DeyPoS: Deduplicatable Dynamic Proof of Storage for Multi-User Environments

Kun He, Jing Chen, Ruiying Du, Qianhong Wu, Guoliang Xue, and Xiang Zhang

Abstract—Dynamic Proof of Storage (PoS) is a useful cryptographic primitive that enables a user to check the integrity of outsourced files and to efficiently update the files in a cloud server. Although researchers have proposed many dynamic PoS schemes in single-user environments, the problem in multi-user environments has not been investigated sufficiently. A practical multi-user cloud storage system needs the secure client-side cross-user deduplication technique, which allows a user to skip the uploading process and obtain the ownership of the files immediately, when other owners of the same files have uploaded them to the cloud server. To the best of our knowledge, none of the existing dynamic PoSs can support this technique. In this paper, we introduce the concept of deduplicatable dynamic proof of storage and propose an efficient construction called DeyPoS, to achieve dynamic PoS and secure cross-user deduplication, simultaneously. Considering the challenges of structure diversity and private tag generation, we exploit a novel tool called Homomorphic Authenticated Tree (HAT). We prove the security of our construction, and the theoretical analysis and experimental results show that our construction is efficient in practice.

Index Terms—Cloud storage, dynamic proof of storage, deduplication.

1 INTRODUCTION

STORAGE outsourcing is becoming more and more attractive to both industry and academia due to the advantages of low cost, high accessibility, and easy sharing. As one of the storage outsourcing forms, cloud storage gains wide attention in recent years [1] [2]. Many companies, such as Amazon, Google, and Microsoft, provide their own cloud storage services, where users can upload their files to the servers, access them from various devices, and share them with the others. Although cloud storage services are widely adopted in current days, there still remain many security issues and potential threats [3] [4].

Data integrity is one of the most important properties when a user outsources its files to cloud storage. Users should be convinced that the files stored in the server are not tampered. Traditional techniques for protecting data integrity, such as message authentication codes (MACs) and digital signatures, require users to download all of the files from the cloud server for verification, which incurs a heavy communication cost [5]. These techniques are not suitable for cloud storage services where users may check the integrity frequently, such as every hour [6]. Thus, researchers introduced *Proof of Storage* (PoS) [7] for checking the integrity without downloading files from the cloud server. Furthermore, users may also require several dynamic operations, such as modification, insertion, and deletion, to update their files, while maintaining the capability of PoS. Dynamic PoS [8] is proposed for such dynamic operations. In contrast with PoS, dynamic PoS employs *authenticated structures* [9], such as the Merkle tree [10]. Thus, when dynamic operations are executed, users regenerate *tags* (which are used for integrity checking, such as MACs and signatures) for the updated blocks only, instead of regenerating for all blocks.

To better understand the following contents, we present more details about PoS and dynamic PoS. In these schemes [5] [8] [11], each block of a file is attached a (cryptographic) tag which is used for verifying the integrity of that block. When a verifier wants to check the integrity of a file, it randomly selects some block indexes of the file, and sends them to the cloud server. According to these challenged indexes, the cloud server returns the corresponding blocks along with their tags. The verifier checks the block integrity and index correctness. The former can be directly guaranteed by cryptographic tags. How to deal with the latter is the major difference between PoS and dynamic PoS. In most of the PoS schemes [5] [11] [12], the block index is "encoded" into its tag, which means the verifier can check the block integrity and index correctness simultaneously. However, dynamic PoS cannot encode the block indexes into tags, since the dynamic operations may change many indexes of non-updated blocks, which incurs unnecessary computation and communication cost. For example, there is a file consisting of 1000 blocks, and a new block is inserted behind the second block of the file. Then, 998 block indexes of the original file are changed, which means the user has to generate and send 999 tags for this update. Authenticated structures are introduced in dynamic PoSs [8] [13] [14] to solve this challenge. As a result, the tags are attached to the authenticated structure rather than the block indexes. Taking the Merkle tree in Fig. 1a as an example (Merkle tree is one of the most efficient authenticated structures in dynamic PoS [14]), the tag corresponding to the second file block involves the index of the Merkle tree node ν_5, that is 5, rather than 2. When a new block is inserted behind the second file block, the authenticated structure turns into the

- *K. He, J. Chen and R. Du are with State Key Laboratory of Software Engineering, Computer School, Wuhan University, Wuhan 430072, China. E-mail: chenjing@whu.edu.cn*
- *Q. Wu is with the Computer School, Beihang University, Beijing, China.*
- *G. Xue and X. Zhang are with the School of Computing, Informatics, and Decision Systems Engineering, Arizona State University, AZ, USA.*

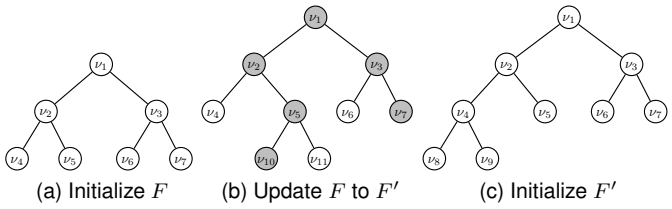


Fig. 1. An Overview of Tree-based Authenticated Structures

structure in Fig. 1b. Then, the index in the tag corresponding to the second file block changes, and the user only has to generate 2 tags for this update. This figure provides an instance that authenticated structure used in dynamic PoS reduces the computation cost in the update process.

However, dynamic PoS remains to be improved in a multi-user environment, due to the requirement of *cross-user deduplication* on the client-side [15]. This indicates that users can skip the uploading process and obtain the ownership of files immediately, as long as the uploaded files already exist in the cloud server. This technique can reduce storage space for the cloud server [16], and save transmission bandwidth for users. To the best of our knowledge, there is no dynamic PoS that can support secure cross-user deduplication.

There are two challenges in order to solve this problem. On one hand, the authenticated structures used in dynamic PoSs, such as skip list [8] and Merkle tree [14], are not suitable for deduplication. We call this challenge *structure diversity*, which means the authenticated structure of a file in dynamic PoS may have some conflicts. For instance, the authenticated structure of a file F is shown in Fig. 1a. When the file is updated to F', the authenticated structure stored on the server-side may turn into the structure in Fig. 1b. However, an owner who intends to upload F' usually generates a structure as shown in Fig. 1c, which is different from the structure stored in the cloud server. Thus, the owner cannot execute deduplication unless the owner and the cloud server synchronize the authenticated structure. On the other hand, even if cross-user deduplication is achieved (for example, the cloud server sends the entire authenticated structure to the owner), *private tag generation* is still a challenge for dynamic operations. In most of the existing dynamic PoSs, a tag used for integrity verification is generated by the secret key of the uploader. Thus, other owners who have the ownership of the file but have not uploaded it due to the cross-user deduplication on the client-side, cannot generate a new tag when they update the file. In this situation, the dynamic PoSs would fail.

If we take dynamic PoS and cross-user deduplication on the client-side as orthogonal issues, we may simply combine the existing dynamic PoS schemes and deduplication techniques. Then, structure diversity is solved via deduplication scheme. For solving private tag generation, each owner can generate its own authenticated structure and upload the structure to the cloud server, which means that the cloud server stores multiple authenticated structures for each file. Also, when a file is updated by a user, the cloud server has to update the corresponding authenticated structure in dynamic PoS, and construct a new authenticated structure for deduplication. As a result, this trivial combination introduces unnecessary computation and storage cost to the cloud server. Taking the combination of [10] and [15] as example, [10] is a dynamic PoS scheme which employs Merkle tree as its authenticated structure, and [15] is a cross-user deduplication scheme which also employs Merkle tree as its authenticated structure. Suppose Alice and Bob independently own a file F, a Merkle tree T_F is generated and stored by the cloud server for deduplication, and two Merkle trees T_A and T_B are generated by Alice and Bob respectively, and stored in the cloud server for PoS. When Alice updates F to F', the cloud server updates T_A to T'_A for PoS and generates a new Merkle tree $T_{F'}$ for deduplication. Thus, the number of Merkle trees grows with the version numbers and the number of owners, which is 4 (T_F, T'_A, T_B, and $T_{F'}$) in the above example. Also, the cloud server has to generate two Merkle trees in the above example which is more time-consuming than update the Merkle trees. As a summary, existing dynamic PoSs cannot be extended to the multi-user environment.

1.1 Related Work

The concept of proof of storage was introduced by Ateniese *et al.* [5], and Juels and Kaliski [17], respectively. The main idea of PoS is to randomly choose a few data blocks as the challenge. Then, the cloud server returns the challenged data blocks and their tags as the response. Since the data blocks and the tags can be combined via homomorphic functions, the communication costs are reduced. The subsequent works [11] [18] [19] [20] [21] [22] [23] [24] [25] extended the research of PoS, but those works did not take dynamic operations into account. Erway *et al.* [8] and later works [13] [14] [26] [27] [28] [29] focused on the dynamic data. Among them, the scheme in [14] is the most efficient solution in practice. However, the scheme is stateful, which requires users to maintain some state information of their own files locally. Hence, it is not appropriate for a multi-user environment.

Halevi *et al.* [15] introduced the concept of proof of ownership which is a solution of cross-user deduplication on the client-side. It requires that the user can generate the Merkle tree without the help from the cloud server, which is a big challenge in dynamic PoS. Pietro and Sorniotti [30] proposed another proof of ownership scheme which improves the efficiency. Xu *et al.* [31] proposed a client-side deduplication scheme for encrypted data, but the scheme employs a deterministic proof algorithm which indicates that every file has a deterministic short proof. Thus, anyone who obtains this proof can pass the verification without possessing the file locally. Other deduplication schemes for encrypted data [32] [33] [34] were proposed for enhancing the security and efficiency. Note that, all existing techniques for cross-user deduplication on the client-side were designed for static files. Once the files are updated, the cloud server has to regenerate the complete authenticated structures for these files, which causes heavy computation cost on the server-side.

Zheng and Xu [35] proposed a solution called proof of storage with deduplication, which is the first attempt to design a PoS scheme with deduplication. Du *et al.* [36] introduced proofs of ownership and retrievability, which are

similar to [35] but more efficient in terms of computation cost. Note that neither [35] nor [36] can support dynamic operations. Due to the problem of structure diversity and private tag generation, [35] and [36] cannot be extended to dynamic PoS.

Wang *et al.* [37] [38], and Yuan and Yu [39] considered proof of storage for multi-user updates, but those schemes focus on the problem of sharing files in a group. Deduplication in these scenarios is to deduplicate files among different groups. Unfortunately, these schemes cannot support deduplication due to structure diversity and private tag generation. In this paper, we consider a more general situation that every user has its own files separately. Hence, we focus on a deduplicatable dynamic PoS scheme in multi-user environments.

The major techniques used in PoS and dynamic PoS schemes are homomorphic Message Authentication Codes [40] and homomorphic signatures [41] [42]. With the help of homomorphism, the messages and MACs/signatures in these schemes can be compressed into a single message and a single MAC/signature. Therefore, the communication cost can be dramatically reduced. These techniques have been used in PoS [7] [14] [18] and secure network coding [43] [44] [45]. A brief survey of homomorphic MACs and signatures could be referred in [46].

1.2 Contributions

The main contributions of this paper are as follows.

1) To the best of our knowledge, this is the first work to introduce a primitive called *deduplicatable dynamic Proof of Storage* (deduplicatable dynamic PoS), which solves the structure diversity and private tag generation challenges.
2) In contrast to the existing authenticated structures, such as skip list [8] and Merkle tree [14], we design a novel authenticated structure called *Homomorphic Authenticated Tree* (HAT), to reduce the communication cost in both the proof of storage phase and the deduplication phase with similar computation cost. Note that HAT can support integrity verification, dynamic operations, and cross-user deduplication with good consistency.
3) We propose and implement the first efficient construction of deduplicatable dynamic PoS called DeyPoS, which supports unlimited number of verification and update operations. The security of this construction is proved in the random oracle model, and the performance is analyzed theoretically and experimentally.

1.3 Organization

The rest of this paper is organized as follows. In Section 2, we introduce the models of deduplicatable dynamic proof of storage. An authenticated structure, called HAT, is designed in Section 3. In Section 4, we propose a concrete scheme, named DeyPoS. The security analysis and performance evaluation results are presented in Section 5 and Section 6, respectively. In the last section, we conclude this paper.

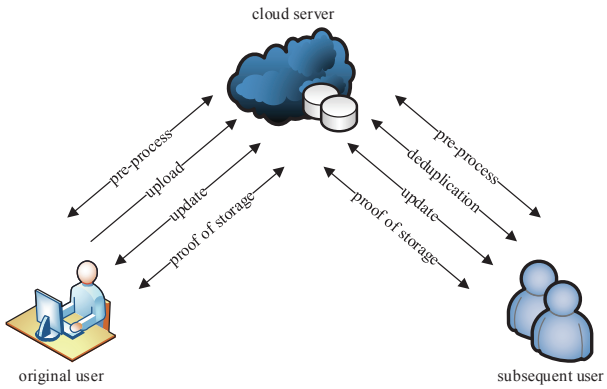


Fig. 2. The system model of deduplicatable dynamic PoS

2 Deduplicatable Dynamic PoS

As discussed in Section 1, no trivial extension of dynamic PoS can achieve cross-user deduplication. To fill this void, we present a novel primitive called deduplicatable dynamic proof of storage in this section.

2.1 System Model

Our system model considers two types of entities: the cloud server and users, as shown in Fig. 2. For each file, *original user* is the user who uploaded the file to the cloud server, while *subsequent user* is the user who proved the ownership of the file but did not actually upload the file to the cloud server. There are five phases in a deduplicatable dynamic PoS system: *pre-process*, *upload*, *deduplication*, *update*, and *proof of storage*.

In the *pre-process* phase, users intend to upload their local files. The cloud server decides whether these files should be uploaded. If the upload process is granted, go into the upload phase; otherwise, go into the deduplication phase.

In the *upload* phase, the files to be uploaded do not exist in the cloud server. The original users encodes the local files and upload them to the cloud server.

In the *deduplication* phase, the files to be uploaded already exist in the cloud server. The subsequent users possess the files locally and the cloud server stores the authenticated structures of the files. Subsequent users need to convince the cloud server that they own the files without uploading them to the cloud server.

Note that, these three phases (pre-process, upload, and deduplication) are executed only once in the life cycle of a file from the perspective of users. That is, these three phases appear only when users intend to upload files. If these phases terminate normally, i.e., users finish uploading in the upload phase, or they pass the verification in the deduplication phase, we say that the users have the ownerships of the files.

In the *update* phase, users may modify, insert, or delete some blocks of the files. Then, they update the corresponding parts of the encoded files and the authenticated structures in the cloud server, even the original files were not uploaded by themselves. Note that, users can update the files only if they have the ownerships of the files, which means that the users should upload the files in the upload phase or pass the verification in the deduplication

phase. For each update, the cloud server has to reserve the original file and the authenticated structure if there exist other owners, and record the updated part of the file and the authenticated structure. This enables users to update a file concurrently in our model, since each update is only "attached" to the original file and authenticated structure.

In the *proof of storage* phase, users only possess a small constant size metadata locally and they want to check whether the files are faithfully stored in the cloud server without downloading them. The files may not be uploaded by these users, but they pass the deduplication phase and prove that they have the ownerships of the files.

Note that, the update phase and the proof of storage phase can be executed multiple times in the life cycle of a file. Once the ownership is verified, the users can arbitrarily enter the update phase and the proof of storage phase without keeping the original files locally.

2.2 Threat Model

We present the threat model briefly as follows. The cloud server and users do not fully trust each other. A malicious user may cheat the cloud server by claiming that it has a certain file, but it actually does not have it or only possesses parts of the file. A malicious cloud server may try to convince users that it faithfully stores files and updates them, whereas the files are damaged or not up-to-date. The goal of deduplicatable dynamic PoS is to detect these misbehaviors with overwhelming probability. The formal threat model is described in Section 2.4 via various security definitions.

2.3 Syntax and Correctness

The deduplicatable dynamic PoS is a comprehensive storage outsourcing approach which establishes mutual confidence between users and the cloud server in a multi-user environment. In this subsection, we give the definition of deduplicatable dynamic PoS and describe its syntax. We also present five algorithms which correspond to the five phases in our system model.

If Alg is a probabilistic polynomial-time (PPT) algorithm, we denote the action of running Alg on input x and assigning the output to the variable y by $y \leftarrow \mathrm{Alg}(x)$. If $\mathrm{Protocol}$ is a two-party protocol, we denote the action of running $\mathrm{Protocol}$ between two entities $\mathcal{A}$ and $\mathcal{B}$ by $\mathrm{Protocol}\langle\mathcal{A}(x_1), \mathcal{B}(x_2)\rangle$, where the corresponding algorithm $\mathcal{A}$ takes as input x_1 and the corresponding algorithm $\mathcal{B}$ takes as input x_2. Function $\epsilon : \mathbb{Z}^+ \to \mathbb{R}$ is negligible if for every positive $\mu \in \mathbb{R}$, there exists an integer N_μ such that $\epsilon(x) < x^{-\mu}$ for all $x > N_\mu$.

Definition 1. *A deduplicatable dynamic PoS scheme consists of the following two polynomial time algorithms and three polynomial time protocols:*

- $(id, e) \leftarrow \mathrm{Init}(1^\lambda, F)$. *This deterministic initialization algorithm is run by a user. It takes as input a security parameter λ and an original file F, and outputs a public identity id and a secret metadata e.*
- $(C, T) \leftarrow \mathrm{Encode}(e, F)$. *This encoding algorithm is run by a user. It takes as input the metadata e and the original file F, and outputs an encoded file C and a corresponding authenticator T.*
- $res \in \{0, 1\} \leftarrow \mathrm{Deduplicate}\langle\mathcal{U}(e, F), \mathcal{S}(T)\rangle$. *This randomized deduplication protocol is run between a user $\mathcal{U}$ and a cloud server $\mathcal{S}$. $\mathcal{U}$ takes as input the metadata e and the original file F. $\mathcal{S}$ takes as input the authenticator T. The protocol outputs 1 if the user convinces the cloud server that it possesses the complete file F locally, and 0 otherwise.*
- $res \in \{\langle e^*, (C^*, T^*)\rangle, \perp\} \leftarrow \mathrm{Update}\langle\mathcal{U}(e, \iota, m, \mathrm{OP}), \mathcal{S}(C, T)\rangle$. *This randomized update protocol is run between a user $\mathcal{U}$ and a cloud server $\mathcal{S}$. $\mathcal{U}$ takes as input the metadata e, a block index ι, an updated block m, and an operation mode OP. $\mathcal{S}$ takes as input the encoded file C and the authenticator T. If the operation succeeds, $\mathcal{U}$ outputs a new metadata e^*, and $\mathcal{S}$ outputs a new encoded file C^* and a new authenticator T^*; otherwise both $\mathcal{U}$ and $\mathcal{S}$ output $\perp$.*
- $res \in \{0, 1\} \leftarrow \mathrm{Check}\langle\mathcal{S}(C, T), \mathcal{U}(e)\rangle$. *This randomized checking protocol is run between a cloud server $\mathcal{S}$ and a user $\mathcal{U}$. $\mathcal{S}$ takes as input the encoded file C and the authenticator T. $\mathcal{U}$ takes as input the metadata e. The protocol outputs 1 if the cloud server convinces the user that C stored in the server is not tampered and is up-to-date, and 0 otherwise.*

Given a file, each user who has the entire original file can obtain the same metadata via the initialization algorithm and pass the deduplication protocol if the file exists in the cloud server. Once a user has uploaded the file or passed the deduplication protocol, it can prove to the cloud server that it has the ownership of the file, and may delete the file from its local storage. No matter who runs the encoding algorithm and uploads the encoded file to the cloud server, the user can run the update protocol and the checking protocol at any time without possessing the file locally, which indicates that our model is suitable to multi-user environments.

Before describing the following definitions, we first explain the update protocol in detail, in which all dynamic operations (modification, insertion, and deletion) should be supported. The file F consists of a sequence of blocks, that is, $F = (m_1, \ldots, m_n)$. If $\mathrm{OP} = \mathrm{mod}$, the original ι-th block is replaced by m. If $\mathrm{OP} = \mathrm{ins}$, m is inserted in front of the original ι-th block. If $\mathrm{OP} = \mathrm{del}$, the original ι-th block is deleted. Note that our model is different from the one in [39] where all users share a file, and an update must be synchronized among the users. In our model, all users have the ownerships of the same file independently, and the update by one user should not affect the other users. This indicates that the cloud server should store the original version and the updated version of the file simultaneously when the original file has multiple owners. It can be done by employing version control techniques with which our model can easily integrate.

At the end of this subsection, we present the correctness of deduplicatable dynamic PoS as follows.

Definition 2. *A deduplicatable dynamic PoS scheme is* correct *if the following two conditions hold for any positive integer λ, any file F, any metadata e generated from $\mathrm{Init}(1^\lambda, F)$, and any (C, T) generated from $\mathrm{Encode}(e, F)$:*

1) $\Pr[\mathrm{Deduplicate}\langle\mathcal{U}(e, F), \mathcal{S}(C, T)\rangle = 1] \geq 1 - \epsilon_1(\lambda)$,

2) $\Pr[\text{Check}\langle \mathcal{S}(C,T), \mathcal{U}(e)\rangle = 1] \geq 1 - \epsilon_2(\lambda)$,

where F, e, and (C,T) can be updated by the Update *protocol for an arbitrary number of times, and $\epsilon_1(\cdot)$ and $\epsilon_2(\cdot)$ are two negligible functions.*

2.4 Security Definitions

In this subsection, we present security definitions of deduplicatable dynamic PoS. The definitions consist of two parts: *uncheatability* and *unforgeability*.

Uncheatability captures the property of authenticity for cross-user deduplication on the client-side. Unlike the definition in [15], we consider a more complex situation in which the adversary has the original file. We also assume that the source is unpredictable as in [15] which means the files from the source have high min-entropy. Intuitively, we require that the user cannot cheat the cloud server that it possesses the whole file except a negligible probability. Let $\Pi = (\text{Init}, \text{Encode}, \text{Deduplicate}, \text{Update}, \text{Check})$ be a deduplicatable dynamic PoS scheme. The uncheatability experiment $Exp^{uc}_{\mathcal{A},\Pi}$ between an imaginary challenger $\mathcal{C}$ (the cloud server) and an adversary $\mathcal{A}$ (the user) is described as follows:

1) $\mathcal{A}$ chooses an unpredictable source S and a file $F \in S$, and gives (S, F) to $\mathcal{C}$.
2) $\mathcal{C}$ runs $(id, e) \leftarrow \text{Init}(1^\lambda, F)$, $(C,T) \leftarrow \text{Encode}(e, F)$, and sends (id, e) to $\mathcal{A}$.
3) $\mathcal{A}$ can run $\text{Check}\langle \mathcal{C}(C,T), \mathcal{A}(e)\rangle$ with $\mathcal{C}$ for $q(\lambda)$ times where $q(\cdot)$ is a polynomial function.
4) $\mathcal{C}$ runs $(C^*, T^*) \leftarrow \text{Update}\langle \mathcal{C}(e, \iota, m, \text{OP}), \mathcal{C}(C,T)\rangle$, where (ι, m, OP) can update F to another element F^* in S. Then, $\mathcal{C}$ runs $id^* \leftarrow \text{Init}(1^\lambda, F^*)$, and sends id^* to $\mathcal{A}$.
5) $\mathcal{A}$ runs $\text{Deduplicate}\langle \mathcal{A}(id^*), \mathcal{C}(C^*, T^*)\rangle$ with $\mathcal{C}$.
6) The experiment outputs 1 if the Deduplicate protocol in Step 5 outputs 1, and 0 otherwise.

Definition 3. *A deduplicatable dynamic PoS scheme is* uncheatable *if for any PPT adversary $\mathcal{A}$ and any positive integer λ,*

$$\Pr[Exp^{uc}_{\mathcal{A},\Pi}(1^\lambda) = 1] \leq \epsilon(\lambda),$$

where $\epsilon(\cdot)$ is a negligible function.

Unforgeability definition captures the integrity property. As in [7], we require an extractor who can extract the challenged blocks from the response if the verification successes. The difference between Definition 3 and the definition in [7] is that we need to consider the impact of the deduplication phase. The unforgeability experiment $Exp^{uf}_{\mathcal{A},\Pi}$ between an imaginary challenger $\mathcal{C}$ (the user) and an adversary $\mathcal{A}$ (the cloud server) is described as follows:

1) $\mathcal{A}$ chooses an unpredictable source S and sends S to $\mathcal{C}$.
2) $\mathcal{C}$ randomly selects a file $F \in S$, runs $(id, e) \leftarrow \text{Init}(1^\lambda, F)$, $(C,T) \leftarrow \text{Encode}(e, F)$, and sends $(id, (C,T))$ to $\mathcal{A}$.
3) $\mathcal{C}$ runs $(C^*, T^*) \leftarrow \text{Update}\langle \mathcal{C}(e, \iota, m, \text{OP}), \mathcal{C}(C,T)\rangle$, where (ι, m, OP) can update F to another element F^* in S. Then, $\mathcal{C}$ runs $id^* \leftarrow \text{Init}(1^\lambda, F^*)$, and sends id^* to $\mathcal{A}$.
4) $\mathcal{A}$ can run $\text{Deduplicate}\langle \mathcal{C}(e^*, F^*), \mathcal{A}(id^*)\rangle$ with $\mathcal{C}$ for $q(\lambda)$ times, where $q(\cdot)$ is a polynomial function.
5) $\mathcal{A}$ runs $\text{Check}\langle \mathcal{A}(id^*), \mathcal{C}(e^*)\rangle$ with $\mathcal{C}$.
6) The experiment outputs 1 if the Check protocol in Step 5 outputs 1, and 0 otherwise.

Definition 4. *A deduplicatable dynamic PoS scheme is* unforgeable *if for any PPT adversary $\mathcal{A}$ with non-negligible probability $\Pr[Exp^{uf}_{\mathcal{A},\Pi}(1^\lambda) = 1]$, there exists an extractor who can recover C^* from $\mathcal{A}$ except negligible probability.*

3 Homomorphic Authenticated Tree

3.1 Overview

To implement an efficient deduplicatable dynamic PoS scheme, we design a novel authenticated structure called *homomorphic authenticated tree* (HAT). A HAT is a binary tree in which each leaf node corresponds to a data block. Though HAT does not have any limitation on the number of data blocks, for the sake of description simplicity, we assume that the number of data blocks n is equal to the number of leaf nodes in a full binary tree. Thus, for a file $F = (m_1, m_2, m_3, m_4)$ where m_ι represents the ι-th block of the file, we can construct a tree as shown in Fig. 1a.

Each node in HAT consists of a four-tuple $\nu_i = (i, l_i, v_i, t_i)$. i is the unique index of the node. The index of the root node is 1, and the indexes increases from top to bottom and from left to right. l_i denotes the number of leaf nodes that can be reached from the i-th node. v_i is the version number of the i-th node. t_i represents the tag of the i-th node. When a HAT is initialized, the version number of each leaf is 1, and the version number of each non-leaf node is the sum of that of its two children. For the i-th node, m_i denotes the combination of the blocks corresponding to its leaves. The tag t_i is computed from $\mathcal{F}(m_i)$, where $\mathcal{F}$ denotes a tag generation function. We require that for any node ν_i and its children ν_{2i} and ν_{2i+1}, $\mathcal{F}(m_i) = \mathcal{F}(m_{2i} \odot m_{2i+1}) = \mathcal{F}(m_{2i}) \otimes \mathcal{F}(m_{2i+1})$ holds, where $\odot$ denotes the combination of m_{2i} and m_{2i+1}, and $\otimes$ indicates the combination of $\mathcal{F}(m_{2i})$ and $\mathcal{F}(m_{2i+1})$, which is why we call it a "homomorphic" tree. An implementation of the tag generation function is described in Section 4.3.

3.2 Path and Sibling Search

To facilitate operations on HAT structures, we exploit two major algorithms for path search and sibling search.

We define the path search algorithm $\rho_\iota \leftarrow \text{Path}(T, \iota)$. It takes a HAT T and a block index ι of a file as input, and outputs the index set of nodes in the path from the root node to the ι-th leaf node among all the leaves which corresponds to the ι-th block of the file. We extend the path search algorithm to support multi-path search as Algorithm 1, where the i-th node in T consists of $\nu_i = (i, l_i, v_i, t_i)$. The algorithm takes as input a HAT and an ordered list of the block indexes, and outputs an ordered list of the node indexes. Lines 2-5 initialize two auxiliary variables for each legal block index ι where i_ι defines a subtree whose root is the i_ι-th node in T, and ord_ι indicates the location of the corresponding leaf node in that subtree. Line 6 initializes a path ρ and a state st. The loop of lines 7-18 calculates the node that should be inserted into ρ by breadth-first search.

Algorithm 1 Path search algorithm

```
1: procedure PATH(T, I)
2:     for ι ∈ I do
3:         if ι > l_1 then
4:             return 0
5:         i_ι ← 1, ord_ι ← ι
6:     ρ ← {1}, st ← TRUE
7:     while st do
8:         st ← FALSE
9:         for ι ∈ I do
10:            if l_{i_ι} = 1 then
11:                continue
12:            else if ord_ι ≤ l_{2i_ι} then
13:                i_ι ← 2i_ι
14:            else
15:                ord_ι ← ord_ι − l_{2i_ι}, i_ι ← 2i_ι + 1
16:            ρ ← ρ ∪ {i_ι}
17:            if l_{i_ι} > 1 then
18:                st ← TRUE
19:    return ρ
```

Algorithm 2 Sibling search algorithm

```
1: procedure SIBLING(ρ)
2:     ψ ← ∅, ρ ← ρ \ {1}, ϱ ← ∅, i ← 1
3:     while ρ ≠ ∅ ∨ ϱ ≠ ∅ do
4:         if 2i ∈ ρ then
5:             i ← 2i, ρ ← ρ \ {i}
6:             if i + 1 ∈ ρ then
7:                 ϱ ← ϱ ∪ {(i + 1, FALSE)}, ρ ← ρ \ {i + 1}
8:             else
9:                 ϱ ← ϱ ∪ {(i + 1, TRUE)}
10:        else if 2i + 1 ∈ ρ then
11:            i ← 2i + 1, ρ ← ρ \ {i}, ψ ← ψ ∪ {i − 1}
12:        else if ϱ ≠ ∅ then
13:            pop the last inserted (α, β) in ϱ
14:            i ← α
15:            if β = TRUE then
16:                ψ ← ψ ∪ {i}
17:    return ψ
```

For each level of T, the loop of lines 9-18 calculates the node in ρ for each block index ι. For example, the path (gray nodes) to the 2nd leaf (the 10th node in the HAT) and the 5th leaf (the 7th node in the HAT) in Fig. 1b is $\rho_{2,5} = \mathrm{Path}(T, \{2, 5\}) = \{1, 2, 3, 5, 7, 10\}$.

We define the sibling search algorithm $\psi \leftarrow \mathrm{Sibling}(\rho)$ as Algorithm 2. It takes the path ρ as input, and outputs the index set of the siblings of all nodes in the path ρ. Note that, the output of the sibling search algorithm is not an ordered list. It always outputs the leftmost one in the remaining siblings. Line 2 initializes the sibling set ψ and an auxiliary set ϱ. The loop of lines 3-16 first determines how many children of a node in ρ also appears in ρ (line 4, 6, and 10). If the answer is two, the algorithm removes these children from ρ and inserts the right child into ϱ for further validation (line 4-7). If the answer is one, the algorithm removes this child from ρ and inserts the other child into the sibling set ψ (line 8-11). However, there is a subtle difference between the left child is in ρ and the right child is in ρ. In the former, the right child will be inserted into ψ (line 15-16) later, while the left child will be immediately inserted into ψ (line 11) in the latter since the left child is the leftmost sibling in the remaining siblings. Lines 12-16 process the node in ϱ. From Fig. 1b, we have $\psi = \mathrm{Sibling}(\rho_{2,5}) = \{4, 11, 6\}$.

From Algorithm 1 and Algorithm 2, it is clear that both the path search algorithm and the sibling search algorithm have the same computation complexity $O(b \log(n))$, where b is the number of block indexes (i.e., the size of $\mathcal{I}$) and n is the number of leaf nodes.

3.3 Node Update

As mentioned in Section 2, there are three dynamic operations: modification, insertion, and deletion. We describe how HAT supports these operations with a single data block. The method is similar to the one used when multiple data blocks are updated.

When a user modifies the ι-th data block, it finds a path $\rho = \mathrm{Path}(T, \iota)$ and obtains the siblings $\psi = \mathrm{Sibling}(\rho)$. Each version number of the nodes in path ρ increases by 1, and each tag of the nodes in path ρ is updated with the help of the modified block and the node value $\{\nu_i \mid i \in \psi\}$. For example, if the 5th data block is modified in Fig. 1b, the values of the nodes in path $\rho = \{1, 3, 7\}$ (i.e., the 1st node, the 3rd node, and the 7th node) should be updated.

To insert a block in front of the ι-th block, the user finds a path $\rho = \mathrm{Path}(T, \iota)$ and obtains the siblings $\psi = \mathrm{Sibling}(\rho)$. Assume that the index of the leaf is i_ι. The user generates two children under the i_ι-th node. The version numbers of the children under the i_ι-th node are set as v_{i_ι}, and each version number of the nodes in path ρ is computed via their children. The inserted block corresponds to the $2i_\iota$-th node (namely the left child), and the original ι-th block is migrated to the $(2i_\iota + 1)$-th node (namely the right child). Then, the user computes all affected node tags in the modification operation. If a data block is inserted in front of the 2nd data block in Fig. 1a, the 10th and the 11th node will be inserted into the tree as shown in Fig. 1b, and the values of the nodes in $\rho \cup \{10, 11\} = \{1, 2, 5, 10, 11\}$ should be updated.

The deletion operation is similar to the previous two operations. When a user deletes the ι-th data block, it first finds a path $\rho = \mathrm{Path}(T, \iota)$ and obtains the siblings $\psi = \mathrm{Sibling}(\rho)$. The version number of nodes in path ρ increases by 1. The user moves the sibling of the deleted leaf node to its parent, and computes all tags of the affected nodes. If the 5th data block is deleted in Fig. 1b, the 6th node will be migrated to the 3rd node, and the values of the nodes in $\{1, 3\}$ should be updated.

3.4 Structure Analysis

Both skip list and Merkle tree are the classical structures in dynamic PoSs. Since there is no deduplication scheme based on skip list and the asymptotic performance of skip list is similar with that of Merkle tree in dynamic PoSs [8], [10], we only discuss the Merkle tree in our paper.

Let $F = (m_1, m_2, m_3, m_4)$, then the initial authenticated structure (HAT or Merkle tree) in dynamic PoSs or deduplication schemes is shown in Fig. 1a. If the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2560812, IEEE Transactions on Computers

owner Alice inserts m_5 before m_2 in dynamic PoSs, i.e., $F' = (m_1, m_5, m_2, m_3, m_4)$, the authenticated structure in Fig. 1a is transformed into the tree in Fig. 1b. However, another owner Bob who possesses F' as its original file can only generate the authenticated structure shown in Fig. 1c. Though F' (which is updated by Alice) already exists in the cloud server, Bob cannot execute deduplication if the deduplication scheme employs Merkle tree [15]. For example, to prove the integrity of m_3, the prover in [15] need to transmit ν_2 and ν_7, who are the siblings of the path from the root to ν_6, to the verifier. Since ν_2 in Fig. 1b is different from ν_2 in Fig. 1c, the cloud server has to send all leaf positions to the user for constructing ν_2 and ν_7, which costs $O(n)$ bandwidth. Thus, Merkle tree is not suitable for deduplication in dynamic PoS due to the structure diversity.

The aim of HAT is to reduce the communication cost in deduplication to $O(b\log(n))$, where b is the number of the challenged blocks. To this end, we make ν_2 in Fig. 1b and ν_2 in Fig. 1c have the same value by employing the homomorphic technique. With a careful design, this homomorphic property does not weaken the security of dynamic PoS and deduplication.

4 The Construction of DeyPoS

In this section, we propose a concrete scheme of deduplicatable dynamic PoS called DeyPoS. It consists of five algorithms as described in Section 2: Init, Encode, Deduplicate, Update, and Check.

4.1 Building Blocks

We employ the following tools as our building blocks:

1) **Collision-resistant hash functions**: A hash function $\mathrm{H} : \{0,1\}^* \to \{0,1\}^*$ is collision-resistant if the probability of finding two diffenent values x and y that satisfy $\mathrm{H}(x) = \mathrm{H}(y)$ is negligible.
2) **Deterministic symmetric encryption**: The encryption algorithm takes a key k and a plaintext m as input, and outputs the ciphertext. We use the notation $\mathrm{Enc}_k(m)$ to denote the encryption algorithm.
3) **Hash-based message authentication codes**: A hash-based message authentication code HMAC : $\{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ is a deterministic function that takes a key k and an input x, and outputs a value y. We define $\mathrm{HMAC}_k(x) \stackrel{def}{=} \mathrm{HMAC}(k, x)$.
4) **Pseudorandom functions**: A pseudorandom function $f : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ is a deterministic function that takes a key k and a value x, and outputs a value y that is indistinguishable from a truly random function of the same input x. We define $f_k(x) \stackrel{def}{=} f(k, x)$.
5) **Pseudorandom permutations**: A pseudorandom permutation $\pi : \{0,1\}^* \times [1,n] \to [1,n]$ is a deterministic function that takes a key k and an integer x where $1 \le x \le n$, and outputs a value y where $1 \le y \le n$ that is indistinguishable from a truly random permutation of the same input x. We define $\pi_k(x) \stackrel{def}{=} \pi(k, x)$.

Algorithm 3 The tag generation algorithm for a leaf node

```
1: procedure LeafTag(α_s, k_c, α_c, c_ι, i_ι, l_{i_ι}, v_{i_ι})
2:     τ_ι ← α_s c_ι
3:     t_{i_ι} ← f_{k_c}(i_ι ‖ l_{i_ι} ‖ v_{i_ι}) + α_c τ_ι
4:     return τ_ι, t_{i_ι}
```

Algorithm 4 The tag generation algorithm for a non-leaf node

```
1: procedure NonLeafTag(k_c, i, l_i, v_i)
2:     τ_{2i} ← t_{2i} − f_{k_c}(2i ‖ l_{2i} ‖ v_{2i})
3:     τ_{2i+1} ← t_{2i+1} − f_{k_c}(2i+1 ‖ l_{2i+1} ‖ v_{2i+1})
4:     return t_i ← f_{k_c}(i ‖ l_i ‖ v_i) + τ_{2i} + τ_{2i+1}
```

6) **Key derivation functions**: A key derivation function $\mathrm{KDF} : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ is a deterministic function that can derive a secret key from two secret values.

4.2 The Pre-process Phase

In the pre-process phase, a user runs the initialization algorithm $(id, e) \leftarrow \mathrm{Init}(1^\lambda, F)$ which computes:

$$e \leftarrow \mathrm{H}(F), id \leftarrow \mathrm{H}(e).$$

Then, the user announces that it has a certain file via id. If the file does not exist, the user goes into the upload phase. Otherwise, the user goes into the deduplication phase.

4.3 The Upload Phase

Let the file $F = (m_1, \ldots, m_n)$. The user first invokes the encoding algorithm $(C, T) \leftarrow \mathrm{Encode}(e, F)$ which is executed as follows.

1) Generate a random key $k \leftarrow \{0,1\}^{|e|}$, and compute $r \leftarrow k \oplus e$.
2) Compute an encryption key $k_e \leftarrow \mathrm{KDF}(k, 0)$. For each block m_ι $(1 \le \iota \le n)$, compute $c_\iota \leftarrow \mathrm{Enc}_{k_e}(m_\iota)$.
3) Build a HAT from F where the tags in HAT are unassigned.
4) Compute $\alpha_s \leftarrow \mathrm{KDF}(k, 1)$, $k_c \leftarrow \mathrm{KDF}(k, 2)$, and $\alpha_c \leftarrow \mathrm{KDF}(k, 3)$. Compute all tags of leaf nodes in HAT with $(c_1, \ldots, c_n)$ via Algorithm 3.
5) Based on the tags of leaf nodes, compute all tags of non-leaf nodes in HAT via Algorithm 4.
6) Compute $\omega \leftarrow \mathrm{HMAC}_e(t_1)$.
7) Set $C = \{c_1, \ldots, c_n\}$ and $T = (r, \alpha_s, \mathcal{T}, \omega, \mathcal{N})$, where $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$ and $\mathcal{N} = \{\nu_1, \nu_2, \ldots\}$ is the set of all HAT nodes.

We use a random key k rather than the hash value e as the encryption key. Thus, the encoding algorithm is probabilistic, which is very important to dynamic operations. Note that identical blocks always lead to the same ciphertext, but it is not a security issue because data confidentiality is not the goal of deduplicatable dynamic PoS, and the encryption algorithm in our construction is used for protecting e.

Algorithm 3 is designed for computing the tags of leaf nodes. Line 2 randomizes the data block which is used

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2560812, IEEE Transactions on Computers

Algorithm 5 The deduplication proving algorithm

1: **procedure** DEDUPPROVE($\alpha_s, k_c, \alpha_c, \{c_1, \ldots, c_n\}, \mathcal{I}, \mathcal{Q}$)
2: $c \leftarrow 0, t \leftarrow \emptyset, \zeta \leftarrow 1, l \leftarrow 1$
3: **while** $\zeta \leq n$ **do**
4: $\delta \leftarrow 0$
5: **while** $\zeta < \iota_{j_l}$ **do**
6: $\delta \leftarrow \delta + c_\zeta, \zeta \leftarrow \zeta + 1$
7: pop the first element in $\mathcal{Q}$
8: $t \leftarrow t \cup \{f_{k_c}(i \| l_i \| v_i) + \alpha_c \alpha_s \delta\}$
9: $c \leftarrow c + c_\zeta$
10: $l \leftarrow l + 1, \zeta \leftarrow \zeta + 1$
11: **return** c, t

Algorithm 6 The response algorithm

1: **procedure** RESPONSE($\mathcal{I}$)
2: $\rho \leftarrow$PATH($T, \mathcal{I}$), $\psi \leftarrow$SIBLING(ρ)
3: $c \leftarrow 0, t \leftarrow 0$
4: **for** $\iota \in \mathcal{I}$ **do**
5: $c \leftarrow c + c_\iota$
6: **for** $i \in \psi$ **do**
7: $t \leftarrow t + t_i$
8: **return** $resp \leftarrow (c, t, \{v_{i_\iota} \mid \iota \in \mathcal{I}\}, \{(i, l_i, v_i) \mid i \in \psi\})$

for deduplication, and line 3 calculates the tag of the data block which is attached to the node index of HAT i_ι and other information, such as version number. Algorithm 4 is designed for computing the tags of non-leaf nodes. Lines 2-3 calculate the function of its children. Note that, τ_{2i} and τ_{2i+1} satisfy homomorphism. Then, line 4 binds the tag to the node index of HAT and other information.

At the end of the upload phase, the user uploads C and T to the cloud server and only stores e locally. Note that, e is an element of small constant size and can be encrypted and stored in the cloud server. In contrast with [14] that requires users possessing or downloading a structure which has logarithmic size of the number of file blocks, all owners of the file can run the deduplication protocol, the checking protocol, and the update protocol without the complete structure of HAT in our scheme.

4.4 The Deduplication Phase

If a file announced by a user in the pre-process phase exists in the cloud server, the user goes into the deduplication phase and runs the deduplication protocol $res \in \{0,1\} \leftarrow \mathrm{Deduplicate}\langle \mathcal{U}(e, F), \mathcal{S}(T) \rangle$ as follows.

1) $\mathcal{S}$ executes the following instructions.
   a) Choose $b \leftarrow [1, n]$ and $\kappa \leftarrow \{0,1\}^\lambda$. For each j $(1 \leq j \leq b)$, compute $\iota_j \leftarrow \pi_\kappa(b)$.
   b) Compute the path $\rho = \mathrm{Path}(\mathcal{I})$, where $\mathcal{I} = \{\iota_1, \ldots, \iota_b\}$, and the siblings $\psi = \mathrm{Sibling}(\rho)$.
   c) Send $(r, b, \kappa, \mathcal{Q})$ to $\mathcal{U}$, and keep $\mathcal{L}$ local, where $\mathcal{Q}$ is the set of (i, l_i, v_i) and $\mathcal{L}$ is the set of t_i for all $i \in \psi$.
2) $\mathcal{U}$ executes the following instructions.
   a) Compute $k \leftarrow r \oplus e$, $k_e \leftarrow \mathrm{KDF}(k, 0)$, $\alpha_s \leftarrow \mathrm{KDF}(k, 1)$, $k_c \leftarrow \mathrm{KDF}(k, 2)$, and $\alpha_c \leftarrow \mathrm{KDF}(k, 3)$. For each block m_ι $(1 \leq \iota \leq n)$, compute $c_\iota \leftarrow \mathrm{Enc}_{k_e}(m_\iota)$.
   b) For each j $(1 \leq j \leq b)$, compute $\iota_j \leftarrow \pi_\kappa(b)$. Let $\mathcal{I} = \{\iota_{j_1}, \ldots, \iota_{j_b}\}$ be the ordered challenged index set. Compute the proof by Algorithm 5.
   c) Send (c, t) to $\mathcal{S}$.
3) If $\alpha_s c$ equals to $\sum \tau_{\iota_j}$, and t is identical to $\mathcal{L}$, $\mathcal{S}$ outputs 1, otherwise outputs 0.

Algorithm 5 generates a proof for deduplication. Line 2 initializes a proof c and t. The loop of line 3-10 calculates the homomorphic data of unchallenged file blocks (line 4-6) and the homomorphic data of challenged file blocks (line 9), respectively. The algorithm also computes all tags of the nodes in the sibling set ψ (line 7-8). Note that the cloud server only accesses the HAT of the file for deduplication, which avoids unnecessary block access. The computation costs to generate a challenge on the server-side, to generate a proof on the client-side, and to verify the proof on the server-side in this phase are $O(b \log n)$, $O(n)$, and $O(b \log n)$, respectively. The communication cost is $O(b \log n)$.

4.5 The Update Phase

In this phase, a user can arbitrarily update the file, such as modify a block, insert a batch of blocks, and delete some blocks, by invoking the update protocol $res \in \{\langle e^*, (C^*, T^*) \rangle, \perp\} \leftarrow \mathrm{Update}\langle \mathcal{U}(e, \iota, m, \mathrm{OP}), \mathcal{S}(C, T) \rangle$. After all operations are finished, the user uploads the updated blocks of the file and the updated nodes of the HAT to the cloud server as shown in Section 3.3. Then, the user computes the updated metadata e^* and verifies the updated blocks via the checking protocol (described in Section 4.6).

4.6 The Proof of Storage Phase

At any time, users can go into the proof of storage phase if they have the ownerships of the files. The users and the cloud server run the checking protocol $res \in \{0,1\} \leftarrow \mathrm{Check}\langle \mathcal{S}(C, T), \mathcal{U}(e) \rangle$ interactively to check the file integrity in the cloud server as follows.

1) $\mathcal{U}$ chooses $b \in [1, n]$, $\kappa \in \{0,1\}^\lambda$ and sends (b, κ) to $\mathcal{S}$.
2) For each j $(1 \leq j \leq b)$, $\mathcal{S}$ computes $\iota_j \leftarrow \pi_\kappa(b)$. Then, the cloud server invokes the response algorithm in Algorithm 6, where $\mathcal{I} = \{\iota_1, \ldots, \iota_b\}$, and sends the proof $resp$ to $\mathcal{U}$ with (r, ν_1, ω).
3) $\mathcal{U}$ computes $k \leftarrow r \oplus e$, $\alpha_s \leftarrow \mathrm{KDF}(k, 1)$, $k_c \leftarrow \mathrm{KDF}(k, 2)$, and $\alpha_c \leftarrow \mathrm{KDF}(k, 3)$. Then, it verifies ν_1, and invokes the verification algorithm in Algorithm 7 to accomplish the verification. It outputs 1 if verification succeeds and 0 otherwise.

Algorithm 6 generates a proof on the server-side, which consists of the combination of the challenged file blocks (line 4-5) and the combination of corresponding tags (6-7). The algorithm also returns other information about the HAT (line 8), such as the version number of challenged leaf nodes. Algorithm 7 is designed for verifying the proof generated

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2560812, IEEE Transactions on Computers

Algorithm 7 The verification algorithm

1: **procedure** VERIFY($\alpha_s, k_c, \alpha_c, \nu_1, \mathcal{I}, resp$)
2: $ctr \leftarrow 1, \tau \leftarrow 0$
3: **for** $\iota \in \mathcal{I}$ **do**
4: **while** $ctr < \iota$ **do**
5: pop the first element in $\{(i, l_i, v_i) \mid i \in \psi\}$
6: $ctr \leftarrow ctr + l_i, \tau \leftarrow \tau + f_{k_c}(i \| l_i \| v_i)$
7: **if** $ctr \neq \iota$ **then**
8: **return** 0
9: **else**
10: $ctr \leftarrow ctr + 1$
11: **for** $(i, l_i, v_i) \in \{(i, l_i, v_i) \mid i \in \psi\}$ **do**
12: $ctr \leftarrow ctr + l_i, \tau \leftarrow \tau + f_{k_c}(i \| l_i \| v_i)$
13: **if** $ctr \neq n + 1$ **then**
14: **return** 0
15: **else if** $t + f_{k_c}(1 \| l_1 \| v_1) - \tau + \alpha_s \alpha_c c \neq t_1$ **then**
16: **return** 0
17: **else**
18: **return** 1

by Algorithm 6. The loop of line 3-10 first calculates current indexes of file blocks and the combination of tags (line 5-6). If the indexes of file blocks do not match the challenged indexes, the algorithm is terminated (line 7-8). The loop of line 11-12 calculates the remaining indexes and tags. In the end, the algorithm checks whether the number of file blocks is correct (line 13-14), and whether the HAT is authentic and up-to-date (line 15-16). The computation costs to generate challenge on the client-side, to generate a proof on the server-side, and to verify the proof on the client-side in this phase are $O(1)$, $O(b \log n)$, and $O(b \log n)$, respectively. The communication cost is $O(b \log n)$.

5 Security Analysis

In this section, we investigate the security of DeyPoS. As mentioned in Section 2.4, the security consists of two parts: uncheatability and unforgeability. We first examine uncheatability via the following theorem.

Theorem 1. *Let* H *be a random oracle, for any source $\mathcal{S}$ with min-entropy $\mu(\cdot)$ and any PPT adversary who makes $q_H(\cdot)$ queries to* H*, DeyPoS is uncheatable, and* $\Pr[Exp^{uc}_{\mathcal{A},\Pi}(1^\lambda) = 1] \leq (1 + q_H)/2^\mu$*, where $q_H = q_H(\lambda)$, $\mu = \mu(\lambda)$, where λ is the security parameter.*

Proof. The proof begins with the uncheatability experiment $Exp^{uc}_{\mathcal{A},\Pi}$ in Definition 3. In this experiment, the random oracle $\mathrm{H}(X)$ takes as input X, and returns an arbitrary element Y if X is never queried, and $\mathrm{H}(X)$ otherwise. The challenger runs $e \leftarrow \mathrm{H}(F)$ and $id \leftarrow \mathrm{H}(e)$, and sends (e, id) to the adversary. Then, F is updated to F^*, and the challenger sends $id^* \leftarrow \mathrm{H}(\mathrm{H}(F^*))$ to the adversary. The advantage that the adversary wins is

$$\Pr[Exp^{uc}_{\mathcal{A},\Pi}(1^\lambda) = 1].$$

The second experiment $Exp^{uc2}_{\mathcal{A},\Pi}$ is identical with $Exp^{uc}_{\mathcal{A},\Pi}$ except that when F is updated to F^*, the challenger declares a failure if $\mathrm{H}(F^*)$ has been queried before. Since the source has min-entropy $\mu = \mu(\lambda)$ [47] [48], we have

$$\Pr[\text{The challenger declares a failure}] \leq \frac{q_H}{2^\mu}.$$

Then, we can obtain the following inequation:

$$\Pr[Exp^{uc}_{\mathcal{A},\Pi}(1^\lambda) = 1] \leq \Pr[Exp^{uc2}_{\mathcal{A},\Pi}(1^\lambda) = 1] + \frac{q_H}{2^\mu}.$$

The advantage of the second experiment can be determined by the definition of min-entropy, i.e.,

$$\Pr[Exp^{uc2}_{\mathcal{A},\Pi}(1^\lambda) = 1] \leq \frac{1}{2^\mu}.$$

Thus, the adversary cannot generate a valid proof except with probability $(1 + q_H)/2^\mu$, and the uncheatability of DeyPoS is guaranteed. □

Before examining unforgeability, we analyze an important property of HAT. That is, the verifier is confident that the indexes of tree nodes in the response from the prover correspond to the indexes of the challenged data blocks. Thus, if the adversary cannot forge the indexes of tree nodes, the verifier can verify that the prover possess the challenged data blocks.

Lemma 1. *For a given sibling set $\{(i, l_i) \mid i \in [1, 2^{\lceil \log_2 n \rceil + 1}]\}$, one can determine a unique set $\{(\iota, i_\iota) \mid \iota \in [1, n]\}$ with n, where n is the total number of the leaves, ι denotes the ι-th leaf node, and i_ι denotes its index in the tree.*

Proof. Note that every node in HAT has a unique index and every node except the root node has a sibling and a parent. Let $\mathcal{I}$ be the set of indexes in the given sibling set. We can construct an index set that contains all siblings and parents of $\mathcal{I}$ and itself. The construction algorithm is as follows. We first select an unprocessed index i, and compute $i_{parent} \leftarrow \lfloor i/2 \rfloor$ and $i_{sibling} \leftarrow i + (-1)^{i \bmod i_{parent}}$. If i_{parent} or $i_{sibling}$ does not exist in $\mathcal{I}$, insert them to $\mathcal{I}$ for future process. After subtracting the original $\mathcal{I}$ from the index set, we can obtain a unique set (the uniqueness can be guaranteed from the fact that the algorithm is deterministic and convergent) where all nodes are in the path.

Next we re-construct paths by connecting a node with its parent from the set. If a path does not begin with the root node whose index is 1, we return an empty set which indicates that the given sibling set is invalid. We also compute the sum L_l of l_i which exists in the sibling set. Note that, if (i, l_i) exists in the given sibling set where $i \neq 1$, none of its descendant nodes exists in the sibling set $\{(i, l_i)\}$ (If all of its descendant nodes are in the sibling set, we can remove them in this case). Let the number of paths be L_p, if $L_l + L_p \neq n$, we return an empty set which indicates that the given sibling set is invalid. Otherwise, each path must start from the root node to a leaf node whose value is $(i_\iota, l_{i_\iota} = 1)$ for some ι. By using l_i in the given sibling set, one can compute the value for each node in the paths, obtain all ι, and return a unique set $\{(\iota, i_\iota) \mid \iota \in [1, n]\}$. □

Lemma 1 assures that if the given sibling set is correct, the response in Algorithm 6 is the exact response to the challenged data blocks. The following theorem guarantees that the adversary cannot forge the sibling set, and we can

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2560812, IEEE Transactions on Computers

extract the challenged data blocks from the response as required in Definition 4.

Theorem 2. *Let* H *be a random oracle with output length* $\delta(\cdot)$*, for any source* $\mathcal{S}$ *with min-entropy* $\mu(\cdot)$*, any pseudorandom function* f*, and any PPT adversary who makes* $q_H(\cdot)$ *queries to* H*, DeyPoS is unforgeable.*

Proof. The proof begins with the unforgeability experiment $Exp^{uf}_{\mathcal{A},\Pi}$ in Definition 4. In this experiment, the random oracle $\mathrm{H}(X)$ takes as input X, and returns an arbitrary element Y if X is never queried, and $\mathrm{H}(X)$ otherwise. The challenger chooses a file F and updates it to F^*. $id \leftarrow \mathrm{H}(\mathrm{H}(F))$, $id^* \leftarrow \mathrm{H}(\mathrm{H}(F^*))$, and $(C, T) \leftarrow \mathrm{Encode}(\mathrm{H}(F), F)$ are sent to the adversary. The advantage that the adversary wins is

$$\Pr[Exp^{uf}_{\mathcal{A},\Pi}(1^\lambda) = 1].$$

The second experiment $Exp^{uf2}_{\mathcal{A},\Pi}$ is identical to $Exp^{uf}_{\mathcal{A},\Pi}$ except that if $H(X)$ has been queried before, the random oracle declares a failure. We assume that the adversary does not query the same X twice without loss of generality, and obtain

$$\Pr[\text{The random oracle declares a failure}] \leq \frac{2q_H}{2^\mu} + \frac{2q_H}{2^\delta},$$

where $\mu = \mu(\lambda)$ and $\delta = \delta(\lambda)$. Then, we have

$$\Pr[Exp^{uf}_{\mathcal{A},\Pi}(1^\lambda) = 1] \leq \Pr[Exp^{uf2}_{\mathcal{A},\Pi}(1^\lambda) = 1] + \frac{2q_H}{2^\mu} + \frac{2q_H}{2^\delta},$$

which means the adversary cannot distinguish the first experiment and the second experiment except negligible probability.

The third experiment $Exp^{uf3}_{\mathcal{A},\Pi}$ is identical to $Exp^{uf2}_{\mathcal{A},\Pi}$ except that we replace the outputs of f in the tag generation algorithm for leaf node with random values. Since f is a pseudorandom function, the adversary cannot distinguish the second experiment and the third experiment. Formally, we can construct an experiment $Exp^{uf'}_{\mathcal{A},\Pi}$ which is identical to $Exp^{uf2}_{\mathcal{A},\Pi}$ except that the outputs of f is provided by a challenger $\mathcal{C}_{prf}$ in experiment Exp^{prf} with a pseudorandom function scheme. Let Exp^{prf0} be the experiment that $\mathcal{C}_{prf}$ uses pseudorandom values, and Exp^{prf1} be the experiment that $\mathcal{C}_{prf}$ uses random values. The advantage can be defined as $|\Pr[\text{output 1 in } Exp^{prf0}] - \Pr[\text{output 1 in } Exp^{prf1}]|$, which should be negligible if f is a pseudorandom function. We define that the experiment of pseudorandom function scheme outputs 1 only if the checking protocol outputs 1 in $Exp^{uf'}_{\mathcal{A},\Pi}$. Note that, if we run Exp^{prf0}, then $Exp^{uf'}_{\mathcal{A},\Pi}$ is identical to $Exp^{uf3}_{\mathcal{A},\Pi}$, otherwise, $Exp^{uf'}_{\mathcal{A},\Pi}$ is identical to $Exp^{uf2}_{\mathcal{A},\Pi}$. Thus, we have $|\Pr[Exp^{uf2}_{\mathcal{A},\Pi}(1^\lambda) = 1] - \Pr[Exp^{uf3}_{\mathcal{A},\Pi}(1^\lambda) = 1]| = |\Pr[\text{output 1 in } Exp^{prf0}] - \Pr[\text{output 1 in } Exp^{prf1}]| \leq \epsilon(\cdot)$, where $\epsilon(\cdot)$ is a negligible function.

In the fourth experiment, we replace the outputs of f in the tag generation algorithm for non-leaf nodes with random values. Again, the adversary cannot distinguish the third experiment and the fourth experiment. Thus, the adversary cannot forge a proof without the knowledge of underlying C^*, and the proof is exactly the same to that of the challenge by Lemma 1. Thus, we can construct an extractor who challenges $q'(n)$ times where $q'(\cdot)$ is a polynomial function. Then, it can get linear equations and solve all of the blocks with overwhelming probability [12]. □

6 Performance Evaluation

In this section, we evaluate the performance of DeyPoS. The evaluation is divided into two aspects: theoretical and experimental evaluations.

6.1 Theoretical Comparison

Table 1 summarizes the asymptotic performance of our scheme in comparison with related schemes, where n denotes the number of blocks, b denotes the number of the challenged blocks, and $|m|$ denotes the size of one block. From the table, we observe that our scheme is the only one satisfying the cross-user deduplication on the client-side and dynamic proof of storage simultaneously. Furthermore, the asymptotic performance of our scheme is better than the other schemes except [31], which only provides weak security guarantee.

6.2 Experimental Comparison

6.2.1 Experiment Environment

We implemented our construction by OpenSSL 1.0.1 on a computer with an Intel 3.2GHz CPU and 8GB DDR4 memory. Each data point is the average of ten experiment results. The evaluation consists of three aspects, including the cost in the upload phase, the cost in the deduplication phase, and the cost in the proof of storage phase. The cost in the update phase is similar to the cost in the proof of storage phase, thus, we do not present the cost in the update phase.

From Table 1, we know that the skip list is only used in dynamic PoS [8], while the Merkle tree is used in both dynamic PoS [14] and deduplication [15]. In addition, the theoretical performance of the skip list is similar to the Merkle tree. Hence, we only compare our scheme with the Merkle tree based solutions. Since there is no Merkle tree based solution that supports both dynamic PoS and deduplication, we compare our scheme with the one based on Merkle tree (like [14] [15]).

6.2.2 Result Analysis

We first evaluate the cost in the upload phase. Fig. 3 presents the initialization time for constructing Merkle trees and HATs with different sizes of files and blocks. The initialization time is similar in all schemes. For example, the initialization time for constructing Merkle tree and HAT is 6.7s and 7.9s, respectively, for a 1GB file of 4kB block size. The storage cost of the client is $O(1)$, and the storage cost of the server is shown in Fig. 4. The authenticator size of HAT is lager than that of the Merkle tree. However, when Merkle tree is employed in PoS scheme, it requires more space for storing tags of file blocks. As a result, the storage cost of our scheme is similar to other Merkle tree based PoS schemes. When the block size is 4kB, the authenticator size is less than 3% of the file size in our scheme.

Next, we evaluate the cost in the deduplication phase. Fig. 5 presents the communication cost when the file size is 1GB. The communication cost considers the data sent from users and the data sent from the cloud server. The

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2560812, IEEE Transactions on Computers

TABLE 1
Comparison with related schemes

Scheme	Deduplication on the client-side			Proof of Storage		
	Client	Server	Comm	Client	server	Comm
[15]	$O(n^2)$	$O(b \log n)$	$O(b \log n) + b\|m\|$	N/A		
[31]	$O(n)$	$O(1)$	$O(1)$	N/A		
[8]	N/A			$O(b \log n)$	$O(b \log n)$	$O(b \log n) + b\|m\|$
[14]	N/A			$O(b \log n)$	$O(b \log n)$	$O(b^2 \log n) + \|m\|$
This paper	$O(n)$	$O(b \log n)$	$O(b \log n) + \|m\|$	$O(b \log n)$	$O(b \log n)$	$O(b \log n) + \|m\|$

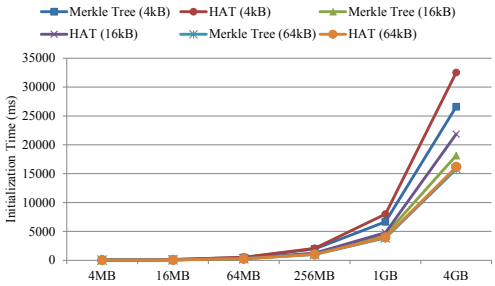


Fig. 3. Initialization time in different file sizes

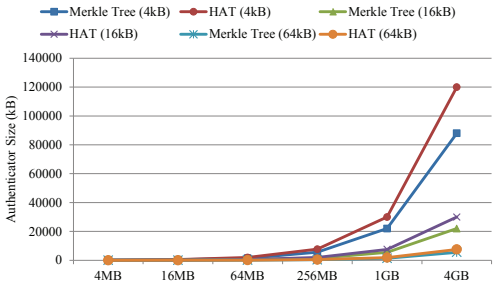


Fig. 4. Authenticator size in different file sizes

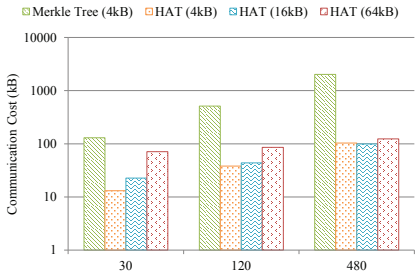


Fig. 5. Communication cost of 1GB file in the deduplication phase, when the number of challenged blocks are 30, 120, and 480, respectively

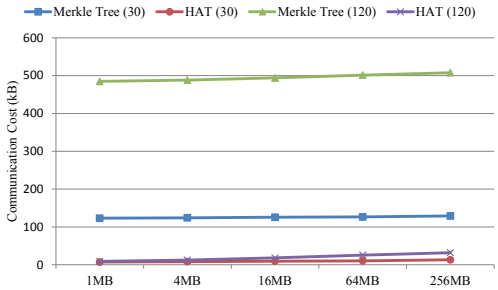


Fig. 6. Communication cost of 4kB block in the deduplication phase, when the number of challenged blocks are 30 and 120, respectively

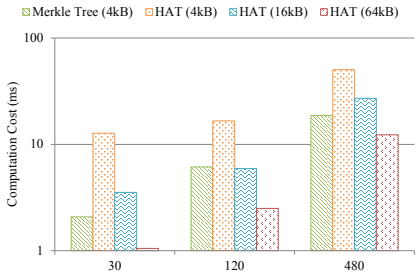


Fig. 7. Computation cost of 1GB file on the server-side in the deduplication phase, when the number of challenged blocks are 30, 120, and 480, respectively

communication cost in our scheme is more efficient than the cost of Merkle tree based schemes, since users has to send all challenged file blocks to the cloud server for generating leaf nodes of Merkle tree in those schemes. When the block size is 4kB and the number of the challenged blocks is 480, the communication cost of Merkle tree based solution [15] is almost 2MB, while the cost of DeyPoS is 104kB. Fig. 6 shows the communication cost of different file sizes, where the block size is fixed on 4kB. When the number of challenged file blocks are fixed, the communication cost stays at a steady level in Merkle tree based schemes since the major cost is to transmit the corresponding file blocks. However, the communication cost grows logarithmically with respect to the file size because the number of nodes in the sibling set grows logarithmically.

The computation cost on the server-side and the client side in the deduplication phase are shown in Fig. 7 and Fig. 8, respectively. The file size is 1GB. The computation cost of Merkle tree based schemes on both the server-side and the client-side have better performances. However, employing Merkle tree based solutions for deduplication alone is not a good choice in dynamic PoSs since the cloud server has to generate Merkle tree for each update, which is time-consuming. The major computation cost on the client-side in Merkle tree based schemes is the Merkle tree generation process, while the major computation cost in DeyPoS is the whole file encryption process. Fig. 9 presents the comparison of the performance among DeyPoS, Merkle tree based schemes, and directly transmitting the file with 100Mbps upload speed. The overall performance is calculated with $T_c + T_s + T_n$, where T_c is the computation cost on the client-side, T_s is the computation cost on the server-side, and T_n is the transmission delay on the 100Mbps channel. Even if the file size is 1MB, DeyPoS and Merkle tree based deduplication schemes have advantages in terms of overall performance.

Finally, we show the experimental results in the proof of

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2560812, IEEE Transactions on Computers

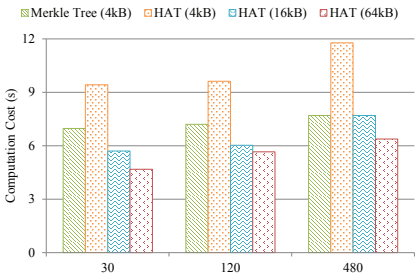


Fig. 8. Computation cost of 1GB file on the client-side in the deduplication phase, when the number of challenged blocks are 30, 120, and 480, respectively

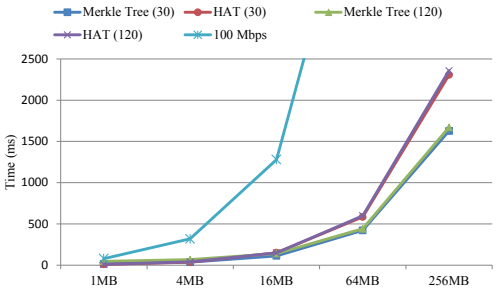


Fig. 9. Performance of 4kB block in the deduplication phase, when the number of challenged blocks are 30 and 120, respectively

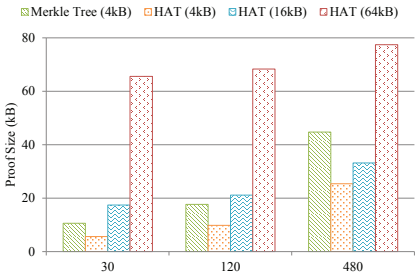


Fig. 10. Proof size of 1GB file in the proof of storage phase, when the number of challenged blocks are 30, 120, and 480, respectively

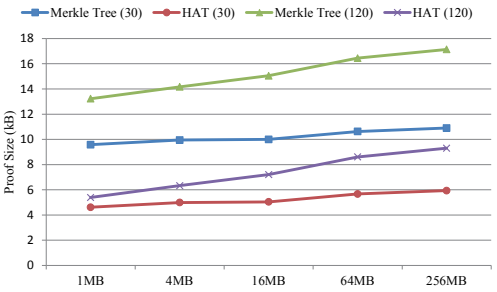


Fig. 11. Proof size of 4kB block in the proof of storage phase, when the number of challenged blocks are 30 and 120, respectively

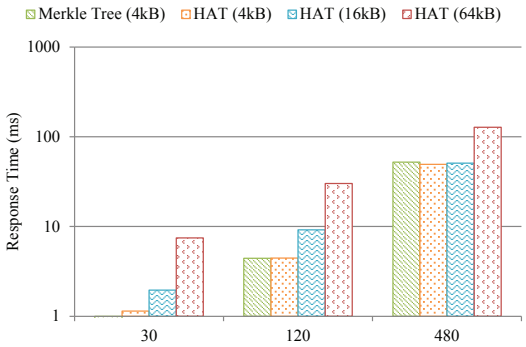


Fig. 12. Response time of 1GB file in the proof of storage phase, when the number of challenged blocks are 30, 120, and 480, respectively

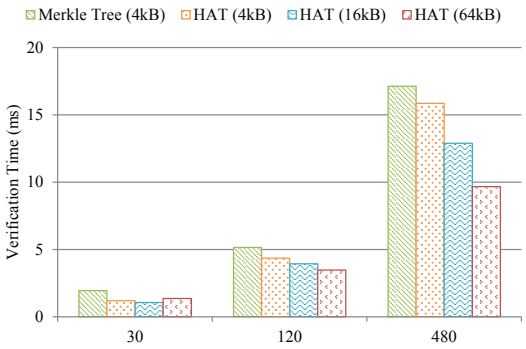


Fig. 13. Verification time of 1GB file in the proof of storage phase, when the number of challenged blocks are 30, 120, and 480, respectively

storage phase. Since the challenge size which is the size of data sent from users is constant and negligible (less than 100B) in both DeyPoS and Merkle tree based solutions, Fig. 10 only depicts the proof size which is the amount of data sent from the cloud server. DeyPoS requires a lower cost than Merkle tree based scheme because the tags in HAT are homomorphic. When we challenge 480 blocks (as shown in [5], challenging 460 blocks can detect 1% data loss with probability 99%), the proof size is less than 80kB, which is negligible small in practice. Fig. 11 presents the proof size of different file sizes, where the block size is fixed on 4kB. Obviously, DeyPoS requires less bandwidths in all situations. When the block size is 4kB [5] [14], the block size is less than 10kB.

Fig. 12 and Fig. 13 show the computation cost in the proof of storage phase for the cloud server and users, respectively. The computation cost on the server-side are almost the same in DeyPoS and Merkle tree based solutions. In the implementation of DeyPoS and Merkle tree based solutions, each file block consists of a number of group elements [12]. The cloud server processes $b \times s$ group elements, where s is the number of elements in a file block and b is the number of challenged file blocks. Thus, for a fixed number of challenged file blocks, the computation cost on the server-side is different from the theoretical result in Table 1. However, if the group element size equals to the block size, the theoretical result holds. The computation cost on the client-side in DeyPoS is slightly lower than the cost in Merkle tree based schemes as shown in Fig. 13. Since users only need to process one (combined) file block, the number of elements in a file block does not influence the total computation cost significantly. Fig. 14 and Fig. 15 present the computation cost of different file sizes, where the block size is fixed on 4kB. These experimental results support our analysis of Fig. 12 and Fig. 13. Also, when the file size is small, such as 1MB, DeyPoS is still more efficient than downloading the entire file (see Fig. 9).

As a consequence, our scheme, DeyPoS, which is based on a HAT, reduces the communication cost in both the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2560812, IEEE Transactions on Computers

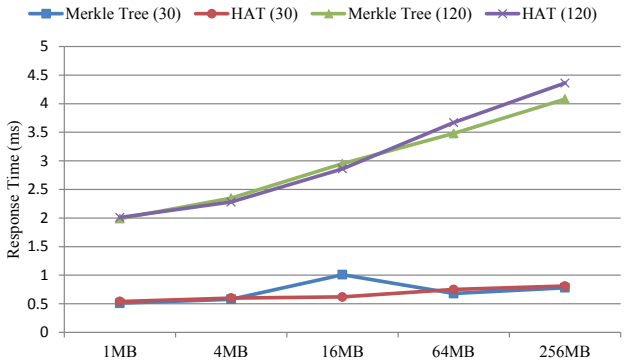


Fig. 14. Response time of 4kB block in the proof of storage phase, when the number of challenged blocks are 30 and 120, respectively

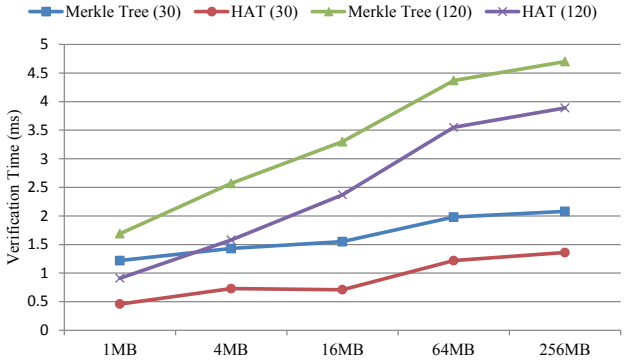


Fig. 15. Verification time of 4kB block in the proof of storage phase, when the number of challenged blocks are 30 and 120, respectively

deduplication phase and the proof of storage phase. The computation cost is as efficient as the one in Merkle tree based dynamic PoSs.

7 Conclusion

We proposed the comprehensive requirements in multi-user cloud storage systems and introduced the model of deduplicatable dynamic PoS. We designed a novel tool called HAT which is an efficient authenticated structure. Based on HAT, we proposed the first practical deduplicatable dynamic PoS scheme called DeyPoS and proved its security in the random oracle model. The theoretical and experimental results show that our DeyPoS implementation is efficient, especially when the file size and the number of the challenged blocks are large.

Acknowledgements

This research was supported in part by NSF grants 1217611, the Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education, the National Natural Science Foundation of China under Grant No. 61272451, 61572380, U1536204, the Major State Basic Research Development Program of China under Grant No. 2014CB340600, and the National High Technology Research and Development Program (863 Program) of China under Grant No. 2014BAH41B00.

References

[1] S. Kamara and K. Lauter, "Cryptographic cloud storage," in *Proc. of FC*, pp. 136–149, 2010.
[2] Z. Xia, X. Wang, X. Sun, and Q. Wang, "A Secure and Dynamic Multi-Keyword Ranked Search Scheme over Encrypted Cloud Data," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 2, pp. 340–352, 2016.
[3] Z. Xiao and Y. Xiao, "Security and privacy in cloud computing," *IEEE Communications Surveys Tutorials*, vol. 15, no. 2, pp. 843–859, 2013.
[4] C. A. Ardagna, R. Asal, E. Damiani, and Q. H. Vu, "From Security to Assurance in the Cloud: A Survey," *ACM Comput. Surv.*, vol. 48, no. 1, pp. 2:1–2:50, 2015.
[5] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song, "Provable data possession at untrusted stores," in *Proc. of CCS*, pp. 598–609, 2007.
[6] G. Ateniese, R. Di Pietro, L. V. Mancini, and G. Tsudik, "Scalable and Efficient Provable Data Possession," in *Proc. of SecureComm*, pp. 1–10, 2008.
[7] G. Ateniese, S. Kamara, and J. Katz, "Proofs of storage from homomorphic identification protocols," in *Proc. of ASIACRYPT*, pp. 319–333, 2009.
[8] C. Erway, A. Küpcü, C. Papamanthou, and R. Tamassia, "Dynamic provable data possession," in *Proc. of CCS*, pp. 213–222, 2009.
[9] R. Tamassia, "Authenticated Data Structures," in *Proc. of ESA*, pp. 2–5, 2003.
[10] Q. Wang, C. Wang, J. Li, K. Ren, and W. Lou, "Enabling public verifiability and data dynamics for storage security in cloud computing," in *Proc. of ESORICS*, pp. 355–370, 2009.
[11] F. Armknecht, J.-M. Bohli, G. O. Karame, Z. Liu, and C. A. Reuter, "Outsourced proofs of retrievability," in *Proc. of CCS*, pp. 831–843, 2014.
[12] H. Shacham and B. Waters, "Compact Proofs of Retrievability," *Journal of Cryptology*, vol. 26, no. 3, pp. 442–483, 2013.
[13] Z. Mo, Y. Zhou, and S. Chen, "A dynamic proof of retrievability (PoR) scheme with o(logn) complexity," in *Proc. of ICC*, pp. 912–916, 2012.
[14] E. Shi, E. Stefanov, and C. Papamanthou, "Practical dynamic proofs of retrievability," in *Proc. of CCS*, pp. 325–336, 2013.
[15] S. Halevi, D. Harnik, B. Pinkas, and A. Shulman-Peleg, "Proofs of ownership in remote storage systems," in *Proc. of CCS*, pp. 491–500, 2011.
[16] J. Douceur, A. Adya, W. Bolosky, P. Simon, and M. Theimer, "Reclaiming space from duplicate files in a serverless distributed file system," in *Proc. of ICDCS*, pp. 617–624, 2002.
[17] A. Juels and B. S. Kaliski, Jr., "PORs: Proofs of retrievability for large files," in *Proc. of CCS*, pp. 584–597, 2007.
[18] H. Shacham and B. Waters, "Compact proofs of retrievability," in *Proc. of ASIACRYPT*, pp. 90–107, 2008.
[19] Y. Dodis, S. Vadhan, and D. Wichs, "Proofs of retrievability via hardness amplification," in *Proc. of TCC*, pp. 109–127, 2009.
[20] K. D. Bowers, A. Juels, and A. Oprea, "HAIL: A high-availability and integrity layer for cloud storage," in *Proc. of CCS*, pp. 187–198, 2009.
[21] C. Wang, Q. Wang, K. Ren, and W. Lou, "Privacy-preserving public auditing for data storage security in cloud computing," in *Proc. of INFOCOM*, pp. 1–9, 2010.
[22] G. Ateniese, R. Burns, R. Curtmola, J. Herring, O. Khan, L. Kissner, Z. Peterson, and D. Song, "Remote data checking using provable data possession," *ACM Transactions on Information System Security*, vol. 14, no. 1, pp. 1–34, 2011.
[23] Y. Zhu, H. Hu, G.-J. Ahn, and M. Yu, "Cooperative provable data possession for integrity verification in multicloud storage," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 12, pp. 2231–2244, 2012.
[24] J. Xu and E.-C. Chang, "Towards efficient proofs of retrievability," in *Proc. of ASIACCS*, pp. 79–80, 2012.
[25] J. Chen, L. Zhang, K. He, R. Du, and L. Wang, "Message-locked proof of ownership and retrievability with remote reparing in cloud," *Security and Communication Networks*, 2016.
[26] E. Stefanov, M. van Dijk, A. Juels, and A. Oprea, "Iris: A scalable cloud file system with efficient integrity checks," in *Proc. of ACSAC*, pp. 229–238, 2012.
[27] D. Cash, A. Küpçü, and D. Wichs, "Dynamic proofs of retrievability via oblivious RAM," in *Proc. of EUROCRYPT*, pp. 279–295, 2013.

[28] M. Azraoui, K. Elkhiyaoui, R. Molva, and M. Önen, "StealthGuard: Proofs of Retrievability with Hidden Watchdogs," in *Proc. of ESORICS*, pp. 239–256, 2014.

[29] Z. Ren, L. Wang, Q. Wang, and M. Xu, "Dynamic Proofs of Retrievability for Coded Cloud Storage Systems," *IEEE Transactions on Services Computing*, vol. PP, no. 99, pp. 1–1, 2015.

[30] R. Di Pietro and A. Sorniotti, "Boosting Efficiency and Security in Proof of Ownership for Deduplication," in *Proc. of ASIACCS*, pp. 81–90, 2012.

[31] J. Xu, E.-C. Chang, and J. Zhou, "Weak leakage-resilient client-side deduplication of encrypted data in cloud storage," in *Proc. of ASIACCS*, pp. 195–206, 2013.

[32] S. Keelveedhi, M. Bellare, and T. Ristenpart, "DupLESS: Server-aided encryption for deduplicated storage," in *Proc. of USENIX Security*, pp. 179–194, 2013.

[33] J. Li, X. Chen, M. Li, J. Li, P. Lee, and W. Lou, "Secure Deduplication with Efficient and Reliable Convergent Key Management," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1615–1625, 2014.

[34] J. Li, Y. K. Li, X. Chen, P. Lee, and W. Lou, "A Hybrid Cloud Approach for Secure Authorized Deduplication," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 5, pp. 1206–1216, 2015.

[35] Q. Zheng and S. Xu, "Secure and efficient proof of storage with deduplication," in *Proc. of CODASPY*, pp. 1–12, 2012.

[36] R. Du, L. Deng, J. Chen, K. He, and M. Zheng, "Proofs of ownership and retrievability in cloud storage," in *Proc. of TrustCom*, pp. 328–335, 2014.

[37] B. Wang, B. Li, and H. Li, "Public auditing for shared data with efficient user revocation in the cloud," in *Proc. of INFOCOM*, pp. 2904–2912, 2013.

[38] B. Wang, B. Li, and H. Li, "Oruta: privacy-preserving public auditing for shared data in the cloud," *IEEE Transactions on Cloud Computing*, vol. 2, no. 1, pp. 43–56, 2014.

[39] J. Yuan and S. Yu, "Efficient public integrity checking for cloud data sharing with multi-user modification," in *Proc. of INFOCOM*, pp. 2121–2129, 2014.

[40] R. Gennaro and D. Wichs, "Fully Homomorphic Message Authenticators," in *Proc. of ASIACRYPT*, pp. 301–320, 2013.

[41] D. Boneh and D. M. Freeman, "Homomorphic Signatures for Polynomial Functions," in *Proc. of EUROCRYPT*, pp. 149–168, 2011.

[42] D. Catalano, D. Fiore, and B. Warinschi, "Homomorphic Signatures with Efficient Verification for Polynomial Functions," in *Proc. of CRYPTO*, pp. 371–389, 2014.

[43] A. Yun, J. H. Cheon, and Y. Kim, "On Homomorphic Signatures for Network Coding," *IEEE Transactions on Computers*, vol. 59, no. 9, pp. 1295–1296, 2010.

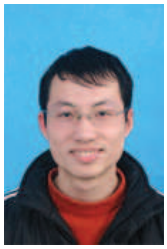
[44] C. Cheng and T. Jiang, "An Efficient Homomorphic MAC with Small Key Size for Authentication in Network Coding," *IEEE Transactions on Computers*, vol. 62, no. 10, pp. 2096–2100, 2013.

[45] J. Chen, K. He, R. Du, M. Zheng, Y. Xiang, and Q. Yuan, "Dominating Set and Network Coding-Based Routing in Wireless Mesh Networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 2, pp. 423–433, 2015.

[46] D. Catalano, "Homomorphic Signatures and Message Authentication Codes," in *Proc. of SCN*, pp. 514–519, 2014.

[47] M. Bellare, S. Keelveedhi, and T. Ristenpart, "Message-locked encryption and secure deduplication," in *Proc. of EUROCRYPT*, pp. 296–312, 2013.

[48] M. Abadi, D. Boneh, I. Mironov, A. Raghunathan, and G. Segev, "Message-locked encryption for lock-dependent messages," in *Proc. of CRYPTO*, pp. 374–391, 2013.



Kun He is a PH.D student of Wuhan University. His research interests include cryptography, network security, mobile computing, and cloud computing. He has published research papers in IEEE Transactions on Parallel and Distributed System, International Journal of Communication Systems, Security and Communication Networks, and IEEE TRUSTCOM.

Jing Chen received the Ph.D. degree in computer science from Huazhong University of Science and Technology, Wuhan. He worked as an associate professor from 2010. His research interests in computer science are in the areas of network security, cloud security. He is the Chief Investigator of several projects in network and system security, funded by the National Natural Science Foundation of China (NSFC). He has published more than 60 research papers in many international journals and conferences, such as IEEE Transactions on Parallel and Distributed System, International Journal of Parallel and Distributed System, INFOCOM, SECON, TrustCom, NSS. He acts as a reviewer for many Journals and conferences, such as IEEE Transactions on Wireless Communication, IEEE Transactions on Industrial Informatics, Computer Communications, and GLOBCOM.

Ruiying Du received the BS, MS, PH.D degrees in computer science in 1987, 1994 and 2008, from Wuhan University, Wuhan, China. She is a professor at computer school, Wuhan University. Her research interests include network security, wireless network, cloud computing and mobile computing. She has published more than 80 research papers in many international journals and conferences, such as IEEE Transactions on Parallel and Distributed System, International Journal of Parallel and Distributed System, INFOCOM, SECON, TrustCom, NSS.

Qianhong Wu is a senior researcher with the UNESCO Chair in Data Privacy, Department of Computer Science and Mathematics, Universitat Rovira i Virgili (URV), Tarragona, Catalonia. He received his M.Sc. in Applied Mathematics from Sichuan University in 2001. He got his Ph.D. degree in Cryptography from Xidian University in 2004. He has been a postdoctoral researcher with Wollongong University in Australia and Wuhan University in China. His research interests include public key cryptography, e- commerce security, security and privacy in networks, and private information retrieval. He has been a principal investigator or co-investigator of several Chinese-funded and Australian-funded projects. He has coauthored over 40 publications and has been a reviewer for several international journals.

Guoliang Xue , an IEEE fellow, is a professor of computer science at Arizona State University. His research interests include survivability, security and resource allocation issues in wireless networks, social net-works and smart grid. With over 200 pub-lished papers in those areas, he is an associate editor of the IEEE/ACM Transactions on Networking and of IEEE Network magazine. He served as technical program co-chair of IEEE INFOCOM'2010, which took place in San Diego.

Xiang Zhang (Student Member 2013) received his B.S. degree in Computer Science from University of Science and Technology of China, Hefei, China, in 2012. Currently he is a Ph.D student in the School of Computing, Informatics, and Decision Systems Engineering at Arizona State University. His research interests include network economics, incentive mechanism design, and game theory in crowdsourcing and cognitive radio networks. He has published papers in IEEE INFOCOM, IEEE GLOBECOM, IEEE ICC, IEEE TVT, IEEE IOTJ, and IEEE Network. He served as a reviewer for IEEE INFOCOM, IEEE GLOBECOM, IEEE ICC, and IEEE ICNC. He served as a TPC member for IEEE ICNC 2016.