Xu *et al. EURASIP Journal on Information Security* (2018) 2018:12
https://doi.org/10.1186/s13635-018-0083-x

EURASIP Journal on Information Security

RESEARCH Open Access

A generic integrity verification algorithm of version files for cloud deduplication data storage

Guangwei Xu*†, Miaolin Lai, Jing Li†, Li Sun† and Xiujin Shi

Abstract

Data owners' outsourced data on cloud data storage servers by the deduplication technique can reduce not only their own storage cost but also cloud's. This paradigm also introduces new security issues such as the potential threat of data lost or corrupted. Data integrity verification is utilized to safeguard these data integrity. However, the cloud deduplication storage only focuses on file/chunk level to store one copy of the same data hosted by different data owners, and is not concerned with the same part of different data, e.g., a series of version files. We propose an integrity verification algorithm of different version files. The algorithm establishes the generic storage model of different version control methods to improve the universality of data verification. Then, the methods of verification tags and proofs generating are improved based on the index pointers corresponding to the storage relationship in the version groups and chained keys. Finally, the random diffusion extraction based on the random data sampling in the version group is proposed to improve the verification efficiency. The results of theoretical and experimental analysis indicate that the algorithm can achieve fast and large-scale verification for different version data.

Keywords: Data integrity verification, Version file storage, Version group, Random diffusion extraction

1 Introduction

With the rapid development of the cloud computing, cloud storage as a new generation of computing infrastructure has received more and more attention. At the same time, more and more cloud storage services spring up which can provide users with low cost but huge data storage space. Although cloud storage can provide convenient storage and fast access to data at any time and etc., the paradigm of outsourced data service also introduces new security challenges. No matter how high degree of reliable measures cloud service providers would take, data loss or corruption could happen in any storage infrastructure due to natural corruption and malicious corruption [1]. Sometimes, in order to save storage space, the malicious storage service provider may delete the data that has not been accessed or accessed less, but claim that these data are completely stored on the remote servers. These misgivings have prompted the data owners to worry whether the outsourced data are intact or corrupted on the remote servers since they are deprived of the direct control of these data [2]. Data integrity verification [2–12] has been proposed to check the integrity of owners' remote stored data. These existing verification algorithms based on homomorphism technology can exactly identify the corrupted data (i.e., each block or file) in the verification. Recently, many commercial cloud storage services, such as Google Drive and Dropbox, utilize deduplication technique at the file/chunk level to store one copy of the same data hosted by different data owners. Liu et al. [13] proposed one-tag checker to audit the data integrity on encrypted cloud deduplication storage.

1.1 Motivation

As it is described above, the existing cloud deduplication storage is not concerned with the same data part of different files, e.g., a series of version files. As we know, a project such as documents, computer programs, and other collections of information needs to be continuously modified from original designing to final realization over a long

*Correspondence: dh.xuguangwei@gmail.com
†Guangwei Xu, Jing Li and Li Sun contributed equally to this work.
School of Computer Science and Technology, Donghua University, 201620 Shanghai, China

period of time. A series of changes of the project are usually identified by a number or letter code, which is termed the "revision number" in revision control [14]. The revision number can track and provide control over changes. Moreover, some important information like medical or trade records requires more complete preservation. Also, old version files need to be integrally saved in order to review and recover the key information in different periods. Nowadays, there are two major version control methods. One is the incremental storage, e.g., subversion. In this method, the earliest version file is stored fully, and then, the subsequent version files are stored incrementally. Another is the opposite of incremental storage, e.g., git, in which the latest version is stored fully.

These major version control methods bring infeasibility and insecurity to the existing cloud deduplication storage and data integrity, respectively. On the one hand, the existing cloud deduplication storage only focuses on file/chunk level to store one copy of the same data hosted by different data owners rather than the same part of different data [13] so that the technique is disabled in decreasing the efficiency of cloud storage. On the other hand, these version control methods incur some security issues for the data integrity even though the restorability of version files is improved. First, the security of version data is still weakened for these version control methods. Once a middle version file is corrupted, any subsequent version file cannot recover the full content of the project due to the insufficient middle version file storage. Second, the little opportunity of each version file being verified decreases the integrity of the full content of version files even though a verified version file is intact after the data verification. The storage space of version data is boosted with the increase of the version files. At the same time, the opportunity of any corrupted version file being identified is decreased since the existing verification algorithms are only oriented to every file instead of a set of version files [10, 11] while these files are randomly extracted. A large number of version files result in more verification cost. In this case, the random data extraction which is applied in the existing verification algorithms reduces the security of full version files.

1.2 Contributions

To improve the efficiency and universality of data verification on the cloud deduplication storage, in this paper, we propose a verification algorithm of different version files (VDVF) that can verify the integrity of version data in remote storage while protecting users' privacy. In summary, the contributions of the proposed algorithm are described as follows.

1) We improve the method of verification tags generating which is based on the generic storage model of different version files and established by combining the full storage and the incremental storage. It can improve the universality of data verification to meet the needs of version data controlled by different storage methods, i.e., full storage, incremental storage, and differential storage.

2) We improve the method of verification proofs generating which is based on the version groups and chained keys. In the verification, verification tags and proofs can be gained by index pointers corresponding to the storage relationship in the version groups and chained keys.

3) We design a novel random diffusion extraction method in the verification, which is based on the random data sampling and the version storage group. It can find all version files related to the chosen data through the link relationship in the version storage group and then extends the Boneh-Lynn-Shacham (BLS) signature scheme to support batch verification for multiple files. In this way, we improve the number of the verified data to effectively protect the integrity of version data under the condition of limited verification cost.

The remaining of the paper is organized as follows. In Section 2, we briefly summarize the current researches on the verification of data integrity. Section 3 outlines the system model and problem statements. In Section 4, we propose an efficient and generic integrity verification algorithm of different version files. Section 5 gives the analysis of our proposed algorithm in terms of security. Section 6 evaluates the performance of algorithm by simulations. Finally, the conclusion and future extension to the work is given in Section 7.

2 Related work

At present, the integrity verification methods for remote storage data can be divided into two types, i.e., provable data possession (PDP) [3] and proof of retrievability (POR) [4]. PDP can guarantee the data integrity from probability and certainty, but it cannot ensure the recoverability of the data. POR indicates that the server can give the recoverable proof of the data, and the data which have a certain degree of damage can be recovered through data coding technique.

- Verification schemes based on remotely stored data. Deswarte Y. et al. [5] first proposed two data integrity verification schemes for remotely stored data. One is to preprocess the files which are going to be verified with hash, and then, multiple challenge-response modes are mainly applied in the verification process. This scheme needs to store a large amount of checksum in order to prevent replay attacks of malicious servers. The other is based on the Diffie-Hellman key exchange protocol. But with the amount of data increasing, the cost of the server's calculation will grow at exponential rates. In 2007, Ateniese et al. [3] proposed PDP, and they applied homomorphic

verifiable tag in this scheme. Users generate a tag for each data block, servers then store these data blocks and tags. When users ask for data verification, servers generate the proofs for pairs of data blocks and tags that needed to be verified according to challenge information. Users can verify the data integrity by verifying the proofs returned from servers without getting the data back. In 2008, Ateniese et al. [6] improved their scheme by adding symmetric key. They proposed a dynamic provable data possession protocol based on cryptographic hash function and symmetric key encryption. A certain number of metadata need to be calculated in advance during the initialization phase, so that the number of updates and challenges is limited and fixed. Each update operation needs to recreate the existing metadata which is not applicable to large files. Moreover, their protocol only allows append-type insertions. Erway et al. [7] also extended the PDP model to support dynamic updates on the stored data.

Juel and Kaliski first proposed POR [8], the main idea of which is to encode the file first and then add "sentinels" which cannot be distinguished from the file data into the file randomly. When verifying the entire file, the verifier only needs to verify these sentinels. But this scheme can only do limited times challenge. Afterwards, Shacham et al. [4] applied homomorphic authentication in two improved schemes so that it can not only reduce the overhead of communication, but carry out unlimited challenges as well. Wang et al. [9] proposed a scheme based on homomorphic token value and RS codes which cannot only verify the integrity of the file but also locate the damaged data blocks. After that, Yang et al. [10] proposed an efficient and secure dynamic auditing protocol which supports data privacy-preserving and dynamic update. This scheme is also applicable to multiple users and multiple clouds. Ziad et al. [11] proposed a scheme based on game theory in order to find an optimal data verification strategy. Recently, Zhang et al. [15] used indistinguishability obfuscation to reduce the auditor's computation overhead while the auditor is equipped with a low-power device.

- Verification schemes based on deduplication storage. Data deduplication in computer storage refers to the elimination of redundant data. In [16], each user computes the integrity tags of each file with his private key, even though the file has been stored in the cloud. Then, the tags on the same block are aggregated into one tag by the cloud to reduce the cost of tag storage and computation. However, the public key aggregated by the multiplication of all the file owners' associated public keys easily reveals the file ownership information while the adversary launches brute-force attack. Moreover, Liu et al. [13] proposed a message-locked integrity auditing scheme based on proxy re-signature techniques in encrypted cloud deduplication storage without an additional proxy server. However, their algorithm is only applicable to the deduplicated storage which stores one copy of the same data hosted by different data owners.

All above schemes only deal with each individual file or block rather than a set of version files. Certainly, these verification algorithms can simply apply into the verification of version files if each version file is dealt with like a general file. In this way, all the files which include the version files are treated equally in the verification. However, if a middle version file is not verified or worse yet it is actually corrupted, the integrity of the subsequent version files will be hard to be ensured. Certainly, if all the version files are verified, it will produce large computation cost. In this paper, we improve the verification of different version files to reduce the impact of these problems on the integrity of version files.

3 System model and problem statements

3.1 Data integrity verification model

The basic model of remote data storage service is that after an agreement is reached between the user and the data storage provider, the user hosts his data and verification tags generated by data units to remote servers provided by the data storage provider. After that, according to the requests from the user, the server provides access to related data.

In traditional data integrity verification model, it is based on the agreement between the two sides of the data storage service to verify the integrity of the stored data. As either side of user or data service provider may provide biased verification results to cheat the other side, our verification model introduces the third party verifier like in [12] for the fairness of data verification. Thus, there are three participants in our data verification model, namely, user, data service provider (DSP), and third party verifier (TPV). Assume that the TPV can be fully trusted while the TPV is composed of a public verifier group rather than an individual to execute the verification, i.e., the number of group's members exceeds a certain value. Therefore, we can consider that the verification result provided by the TPV is fair and credible. The details about the number of the public verifier group members are not discussed in this paper due to the limitation of space.

In the verification of the remote data storage, we apply Challenge-Response model to perform the data verification, which is shown in Fig. 1. Firstly, the TPV sends data verification challenges to the DSP. Secondly, the DSP

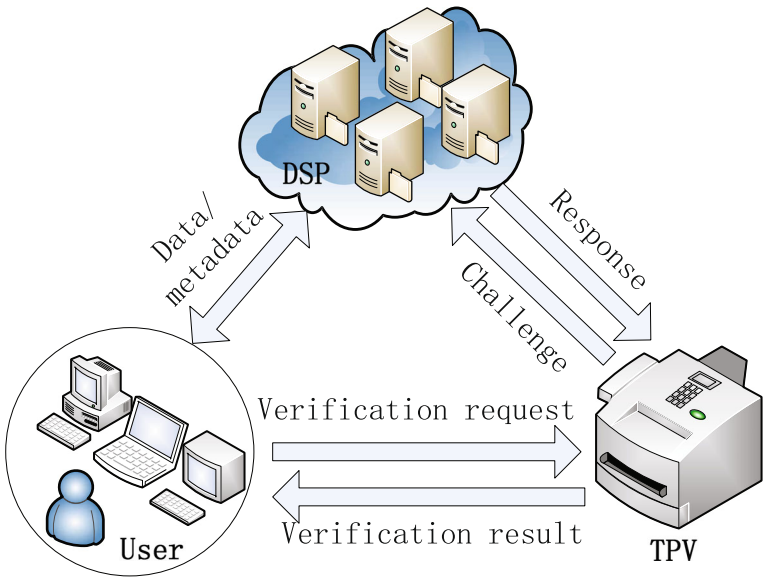

Fig. 1 Data integrity verification model

generates the integrity proofs of the challenged data based on the received challenges and then sends the proofs back to the TPV. Finally, the TPV judges whether the data have been intact or not by comparing the proofs from the DSP with the corresponding verification tags.

3.2 Version file storage model

The storage of different version files is one of the key technologies in version control. It is the management of changes to a set of files. Each version file is associated with a timestamp and the person who made the change. Certainly, version files can be compared, restored, and merged with some types of files. Version control also enables data administrators to easily track data changes made and roll back to earlier version files when required. In version control, unique version names or version numbers are assigned to keep track of different version files of electronic information incrementally such as semantic versioning. The revision number of version file includes major version number, minor version number, file phase identification, and revision version number. For example, the revision number of a file is "1.1.1."

In the process of version files generating, an efficient data structure is needed to link different version files. In terms of graph theory, version files are stored by a line of a trunk with branches. Version relationship model can be divided into two kinds based on the generation of the version files. One is based on the generation time of the version files to build the file chain and finally form the linear version chain model [17] as shown in Fig. 2. In reality, the structure is more complicated to form a directed acyclic or branched graph. If multiple people are working on a single data set or document, they implicitly create branches of the data (in their working copies) which will finally form the other one, i.e., split version tree model [18], as shown in Fig. 3, in which black solid lines are the trunk of the tree, black dotted lines are branches, and the blue dotted lines represent the merge of branches.

Definition 1 (storage node of version files). Each version file in the version chain or version tree is called storage node of version files which is shorten to storage node.

Each storage node is corresponding to each version file.

There are three main types of version files' storage, i.e., full storage, incremental storage, and differential storage. The full storage is the starting point for all other storages and contains all the data in a file that are selected to be stored. The incremental storage is to store all the changed data in a file since the last storage was made. They do this whether the last storage was a full one or an incremental copy. For example, if a full storage was done by node 1, node 2 stores all the changed data of the file after node 1 by the incremental storage. Likewise, node 3 only copies those changed data in the file after node 2's incremental storage. The differential storage copies those changed data in the file since the last full storage took place. For example, if a full storage was done by node 1, node 2 will copy all changed data of the file by the differential storage since

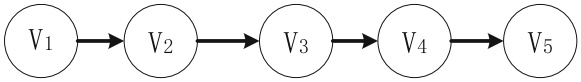

Fig. 2 Linear version chain

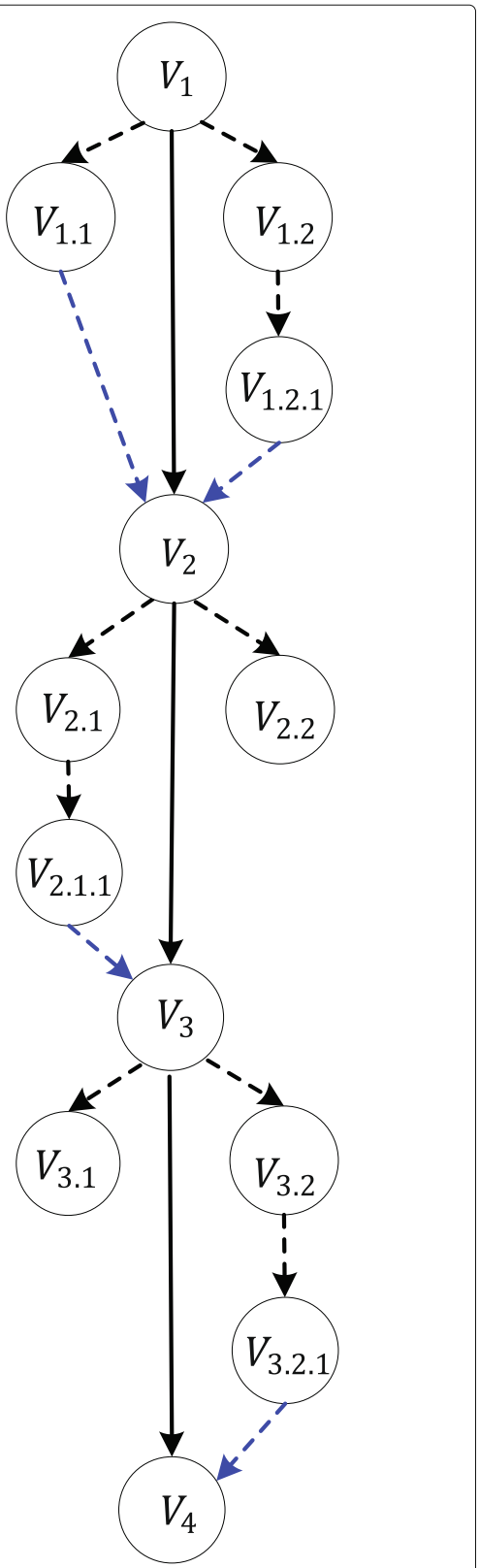

Fig. 3 Split version tree

node 1's storage copied everything. Also, node 3 will copy all changed data of the file by the differential storage since node 1's full storage was made.

In version file storage model, we make two following assumptions. (1) Once some data of the file are changed, a new version file will be inevitably generated. Changing the file content without updating the version file will lead to the conflict between version files. (2) Every version file has a unique predecessor and successor. Only if the file content changes, a new and unique version number of the file is increased. To conveniently describe the key problem, we focus on the linear version chain or the trunk chain of the version tree in the following discussion. The branches' chain of the version tree has a similar approach to it. While the version tree is boiled down to the trunk chain, it is equivalent to the linear version chain. In this paper, we uniformly call the linear version chain or the trunk chain of the version tree the version chain.

3.3 Attack model

The main threats of untrusted remote data storage come from that DSPs intentionally or unintentionally lost or corrupt users' data. There exist several following possible attack scenarios [19] on the integrity of the storage of different version files: (1) Attacker corrupts users' data and then forge the verification proofs by the revealed private key to pass the data verification. If only one private key is reused to generate the verification tags in all version files, the DSP can forge verification proofs of the verified data blocks in these version files and deceive the TPV while the private key is revealed by a sloppy user which is shared by all users in a project development team. (2) Attacker reduces the number of copies of the same data to save the storage space so that any part in a series of version files being corrupted will be unable to guarantee the integrity of version files. Generally, three copies of the same data are stored to maintain data service availability since the availability of users' data depends heavily on the number of copies [11]. The unreasonable storage model of version files further increases the potential risk of data corruption. For example, in the incremental storage, if any predecessor version file in a version chain is corrupted, the sequent version files will not be restored completely.

(3) Attacker easily escapes data corruption identification from the verification even if he corrupts key data in a series of version files. Due to the limited cost and number of checked data in each verification, all the verification algorithms only execute the probabilistic verification. In this way, many data cannot be verified while only a part of the data is extracted to be checked by the random data sampling. Moreover, considering that the DSP may not recognize that the verified data are corrupted only if the user declares, we introduce the TPV to verify the integrity of these data.

3.4 Problem statements

The existing verification algorithms treat all the files equally whether the file is one of the version files or not and is only corresponding to its expiry date or not. Due to the particularity of version files, the traditional verification results in the lack of computational efficiency and reliability in the integrity protection while these existing verification algorithms are directly applied in the verification of version files. We summarize the following problems.

(1) Although each verified file or block can be guaranteed whether it is intact, the complete content may still be corrupted if only one of the version files is checked in the verification.

In addition to the full storage of version files, the other storage methods only store the increment or difference to their precursor file, i.e., most of version files in a set of version files only store part of their complete content. Thus, the change of version files depends on the base version.

Definition 2 (base version file). A version file which is stored by the full storage is called the base version file.

For example, in git, the latest version file which is fully stored is the base version file. On the contrary, the earliest version file is the base version file in subversion. Thus, the verification of each independent file or block can only ensure the integrity of the verified file or block rather than the integrity of its actual content. Under the existing random verification mechanism, every version file is independently extracted to execute the verification. Thus, the integrity of only a part of version files can be guaranteed rather than the full content contained in the set of version files. Thus, even if a verified file or block is intact, it can also cause corruption or loss of important data related to the file or block in the version chain since there exists a predecessor and successor linkage in a version chain. Although a file is judged to be intact after the verification, the integrity of its precursor and successor file cannot be guaranteed so that user cannot accurately recover the original data content from remote version files' storage due to the loss or corruption of the key version data. This uncertainty is more serious especially while the number of version files in a version chain is large.

(2) There exists a potential danger that only one fixed private key is utilized to generate the verification tags of all the version files. As it is discussed in attack model, a group of users in a project development team share a private key. In this case, any sloppy user can reveal the private key so that the attacker can obtain the private key and forge the verification proofs to deceive the TPV.

(3) Even if the integrity of each version file stored by the full storage can be enforced by the integrity verification, it results in excessive computation and transmission overhead. Thus, it is very important to the choice of the base version.

In view of the above problems, we need to improve traditional verification methods. By improving the storage method of version files, we ensure the integrity of the content of version files in the version chain.

4 Data integrity verification algorithm for different version files

4.1 Constructing the generic storage model of version files

Assume that there is a series of files in a version chain, e.g., $\{V_{1.1}, V_{1.2}, V_{1.2.1}, V_{2.1}, V_{2.1.1}, V_{2.2}, V_{3.1}, V_{3.2}, V_{3.2.1}, ...\}$. Referring to the common version files' storage methods, they can be stored by the incremental storage (e.g., subversion) or the inverse incremental storage (e.g., git). If these files are stored by the former, the earliest version file is the base version, and the successor version files are stored by the incremental storage. If these files are stored by the latter, the latest version file is the base version, and the predecessor version files are stored by the inverse incremental storage. Thus, for the former, if any version file is corrupted, the full content of version files cannot be recovered. In this case, the security of version files is poor. For the latter, even though the security of version files is improved, the storage space of version data is increased. Especially, the security of version data is weakened while there are many version files and each version file is only randomly extracted in each verification.

Therefore, to improve the security of version data and reduce the storage space of version data, we redesign the base version of version files to meet the needs of different storage methods of version data in this paper. We call each version file in the version chain the storage node, which can also be shorten to node. Moreover, in order to protect data privacy, when TPV performs the data verification, we apply bilinear maps and homomorphic encryption to guarantee the security of the verification and the reliability of the verification results, and meanwhile, effectively reduce the traffic cost in network communication. The homomorphic encryption generates an encrypted result on ciphertexts, which matches the result of the operations on the plaintext when decrypted. The purpose of homomorphic encryption is to allow computation on encrypted data.

4.1.1 Grouping version files based on storage threshold

Considering the advantage of several storage ways and less storage overhead, we combine the full storage and incremental storage when users store version files to remote servers. Thus, we partition the version chain based on version storage threshold into several version groups referring to [20].

Definition 3 (version group). A version group which is a part of a version chain is composed of several version files sorted by the version number.

The number of members in a version group depends on version storage threshold.

Definition 4 (version storage threshold). The version storage threshold which is a value controls the maximum number of the version group members.

By selecting the appropriate version storage threshold, we store the first version file in each version group by the full storage and then make an iterative incremental storage to every successor version file in the group. Thus, in the storage of version files, the version store threshold determines the size of each version group. For example, on the left side of Fig. 4, we set the version storage threshold to 4. Therefore, the version chain is partitioned into two version groups. The first version group includes four members, i.e., nodes V_1, V_2, V_3, and V_4. The second version group includes two members, i.e., nodes V_5 and V_6. In the first version group, node V_1 which is the basic version is stored by the full storage, and nodes V_2, V_3, and V_4 are all stored by the incremental storage and based on their respective precursor version file in a form of the incremental storage. Likewise, node V_5 is the basic version and stored by the full storage. Node V_6 is stored by the incremental storage based on node V_5. So the version tree is finally formed as shown on the right side of Fig. 4. The shadow nodes represent the basic versions, and others represent the incremental storage nodes based on the basic versions.

According to previous analysis, the version storage threshold and the partition of the version group are the most important selection in version file storage. The research on the storage threshold has been studied in [20], and in this paper, we will not discuss it. We set the version storage threshold to T.

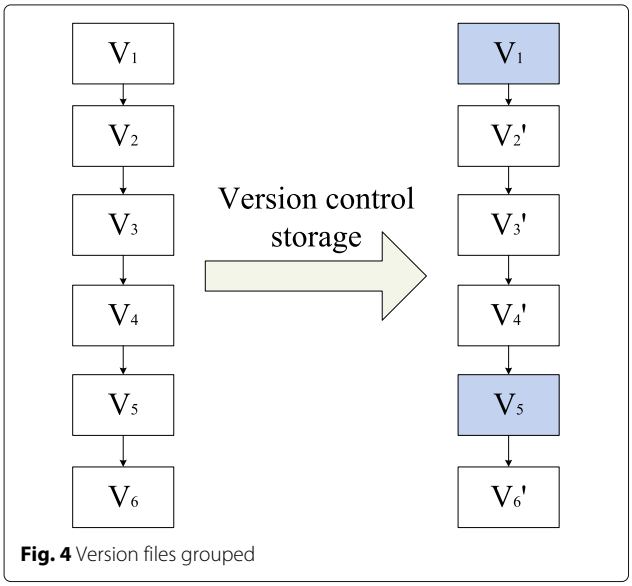

Fig. 4 Version files grouped

The process of version files grouping is as follows. Before the partition of the version group is performed, the set of version files should be first ensured referring to the same file description series. Secondly, the length (or depth) of the version chain can be determined. Finally, the number of members in each version group is calculated based on the version storage threshold T and then splits the version chain into several version groups.

For example, given a series of files V_i with file name V and version numbers i ($i \in [1, n]$) in a linear version chain. Assume V_1 is identified as the first version file. V_i's ($i \in [2, n]$) offset compared to V_1 in the version chain is $L = f(V_i)$, where f is the offset calculate function to figure out the offset of V_i relative to the first version file in the version chain. When the version number i is the biggest one, its offset value relative to the first version file in the chain is the maximum length of the version chain $L_{\max}$. In a version chain, we can simply carry out the version group by dividing the maximum length of the chain according to the equal threshold. Thus, once the version storage threshold is given, the number of version groups c can be calculated by

$$c = \lceil L_{\max}/T \rceil, \tag{1}$$

where $\lceil . \rceil$ means that the value takes an integer being greater than or equal to the current value. In version tree model, we need to take the vertical and horizontal development of the version tree into account to perform the version group. The depth of the version tree is determined by the maximum value of the trunk on the version tree, namely, the number of the major version. Therefore, the depth of the version tree is calculated based on the number of the main version, which is similar to the linear version chain, i.e., $L = f(\text{getmajorversionnumber}(V_{\{.\}}))$, where $\{.\}$ is file's version number.

4.1.2 *Storage and restoration of deduplication-based version files*

According to the sequence of version number and content correlation between two version files of adjacent version numbers, we can set up the associated storage of version files in practical. For the sake of explaining the storage of version files, let us see an example. We suppose that there exist two successive version files, i.e, Version 1 and Version 2, respectively. These two version files are locally stored as independent and full files as shown in the left of Fig. 5. When users want to store them on remote servers, they should do block processing on these version files referring to [5]. Providing that Version 1 can be divided into 4 data blocks, namely, Block_1, Block_2, Block_3, and Block_4. Also, Version 2 can be divided into 4 data blocks, namely, Block_2, Block_4, Block_5 and Block_6, where Version 2's duplicate block numbers comparing with Version 1's means that the content of blocks is the same. Compared

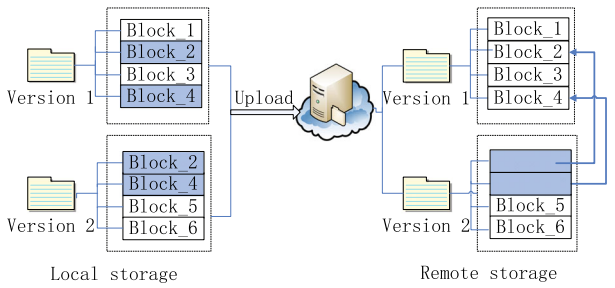

Fig. 5 Comparison between local and remote storage of version files

to Version 1, Version 2 deletes Block_1 and Block_3 and adds Block_5 and Block_6 as shown in the shaded rectangles on the left side in Fig. 5. In remote storage, Version 1 is selected as the base version, and all blocks in the file (i.e., Block_1, Block_2, Block_3 and Block_4) are fully stored as shown on the right side of Fig. 5. However, only Block_5 and Block_6 in Version 2 are stored by the incremental storage, and in addition, the index pointers of Block_2 and Block_4 pointing to corresponding data blocks in Version 1 are put. Certainly, if Block_1 in Version 1 is updated into Version 2, it will be firstly deleted and then replaced by a new added block (i.e., Block_5) in Version 2. If we need to restore Version 2 in the local station, we can directly download the stored Block_5 and Block_6 and indirectly download Block_2 and Block_4 by their index pointers pointing to the corresponding data blocks. Thereby, the entire content of Version 2 is restored.

In order to further discuss the data storage and data restoration in the version control mode, we give a set of version files consisting of four version files, as shown in Fig. 6. The left side of the figure indicates that all of the version files adopt the full storage when they are stored locally, that is, each file is independent and fully stored. The middle of the figure indicates the content of each version file when they are uploaded to remote servers. Let the version storage threshold be set to 4, and all version files form a version group. V_1 is the first file and stored by the full storage, and other files in this group V_2, V_3, and V_4 adopt the incremental storage. Let $\Delta_{i+1|V_i}$, $i \in [1,3]$ indicate the contents of the incremental storage of version file V_{i+1} with respect to its predecessor version file V_i, namely, the differential content from V_i to V_{i+1}. Followed by analogy, the contents of the other version files can be got. When getting a version file, we find the version group it belongs to and the first version file in the group, and then restore the full files in sequence according to the version chain as shown in the right side of Fig. 6. For example, if the version file V_3 needs to be fully restored, it first gets the version file V_1, and then supplements Δ_2 and Δ_3, i.e., $V_3 = V_1 + \Delta_2 + \Delta_3$. Certainly, Δ_i relative to V_1 includes the added data as well as the deleted data.

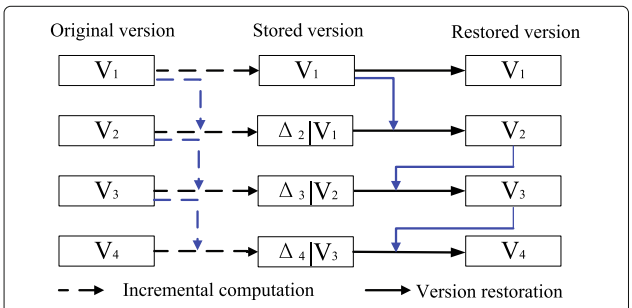

Fig. 6 Storage and restoration of a version file

In the process of the incremental storage between different version files, the difference information between two files needs to be calculated. Considering that rsync as a type of delta encoding is used for determining what parts of the file need to be transferred over the network connection, we apply the core idea of rsync algorithm [21] to distinguish the difference information between two files.

4.2 Generating the chained keys

To ensure the association between different version files, we apply chained keys in the generation of keys. In [22], the authors combined the hysteresis signatures with chained keys to build a link structure between signatures. However, in the verification, it needs to verify both timestamps and signatures so that it will add computation and communication overhead. In particular, the larger set of version files (i.e., more revisions) needs more verification time. Thus, we design a scheme of chained keys. On the basis of the basic key, the processing key of every version file is calculated by hashing key of its predecessor version file. Figure 7 illustrates the generation process of the chain keys in which the hash operation is a one-way hash function $h(.)$ and considered to be secure since it is easy to compute $y = h(x)$ with given x and hard to compute $x = h^{-1}(y)$ with given y. *Key* in the figure indicates the basic key, and it can get a pair of private-public key (sk, pk) through the key generation algorithm in Section 4.4. Key_1 indicates the key of version file V_1 and hashes from *Key*. Straight arrow in this figure indicates the hash function $h(.)$. Every $Key_i (i \geq 2)$ is hashed from Key_{i-1}, where i indicates the version number. At last, all the keys $Key_i (i > 1)$ on the key chain can be obtained.

4.3 Extracting the challenged data

Existing data integrity verification algorithms mainly extract some data from the entire set of data randomly

Fig. 7 Generation of the chained keys

to execute the verification [23]. Thus, even though there exist different version files, each time the data extraction is independent and random.

Considering the combination storage method of the full storage and the incremental storage is applied by these version files, we do not just verify the chosen file, but scale up the extraction range to all version files related to the chosen file by the link relationship in the version group while one of these files is extracted to be verified. Thus, we need to locate the version group the chosen file belongs to and the relative version files in the version group. All the files in the current version group are determined whether they are intact. Therefore, while a file in a version group is randomly extracted and verified, all the files in the version group will be extracted to execute the verification actually. We define this data extraction as *random diffusion extraction* corresponding to *random extraction*. The random diffusion extraction collects the results of random extraction and then finds a relevant version group by the version relationship model to determine the final set of extracted files. By performing batch verification of all these extracted files, we can verify the integrity of the full content of the chosen file. Meanwhile, once the verification can cover multiple continuous version files even all version files, the reliability of data storage can be further enhanced.

We define the random extraction data set as V_{chall}, the random diffusion extraction data set as V'_{chall}. Before the verification, we need to extract the set of challenged files V_{chall} by the random extraction. If there exists a file in a version chain being extracted in V_{chall}, we locate the version group the chosen file belongs to according to the file and storage threshold T, and then put its precursor version files in the group into V'_{chall} to prepare for the data verification. For example, in Fig. 2, let the storage threshold T be 4. If V_3 exists in V_{chall}, all precursor version files of V_3 in the same version group should be put into V'_{chall}, namely, $V_{\text{chall}} = \{V_3\} \rightarrow V'_{\text{chall}} = \{V_1, V_2, V_3\}$.

If there exists a file in a split version tree model being extracted in V_{chall}, we need to determine the major version number of the file firstly and then locate the version group which includes version files on the trunk and branches of the split version tree. Finally, put all of these precursor version files into V'_{chall} to prepare for the data verification. For example, in Fig. 3, let the storage threshold be set to 3. If V_3 is extracted in V_{chall}, all precursor version files in the current version group should be put into V'_{chall}, namely, $V_{\text{chall}} = \{V_3\} \rightarrow V'_{\text{chall}} = \{V_{1.1}, V_{1.2}, V_{1.2.1}, V_{2.1}, V_{2.1.1}, V_{2.2}, V_{3.1}, V_{3.2}, V_{3.2.1}\}$.

Of course, the amount of files actually extracted will increase accordingly while the data set V_{chall} extracted by the random extraction extends to the data set V'_{chall} by the random diffusion extraction. However, the number of files being verified is limited by the data verification overhead. Thus, how to determine the size of extracted data set reasonably is also a key problem to be considered. In order to control the amount of data being verified under the limitation of data verification overhead, we need to restrict the size of the extracted data set in the process of the random data extraction. Providing that the standard size of each data block is fixed, the number of data blocks in the base version file of a version group is n, the storage threshold is T, and the modified probability of each successor version file based on the predecessor file is α. In addition, we assume that with the constraints of data verification overhead, the total amount of data blocks verified each time is q. Thus, the amount of files randomly extracted every time t is

$$t = \frac{q}{n + (T-1)\alpha n}. \tag{2}$$

Accordingly, the total number of files in random diffusion extraction $\bar{t}$ is

$$\bar{t} = T \times t. \tag{3}$$

After determining the amount of files in the random extraction, we put all the chosen files into the set V_{chall}. Then, each file is extracted from the set to analyze its version group, and all the precursor version files in the version group are put into the set V'_{chall}. Finally, the data set V'_{chall} is formed and challenged the data integrity.

4.4 Algorithm design

Because the traditional verification methods do not distinguish the different version files, after introducing version files, the corresponding verification algorithm needs to take into account the impact of version files' storage. Namely, the bigger the version number is, the newer the file is. In this algorithm, we combine the batch verification and short signature technology based on bilinear map. The algorithm needs to improve the efficiency of the verification while ensuring its security and correctness. Our algorithm designs as follows.

Let G_1, G_2, and G_T be the multiplicative groups with the same prime order p and $e : G_1 \times G_2 \rightarrow G_T$ be the bilinear map. Let g_1 and g_2 be the generators of G_1 and G_2, respectively. Let $h : \{0,1\}^* \rightarrow G_1$ be a keyed secure hash function that maps $\{0,1\}^*$ to a point in G_1, where $\{0,1\}^*$ is the abstract information of the data.

(1) Chained key generation algorithm $CKeyGen(V_i) \rightarrow (sk_i, pk_i)$. The user first selects a random number λ and then acquires a secure random big number $sk \in Z_p$ as the private key and computes $pk = g_2^{sk} \in G_2$ as the public key through the key generation algorithm $KenGen(\lambda) \rightarrow (sk, pk)$. On the basis of a basic pair of private-public key (sk, pk), we can get the key pair of the version file V_i by chained key generation function $h((sk, pk), V_i) = (sk_i, pk_i)$.

(2) Tag generation algorithm $TagGen(\mathbb{M}, sk_i) \rightarrow \mathbb{T}$. Let $\mathbb{M}$ be the set of outsourced data. The tag generation algorithm generates a tag t_{ij} for every data block m_{ij} in the encrypted version file $V_i \in \mathbb{M}$ which is composed of n data blocks, i.e., $m_{i1}, m_{i2}, ..., m_{in}$, where j represents the identification number of the data block m_{ij} and $j \in [1, n]$.

It first chooses random value $x_i \in Z_p$ for each version file and computes $u_i = g_1^{x_i} \in G_1$. For each data block m_{ij} ($m_{ij} = \{0,1\}^*$), it computes a data tag t_{ij} for the data block as

$$t_{ij} = \left(h(m_{ij.ID}||j) \times u_i^{m_{ij}}\right)^{sk_i}, \tag{4}$$

where $m_{ij.ID}$ is the identification of the data block m_{ij}, and $||$ denotes the concatenation operation. It outputs the set of data tags $\mathbb{T} = \{t_{ij}\}_{i\in[1,d],j\in[1,n]}$, where d is the maximum version number of the set of version files.

(3) Batch challenge algorithm $BChall(\mathbb{M}, \beta) \rightarrow \mathbb{C}$. In all outsourced data $\mathbb{M}$, applying the ratio of data extracted β, it first gets the random extraction data set V_{chall}. Referring to the set V_{chall}, it then assigns the random diffusion extraction data set V'_{chall} which is the challenge set of version groups that related to those chosen files through organization of challenged data algorithm. Afterwards, it constructs a set of extracted data blocks as the challenged set $\mathbb{Q}$ for the set V'_{chall}. Assume that there are $\bar{t}$ files in V'_{chall}. In addition, it generates a random number u_i for each extracted file V_i and chooses a random number $r \in Z_p$, and then computes the set of challenge stamp $R_i = pk_i^r$. Finally, it outputs the challenge as $\mathbb{C} = \left(\bar{t}, V'_{\text{chall}}, \mathbb{Q}, \{R_i\}\right)$.

(4) Prove algorithm $\text{Prove}(\mathbb{C}) \rightarrow \mathbb{P}$. After receiving the challenge, the DSP computes the proofs of all challenge data $\mathbb{P}$, which consist of the tag proof TP_{V_i} and the data proof DP_{V_i}. Assume each version file includes n data blocks. The tag proof is generated as $TP_{V_i} = \prod_{j\in[1,n]} t_{ij}^{v_{ij}}$, where v_{ij} is a series of chosen random numbers. To generate the data proof, it first computes the linear combination of all the challenged data blocks as $MP_{V_i} = \sum_{j\in[1,n]} v_{ij}m_{ij}$ and then computes the data proof as $DP_{V_i} = e(u_i, R_i)^{MP_{V_i}}$. It gets the proof of each version file $P_{V_i} = \{TP_{V_i}, DP_{V_i}\}$ and then outputs the set of proof of all extracted files $\mathbb{P} = \{P_{V_i}\}_{V_i\in V'_{\text{chall}}}$ and sends it to TPV.

(5) Batch verification algorithm $BVerify\left(\mathbb{C}, \mathbb{P}, pk_i, \bar{t}, r\right) \rightarrow 0/1$. For each file that needs to be verified V_i, it computes the set of identifier's hash values $h(m_{ij.ID}||j)$, and then uses these hash values to compute a challenge value H_{V_i} by $H_{V_i} = \prod_{j\in[1,n]} h(m_{ij.ID}||j)^{rv_{ij}}$. When finished with the calculation of all the challenged files, it verifies the proofs by the verification equation as

$$\prod_{V_i\in V'_{\text{chall}}} DP_{V_i} = \frac{\prod e\left(TP_{V_i}, g_2^r\right)}{\prod e\left(H_{V_i}, pk_i\right)}. \tag{5}$$

If Eq. (5) is true, it outputs 1 and indicates that all the verified files are intact. Otherwise, it outputs 0 and indicates there exist corrupted files.

If there exist corrupted files in the verification result, it can locate these corrupted files by challenging level by level. That is, each version group is challenged separately to locate the corrupted version group, and then each version file in the group is verified separately to locate the corrupted version files.

5 Security analysis

The proposed algorithm should ensure the security in terms of the correctness of the algorithm, the security of the data storage, data privacy-protecting, the security of the batch verification, and the timeliness of the algorithm. To facilitate the analysis, we first give a security definition referring to [24] as follow.

Definition 5 (computational Diffie-Hellman (CDH) problem). If g, g^a, and g^b are known, it is computationally infeasible to calculate g^{ab} with unknown $a, b \in Z_p$.

To hide the corrupted data, the DSP must try to forge the verification data to pass the verification Eq. (5). The security of the proposed algorithm is analyzed respectively in Sections 5.2, 5.3, and 5.4.

5.1 Correctness of algorithm

We can judge the correctness of the algorithm by verifying the correctness of the verification equation. The proof of the verification formula is as follows.

$$\begin{aligned}
&\prod DP_{V_i} \prod e(H_{V_i}, pk_i) \\
&= \prod DP_{V_i} \cdot e(H_{V_i}, pk_i) \\
&= \prod e(u_i, R_i)^{MP_{V_i}} \cdot e(H_{V_i}, pk_i) \\
&= \prod e\left(u_i, pk_i^r\right)^{\sum_{j\in[1,n]} v_{ij}m_{ij}} \cdot e\left(\prod_{j\in[1,n]} h(m_{ij.ID}||j)^{rv_{ij}}, pk_i\right) \\
&= \prod e\left(u_i, g_2^{sk_i r}\right)^{\sum v_{ij}m_{ij}} \cdot e\left(\prod h(m_{ij.ID}||j)^{rv_{ij}}, g_2^{sk_i}\right) \\
&= \prod e\left(u_i^{r\sum v_{ij}m_{ij}}, g_2^{sk_i}\right) \cdot e\left(\prod h(m_{ij.ID}||j)^{rv_{ij}}, g_2^{sk_i}\right) \\
&= \prod e\left(u_i^{r\sum v_{ij}m_{ij})} \cdot \prod h(m_{ij.ID}||j)^{rv_{ij}}, g_2^{sk_i}\right) \\
&= \prod e\left(\left(u_i^{\sum v_{ij}m_{ij}} \prod h(m_{ij.ID}||j)^{v_{ij}}\right)^r, g_2^{sk_i}\right) \\
&= \prod e\left(\left(u_i^{\sum v_{ij}m_{ij}} \prod h(m_{ij.ID}||j)^{v_{ij}}\right)^{sk_i}, g_2^r\right) \\
&= \prod e\left(\prod h(m_{ij.ID}||j)^{sk_i v_{ij}} \cdot u_i^{sk_i \sum v_{ij}m_{ij}}, g_2^r\right) \\
&= \prod e\left(\prod h(m_{ij.ID}||j) \cdot u_i^{m_{ij}})^{sk_i v_{ij}}, g_2^r\right) \\
&= \prod e\left(\prod t_{ij}^{v_{ij}}, g_2^r\right) \\
&= \prod e\left(TP_{V_i}, g_2^r\right)
\end{aligned}$$

According to the proof of the verification formula, homomorphism can be used to verify the integrity of data. In every verification, our method can generate an encrypted proof with the challenge stamp by using the property of the bilinear pairing. Even though the TPV cannot decrypt it, she can verify the correctness of the proof without decrypting it. In addition, when challenging the integrity of data, the TPV randomly generates the value of each challenge. The DSP cannot predict it or save the corresponding data proofs and tag proofs after calculating these data. Due to the randomness of the value of the challenge, the repetition rate of challenge in different verification is very small which can effectively resist the replay attack on the DSP. Only if the received challenge and the corresponding data blocks are used while the DSP generates the proofs, the verification can be passed. We can draw conclusion from the previous analysis that our proposed algorithm is correct.

5.2 Security of data storage

In order to ensure the security of the data storage, we need to prove that only when the remote server stores the user's data correctly, the verification formula can be passed through. Based on Definition 5, it is computationally infeasible to forge DP_{V_i}, TP_{V_i}, and H_{V_i} in our algorithm. Thus, the DSP cannot forge $P_{V_i} = \{TP_{V_i}, DP_{V_i}\}$ effectively and get the true results.

We introduce an encrypted proof with the challenge stamp $R_i = pk_i^r$ in our algorithm. The DSP cannot decrypt r from R_i and pk_i. Meanwhile, we can see from the formula $DP_{V_i} = e(u_i, R_i)^{MP_{V_i}}$ that if DP_{V_i} is effective, MP_{V_i} and m_{ij} are effective in $MP_{V_i} = \sum_{j\in[1,n]} v_{ij}m_{ij}$. In every verification, v_{ij} is generated randomly. According to the small index technique mentioned in [25], the correctness of the data blocks that extracted randomly can be guaranteed.

Based on the analysis above, the storage security of user's data is guaranteed, and the algorithm can effectively resist the forge attack and replace attack.

5.3 Data privacy-protecting

During a verification process, we need to ensure that the user's data and tags cannot be acquired by the TPV from the proof returned from the DSP. We can see from $MP_{V_i} = \sum_{j\in[1,n]} v_{ij}m_{ij}$ that if TPV can acquire MP_{V_i}, she can solve the user's data by obtaining the linear combination equation of the data blocks.

In $DP_{V_i} = e(u_i, R_i)^{MP_{V_i}}$, MP_{V_i} is hidden. In $u_i = g_1^{x_i}$, x_i is randomly selected when uploading the file and unknown to TPV. Even if TPV knows the value of u_i, she is difficult to solve MP_{V_i} based on Definition 5. In formula $TP_{V_i} = \prod_{j\in[1,n]} t_{ij}^{v_{ij}} \left(\prod_{j\in[1,n]} h(m_{ij.ID}||j)^{v_{ij}}\right)^{sk_i} \cdot \left(u_i^{\sum_{j\in[1,n]} v_{ij}m_{ij}}\right)^{sk_i}$, we can see that $\left(u_i^{\sum_{j\in[1,n]} v_{ij}m_{ij}}\right)^{sk_i}$ is hidden by $\left(\prod_{j\in[1,n]} h(m_{ij.ID}||j)^{v_{ij}}\right)^{sk_i}$, and it is a computational Diffie-Hellman problem for TPV to calculate $\left(\prod_{j\in[1,n]} h(m_{ij.ID}||j)^{v_{ij}}\right)^{sk_i}$ by $\prod_{j\in[1,n]} h(m_{ij.ID}||j)^{v_{ij}}$ and $pk_i = g_2^{sk_i}$ based on Definition 5. On the basis of Definition 5 and the previous analysis, we can find that TPV cannot acquire $\sum_{j\in[1,n]} v_{ij}m_{ij}$ and $\left(u_i^{\sum_{j\in[1,n]} v_{ij}m_{ij}}\right)^{sk_i}$.

Based on the analysis above, the privacy of user's data can be protected well.

5.4 Security of batch verification

The batch verification method proposed in our algorithm is based on the aggregate signature. Thus, the security of the storage and privacy-protecting of users' multiple files in batch verification are the same as that of single file. From the previous analysis, the security of the storage and privacy-protecting of users' single file in the verification are guaranteed. Similarly, it is also satisfied for multiple files. Thus, the security of batch verification is also guaranteed.

5.5 Timeliness of algorithm

We assume that after uploading files to remote servers, user will delete the local files. So before generating a new version file, it is necessary to download the newest version file stored on the server. Thus, before modifying the previous version file, the user needs to verify the integrity of the file. If the result is true, user can modify the file. Otherwise, user cannot do the modification. In this way, the version files applying the incremental storage will be able to restore the entire content successfully. In addition, it prevents any predecessor version file in version chain from being corrupted.

5.6 Computational complexity

The computation cost of the algorithm is mainly composed of three parts, i.e., the computation cost of the user, the computation cost of the DSP, and the computation cost of the TPA. Many algorithms divided data blocks into sectors like in [10, 13]. Assume that s is the number of sectors in one block. Let $s = 1$ so that our algorithm can compare with the computation cost of these algorithms in the same way. The computation cost of users mainly focuses on the generation time of verification tags which depends on the execution time of formula (4). Thus, our algorithm needs the computation cost $O(nd)$ while the verification tags of all blocks of d versions are generated, where n is the number of blocks in one version file. Similarly, the computation cost of users are both $O(nd)$ in [10, 13] while the verification tags of d files and each file with n blocks are generated. Secondly, the computation cost of DSP is analyzed. Assume that q blocks are extracted to execute the verification. The computation

cost of our algorithm is $O(q)$ by the random diffusion extraction, and the computation cost in [10] and [13] is both $O(q)$ by the random extraction. Finally, TPA's computation cost is mainly concentrated on the judgment stage, which is determined by formula (5). The computation cost of our algorithm is the same with the algorithms [10, 13] and all $O(q)$. Therefore, the computation cost of our algorithm is the same as these algorithms. Even so, our algorithm verifies the files and its corresponding version files precisely, but these algorithms only verify q blocks which are randomly extracted from all files.

6 Simulation

All the tests are performed on a cloud storage platform and two laptops. The cloud storage platform which is composed of two servers, each equipped with Xeon E5-2403 1.8 GHz CPU and 32 GB RAM, is built as the DSP. The two laptops which are equipped with Intel Core i5-4210M 2.60 GHz CPU and 4 GB memory work as the User and the TPV, respectively. In the process of the experiment, to reduce the experimental time, we set the size of every stored file to 1 MB. In the incremental storage, the ratio of the version file modified to its precursor version file α is 10%, so that the size of each incremental storage file is 0.1 MB. Let the size of each data block be fixed to 512 bytes, the hash algorithm be the secure hash algorithm (e.g., , SHA-256) with high security, and the version storage threshold be set to 4. All the simulation results are the mean of 20 trials.

Considering the relativity between our algorithm with the existing verification algorithms, we call the verification algorithms based on deduplication storage like in [10] P-PAP. We call the verification algorithms based on remotely stored data like in [13] VRSD. P-PAP and VRSD are both oriented to the verification of each individual file. However, our algorithm verifies the version files. We compare the performance of our algorithm with P-PAP in terms of the generation of tags and batch verification of multiple files. We will analyze the performance of the algorithms from several following aspects, i.e., verification storage and transmission overhead, time efficiency, the coverage of the verified file, and verification efficiency.

6.1 Verification storage and transmission overhead

The storage overhead during the verification is mainly the storage of tags of the data blocks that the user generates for all files. We set the size of each tag to z. Let the size of each file be 1M. For P-PAP, the space required to store tags is $2048z$. For VDVF, the space required to store tags of four files in one version group is $2662.4z$. The space storing the same four files for VRSD is $8192z$ which is almost three times of that for VDVF. The storage space for P-PAP is approximately similar to that for VDVF. With the increase of amount of files, the storage space of tags also increases, and the space required in VRSD is almost three times of that in VDVF and P-PAP all the time. This is mainly because that most of the version files apart from the base version in a version group which is totally stored (1 MB) apply incremental storage (0.1 MB). Thus, the storage space of the tags is saved.

In terms of the transmission overhead in the verification, under the premise of the same verification data size, the data transmission overhead of P-PAP and VDVF in the network is the same.

6.2 Time efficiency

In our combination storage model based on the full storage and incremental storage, apart from the first file in a version group, others only deal with the incremental data when generating data tags. Thus, for the same file, only less data blocks need to be treated in our algorithm. In Fig. 8, we compare VDVF with VRSD and P-PAP in terms of the tag generation time with modification ratio and the number of version files. From the figure, we can see that VDVF and P-PAP costs the same time which is less than VRSD in Fig. 8a. However, VDVF and VRSD cost the same time which is more than P-PAP in Fig. 8b while the number of version files is 5. The reason is that VDVF needs to generate the tags of the total file of 1 MB according to our combination storage model.

Alike in the verification, apart from the first file in a version group, others only deal with the incremental file in the verification. The verification execution time with different modification ratio and different version number among VRSD, P-PAP, and VDVF is compared in Fig. 9. Also, VDVF and P-PAP normally cost less time than VRSD. However, VDVF and VRSD cost the same time which is more than P-PAP in Fig. 9b while the number of version files is 5. The reason is that our algorithm generally verifies less data blocks apart from the number of version files being 5.

6.3 Coverage of the verified file

Definition 6 (coverage of the verified file). The number of the checked file in the random extraction or the random diffusion extraction is extracted in the verification, where the random diffusion extraction is extended based on the random extraction according to the relationship among version files.

The coverage of the verified file is closely related to the storage model. There are four files in a version group due to the version storage threshold being 4. The total size of four files in a version group is 1.3 MB. In the case of the same amount of verification data each time, the results of our algorithm VDVF compared with VRSD and P-PAP are shown in Fig. 10. We can see from the figure that

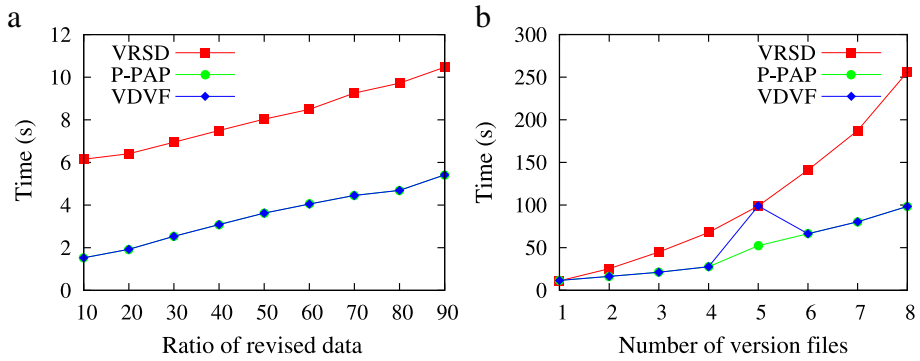

Fig. 8 Tag generation time. **a** different modification ratio. **b** different version number

in the same amount of verified data, P-PAP and VDVF can execute the verification of more files than VRSD. It is mainly because these algorithms apply the different data storage model. VDVF expands the amount of verified files and the coverage of the verified files due to the relationship between version files under the same data verification quantity.

6.4 Verification efficiency

The verification efficiency refers to the proportion of reliable data storage after the data verification, which is the ratio of the number of the verified files to the total number of stored files. In data batch verification, the more the number of files that eventually accept the verification, the higher the reliability of the data and the greater the efficiency of data verification. We set the version storage threshold to 4, and thus, there are four files in a version group. We change the number of files from 1 to 4 in the experiment. For VDVF, if the user randomly selects a number from 1 to 4, it is the number of files that are finally verified. If the version number of chosen file is 5, the number of files that are finally verified is 1 due to 5 minus 4. However, for VRSD and P-PAP, the verification is executed fully in accordance with the files that user randomly extracts. Assume that there are 1000 sets of version files stored on the server, every set has four version files, the total number of the files is 4000. After a certain number of files is randomly selected by the user, these algorithms acquire the proportion of the number of files received verification actually to the total number of the stored files according to respective operation situation. The experimental results are shown in Fig. 11. We can see that in the case of the same number of chosen files, the quantity of the verified files in VDVF is three times more than that in VRSD and P-PAP. It is because that under the condition of the same data extraction, the number of the verified files is expanded due to the version group relationship between different version files. Thus, VDVF realizes the integrity verification of more files and has a higher efficiency than the other two algorithms.

7 Conclusions

Although the cloud deduplication storage can reduce storage cost, the reliability of files will decrease while traditional verification schemes are applied into the integrity

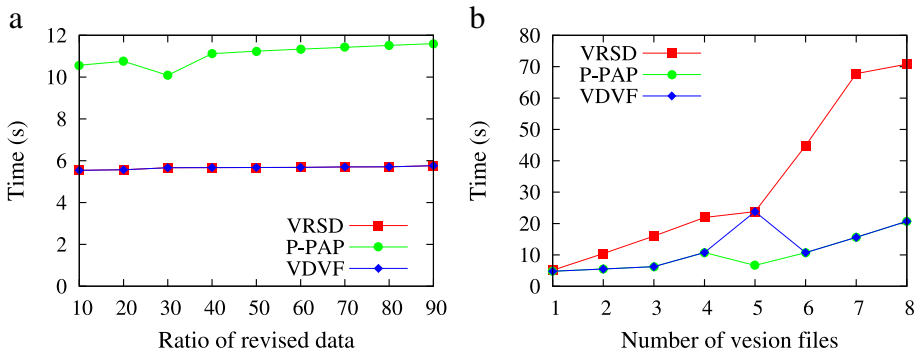

Fig. 9 Verification execution time. **a** different modification ratio. **b** different version number

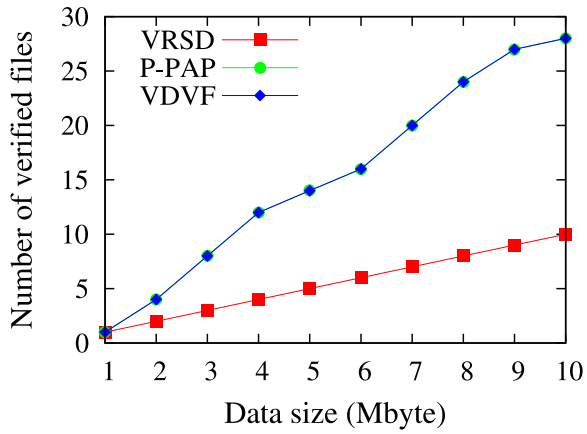

Fig. 10 Number of verified files under the same data verification quantity

verification of version files. The reason is that these schemes do not distinguish version files and treat them the same as common files. These common files can utilize the traditional verification schemes based on the cloud deduplication storage to decrease the verification cost. However, these version files have the same data among each others so that the traditional verification schemes cannot still be used directly. In this paper, we proposed an efficient integrity verification algorithm for remote data storage of version files. This algorithm adopts the combination storage model based on the full storage and incremental storage to reduce the size of file storage and tag storage. Also, it reduces the transmission overhead in the verification due to less verification costs at the same time. The chained key is used to improve security of the storage keys of different version files. The data security in the process of the verification is assured by applying BLS technology. From theoretical and experimental analysis, we can figure out that the algorithm effectively expands the coverage of verified files, improves the efficiency of file verification, and meets the security request at the same time. In the future, we will further make a trade-off between the optimal storage of version files and data integrity protecting to meet the needs of rapid restoration of any version file at any time.

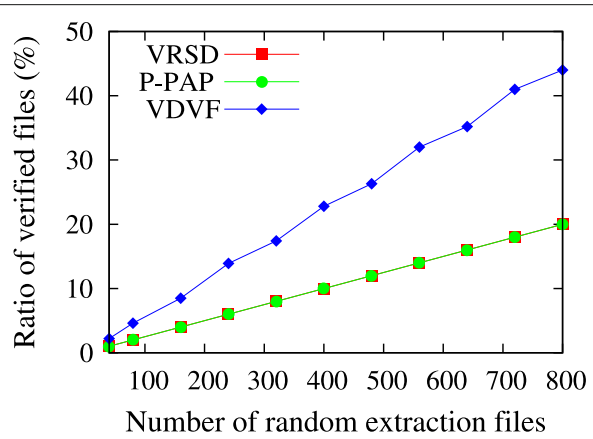

Fig. 11 Proportion of files that actually execute the verification after random extraction

Acknowledgements
The work was sponsored by Natural Science Foundation of Shanghai (Nos.15ZR1400900 and 16ZR1401100), Education and Scientific Research Project of Shanghai (C160076), National Natural Science Foundation of China (No.61772128).

Funding
The work was sponsored by the Natural Science Foundation of Shanghai (nos. 15ZR1400900 and 17ZR1400200), Education and Scientific Research Project of Shanghai (no. C160076), and National Natural Science Foundation of China (no. 61772128).

Availability of data and materials
The datasets used and/or analyzed during the current study are available from the corresponding author on reasonable request.

Authors' contributions
GX carried out the conception and design of the proposed verification algorithm and drafted the manuscript. ML and JL carried out the analysis of simulation and results. LS and XS participated the evaluation of algorithm's correctness. All authors read and approved the final manuscript.

Authors' information
Guangwei Xu is an associate professor in the School of Computer Science and Technology at Donghua University, Shanghai, China. He received the M.S. degree from Nanjing University, Nanjing, China, in 2000, and the Ph.D. from Tongji University, Shanghai, China, in 2003. His research interests include the data integrity verification and data privacy protection, QoS, and routing of the wireless and sensor networks. E-mail: gwxu@dhu.edu.cn

Competing interests
The authors declare that they have no competing interests.

Publisher's Note

Received: 16 February 2018 Accepted: 29 August 2018
Published online: 20 September 2018

References
1. G. Xu, Y. Yang, C. Yan, Y. Gan, A rapid locating protocol of corrupted data for cloud data storage. Ksii Trans. Internet Inf. Syst. **10**(10), 4703–4723 (2016)
2. C. Wang, Q. Wang, K. Ren, W. Lou, in *proceedings of IEEE INFOCOM 2010*. Privacy-preserving public auditing for data storage security in cloud computing (IEEE, San Diego, 2010)
3. G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, D. Song, in *Proceedings of the 14th ACM conference on Computer and Communications Security*. Provable data possession at untrusted stores (ACM, New York, 2007), pp. 598–609
4. H. Shacham, B. Waters, in *Proceedings of the 14th International Conference on the Theory and Application of Cryptology and Information Securit*. Compact proofs of retrievability (Springer, Berlin, 2008), pp. 90–107
5. Y. Deswarte, J.-J. Quisquater, A. Saidane, in *Proceedings of IFIP Advances in Information and Communication Technology*. Remote integrity checking (Springer, Boston, 2003), pp. 1–11
6. G. Ateniese, R. D. Pietro, L. V. Mancini, G. Tsudik, in *Proceedings of the 4th International ICST Conference on Security and Privacy in Communication Networks*. Scalable and efficient provable data possession (ACM, Istanbul, 2008), pp. 1–10

7. C. Erway, A. Küpçü, C. Papamanthou, R. Tamassia, in *Proceedings of the 16th ACM conference on Computer and communications security*. Dynamic provable data possession (ACM, Chicago, 2009), pp. 213–222
8. A. Juels, B. S. Kaliski, in *Proceedings of the 14th ACM conference on Computer and communications security(CCS'07)*. Pors: proof of retrievability for large files (ACM, Alexandria, 2007), pp. 584–597
9. Q. Wang, C. Wang, J. Li, K. Ren, W. Lou, in *Proceedings of the 14th European Symposium on Research in Computer Security*. Enabling public verifiability and data dynamics for storage security in cloud computing (Springer-Verlag Berlin Heidelberg, Saint-Malo, 2009), pp. 355–370
10. K. Yang, X. Jia, An efficient and secure dynamic aduiting protocol for data storage in cloud computing. IEEE Trans. Parallel Distrib. Syst. **24**(9), 1717–1726 (2013)
11. Z. Ismail, C. Kiennert, J. Leneutre, L. Chen, Auditing a cloud provider's compliance with data backup requirements: a game theoretical analysis. IEEE Trans. Inf. Forensic Secur. **11**(8), 1685–1699 (2016)
12. C. Wang, Q. Wang, K. Ren, W. Lou, in *Proceedings of the 17th International Workshop on Quality of Service*. Ensuring data storage securityin cloud computing (IEEE, Charleston, 2009), pp. 1–9
13. X. Liu, W. Sun, W. Lou, Q. Pei, Y. Zhang, in *proceedings of IEEE INFOCOM 2017*. One-tag checker: message-locked integrity auditing on encrypted cloud deduplication storage (IEEE, Atlanta, 2017)
14. B. O'Sullivan, *Mercurial: the definitive guide*. (O'Reilly Media, Inc., Sebastopol, 2009)
15. Y. Zhang, C. Xu, X. Liang, H. Li, Y. Mu, X. Zhang, Efficient public verification of data integrity for cloud storage systems from indistinguishability obfuscation. IEEE Trans. Inf. Forensic Secur. **12**(3), 676–688 (2017)
16. J. Yuan, S. Yu, in *Proceedings of IEEE Conference on Communications and Network Security (CNS)*. Secure and constant cost public cloud storage auditing with deduplication (IEEE, Washington DC, 2013), pp. 145–153
17. F. Grandi, in *Proceedings of the Joint EDBT/ICDT Workshops*. Dynamic multi-version ontology-based personalization (ACM, Genoa, 2013), pp. 224–232
18. K. Jea, H. Feng, Y. Yau, S. Chen, J. Dai, in *Proceedings of 1998 Asia Pacific Software Engineering Conference*. A difference-based version model for oodbms (IEEE, Taipei, 1998), pp. 369–376
19. G. Xu, C.Y.Y.G. Yanbin Yang, A probabilistic verification algorithm against spoofing attacks on remote data storage. Int. J. High Perform. Comput. Netw. **9**(3), 218–229 (2016)
20. S. Hou, L. He, W. Zhao, H. Xie, in *Proceedings of the 2nd International Conference on Industrial and Information Systems*. Research on threshold-based version hybrid storage model (IEEE, Dalian, 2010), pp. 136–139
21. A. Ghobadi, E. H. Mahdizadeh, Y. L. Kee, L. K. Wei, M. H. Ghods, in *Proceedings of the 2th International Conference on Advanced Communication Technology*. Pre-processing directory structure for improved rsync transfer performance (IEEE, Jeju Island, 2011), pp. 1043–1048
22. S. TEZUKA, R. UDA, K. OKADA, in *Proceedings of IEEE 26th International Conference on Advanced Information Networking and Applications*. Adec: assured deletion and verifiable version control for cloud storage (IEEE, Fukuoka, 2012), pp. 23–30
23. Y. Zhu, H. Hu, G.-J. Ahn, M. Yu, Cooperative provable data possession for integrity verification in multicloud storage. IEEE Trans. Parallel Distrib. Syst. **23**(12), 2231–2244 (2012)
24. N. Koblitz, A. Menezes, S. Vanstone, The state of elliptic curve cryptography. Des. Codes Crypt. **19**(2), 173–193 (2000)
25. M. Bellare, J.A. Garay, T. Rabin, in *Proceedings of Advances in Cryptology-EUROCRYPT'98*. Fast batch verification for modular exponentiation and digital signatures (Springer, Espoo, 1998), pp. 236–250