Dynamic Provable Data Possession

C. CHRIS ERWAY[1], AppNeta, Inc.
ALPTEKIN KÜPÇÜ[1], Koç University
CHARALAMPOS PAPAMANTHOU[1], ECE and UMIACS, University of Maryland
ROBERTO TAMASSIA, Brown University

15

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 them 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 have not been tampered with or deleted (without downloading the actual data). However, existing PDP schemes apply 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).

Categories and Subject Descriptors: E.3 [**Data Encryption**]—*Public key cryptosystems*; H.3.2 [**Information Storage**]—*File organization*

General Terms: Security, Algorithms

Additional Key Words and Phrases: Coud storage, outsourced storage, provable data possession, proof of retrievability, secure storage, cloud security

ACM Reference Format:
C. Chris Erway, Alptekin Küpçü, Charalampos Papamanthou, and Roberto Tamassia. 2015. Dynamic provable data possession. ACM Trans. Info. Syst. Sec. 17, 4, Article 15 (April 2015), 29 pages.
DOI: http://dx.doi.org/10.1145/2699909

[1]Work mainly done while at Brown University.

A preliminary version of this work appeared in the 16th ACM Conference on Computer and Communications Security (ACM CCS 2009) [Erway et al. 2009]. Work supported in part by the U.S. National Science Foundation under grants CNS–0627553, CNS–1228485, IIS–0713403 and OCI–0724806, by a research gift from NetApp, Inc., by the Center for Geometric Computing and the Kanellakis Fellowship at Brown University, by TÜBİTAK, the Scientific and Technological Research Council of Turkey, under project number 112E115, and by European Union COST Actions IC1306 and IC1206.
Authors' addresses: C. C. Erway, AppNeta, 285 Summer Street, Fourth Floor, Boston, MA 02210, USA; A. Küpçü, Koç Üniversitesi Mühendislik Fakültesi, Sarıyer, İstanbul, 34450, TURKEY; C. Papamanthou, 3409 A.V. Williams Building, University of Maryland, College Park, MD, 20720, USA; R. Tamassia, Department of Computer Science, Brown University, 115 Waterman Street, Providence, RI 02912-1910, USA.

DOI: http://dx.doi.org/10.1145/2699909

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 have 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 the data since this may be prohibitive in terms of bandwidth and time, especially if the client performs this check frequently (therefore *authenticated data structure* solutions [Tamassia 2003] cannot be directly applied in this scenario).

Ateniese et al. [2011] have formalized a model called *Provable Data Possession* (PDP). In this model, data (often represented as a file F) are preprocessed by the client, and meta-data 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 have 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 [Ateniese et al. 2011; Dodis et al. 2009; Juels and Kaliski. 2007; Shacham and Waters 2013] 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. [2008] present a scheme with somewhat limited dynamism, which is discussed in detail in the related work section. Although the static model fits some application scenarios (e.g., libraries and scientific datasets), it is crucial to consider the dynamic case in which 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 [Kallahalla et al. 2003; Li et al. 2004], database services [Maheshwari et al. 2000], and peer-to-peer storage [Kubiatowicz et al. 2000; Muthitacharoen et al. 2002].

In this article, we introduce a 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), 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, thus enabling verification at different levels for different users (e.g., every user can verify her own home directory) and, at the same time, removing the need to download the whole data (as opposed to [Goodrich et al. 2008]). Our scheme yields a provable outsourced versioning system (e.g., CVS), which is evaluated in Section 6 by using traces of CVS repositories from three well-known projects.

1.1. Contributions

The main contributions of this work are summarized as follows:

(1) We introduce a formal framework for DPDP.
(2) We provide the first efficient *fully dynamic* PDP solution.

Table I. Comparison of PDP Schemes

Scheme	Server comp.	Client comp.	Comm.	Model	Block operations: append	modify	insert	delete	Probability of detection
PDP [Ateniese et al. 2011]	$O(1)$	$O(1)$	$O(1)$	RO	✓				$1-(1-f)^C$
Scalable PDP [Ateniese et al. 2008]	$O(1)$	$O(1)$	$O(1)$	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	✓	✓	✓	✓	$1-(1-f)^{\Omega(\log n)}$

Original PDP scheme [Ateniese et al. 2011]; Scalable PDP [Ateniese et al. 2008]; our scheme based on authenticated skip lists (DPDP I) and our scheme based on RSA trees (DPDP II). Asterisk (*) indicates that a certain operation can be performed only a limited (predetermined) number of times. RO means the scheme is proven secure only in the Random Oracle model. 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.

(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 I).
(4) We give an alternative construction (Section 7.1) of a rank-based authenticated dictionary using an RSA tree [Papamanthou et al. 2008]. This construction results in a DPDP scheme with improved detection probability but higher server computation (see DPDP II in Table I).
(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.

Here, 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 meta-data stored locally by the client) is $O(1)$ for both schemes; and 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., Li et al. [2006] and Naor and Nissim [1998]) 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 [Papamanthou and Tamassia 2007] but not for Merkle trees.

1.2. Related Work

The PDP scheme by Ateniese et al. [2011] provides an optimal protocol for the *static* case that achieves $O(1)$ costs for all the complexity measures listed earlier. 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 [Gazzoni and Barreto 2006; Oprea et al. 2005], linear storage for the client [Sebe et al. 2004], or do not provide security guarantees for data possession [Schwarz and Miller 2006]. Note that using [Ateniese et al. 2011] in a dynamic scenario is insecure

due to replay attacks. As observed in Dwork et al. [2009], to avoid replay attacks, an authenticated tree structure that incurs logarithmic costs must be employed, and thus constant costs are not feasible (under certain assumtptions) in a dynamic scenario. The optimal PDP construction was generalized by Ateniese et al. [2009].

Juels and Kaliski present *Proofs of Retrievability* (PORs) [Juels and Kaliski 2007], focusing on static archival storage of large files. Their scheme's effectiveness rests largely on preprocessing steps that 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 [Shacham and Waters 2013], but their solution is also static (see [Dodis et al. 2009] for a summary of POR schemes and related tradeoffs).

In our solution, we regard encryption as external to our system. If the user wants to have confidentiality of her data, she can provide us with a file whose blocks are encrypted independently, for the sake of efficiency. If the file blocks are much larger than the block size of the block cipher used for encryption, which will be the case in reality, then the confidentiality requirement is satisfied without sacrificing performance, especially when our variable block size scheme in Section 5 is employed. Since our construction does not modify the file and assumes no property on it, our system will work in perfect compliance.

But any other use of encrypting the file or employing error-correction codes will result in a huge degradation of performance. For example, Compact POR uses Reed-Solomon codes [Reed and Solomon 1960]. Modification in a single block propagates to $O(n)$ other blocks in the file. Therefore, the cost metrics will all jump to $\Omega(n)$. One immediate idea to overcome this problem is to employ erasure codes with locality. But using only local codes will be insecure, keeping the probability of detecting a cheating adversary low while being able to effectively erase a block by erasing only a small number of encoded blocks, as pointed out by Küpçü [2010a, 2010b].

Simultaneously with our work, Ateniese et al. have developed a dynamic PDP solution called Scalable PDP [Ateniese et al. [2008]. 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 recreating all the remaining challenges, which is problematic for large files. Under these limitations (otherwise the lower bound of Dwork et al. [2009] would have been violated), they provide a protocol with optimal asymptotic complexity $O(1)$ in all complexity measures giving the same probabilistic guarantees as our scheme. Last, their work is in the random oracle model, whereas our scheme is provably secure in the standard model (see Table I for full comparison). A more detailed comparison with variants [Wang et al. 2009; Zheng and Xu 2011] developed after our DPDP construction is included in Section 7.2.

Several other related works deserve to be cited even though they are focused on distributing the storage to more than a single server [Bowers et al. 2009; Curtmola et al. 2008], cloud architecture [Kamara and Lauter 2010], public verifiability via third parties [Shah et al. 2008; Wang et al. 2010], or data recoverability [Ateniese et al. 2014].

Our work is also closely related to **memory checking**, for which lower bounds are presented in Dwork et al. [2009] and Naor and Rothblum [2005]. Specifically,

in Dwork et al. [2009] it is proved that all nonadaptive 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 [Clarke et al. 2003; Tamassia and Triandopoulos 2005]. Related bounds for other primitives have been shown by Blum et al. [1994].

Recently, there were some interesting **dynamic POR** constructions. Iris [Stefanov et al. 2012] is based on the idea of aggregating multiple updates from multiple clients and sending one batch update to the server. The aggregation is performed by a trusted *portal*, and the performance is amortized. PORAM [Cash et al. 2013], on the other hand, employs oblivious RAM techniques to hide the information about which blocks are being updated from the server. Since the server does not get to observe which blocks are interrelated through the encoding, it cannot corrupt such an interrelated set to effectively corrupt an original block. Moreover, PORAM uses erasure codes independently for each block, and thus keeps the construction efficient. Similar, more efficient techniques have also emerged [Shi et al. 2013; Chandran et al. 2014].

Also recently, two other works have improved our version control system extension. First, Etemad and Küpçü [2013] realized that versions are *append-only*, and thus we may employ static PDP to keep the versions. Later, Chen and Curtmola [2014] presented an even more efficient verifiable version control system.

2. MODEL

We build on the PDP definitions from Ateniese et al. [2011]. We begin by introducing a general DPDP scheme and then show how the original PDP model is consistent with this definition.

Definition 2.1 (*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 rewrite, modify block i, delete block i, add a block after block i, etc.), and the previous meta-data 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 meta-data 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 PrepareUpdate. The output is the new version of the file F_i and the meta-data M_i, along with the meta-data 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 PrepareUpdate

algorithm,[2] plus the $M'_c, P_{M'_c}$ sent by the server. It outputs acceptance (F can be deleted in that case) or rejection signals.

—Challenge(sk, 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 meta-data M_c as input, and outputs a challenge c that is then sent to the server.

—Prove(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 meta-data, and the challenge c. It outputs a proof P that is sent to the client.

—Verify(sk, pk, M_c, c, P) $\rightarrow$ {accept, 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 meta-data 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.

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 Ateniese et al. [2011] does not store any meta-data 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 t {accept, reject}, algorithm VerifyUpdate can also output a new client meta-data M_c. In most scenarios, this new meta-data will be set as $M_c = M'_c$.

Retrieval of a (part of a) file is similar to this challenge-response protocol, composed of Challenge, Verify, 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 PrepareUpdate, PerformUpdate, and VerifyUpdate specifying an update that is a full rewrite (or append).

As stated earlier, PDP is a restricted case of DPDP. The PDP scheme of Ateniese et al. [2011] has the same algorithm definition for key generation, defines a restricted version of 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 Challenge (although one is very easy to infer). PerformUpdate consists of performing a full rewrite or an append (so that *replay* attacks can be avoided), and VerifyUpdate is used accordingly (i.e., it always accepts in case of a full rewrite, 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 Ateniese et al. [2011] and Dodis et al. [2009]. Note that the restriction to the PDP scheme gives a security definition for PDP schemes compatible with the ones in Ateniese et al. [2008, 2011].

Definition 2.2 (*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.

[2]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).

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. 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 2.1) 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 here 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 rewrite (this corresponds to the first time the client sends a file to the server). The challenger updates his local meta-data only for the verifying updates (hence, nonverifying updates are considered not to have taken place—data have 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 meta-data 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 that are accepting.

Definition 2.3 (*Alternative Security Definition for DPDP*). A DPDP scheme is secure if for any PPT f-adversary who can win the data possession game with non-negligible probability on f-fraction of blocks, there exists a PPT f-extractor algorithm that can extract f-fraction of blocks of the file with high probability by resetting and challenging the adversary polynomially many times.

THEOREM 2.4. *Definitions 2.2 and 2.3 are equivalent.*

PROOF. The f-extractor employs the extractor (in Definition 2.2) on subsets of all f-fraction of the blocks each time, until all those blocks are extracted. If the f-adversary succeeds with non-negligible probability on those f-fraction of the blocks, then extractor will succeed in extracting subsets of these. For the other direction, as long as the number of challenged blocks is less than or equal to $f * n$, then the extractor can employ the f-extractor for the purposes of extraction. □

Remark 2.5. $1/n \leq f \leq 1$, since the adversary must corrupt at least one block to attack successfully.

Remark 2.6. If $f < 1$, then the extractor cannot extract the whole file. In this case, the DPDP scheme should catch the adversary with some probability. This "probability of detection" will be discussed later.

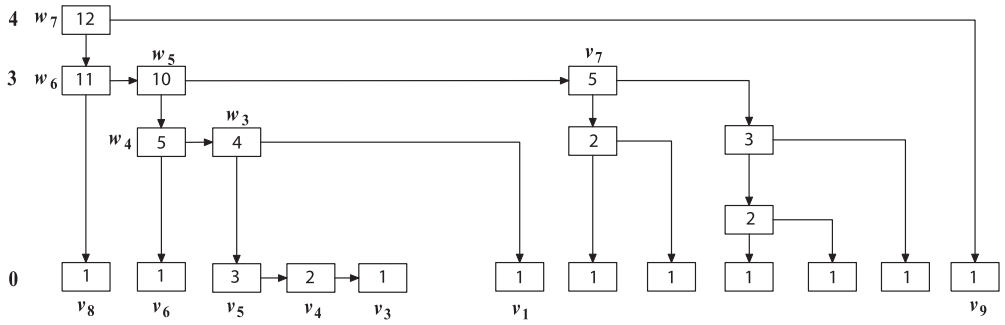


Fig. 1. Example of rank-based skip list.

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 [Dodis et al. 2009; Juels and Kaliski 2007; Shacham and Waters 2013], 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 A.

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, as explained by Küpçü [2013].

3. RANK-BASED AUTHENTICATED SKIP LISTS

In order to implement our first DPDP construction, we use a modified version of the authenticated skip list data structure [Goodrich et al. 2001]. 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 [Merkle 1987]) 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., Li et al. [2006]). Yet such algorithms have been extensively studied for the case of the authenticated skip list data structure [Papamanthou and Tamassia 2007]. Before presenting the new data structure, we briefly introduce authenticated skip lists.

The authenticated skip list is a skip list [Pugh 1990] (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 meta-data 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 [Goodrich et al. 2001], 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.

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 $\mathsf{high}(s) = n$ as 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 Goodrich et al. [2001]. 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 3.1 (*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 $+\infty$.

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 later, to compute $\mathcal{T}(i)$ and a proof for $\mathcal{T}(i)$ (for convenience we use $\mathcal{T}(i)$ to denote $\mathcal{T}(m_i)$).

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 $\mathcal{T}$;

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 II. Due to the properties of skip lists, a proof has expected size $O(\log n)$ with high probability (whp).

Table II. Proof for the 5-th Block of the File F Stored in the Skip List of Figure 1

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

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: {accept, reject} = verify(i, M_c, $\mathcal{T}$, Π)

Let $\Pi = (A_1, \ldots, A_k)$, where $A_j = (l_j, q_j, d_j, g_j)$ for $j = 1, \ldots, k$;
$\lambda_0 = 0$; $\rho_0 = 1$; $\gamma_0 = \mathcal{T}$; $\xi_0 = 0$;
for $j = 1, \ldots, k$ **do**
  $\lambda_j = l_j$; $\rho_j = \rho_{j-1} + q_j$; $\delta_j = d_j$;
  if $\delta_j =$ rgt **then**
    $\gamma_j = h(\lambda_j, \rho_j, \gamma_{j-1}, g_j)$;
    $\xi_j = \xi_{j-1}$;
  else $\{\delta_j =$ dwn$\}$
    $\gamma_j = h(\lambda_j, \rho_j, g_j, \gamma_{j-1})$;
    $\xi_j = \xi_{j-1} + q_j$;
  end if
end for
if $\gamma_k \neq M_c$ **then**
  return reject;
else if $\rho_k - \xi_k \neq i$ **then**
  return reject;
else $\{\gamma_k = M_c$ and $\rho_k - \xi_k = i\}$
  return accept;
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 $\lambda_j = l(v_j)$, that is, the level of v_j;
—integer $\rho_j = r(v_j)$, that is, 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 $\gamma_j = f(v_j)$, that is, 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 3.2. *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, that is, 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.*

Table III. 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"

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

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 atRank(i) (for an insertion or modification) or 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 Papamanthou and Tamassia [2007]. 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).

In Algorithm 3, we outline the update algorithm performed by the server (performUpdate), and in Algorithm 4, we outline 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.

To give some intuition of how Algorithm 4 produces proof $\Pi'(i)$, the reader can verify that Table III corresponds to $\Pi'(5)$, the proof that the client produces from Table II 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 III) 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 3.1 and that the ranks along the search path are increased due to the addition of one more block.

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

1: **if** upd is a deletion **then**
2: set $(\mathcal{T}'_i, \Pi'_i) = \mathsf{atRank}(i)$ and $(\mathcal{T}'_{i-1}, \Pi'_{i-1}) = \mathsf{atRank}(i-1)$;
3: set $\mathcal{T}' = \mathcal{T}'_i \cup \mathcal{T}'_{i-1}$ and $\Pi' = \Pi'_i \cup \Pi'_{i-1}$;
4: **else** {upd is an insertion or modification}
5: set $(\mathcal{T}', \Pi') = \mathsf{atRank}(i)$;
6: **end if**
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** $(\mathcal{T}', \Pi')$;

ALGORITHM 4: {accept, reject} = verUpdate(i, M_c, $\mathcal{T}$, upd, $\mathcal{T}'$, Π')

1: **if** upd is a deletion **then**
2: split $\mathcal{T}'$ into $\mathcal{T}'_i$ and $\mathcal{T}'_{i-1}$. Also split Π' into Π'_i and Π'_{i-1};
3: set $decision$ = verify(i, M_c, $\mathcal{T}'_i$, Π'_i) $\wedge$ verify($i-1$, M_c, $\mathcal{T}'_{i-1}$, Π'_{i-1});
4: **else** {upd is an insertion or modification}
5: set $decision$ = verify(i, M_c, $\mathcal{T}'$, Π');
6: **end if**
7: **if** $decision$ = reject **then**
8: **return** reject;
9: **else** {$decision$ = accept}
10: from i, $\mathcal{T}$, $\mathcal{T}'$, and Π', compute and store the updated label M'_c of the start node;
11: **return** accept;
12: **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 2.1. 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 [Goodrich et al. 2001; Papamanthou et al. 2008]. Most of the material of this section also applies to the DPDP II scheme presented in Section 7.1.

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.

—KeyGen(1^k) $\rightarrow$ {sk, 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.
—PrepareUpdate(sk, pk, F, info, M_c) $\rightarrow$ {e(F), e(info), e(M)}: This is a dummy procedure that outputs the file F and information info it receives as input. M_c and e(M) are empty (not used).
—PerformUpdate(pk, F_{i-1}, M_{i-1}, e(F), e(info), 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). e(F), e(info), e(M), which are output by PrepareUpdate, are sent by the client (e(M) being empty). The procedure updates the file according to e(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 e(info)). Note that the index j and the type of the update upd is taken from e(info) and the new data $\mathcal{T}$ is e(F). Finally, Algorithm 3 outputs M'_c and $P_{M'_c} = \Pi(j)$, which are output by PerformUpdate. The expected runtime is $O(\log n)$ whp.
—VerifyUpdate(sk, pk, F, info, M_c, M'_c, $P_{M'_c}$) $\rightarrow$ {accept, 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.

—Challenge(sk, 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 $O(C)$.

—Prove(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.

—Verify(sk, pk, M_c, c, P) $\rightarrow$ {accept, 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 aforementioned 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 Ateniese et al. [2011]. 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 [Boneh et al. 2001] as in Compact POR [Shacham and Waters 2013].

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 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 Ateniese et al. [2011]. First, instead of sending random values a_j separately, the client can simply send a random key to a pseudorandom function that will generate those values. Second, a key to a pseudorandom permutation can be sent to select the indices of the challenged blocks $1 \le i_j \le n$ $(j = 1, \ldots, C)$. The definitions of these pseudorandom families can be put into the public key. See [Ateniese et al. 2011] for more details on this challenge procedure. We can now outline our main result (for the proof of security, see Appendix A):

THEOREM 4.1. *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;*
(*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 these results hold in expectation and with high probability due to the properties of skip lists [Pugh 1990].

Intuition for the Proof. Before looking at our full proof, we present the intuition for our proof.

(1) Our challenger will have two subentities: 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. 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 client.

(2) Consider the case where the version of our DPDP scheme without the tags is used (hence, blockless-verification is not possible). At the end of the security game, the adversary will reply to the challenge sent by the challenger. The extractor just outputs the block(s) 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 succeeded in outputting the original blocks) or otherwise the reductor

breaks collision-resistance since now the original block together with the extracted block constitute a collision, failing Lemma A.1.

(3) Now we can consider the blockless-verification version of our DPDP construction. But, for simplicity, assume that only one block is challenged. Call the block sent in the proof by the adversary b, and the original challenged block stored at the reductor m. The extractor just outputs b. If the extractor succeeds in extracting the correct block (i.e., $b = m$), then we are done. Now suppose the extractor fails, which means $b \neq m$. Now, if $g^b = g^m \mod N$, then the reductor breaks the factoring assumption, since this means $b = m \mod \phi(N)$, which means $b - m = k\phi(N)$ for some integer $k \neq 0$ (since the extractor failed to extract the original block). Hence, $L = b - m$ can be used in Miller's Lemma [Miller 1975], which leads to factoring N. Otherwise, $g^b \neq g^m \mod N$. This means there are two different tags that can provide a verifying skip list proof. By Lemma A.1, the reductor can break the collision-resistance of the hash function by outputting $(g^b \mod N)$ and $(g^m \mod N)$.

Finally, we extend our simpler proofs above to a proof of the full scheme with multiple challenges (see Appendix). The overall idea will be the same, and the way we will address multiple challenges will be by (i) solving a system of linear equations as in the extractor at the last paragraph of the proof of Theorem 4.3 in Ateniese et al. [2011], and (ii) finding a subset of items that will work exactly as in case (3).

4.3. Generic Construction

At this point, we observe two main points that efficient DPDP constructions should have:

(1) For efficient proofs (i.e., blockless verification), the DPDP scheme should use homomorphic tags over the data. The homomorphism property was previously formalized by Ateniese et al. [2009], and thus we do not want to repeat the formalization here. Instead, we would like to mention possible alternative approaches and an intuitive discussion below.
(2) For dynamism of data, the DPDP protocol should employ a data structure with efficient membership queries (i.e., an authenticated data structure). We will discuss possible alternative data structures in the following text.

Thus, we can say that in general, a DPDP scheme does the following: (1) considers the data as composed of some number of chunks (e.g., as a single whole, or n blocks), (2) creates homomorphic tags over the chunks, and (3) puts a data structure on top of the tags.

When a challenge comes, the server does the following: (1) creates the combined chunk in a way that will match the homomorphic property of the tags, (2) creates membership proofs using the data structure proving that the tags of the challenged blocks are in the latest version of the data structure, and (3) sends the combined chunk, together with the data structure proofs (that include the tags) to the client as the proof.

Alternatives for Homomorphic Tags. As long as the tags have a homomorphism property, this can be used to combine the chunks into a single large chunk, providing *blockless verification*. It is possible that the tags include order information (e.g., PDP [Ateniese et al. 2011] tags contain a hash of the block index in the tag), or the tags do not care about the order information (e.g., in our DPDP construction, since rank-based authenticated skip list will handle the block indexing issues, tags are not related to block indices). However, for a dynamic system to be efficient, it must be the case that the tags do not contain order information, since otherwise an update may cause all tags to be updated.

Tags may be related to the file as a whole, to blocks of the file, or to even smaller units (e.g., sectors in POR [Juels and Kaliski 2007]). As seen in our tests in Section 6, the size of the unit in a tag matters for performance, and should be optimized.

Furthermore, as long as the homomorphic property is there, RSA-based tags, or BLS-signature-based tags may be used, along with other alternatives. If a prime-order group is used for computing tags, it must be remembered that each unit (e.g., block) must be smaller than the order of the group, since otherwise the server may just store the equivalent value that is smaller than the group order and still pass the verification. RSA-type groups where the order is unknown do not have this issue.

Alternatives for Ordered Data Structure. Rank-based authenticated skip list is a neat example of an ordered and authenticated data structure that allows membership queries efficiently ($O(\log n)$ whp where n is the number of leaves, which corresponds to the number of blocks in the file). To the best of our knowledge, rank-based authenticated skip list is the first construction of an ordered authenticated data structure that has $\log n$ performance. Later variants (although not fully detailed) include Merkle trees (used in Wang et al. [2009]) and range-based 2–3 trees [Zheng and Xu 2011]. The most important advantage of a skip list over balanced trees is that the authenticated version is much easier to implement, without the complication of the authenticated rebalancing operations. Since a skip list will have $\log n$ height with high probability, the need for complex authenticated balancing operations to obtain guaranteed $\log n$ performance is not well justified. Recently, a variant of our rank-based authenticated skip list construction called FlexList is also proposed, with improved efficiency, and the ability to handle multiple challenges and updates at once [Kachkeev et al. 2013; Esiner et al. 2013, 2014].

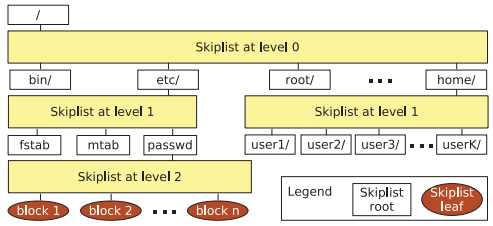
Yet, even more alternative authenticated data structures include hash lists (i.e., an array of hash values of each block), or simply a hash of the whole file (in general, we consider this as a value rather than a data structure because of its simplicity). But, such structures do not provide efficient DPDP: (1) If a single hash of the file is kept at the client, even though this is very efficient in terms of client storage ($O(1)$) and server storage (no extra tags or data structure to keep), the updates are very inefficient. Even a single change in the file would require recomputing the hash of the whole file—an operation that is extremely slow for large files. (2) If a hash list is used, the client will need $O(n)$ space to keep one hash per block, even though the server storage is optimal as the first case. Modification of a block will be an $O(1)$ operation; just requiring the client to recompute the hash of that block. Yet, an insertion into the file may cause $O(n)$ time.

5. EXTENSIONS AND APPLICATIONS

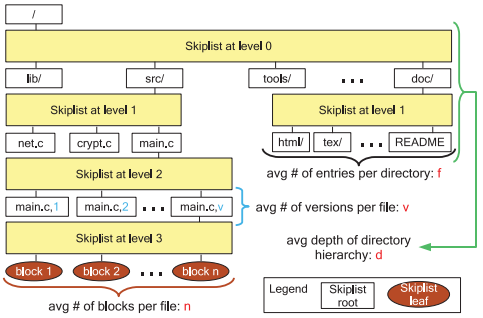
Our DPDP scheme supports a variety of distributed data outsourcing applications where the data is subject to dynamic updates. In this section, we describe extensions of our basic scheme that employ additional layers of rank-based authenticated dictionaries to store hierarchical, application-specific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

5.1. 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.,



(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 skip lists for holding versions of a file. To eliminate redundancy at the version level, persistent authenticated skip lists could be used [Anagnostopoulos et al. 2001]: the complexity of these proofs will then be $O(\log n + \log v + d \log f)$.

Fig. 2. Applications of our DPDP system.

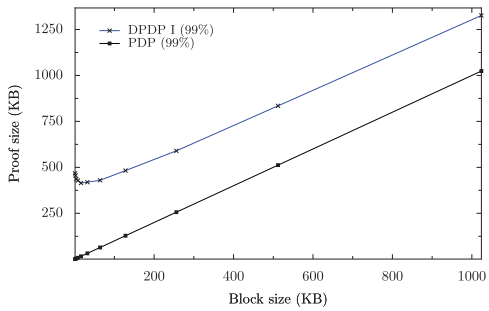
in bytes). Each internal node, in turn, is assigned a rank equivalent to the amount of bytes reachable from it. Queries and proofs proceed the same as before, except that ranks and intervals associated with the search path refer to byte offsets, not block indices, with updates phrased as, for example, "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.

5.2. Directory Hierarchies

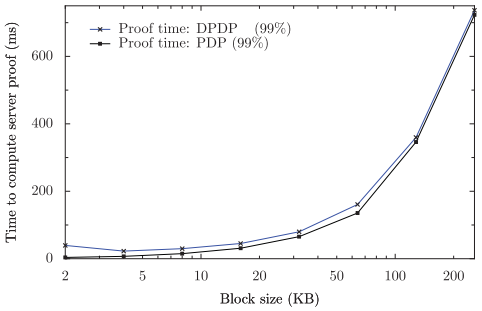
We can also extend our DPDP scheme for use in storage systems consisting of multiple files within a directory hierarchy. The key idea is to place the start node of each file's rank-based authenticated structure (from our single-file scheme) at the bottom node of a parent dictionary used to map file names to files. Using key-based authenticated dictionaries [Papamanthou and Tamassia 2007], 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 multiuser 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.



(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.

Fig. 3. Price of dynamism.

5.3. 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 [Papamanthou and Tamassia 2007], 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 representing file data for each new revision added (since this dictionary is placed at the leaf of each file's version dictionary). To be more space-efficient, we could use *persistent* authenticated dictionaries [Anagnostopoulos et al. 2001] 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.

6. 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 [Ateniese et al. 2011], where a server wishes to prove possession of a 1GB file. As observed in Ateniese et al. [2011], 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.

6.1. 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 Ateniese et al. [2011],[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 II). 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.

6.2. 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 five trials), which were performed on an AMD Athlon X2 3800+ system with 2GHz CPU and 2GB of RAM. As earlier, 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 [Ateniese et al. 2011] 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.

In terms of a practical deployment serving multiple clients, note that with 30ms per 460-block challenge, the server throughput would be 33 clients per second using a single-core machine (with properties similar to the one in our tests). A 16-core server machine can serve more than 500 clients' challenges per second. Considering the proof size of 415KB each, this would require the server to have about 1.65Gbit/s upload bandwidth, which would be the limiting factor in such a server deployment.

6.3. 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 5. Using CVS repositories for the Rsync,[4] Samba,[5] and Tcl[6] 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 [Ateniese et al. 2008]) cannot handle these types of updates. For this evaluation, we

[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.
[4]http://rsync.samba.org/.
[5]http://cvs.samba.org/.
[6]http://www.tcl.tk/.

Table IV. Authenticated CVS Server Characteristics

	Rsync	Samba	Tcl
dates of activity	1996–2007	1996–2004	1998–2008
# of files	371	1,538	1,757
# of commits	11,413	27,534	24,054
# of updates	159,027	275,254	367,105
Total lines	238,052	589,829	1,212,729
Total KB	8,331KB	18,525KB	44,585KB
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%	425KB	395KB	426KB
Proof size per commit	13KB	9KB	15KB
Proof time per commit	1.2ms	0.9ms	1.3ms

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.

Table IV 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 5, 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 IV 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 per commit" and "Proof 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.

7. REMARKS

7.1. Rank-Based RSA Trees

We now describe how we can use ideas from Papamanthou et al. [2008] to implement the DPDP II scheme (see Table I), which has a higher probability of detection, maintains logarithmic communication complexity but has increased update time.

In Papamanthou et al. [2008], 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 because 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 Papamanthou et al. [2008] to support rank information. In Papamanthou et al. [2008], an ϵ is chosen between 0 and 1 and a tree structure[7] is built that has $O(1/\epsilon)$ levels, each node having degree $O(n^\epsilon)$. However, there is no notion of order in Papamanthou et al. [2008]. To introduce a notion of order, we assume that the 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 aforementioned provably secure authenticated data structure based on [Papamanthou et al. 2008] to secure the tags (where security is based on the *strong RSA assumption*), we obtain the following result:

THEOREM 7.1. *Assume the strong RSA assumption and the factoring assumption hold. The dynamic provable data possession scheme presented in this section (DPDP II) has*

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

Table V. Two New DPDP Schemes That Will Incorporate Tags and Proof Techniques Described in This Chapter, Together with Ideas from Papamanthou et al. [2008]
As before, we denote with n the number of the blocks of the file, with f the fraction of the corrupted blocks, and with C being the number of challenged blocks (typically a constant independent of n). In all constructions, the storage space is $O(1)$ at the client and $O(n)$ at the server.

Scheme	DPDP III	DPDP IV
Update time (server)	$O(n^\epsilon)$	$O(1)$
Challenge time (server)	$O(1)$	$O(n^\epsilon)$
Client computation	$O(1)$	$O(1)$
Communication	$O(1)$	$O(1)$
Model	Standard	Standard
Append blocks	✓	✓
Modify blocks	✓	✓
Insert blocks	✓	✓
Delete blocks	✓	✓
Prob. of detection	$1-(1-f)^C$	$1-(1-f)^C$

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.*
(*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 Ateniese et al. [2008, 2011] or DPDP I constructions would increase the communication complexity from $O(1)$ to $O(\log n)$ and from $O(\log n)$ to $O(\log^2 n)$, respectively.

7.2. Other DPDP Constructions

Following ideas from Papamanthou et al. [2008], we can modify our scheme in Section 7.1 to implement DPDP III and DPDP IV schemes (see Table V), which are optimized for challenge-intensive or update-intensive workloads, respectively. Both DPDP III and DPDP IV schemes will achieve the same probability of detection as our DPDP I scheme. The reasons they are presented as future work is that we have not analyzed their efficiency carefully, and hence the following table should be taken only as a good guess of what will happen if ideas in Papamanthou et al. [2008] will be incorporated to our DPDP scheme keeping the same tags and the challenge structure.

7.3. POR vs. PDP

One can generalize most secure cloud storage schemes as PDP-type or POR-type schemes. Both types of schemes are introduced with the same purpose: Alice wants to store her data at a server that may not necessarily be fully trusted. Alice would like to obtain a proof that her data is being kept intact at the server side. The most widely accepted definitions for POR-type schemes appear in Compact POR [Shacham and Waters 2013] and its generalization [Dodis et al. 2009]. We will use the definitions in this article for the PDP-type schemes.

Consider, as an example, Dropbox cloud storage system. When Alice signs up for Dropbox, it creates a directory on the client's computer, with some demonstration files inside, and immediately starts uploading those files. Thus, by choosing these initial demonstration files, the adversary can easily mount a chosen-file attack. Furthermore, with program updates, the adversary may update these files, thus forcing Alice's computer to perform those updates on the server. Therefore, we conclude that an *adaptive chosen-file attack* formulation is necessary. Note that this formulation is similar to a *chosen-plaintext attack* on an encryption scheme, and its history is full of justification for adopting such a definition.

After the adversary mounts this adaptive chosen-file attack, the challenger interacts with the adversary through the challenge-response protocol. There needs to be an extractor, similar to the one in zero-knowledge proof of knowledge (ZKPoK) systems, to make sure the server indeed keeps the file intact. As in ZKPoK systems, the extractor may rewind the adversary and rechallenge polynomially-many times.

Note that the initial POR-type definition [Juels and Kaliski 2007] included the extractor as part of the actual scheme and required the actual server to be stateless to ensure it works. Later, it was realized that this is an unnecessary requirement and that formulating an imaginary extractor as in the ZKPoK systems is enough to ensure security [Ateniese et al. 2011; Shacham and Waters 2013; Erway et al. 2009]. Therefore, we allow the extractor to rewind the adversary, instead of assuming a stateless server. Obviously, we need the probability that the extractor succeeds to be high.

We have seen that POR-type schemes provide very strong security guarantees by *necessarily* sacrificing performance. On the other hand, PDP-type schemes can work well under certain scenarios. In the following, we analyze various scenarios:

—For commercial applications that do not require extremely high guarantees where the server may be semitrusted, PDP-type schemes may provide a sufficient guarantee. For example, Alice may trust that Amazon will not intentionally modify a single bit of her data, but Amazon may try to hide a system failure where a fraction of her file is lost. In such a case, Amazon will be caught cheating using a PDP-type scheme.
—For the cases where the file itself is tolerant to minor modifications, again PDP-type schemes may be sufficient. For example, regular text files may be tolerant to change in a few letters; image, video, or audio files may tolerate a few glitches here and there.
—When employed as a business practice by companies whose reputations matter, there is a great incentive not to get caught, even with a low probability. Consider the Amazon example again. If Amazon gets caught cheating, then the financial losses will be intolerable. Note that, on the other hand, without any provable storage system in use, Amazon is not necessarily afraid of corrupting user data, since there will be no proof.
—Moreover, the law of large numbers is a very important concern for secure cloud storage scenarios. Even though the probability that a single user (Alice) catches Amazon cheating is low, Amazon has too many users. The law of large numbers tells us that some users will catch Amazon cheating. Besides, remember that the earlier analysis represents a single challenge-response scenario. In reality, Alice will challenge the server multiple times. Again, the law of large numbers tells us that even a single user like Alice will catch Amazon cheating after multiple challenge-response protocol executions.

The aforementioned scenarios are in line with the *covert adversay* model for multiparty computation [Aumann and Lindell 2010]. As argued by Aumann and Lindell, this model represents realistic adversaries, those who are afraid of legal penalties. Thus, we can

say PDP-type schemes work well in such a covert adversary setting, and thus we expect them to perform as necessary in real scenarios.

In summary, if performance is the main concern, PDP-type schemes should be employed, with the fact in mind that a single challenge-response is not extremely binding, but multiple random challenges increase the probability of catching dramatically. On the other hand, if immediate conclusion is required and the data is highly-sensitive, then POR-type schemes with built-in erasure- and error-resistance must be employed. Note that, in both cases, if the server corrupts all the data, there is no technical solution possible to retrieve back the original.

APPENDIX

A. 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.

To prove security of our DPDP scheme, we need the following assumptions, definitions, facts, and lemmas. We begin with the following lemma, which follows from the two-party authenticated skip list construction (Theorem 1 of [Papamanthou and Tamassia 2007]) and our discussion in Section 3.

LEMMA A.1. *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 nonnegligible probability.*

Definition A.2 (*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.

Definition A.3. Euler's ϕ function for $N = pq$ where p, q are primes is defined as $\phi(N) = (p-1)(q-1)$.

Definition A.4. Carmichael λ function for $N = pq$ where p, q are primes is defined as $\lambda(N) = \mathsf{lcm}(p-1, q-1)$ where $\mathsf{lcm}(x, y)$ denotes the least common multiple of x and y.

FACT 1. $\lambda(N) \mid \phi(N)$.

LEMMA A.5 (MILLER'S LEMMA [MILLER 1975]). *Let L be a number divisible by $\lambda(N)$. Then, there exists a PPT algorithm that factors N with nonnegligible probability, given L and N.*

THEOREM A.6 (SECURITY OF DPDP PROTOCOL). *The DPDP protocol is secure in the standard model according to Definition 2.2, assuming the existence of a collision-resistant hash function and that the factoring assumption holds.*

PROOF. The challenger is given a hash function h, and an integer $N = pq$ but not p or q. The challenger then samples a high-order element g from $\mathbb{Z}_N^*$. He interacts with the adversary in the data possession game honestly, using the given hash function, and creates and updates the tags while using N as the modulus and g as the base.

Suppose now the challenger challenges C blocks, namely the blocks with indices $i_1, i_2, \ldots, i_C$. We recall that in response to each challenge, the proof contains:

(1) The tags $T_{i_1}, T_{i_2}, \ldots, T_{i_C}$ for each block $i_1, i_2, \ldots, i_C$, along with the respective skip list proofs that correspond to each tag $T_{i_1}, T_{i_2}, \ldots, T_{i_C}$;
(2) A "weighted" sum of the form $B = a_{i_1} b_{i_1} + a_{i_2} b_{i_2} + \cdots + a_{i_C} b_{i_C}$, where a_{i_j} $(j = 1, \ldots, C)$ are random numbers known by the challenger.

According to Definition 2.2, the DPDP scheme is secure if, whenever the verification succeeds with nonnegligible probability (i.e., the adversary wins the data possession game), the challenger can extract the actual blocks (which we denote with $m_{i_1}, m_{i_2}, \ldots, m_{i_C}$) in polynomially-many interactions with the adversary. The idea of the extraction is to reset and challenge with independent a_{i_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 for B, 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 nonnegligible fraction of challenges, since the extractor needs only polynomially-many (indeed, C) equations, by rewinding polynomially-many times, the extractor can extract the original blocks. Now suppose the challenger challenges the adversary for a polynomial number of times and gets C verifying responses. Then, if $B_1, B_2, \ldots, B_C$ are the weighted sums received each time, we have the following equations:

$$\begin{aligned} B_1 &= a_{i_{11}} b_{i_1} + a_{i_{12}} b_{i_2} + \cdots + a_{i_{1C}} b_{i_C} \\ B_2 &= a_{i_{21}} b_{i_1} + a_{i_{22}} b_{i_2} + \cdots + a_{i_{2C}} b_{i_C} \\ &\vdots \\ B_C &= a_{i_{C1}} b_{i_1} + a_{i_{C2}} b_{i_2} + \cdots + a_{i_{CC}} b_{i_C}, \end{aligned}$$

where $a_{i_{j1}}, a_{i_{j2}}, \ldots, a_{i_{jC}}$ for $j = 1, \ldots, C$ are different sets of random numbers sent each time with the challenge and $b_{i_1}, b_{i_2}, \ldots, b_{iC}$ are the blocks that the adversary claims to possess. By solving this system of linear equations we extract the blocks $b_{i_1}, b_{i_2}, \ldots, b_{iC}$. This constitutes the extractor's output, and if they correspond to the original blocks $m_{i_1}, m_{i_2}, \ldots, m_{iC}$, then we are done. Otherwise, meaning there is at least one nonmatching block, we show that the reductor can either break the factoring assumption or the collision-resistance of the hash function.

Suppose now there is a subset of challenged blocks $\{b_1, b_2, \ldots, b_k\} \subseteq \{b_{i_1}, b_{i_2}, \ldots, b_{iC}\}$ such that $b_j \neq m_j$ for all $j = 1, \ldots, k$ (i.e. the extractor failed for those blocks). Let $a_1, a_2, \ldots, a_k$ and $T_1, T_2, \ldots, T_k$ be the random numbers and the tags, respectively, that correspond to some response (i.e., to some linear equation of the system) for blocks $\{b_1, b_2, \ldots, b_k\}$. The reductor first checks to see if there is any tag mismatch: $\mathcal{T}_| \neq g^{m_j} \mod N$, for some $1 \leq j \leq k$. If this is the case, the reductor can output $\mathcal{T}_|$ and $g^{m_j} \mod N$ for that particular j as a collision, using Lemma A.1.

Furthermore, remember that since the adversary's proof verified, we have $T_{i_1}^{a_{i_1}} T_{i_2}^{a_{i_2}} \ldots T_{i_C}^{a_{i_C}} = g^B \mod N$ for all B values. The reductor now computes $M = \sum a_{i_j} m_{i_j}$, and assuming there is no tag mismatch (otherwise we would have broken collision-resistance), we know that $T_{i_1}^{a_{i_1}} T_{i_2}^{a_{i_2}} \ldots T_{i_C}^{a_{i_C}} = g^{a_{i_1} m_{i_1} + a_{i_2} m_{i_2} + \ldots + a_{i_C} m_{i_C}} \mod N$. This means $g^B = g^M \mod N$. Now, if there is the subset $\{b_1, b_2, \ldots, b_k\}$ of blocks that are different from the original blocks (i.e., $B \neq M$), then $B - M$ can be used to factor N, by using Miller's Lemma [Miller 1975].

Therefore, if the adversary can respond to a nonnegligible fraction of challenges, since the extractor needs only polynomially-many equations, by rewinding polynomially-many times, the challenger can either extract the original blocks (using the extractor), or break the collision-resistance of the hash function used or the factoring

assumption (using the reductor) with nonnegligible probability. This concludes the proof of Theorem A.6. □

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 $n-t$ nontampered blocks has probability $((n-t)/n)^C$.

ACKNOWLEDGMENTS

We thank Giuseppe Ateniese, Michael T. Goodrich, Anna Lysyanskaya, and Nikos Triandopoulos for many useful discussions.

REFERENCES

Aris Anagnostopoulos, Michael T. Goodrich, and Roberto Tamassia. 2001. Persistent authenticated dictionaries and their applications. In *Proceedings of the 4th International Conference on Information Security (ISC'01)*. Springer-Verlag, London, 379–393.

Giuseppe Ateniese, Randal Burns, Reza Curtmola, Joseph Herring, Osama Khan, Lea Kissner, Zachary Peterson, and Dawn Song. 2011. Remote data checking using provable data possession. *ACM Transactions on Information and System Security (TISSEC)* 14, 1, Article 12 (June 2011), 12:1–12:34 pages.

Giuseppe Ateniese, Roberto Di Pietro, Luigi V. Mancini, and Gene Tsudik. 2008. Scalable and efficient provable data possession. In *Proceedings of the 4th International Conference on Security and Privacy in Communication Netowrks (SecureComm'08)*. ACM, New York, NY, Article 9, 9:1–9:10 pages.

Giuseppe Ateniese, Michael T. Goodrich, Vassilios Lekakis, Charalampos Papamanthou, Evripidis Paraskevas, and Roberto Tamassia. 2014. Accountable Storage. Cryptology ePrint Archive, Report 2014/886. (2014).

Giuseppe Ateniese, Seny Kamara, and Jonathan Katz. 2009. Proofs of storage from homomorphic identification protocols. In *Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT'09)*. 319–333.

Yonatan Aumann and Yehuda Lindell. 2010. Security against covert adversaries: Efficient protocols for realistic adversaries. *Journal of Cryptology* 23 (2010), 281–343.

M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. 1994. Checking the correctness of memories. *Algorithmica* 12, 2 (1994), 225–244.

Dan Boneh, Ben Lynn, and Hovav Shacham. 2001. Short signatures from the weil pairing. In *Proceedings of the 7th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology (ASIACRYPT'01)*. Springer-Verlag, London, 514–532.

Kevin D. Bowers, Ari Juels, and Alina Oprea. 2009. HAIL: A high-availability and integrity layer for cloud storage. In *Proceedings of the ACM International Conference on Computer and Communications Security (CCS'09)*. 187–198.

David Cash, Alptekin Küpçü, and Daniel Wichs. 2013. Dynamic proofs of retrievability via oblivious RAM. In *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT'13)*. 279–295.

Nishanth Chandran, Bhavana Kanukurthi, and Rafail Ostrovsky. 2014. Locally updatable and locally decodable codes. In *TCC*. Springer, 489–514.

Bo Chen and Reza Curtmola. 2014. Auditable version control systems. In *Proceedings of the ISOC 21st Annual Network and Distributed System Security Symposium (NDSS'14)*.

Dwaine E. Clarke, Srinivas Devadas, Marten van Dijk, Blaise Gassend, and G. Edward Suh. 2003. Incremental multiset hash functions and their application to memory integrity checking. In *Proceedings of the 9th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology (ASIACRYPT'03)*. 188–207.

Reza Curtmola, Osama Khan, Randal Burns, and Giuseppe Ateniese. 2008. MR-PDP: Multiple-replica provable data possession. In *Proceedings of the 28th International Conference on Distributed Computing Systems (ICDCS)*. IEEE Computer Society, Washington, DC, USA, 411–420.

Yevgeniy Dodis, Salil Vadhan, and Daniel Wichs. 2009. Proofs of retrievability via hardness amplification. In *Proceedings of the 6th Theory of Cryptography Conference on Theory of Cryptography (TCC'09)*. Springer-Verlag, Berlin, 109–127.

Cynthia Dwork, Moni Naor, Guy N. Rothblum, and Vinod Vaikuntanathan. 2009. How efficient can memory checking be? In *Proceedings of the 6th Theory of Cryptography Conference on Theory of Cryptography (TCC'09)*. Springer-Verlag, Berlin, 503–520.

Chris Erway, Alptekin Küpçü, Charalampos Papamanthou, and Roberto Tamassia. 2009. Dynamic provable data possession. In *Proceedings of the ACM International Conference on Computer and Communications Security (CCS'09)*. 213–222.

Ertem Esiner, Adilet Kachkeev, Samuel Braunfeld, Alptekin Küpçü, and Öznur Özkasap. 2013. FlexDPDP: FlexList-based optimized dynamic provable data possession. *Cryptology ePrint Archive, Report 2013/645* (2013).

Ertem Esiner, Alptekin Küpçü, and Öznur Özkasap. 2014. Analysis and optimization on FlexDPDP: A practical solution for dynamic provable data possession. In *Intelligent Cloud Computing (ICC'14)*.

Mohammad Etemad and Alptekin Küpçü. 2013. Transparent, distributed, and replicated dynamic provable data possession. In *Proceedings of the Applied Cryptography and Network Security Conference (ACNS'13)*. 1–18.

Décio Luiz Gazzoni and Paulo Sérgio Licciardi Messeder Barreto. 2006. Demonstrating data possession and uncheatable data transfer. Cryptology ePrint Archive, Report 2006/150. (2006).

Michael T. Goodrich, Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. 2008. Athos: Efficient authentication of outsourced file systems. In *Proceedings of the 11th International Conference on Information Security (ISC'08)*. Springer-Verlag, 80–96.

M. T. Goodrich, R. Tamassia, and A. Schwerin. 2001. Implementation of an authenticated dictionary with skip lists and commutative hashing. In *Proceedings of the DARPA Information Survivability Conference & Exposition II (DISCEX'01)*. 68–82.

Ari Juels and Burton S. Kaliski. 2007. PORs: Proofs of retrievability for large files. In *Proceedings of the ACM International Conference on Computer and Communications Security (CCS'07)*. 584–597.

Adilet Kachkeev, Ertem Esiner, Alptekin Küpçü, and Öznur Özkasap. 2013. Energy efficiency in secure and dynamic cloud storage. In *Energy Efficiency in Large Scale Distributed Systems, EE-LSDS*. 125–130.

Mahesh Kallahalla, Erik Riedel, Ram Swaminathan, Qian Wang, and Kevin Fu. 2003. Plutus: Scalable secure file sharing on untrusted storage. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST'03)*. USENIX Association, Berkeley, CA, 29–42.

Seny Kamara and Kristin Lauter. 2010. Cryptographic cloud storage. In *Proceedings of the 14th International Conference on Financial Cryptograpy and Data Security (FC'10)*. Springer-Verlag, Berlin, Heidelberg, 136–149.

John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. 2000. OceanStore: An architecture for global-scale persistent storage. *ACM SIGPLAN Notices* 35, 11 (2000), 190–201.

Alptekin Küpçü. 2010a. *Efficient Cryptography for the Next Generation Secure Cloud*. Ph.D. Dissertation. Brown University.

Alptekin Küpçü. 2010b. *Efficient Cryptography for the Next Generation Secure Cloud: Protocols, Proofs, and Implementation*. Lambert Academic Publishing.

Alptekin Küpçü. 2013. Official arbitration with secure cloud storage application. *Computer Journal* (2013). DOI:http://dx.doi.org/10.1093/comjnl/bxt138

Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid Reyzin. 2006. Dynamic authenticated index structures for outsourced databases. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD'06)*. ACM, New York, NY, 121–132.

Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. 2004. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6 (OSDI'04)*. USENIX Association, Berkeley, CA.

Umesh Maheshwari, Radek Vingralek, and William Shapiro. 2000. How to build a trusted database system on untrusted storage. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4 (OSDI'00)*. USENIX Association, Berkeley, CA, 10–26.

R. C. Merkle. 1987. A digital signature based on a conventional encryption function. In *Proceedings of the International Cryptology Conference (CRYPTO'87)*. 369–378.

Gary L. Miller. 1975. Riemann's hypothesis and tests for primality. In *Proceedings of 7th Annual ACM Symposium on Theory of Computing (STOC'75)*. ACM, New York, NY, 234–239.

Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. 2002. Ivy: A read/write peer-to-peer file system. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI'02)*. ACM, New York, NY, 31–44.

Moni Naor and Kobbi Nissim. 1998. Certificate revocation and certificate update. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7 (SSYM'98)*. USENIX Association, Berkeley, CA.

Moni Naor and Guy N. Rothblum. 2005. The complexity of online memory checking. In *Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS'05)*. IEEE Computer Society, Washington, DC, 573–584.

A. Oprea, M. K. Reiter, and K. Yang. 2005. Space-efficient block storage integrity. In *NDSS*.

Charalampos Papamanthou and Roberto Tamassia. 2007. Time and space efficient algorithms for two-party authenticated data structures. In *Proceedings of the 9th International Conference on Information and Communications Security (ICICS'07)*. Springer-Verlag, Berlin, 1–15.

Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. 2008. Authenticated hash tables. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS'08)*. ACM, New York, NY, 437–448.

William Pugh. 1990. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM* 33, 6 (June 1990), 668–676.

I. Reed and G. Solomon. 1960. Polynomial codes over certain finite fields. *Journal of SIAM* 8 (1960), 300–304.

Thomas S. J. Schwarz and Ethan L. Miller. 2006. Store, forget, and check: Using algebraic signatures to check remotely administered storage. In *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems (ICDCS'06)*. IEEE Computer Society, Washington, DC, USA.

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

Hovav Shacham and Brent Waters. 2013. Compact proofs of retrievability. *Journal of Cryptology* 26, 3 (2013), 442–483. DOI:http://dx.doi.org/10.1007/s00145-012-9129-2

Mehul A. Shah, Ram Swaminathan, and Mary Baker. 2008. *Privacy-Preserving Audit and Extraction of Digital Contents*. Technical Report. HP Labs Technical Report No. HPL-2008-32.

Elaine Shi, Emil Stefanov, and Charalampos Papamanthou. 2013. Practical dynamic proofs of retrievability. In *Proceedings of the ACM International Conference on Computer and Communications Security (CCS'13)*. 325–336.

Emil Stefanov, Marten van Dijk, Ari Juels, and Alina Oprea. 2012. Iris: A scalable cloud file system with efficient integrity checks. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC'12)*. ACM, New York, NY, 229–238.

Roberto Tamassia. 2003. Authenticated data structures. In *Proceedings of the European Symposium on Algorithms (ESA'03)*. 2–5.

Roberto Tamassia and Nikos Triandopoulos. 2005. Computational bounds on hierarchical data processing with applications to information security. In *Proceedings of the 32nd International Conference on Automata, Languages and Programming (ICALP'05)*. Springer-Verlag, Berlin, 153–165.

Cong Wang, Qian Wang, Kui Ren, and Wenjing Lou. 2010. Privacy-preserving public auditing for data storage security in cloud computing. In *Proceedings of the 29th Conference on Information Communications (INFOCOM'10)*. IEEE Press, Piscataway, NJ, 525–533.

Qian Wang, Cong Wang, Jin Li, Kui Ren, and Wenjing Lou. 2009. Enabling public veriability and data dynamics for storage security in cloud computing. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS'09)*. 355–370.

Qingji Zheng and Shouhuai Xu. 2011. Fair and dynamic proofs of retrievability. In *Proceedings of the 1st ACM Conference on Data and Application Security and Privacy (CODASPY'11)*. ACM, New York, NY, 237–248.

Received February 2014; revised August 2014; accepted November 2014