file system(as depicted in Figure 2(a)).

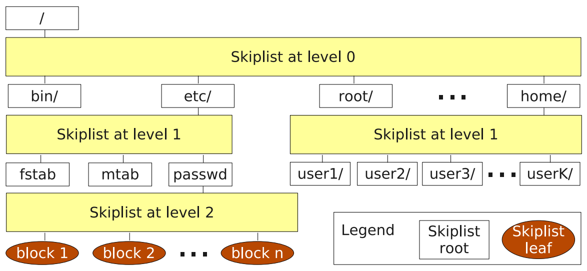
This extension provides added flexibility for multi-user environments. Consider a system administrator who employs an untrusted storage provider. The administrator can keep the authenticated structure's metadata corresponding to the topmost directory, and use it to periodically check the integrity of the whole file system. Each user can keep the label of the start node of the dictionary corresponding to her home directory, and use it to independently check the integrity of her home file system at any time, without need for cooperation from the administrator.

Since the start node of the authenticated structure of the directory hierarchy is the bottom-level node of another authenticated structure at a higher level in the hierarchy, upper levels of the hierarchy must be updated with each update to the lower levels. Still, the proof complexity stays relatively low: For example, for the rank-based authenticated skip list case, 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 file system have expected $O(d \log n)$ size and computation time whp.

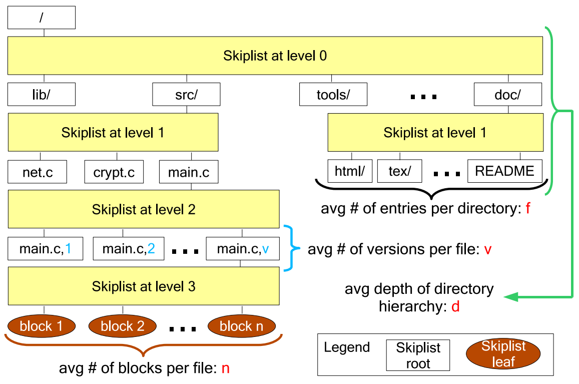
7.3 Version control

We can build on our extensions further to efficiently support a versioning system (e.g., a CVS repository, or versioning filesystem). Such a system can be supported by adding another additional layer of key-based authenticated dictionaries [25], keyed by revision number, between the dictionaries for each file's directory and its data, chaining proofs as in previous extensions. (See Figure 2(b) for an illustration.) As before, the client needs only to store the topmost basis; thus we can support a versioning system for a single file with only $O(1)$ storage at the client and $O(\log n + \log v)$ proof complexity, where v is the number of the file versions. For a versioning system spanning multiple directories, let v be the number of versions and d be the depth of the directory hierarchy. The proof complexity for the versioning file system has expected size $O(d(\log n + \log v))$.

The server may implement its method of block storage independently from the dictionary structures used to authenticate data; it does not need to physically duplicate each block of data that appears in each new version. However, as described, this extension requires the addition of a new rank-based dictionary



(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. To eliminate redundancy at the version level, persistent authenticated skip lists could be used [1]: the complexity of these proofs will then be $O(\log n + \log v + d \log f)$.

Figure 2: Applications of our DPDP system.

representing file data for each new revision added (since this dictionary is placed at the leaf of each file's version dictionary). In order to be more space-efficient, we could use *persistent* authenticated dictionaries [1] along with our rank mechanism. These structures handle updates by adding some new nodes along the update path, while preserving old internal nodes corresponding to previous versions of the structure, thus avoiding unneeded replication of nodes.

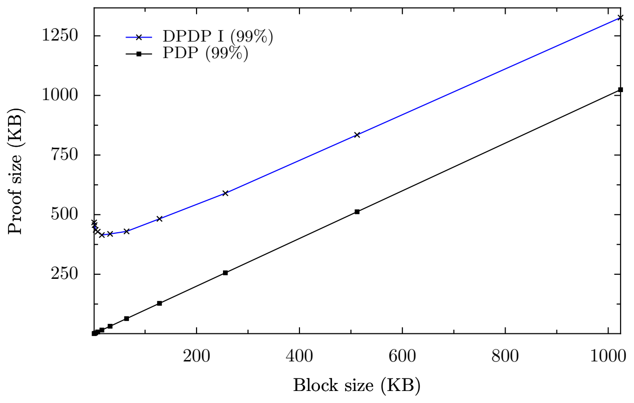
8 Performance evaluation

We evaluate the performance of our DPDP I scheme (Section 4.2) in terms of communication and computational overhead, in order to determine the *price of dynamism* over static PDP. For ease of comparison, our evaluation uses the same scenario as in PDP [2], where a server wishes to prove possession of a 1GB file. As observed in [2], detecting a 1% fraction of incorrect data with 99% confidence requires challenging a constant number of 460 blocks; we use the same number of challenges for comparison.

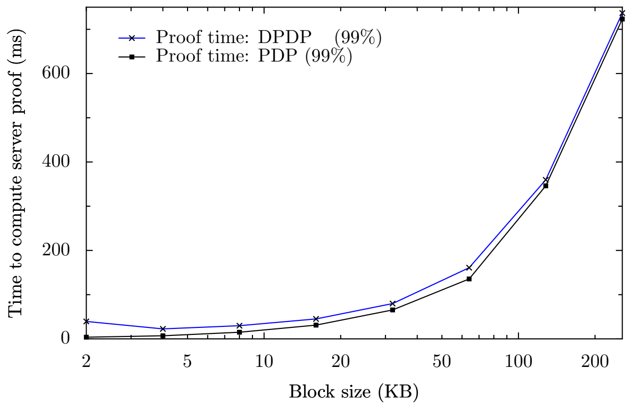
8.1 Proof size

The expected size of proofs of possession 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], [3] and is thus represented by the line labeled PDP. The distance between this line and those for our DPDP I scheme represents our communication overhead—the price of dynamism—which comes from the skip list query responses (illustrated in Table 2). Each response contains on average $1.5 \log n$ rows, so the total size decreases exponentially (but slowly) with increasing block size, providing near-constant overhead except at very small block sizes.

[3]The authors present multiple versions of their scheme. The version without the knowledge of exponent assumption and the random oracle actually sends this M; other versions only compute it.



(a)



(b)

Figure 3: (a) Size of proofs of possession 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.

8.2 Server computation

Next, we measure the computational overhead incurred by the server in answering challenges. Figure 3(b) presents the results of these experiments (averaged from 5 trials), which were performed on an AMD Athlon X2 3800+ system with 2GHz CPU and 2GB of RAM. 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 and increases linearly with the block size; note that static PDP [2] must also compute this M in response to the challenge. Thus the computational price of 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 1GB file with a million blocks of size 1KB, computing the proof for 460 challenged blocks (achieving 99% confidence) requires less than 40ms in total (as small as 13ms with larger blocks). We found in other experiments that even when the server is not I/O bound (i.e., when computing M from memory) the computational cost was nearly the same. Note that any outsourced storage system proving the knowledge of the challenged blocks must reach those blocks and therefore pay the I/O cost, and therefore such a small overhead for such a huge file is more than acceptable.

The experiments suggest the choice of block size that minimizes total communication cost and computation overhead for a 1GB file: a block size of 16KB is best for 99% confidence, resulting in a proof size of

415KB, and computational overhead of 30ms. They also show that the price of dynamism is a small amount of overhead compared to the existing PDP scheme.

8.3 Version control

Finally, we evaluate an application that suits our scheme's ability to efficiently handle and prove updates to versioned, hierarchical resources. Public CVS repositories offer a useful benchmark to assess the performance of the version control system we describe in Section 7. Using CVS repositories for the Rsync [28], Samba [28] and Tcl [24] projects, we retrieved the sequence of updates from the RCS source of each file in each repository's main branch. RCS updates come in two types: "insert m lines at line n" or "delete m lines starting at line n". Note that other partially-dynamic schemes (i.e., Scalable PDP [3]) cannot handle these types of updates. For this evaluation, we consider a scenario where queries and proofs descend a search path through hierarchical authenticated dictionaries corresponding (in order) to the directory structure, history of versions for each file, and finally to the source-controlled lines of each file. We use variable-sized data blocks, but for simplicity, assume a naïve 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, resulting in less overhead.

	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. # updates/commit	13.9	10	15.3
Avg. # commits/file	30.7	17.9	13.7
Avg. # entries/directory	12.8	7	19.8
Proof size, 99%	425 KB	395 KB	426 KB
Proof size per commit	13 KB	9 KB	15 KB
Proof time per commit	1.2ms	0.9ms	1.3ms

Table 4: Authenticated CVS server characteristics.

Table 4 presents performance characteristics of three public CVS repositories under our scheme; while we have not implemented an authenticated CVS system, we report the server overhead required for proofs of possession for each repository. Here, "commits" refer to individual CVS checkins, each of which establish a new version, adding a new leaf to the version dictionary for that file; "updates" describe the number of inserts or deletes required for each commit. Total statistics sum the number of lines (blocks) 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 evaluate the performance of a proof of possession under the DPDP I scheme: as described in Section 7, 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.

As in the previous evaluation, "Proof size, 99%" in Table 4 refers to the size of a response to 460 challenges over an entire repository (all directories, files, and versions). This figure shows that clients of an untrusted CVS server—even those storing none of the versioned resources locally—can query the server to prove possession of the repository using just a small fraction (1% to 5%) of the bandwidth required to download the entire repository. "Proof size and time *per commit*" refer to a proof sent by the server to prove that a single commit (made up of, on average, about a dozen updates) was performed successfully, representing the typical use case. These commit proofs are very small (9KB to 15KB) and fast to compute (around 1ms), rendering them practical even though they are required for each commit. Our experiments show that our DPDP scheme is efficient and practical for use in distributed applications.

Acknowledgments

Work supported in part by the U.S. National Science Foundation under grants CNS–0627553, IIS–0713403 and OCI–0724806, by a research gift from NetApp, Inc., and by the Center for Geometric Computing and the Kanellakis Fellowship at Brown University. We thank Giuseppe Ateniese and Anna Lysyanskaya for many useful discussions.

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.

[3] G. Ateniese, R. D. Pietro, L. V. Mancini, and G. Tsudik. Scalable and efficient provable data possession. SecureComm, 2008.

[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. Boneh, B. Lynn, and H. Shacham. Short signatures from the weil pairing. In *ASIACRYPT*, 2001.

[6] 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.

[7] Y. Dodis, S. Vadhan, and D. Wichs. Proofs of retrievability via hardness amplification. In *TCC*, 2009.

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

[9] D. L. Gazzoni and P. S. L. M. Barreto. Demonstrating data possession and uncheatable data transfer. Cryptology ePrint Archive, Report 2006/150, 2006.

[10] M. T. Goodrich, C. Papamanthou, R. Tamassia, and N. Triandopoulos. Athos: Efficient authentication of outsourced file systems. In *ISC*, pages 80–96, 2008.

[11] M. T. Goodrich, R. Tamassia, and A. Schwerin. Implementation of an authenticated dictionary with skip lists and commutative hashing. In *DISCEX II*, pages 68–82, 2001.

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

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

[14] 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.

[15] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structures for outsourced databases. In *SIGMOD*, pages 121–132, 2006.

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

[17] U. Maheshwari, R. Vingralek, and W. Shapiro. How to build a trusted database system on untrusted storage. In *OSDI*, pages 10–26, Berkeley, CA, USA, 2000. USENIX Association.

[18] R. Merkle. A digital signature based on a conventional encryption function. *LNCS*, 293:369–378, 1987.

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

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

[21] M. Naor and K. Nissim. Certificate revocation and certificate update. In *USENIX Security*, pages 17–17, 1998.

[22] M. Naor and G. N. Rothblum. The complexity of online memory checking. In *FOCS*, pages 573–584, 2005.

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

[24] J. Ousterhout. Tcl/tk. http://www.tcl.tk/.

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

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

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

[28] Samba. Samba.org CVS repository. http://cvs.samba.org/cgi-bin/cvsweb/.

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

[30] 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.

[31] H. Shacham and B. Waters. Compact proofs of retrievability. In *ASIACRYPT*, 2008.

[32] R. Tamassia. Authenticated data structures. In *ESA*, pages 2–5, 2003.

[33] R. Tamassia and N. Triandopoulos. Computational bounds on hierarchical data processing with applications to information security. In *ICALP*, pages 153–165, 2005.