Real-time Proof of Violation for Cloud Storage

Gwan-Hwan Hwang, Wei-Sian Huang, and Jenn-Zjone Peng
Department of Computer Science and Information Engineering
National Taiwan Normal University
Taipei, Taiwan
ghhwang@csie.ntnu.edu.tw

***Abstract*—In this paper we define and explore proof of violation (POV) for cloud storage systems. A POV scheme enables a user or a service provider to produce a precise proof of either the occurrence of the violation of properties or the innocence of the service provider. POV schemes are solutions for obtaining mutual nonrepudiation between users and the service provider in the cloud. Existing solutions for obtaining mutual nonrepudiation only support epoch-based POV. The drawback of an epoch-based POV scheme is that violation of properties can only be detected at the end of an epoch. We propose a real-time POV scheme in which the auditing can be performed at the time of each file operation. To the best of our knowledge, it is the first scheme that can perform real-time POV for cloud storage. In addition, each client device only needs to store a partial hash tree of files in an account. Experimental results are presented that demonstrate the feasibility of the proposed scheme. Service providers of cloud storage can use the proposed scheme to provide a mutual nonrepudiation guarantee in their service-level agreements.**

Keywords—Cloud storage; cloud security; nonrepudiation; proof of violation; service-level agreement

I. Introduction

Cloud storage is a model of networked online storage where data are stored in virtualized storage pools that are hosted by service providers in the cloud. Service providers operate large data centers, and users who require their data to be hosted in these centers buy or lease storage capacity from the service providers. Popular cloud storage systems include Google Drive [1], Dropbox [2], iCloud [3], SugarSync [4], SkyDrive [5], and Box [6].

Security and reliability are two major concerns about cloud storage. Clients are not likely to entrust their data to another company without a guarantee. The basic security features include authentication, confidentiality, data integrity, and nonrepudiation, the first three of which can be easily implemented using cryptographic cloud storage [7]. Cryptographic cloud storage provides confidentiality since the data are secured by cryptographic algorithms. Each file stored in cloud storage needs to be associated with a digital signature that is generated by the private key of the user, so as to guarantee the file integrity.

However, the user still cannot ensure that his/her files will not be lost or that the files retrieved from cloud storage are the latest ones. Consider the situation that some files of the user stored on a cloud storage system are damaged or destroyed because of some kind of internal errors or malicious security attacks, the service provider is likely to then restore the files using a backup of an early version of the files and their associated digital signatures, and can then deny that the user's latest version of files have been lost. This is called a roll-back attack [8], [9], and it is therefore obvious that the *repudiation problem* exists between users and the service provider.

A cloud storage system is responsible to obtain some properties such as integrity, write serializability, and read freshness. However, none of today's cloud storage system provider guarantees for such properties in their service-level agreements (SLAs). For example, the SLA of Amazon S3 guarantees that the service is available with a monthly uptime percentage of at least 99.9% [10]. If the *availability* falls below 99.9%, the customer will be eligible to receive a refund. However, customers do not receive a refund when properties other than availability are violated. For example, if files are damaged or lost, users usually do not have any precise proof to prove it. Since customers cannot make informed decisions about the risk of losing data stored, their incentive to rely on these services is reduced.

If we want to include more guaranteed properties in an SLA, we need a practical scheme that enables the service provider to prove its innocence or the users to prove the service provider's guilt when the user makes false accusations or the service provider violates the desirable properties [7]. Once a *mutual nonrepudiation* is built up based on this scheme, users and the service provider can establish a financial contract in which the cost of the service varies with the level of security desired; the service provider then assures that it will pay an agreed-upon compensation in cases where it is proven that data security has been forfeited or tampered with.

In this paper we define and explore *proof of violation* (*POV*). A POV scheme enables a user or a service provider to produce a precise proof of either the occurrence of the violation of properties or the innocence of the service provider. A POV scheme is with three tuples. First, it should define a set of *properties* which the service provider must not violate when it processes operations requested by users. Second, proofs are based on *attestations*, which are signed messages that bind the

[*] The corresponding author is Gwan-Hwan Hwang (e-mail: ghhwang@csie.ntnu.edu.tw)

users to the requests they make and the service provider. Users and the service provider exchange attestations for every request. Third, *auditing* should be performed according to collected attestations to prove if properties are violated. Auditing can detect the violations of properties of the cloud storage system and the service provider can disprove false accusations made by users; that is, the users of a service with POV cannot frame the cloud. In a cloud storage system, properties for POV may include integrity, write serializability, and read freshness. The framework of a service with POV supports the mutual nonrepudiation guarantee.

The study of POV scheme for cloud systems started from some *epoch-based POV* schemes [11], [12]. When an epoch starts, attestations exchanged between users and the service provider are accumulated. Then, after a period of time, the accumulated attestations are used to perform auditing. The passing of an audit proves that none of the file operations issued in this epoch violated the desirable properties of the service. After the auditing, the accumulated attestations can be discarded and a new epoch starts. An epoch may be several hours or several days. It is perhaps determined by the processing power of the devices or the communication bandwidth.

While epoch-based auditing has been demonstrated to be practical, it is subject to the problem that the violation of properties can only be detected at the end of an epoch. Consider the situation that a document stored in a cloud storage is used to sign an important contact for companies, it may cause huge losses if we use a wrong version. An epoch-based POV scheme can only detect the violation at the end of an epoch rather than at the download time. This situation motivated us to develop a *real-time POV* scheme in which the auditing can be performed when each file operation is executed. An intuitive solution is to store the hash values of all the files in the account on all client devices. Integrity checking can then be performed by verifying a downloaded file *f* according to its hash value. However, it seems impractical for client devices to store all the hash values of files in an account because the files may be accessed by multiple client devices interchangeably, which would also require the cached hash values to be synchronized among the client devices.

To the best of our knowledge, this study is the first to address the issue of real-time POV for cloud storage systems; previous work has only supported epoch-based POV [12] and real-time integrity checking [13]. Integrity checking ensures that the user can always verify if the latest version of the file is retrieved. Real-time POV performs real-time integrity checking and also produces a precise proof when a violation is detected. We propose a real-time POV scheme that can support interchangeable accesses of files by multiple client devices. The cached hash values stored on client devices are synchronized using the accumulated attestations stored by the service provider, which removes the need for client devices to exchange hash values of files. Instead of storing hash values of all files, we propose to store a *partial hash tree* on client devices to reduce memory requirements and the initialization time.

This paper is organized as follows: Section II presents a real-time POV schemes, Section III presents the implementation details and experimental results, Section IV describes related work, and conclusions are drawn in Section V.

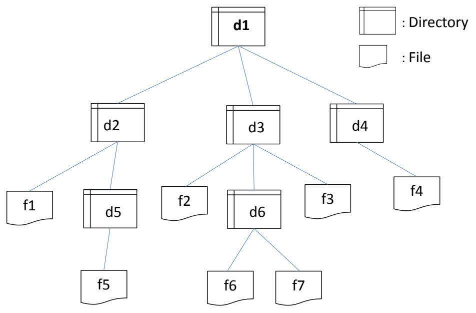

Fig. 1. An example of a file directory

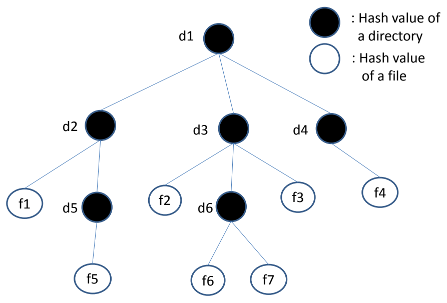

Fig. 2. The hash tree corresponding to directory in Fig. 1

II. A Real-Time POV Scheme

A real-time POV scheme must perform both real-time integrity checking and generate attestations for auditing. An intuitive solution for real-time checking is to have the client device store the hash tree of files of an account. A hash tree is a tree of hashes in which the leaves are hashes of data blocks and the top of the tree is occupied by a *root hash* (or top hash or master hash) [14]. Given the file directory in Fig. 1. Fig. 2 illustrates its corresponding hash tree. Each file in a directory is associated with its hash value. Usually, a cryptographic hash function such as SHA-1, Whirlpool, or Tiger is used for the hashing. This is denoted as *h*(filename); for example, the hash value of file f6 is *h*(f6). The hash value of a directory is the hashing of the concatenation of all the hash values of files and directories in it; such as *h*(d3)=*h*(*h*(f2), *h*(d6), *h*(f3)). The root hash is the hash value of the root directory; that is, *h*(d1)=*h*(*h*(d2), *h*(d3), *h*(d4)).

Once a client device has the latest version of the root hash of the hash tree of the user's directory, it can request a copy of this hash tree from the service provider and verify if the obtained hash tree is valid. It can then check downloaded file *f* by verifying its hash value in the hash tree. After a write operation is successfully performed, it can modify the hash value of the updated file in the hash tree. However, this intuitive solution has the following problems:

1. If files in an account are accessed by multiple client devices interchangeably, we have to synchronize the cached hash trees on all client devices. This is because each client device needs the hash tree of the account when it performs auditing after a file operation.

2. The size of the hash tree increases with the number of files in directories, and so the client device may not have sufficient memory to store the hash tree.

3. Even if the above two problems can be solved, it needs to be possible to produce a precise proof when a violation is detected by a client device.

In this section we propose a real-time POV scheme. The system allows multiple client devices to access files in an account interchangeably. In addition, only a partial hash tree needs to be kept on a client device.

II-1.System Architecture

In this section we present the architecture of the proposed system. We consider the situation where files in a single account are accessed by multiple client devices interchangeably. The system has to be able to perform efficient real-time POV. Since we need to perform POV when a violation is detected, attestations are also necessary.

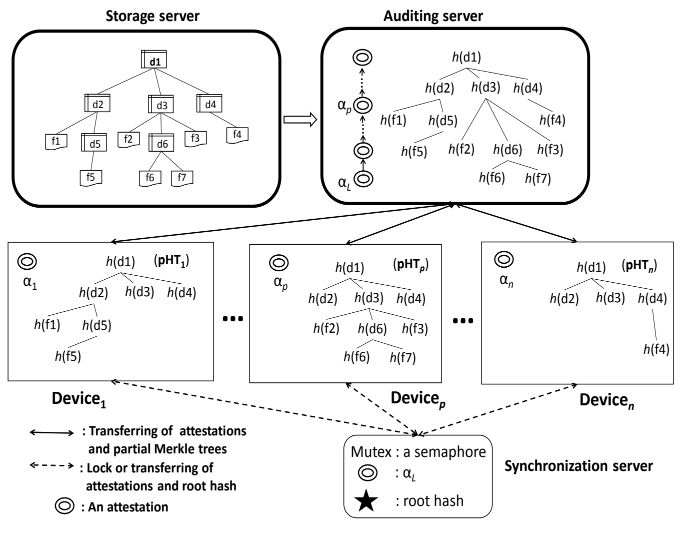

Fig. 3. System architecture

The system architecture is shown in Fig. 3 The system involves a *storage server* (STS), *auditing server* (AS), *synchronization server* (SYS), and some client devices. The files and directories of an account are stored on a STS, while an AS is responsible for exchanging attestations with client devices. The attestations of all the operations are chained in an order by embedding a hash value of the attestation of previous operation in each attestation. They are stored on the AS and we assume that the latest attestation is α_L. The hash tree of the account is also stored on the AS. Fig. 3 shows that files in an account are accessed by *n* devices (i.e., device$_1$ to device$_n$).

Each device needs keep the last attestation it received and a partial hash tree of the account. A partial hash tree is part of a complete hash tree that omits the nodes under some directories. These partial hash trees are stored on client devices. For example, the partial hash tree of device$_1$, pHT$_1$, omits the nodes under directories d3 and d4. We say that a partial hash tree *M covers* a file *f* if the hash value of *f* is in *M*. We can see in Fig. 3 that the partial hash tree of device$_p$, pHT$_p$, covers files f2, f3, f6, and f7. We use pHT(FS) to denote the minimal partial hash tree that covers the files in a file set FS. For example, we have pHT({f1, f5}) = pHT$_1$, pHT({f2, f3, f6, f7}) = pHT$_p$, and pHT({f4}) = pHT$_n$. A partial hash tree can be used to generate the root hash of its corresponding complete hash tree. For example, we can use pHT$_1$ to compute the root hash, which is h(d1)=h(h(d2), h(d3), h(d4))=h(h(h(f1), h(d5)), h(d3), h(d4)). If a file covered by a partial hash tree is updated, we still can compute the new root hash from this partial hash tree. For example, if file f1 is updated and we only have pHT$_1$, we can simply update the hash value of f1 in pHT$_1$ and compute the new root hash correctly from the updated pHT$_1$ because we have the hash values of the directories of these files that are not covered by pHT$_1$.

Devices that may access the same account employ a SYS to exchange the latest attestation and root hash. As shown in Fig. 3, a SYS stores a semaphore Mutex, the latest attestation α_L, and the root hash of the hash tree of a user's account. The STS and AS can be hosted by the same service provider. However, the SYS must not be accessible from the AS. Less than 10 kB of data needs to be stored on the SYS. The user can install this data on another cloud service provider or maintain it himself/herself.

II-2.The Communication Protocol

In this section we present how client devices communicate with the AS and SYS for efficient real-time POV. A file operation by device$_p$ is performed according to the following protocol:

Step 1: Lock(Mutex).
Step 2: Obtain attestation α_L and root hash RH that are stored on the SYS.
Step 3: Assume that α_p is the last attestation received by device$_p$. Send α_L and α_p to the AS.
Step 4: The AS sends back attestations, which are chained from α_L and α_p. We use $\alpha_L \rightarrow \alpha_p$ to denote these attestations.
Step 5: Check if attestations $\alpha_L \rightarrow \alpha_p$ are properly chained.
Step 6: Select all the write operations in attestations $\alpha_L \rightarrow \alpha_p$ and use these operations to update the hash values of files in pHT$_p$. If any write operation to a file is not present in pHT$_p$, update the hash value of the leave node (a directory) in pHT$_p$ where the file is located[1].
Step 7: We assume that device$_p$ is going to perform file operations on files in the account of a user U. Exchange messages with the AS to obtain an attestation as follows:[2]

- Device$_p$ sends a request message Q$_i$, where Q$_i$=(OP$_i$, [OP$_i$]$_{\text{pri(U)}}$), to the AS. Note that OP$_i$

[1] Note that the client device should ask the service provider to answer with the hash value of this directory. In addition, if multiple write operations are performed on the same file, we only have to update the hash value of the file or directory once.

[2] To describe chain hashing formally, [O]$_{\text{pri(x)}}$ is used to denote a digital signature on data object O that is generated by the private key of a subject x, and data objects within square brackets that are separated by commas are first connected and then have cryptographic operations performed on them; for example, [O$_1$, O$_2$, O$_3$]$_{\text{pri(x)}}$ represents a digital signature that signs data objects O$_1$, O$_2$, and O$_3$ in the private key of subject x.

specifies the required operation, such as a file creation, file update, or file read. The hash value of the file to be updated is embedded in Q_i. When the AS receives Q_i, it checks if this is a valid request message by verifying the digital signature, $[OP_i]_{pri(U)}$.

- The STS executes commands in Q_i. Assume that the execution result is L_i. The AS sends a response message R_i, where R_i=(Q_i, L_i, $[Q_i, L_i, R_{i-1}]_{pri(AuditingServer)}$), to user U. The response message R_i, denoted as α_{New}, is the attestation of this file operation. The AS stores this message in the attestation pool of this account. Thus, α_{New} is chained to α_L. Note that Device$_p$ should verify if R_i (α_{New}) is correctly chained to R_{i-1} (α_L).

```
Step 8: IF $Q_i$ is a read operation of file $f$
    IF $pHT_p$ contains the hash value of $f$ THEN
      ● Compute a root hash from $pHT_p$ and check if it equals RH.
      ● Use $L_i$ to compare if the hash value of the obtained file is equal to the hash value of $f$ in $pHT_p$.
      ● Update the attestation stored on the SYS to $\alpha_{New}$.
      ● Unlock(Mutex).
    ELSE /* The hash value of $f$ is absent from $pHT_p$. */
      ● Download $pHT(\{f\})$ from the AS.
      ● Update $pHT_p$ according to $pHT_p = pHT_p \cup pHT(\{f\})$.
      ● Compute a root hash from $pHT_p$ and check if it equals RH.
      ● Use $L_i$ to compare if the hash value of the obtained file is equal to the hash value of $f$ in $pHT_p$.
      ● Update the attestation stored on the SYS to $\alpha_{New}$.
      ● Unlock(Mutex).
    END IF
  END IF
Step 9: IF $Q_i$ is a write operation of file $f$
    IF $pHT_p$ contains the hash value of $f$ THEN
      ● Compute a root hash from $pHT_p$ and check if it equals RH.
      ● Update the hash value of $f$ in $pHT_p$. Compute a root hash RH′ from $pHT_p$.
      ● Update the attestation and the root hash stored on the SYS to $\alpha_{New}$ and RH′, respectively.
      ● Unlock(Mutex).
    ELSE /* The hash value of $f$ is absent fom $pHT_p$. */
      ● Download $pHT(\{f\})$ from the AS.
      ● Update $pHT_p$ according to $pHT_p = pHT_p \cup pHT(\{f\})$.
      ● Compute a root hash from $pHT_p$ and check if it equals RH.
      ● Update the hash value of $f$ in $pHT_p$. Compute a root hash RH′ from $pHT_p$.
      ● Update the attestation and the root hash stored on the SYS to $\alpha_{New}$ and RH′, respectively.
      ● Unlock(Mutex).
    END IF
  END IF
```

The write serializability and read freshness of file accesses in this architecture is maintained based on the chaining of attestations with digital signatures, which occurs in step 7 of the protocol. The protocol uses the Mutex semaphore to ensure that file accesses by the client devices are mutually exclusive. Since only a single client device can lock the semaphore to start the protocol, we can guarantee write serializability and read freshness. Below we show that we can successfully audit each file operation. In step 6, a device$_p$ always updates its partial hash tree pHT_p successfully because we assume that this device can obtain the valid root hash in step 2 and that the attestations will be properly chained. We have the following situations:

- If the file access is a read operation of f and the hash value of f is in pHT_p, device$_p$ can use the hash value of f to verify if the read file is correct.
- If the file access is a read operation of f and the hash value of f is not in pHT_p, device$_p$ downloads the partial hash tree that covers f and extends its partial hash tree accordingly. Since device$_p$ obtains the latest root hash successfully in step 2, it can verify the validity of the updated partial hash tree and then use the hash value of f to verify if the read file is correct.
- If the file access is a write operation of f, device$_p$ simply has to ensure that the attestations are properly chained, its partial hash tree is correctly updated, and the latest attestation and the new root hash are stored on the SYS.

Note that we assume that the SYS is secure. If it is compromised by someone placing incorrect attestations and root hashes on it, it will be possible for the AS to launch a roll-back attack. Over a prolonged period it is possible for client devices to request a huge number of file operations, which could result in an excessive overhead for the service provider to store these attestations. We can employ the method proposed in [12] to discard these attestations.

III. Implementation and Experimental Results

We conducted a series of experiments to demonstrate the feasibility of the proposed real-time POV scheme presented in Section II. The implementation details and experimental results are presented as follows. We used the Java programming language to implement our system. The digest function was "java.security.MessageDigest" with the "SHA-1" algorithm. The RSA algorithm for the public-key encryption function used was "java.security.KeyPairGenerator".

Before the AS starts to serve a user, it needs to compute the hash values of all the files and directories in that user's account so as to load the hash tree into its main memory. Consider the three accounts (A, B, and C) listed in TABLE I. Account A has a small number of files, while account C has a large number of files; actually, account C is the home directory of one of the authors and contains files gathered over a 20-year period.

TABLE I. Three example accounts for performing experiments

	Account A	**Account B**	**Account C**
Number of files	44	22968	28404
Number of directories	6	188	1718
Total size	666 MB	32.9 MB	6.9 GB

We used a high-performance PC running Windows 7. The computation times listed in TABLE II include the time required to generate the hash values for all the files. Consider the situation that the user creates an account and then uploads many files. The service provider can generate the hash for each individual file while the files are being uploaded, so that the hash values of all the files are ready once they have been uploaded, and we only have to compute the hash values of the directories to derive the root hash. TABLE III indicates that the times required to generate a root hash decrease dramatically when using this procedure.

TABLE II. TIMES REQUIRED TO GENERATE THE ROOT HASH FROM NOT-HASHED FILES (IN SECONDS)

Account A	3.404
Account B	16.618
Account C	229.351

TABLE III. TIMES REQUIRED TO GENERATE THE ROOT HASH FROM HASHED FILES (IN SECONDS)

Account A	0.032
Account B	10.49
Account C	52.967

After the hash values of files and directories are computed, the AS needs to load a hash tree into its main memory for the efficient processing of the subtree. In our implementation we employed the Java programming language to construct the AS. The hash tree of an account was stored as a Java object for later access. TABLE IV lists the times required to instantiate a Java object that stores the hash tree based on the hash values of files and directories of the three accounts. We assumed that the hash values of files and directories had been computed.

TABLE IV. TIMES REQUIRED TO INSTANTIATE A JAVA OBJECT THAT STORES THE HASH TREE (IN SECONDS)

Account A	0.051
Account B	0.487
Account C	0.555

TABLE V. SIZES OF XML FILES REPRESENTING THE HASH TREES OF THE THREE ACCOUNTS

Account A	3.2 kB
Account B	2.9 MB
Account C	3.4 MB

We also measured the memory required needed to store a partial hash tree for real accounts. One of the reasons for a client device storing only a partial hash tree is to minimize the memory usage in that device. The size of the complete hash tree for account C was about 3.4 MB, as indicated in TABLE V. TABLE VI lists the sizes of partial hash trees covering some files that were selected randomly from account C. The experimental results indicate that the average size of the partial hash tree to cover 100 randomly selected files was 0.438 MB, which is only about one-tenth of the size of the complete hash tree. If a client device only accesses a small number of files in an account, it only has to cache a small portion of the hash tree.

TABLE VI. SIZES OF PARTIAL HASH TREES COVERING SOME OF THE FILES IN ACCOUNT C

α: Number of randomly picked files
β: Average size of the partial hash tree (in MB)

α	1	10	100	200	300	400	500	1000
β	0.002	0.05	0.438	0.871	1.398	1.676	2.04	2.98

We also evaluated the performance of the protocol presented in Section II-2. TABLE VII lists the times required to perform read and write operations without the auditing protocol. Note that the client devices and the file server were located within the same network segment. The files for reading and writing were selected randomly from account C. These times represent the baseline values used for comparison with the running time when applying our real-time POV scheme.

TABLE VII. AVERAGE TIMES REQUIRED TO PERFORM A READ AND WRITE OPERATIONS WITHOUT THE AUDITING PROTOCOL

Size of files	**Read operation**	**Write operation**
1~100kB	0.012	0.016
100kB~1MB	0.047	0.049
1MB~5MB	0.133	0.154
5MB~10MB	0.216	0.223

TABLE VIII lists the times required to perform read and write operations when the auditing protocol is applied. We first set up a single client device to perform 1000 operations using account C. Files for reads and writes were selected randomly from account C. For different file sizes in a specific range, we computed the average times for reading and writing separately. We also list the run-time separately for files whose hash values are in or not in the partial hash tree of the client devices. TABLE VIII indicates that more time is required when the hash value of the file is not in the partial hash tree, which is due to the client device needing to download a subtree of the complete hash tree from the AS and compute the root hash. According to [12] and TABLE VIII, compared with the C&L scheme, our real-time POV scheme requires an average of about 1.27-fold (i.e., 0.712 seconds/0.557 seconds) longer for performing a read operation (for a 10-MB file) when the hash value of the accessed file is in the partial hash tree stored on the client device. When the hash value of the file is absent in the partial hash tree, a partial hash tree that covers the accessed file needs to be downloaded and the root hash of the partial hash tree on the client device needs to be computed to validate the partial hash tree. This results in the average running time increasing to 1.156 seconds. For a write operation, the client device needs to compute the new hash value and upload to the SYS. The average times increase to 1.367 seconds and 1.721 seconds when the hash value of the file is in and not in the partial hash tree, respectively. The real-time POV is more reliable since each operation is audited immediately. However, there is an overhead associated with refreshing the partial hash tree and checking the downloaded hash values.

TABLE VIII. AVERAGE TIMES REQUIRED TO PERFORM READ AND WRITE OPERATIONS WITH THE AUDITING PROTOCOL (FOR A SINGLE CLIENT DEVICE WITHOUT LOCKING INTO THE SYS)

File operation	Hash value of the file in partial hash tree?	File size	Time
Read	Yes	0~100kB	0.417
		100kB~1MB	0.518
		1MB~5MB	0.651
		5MB~10MB	0.712
		0~100kB	0.751

	No	100kB~1MB	0.922
		1MB~5MB	1.098
		5MB~10MB	1.156
Write	Yes	0~100kB	0.921
		100kB~1MB	1.024
		1MB~5MB	1.239
		5MB~10MB	1.367
	No	0~100kB	1.239
		100kB~1MB	1.359
		1MB~5MB	1.601
		5MB~10MB	1.721

For the results listed in TABLE IX we set up three client devices to access files in an account interchangeably. Files for reading and writing were selected randomly from account C. Each client device performed 333 file operations, thus yielding the execution time for 1000 file operations. Since multiple client devices access files interchangeably, each client device needs to download attestations in steps 4, 5, and 6 as described in Section II-2. This actually involves synchronizing the partial hash trees among client devices. When there is only a single client device, the client device does not need to execute step 6. As listed in TABLE VIII, the average times required to execute a write operation without and with the hash value of the file in the partial hash tree, and a read operation without and with the hash value of the file in the partial hash tree, for files of sizes 5 MB~10 MB, were 0.712 seconds, 1.156 seconds, 1.367 seconds, and 1.721 seconds, respectively. TABLE IX indicates that the corresponding times when multiple client devices were involved were 1.138 seconds, 1.591 seconds, 1.76 seconds, and 2.209 seconds, respectively. These results indicate that the average overhead was about 37.5%. Having three client devices performing file operations concurrently resulted in frequent interchanges. The differences between the values listed in TABLE VIII and TABLE IX reflect the overhead associated with interchangeable file accesses from multiple client devices.

TABLE IX. TIMES REQUIRED TO PERFORM READ AND WRITE OPERATIONS WITH THE AUDITING PROTOCOL (FOR MULTIPLE CLIENT DEVICES)

File operation	Hash value of the file in partial hash tree?	File size	Time
Read	Yes	0~100kB	0.791
		100kB~1MB	0.883
		1MB~5MB	1.098
		5MB~10MB	1.138
	No	0~100kB	1.023
		100kB~1MB	1.268
		1MB~5MB	1.471
		5MB~10MB	1.591
Write	Yes	0~100kB	1.320
		100kB~1MB	1.462
		1MB~5MB	1.618
		5MB~10MB	1.760
	No	0~100kB	1.691
		100kB~1MB	1.719
		1MB~5MB	2.067
		5MB~10MB	2.209

We also performed experiments to demonstrate another advantage of storing the partial hash tree on client devices. If we store the complete hash tree on the client device, a client device needs to download the complete hash tree from the AS when it starts performing file operations. However, according to the protocol in Section II-2, a client device only has to download a partial hash tree that covers the file in the first operation. TABLE X compares the startup times for the two schemes. Account D comprised about 85200 files, 5130 directories, and 21 GB; the number of files in account D was about three times that in account C. The startup time for account D is about five timed longer if we need to cache the complete hash tree. However, the startup time for our scheme is almost constant since we only need to cache a partial hash tree. In this case the required startup time is the time required to fetch a subtree that covers the first accessed file (about 0.002MB according to TABLE VI).

TABLE X. STARTUP TIMES FOR A CLIENT DEVICE STARTING TO PERFORM A FILE OPERATION (IN SECONDS).

	Scheme	Minimum time	Maximum time	Average time
Account C	Partial hash tree	0.034	0.053	0.045
	Complete hash tree	0.681	0.752	0.701
Account D	Partial hash tree	0.034	0.055	0.046
	Complete hash tree	3.428	3.617	3.478

IV. RELATED WORK

Existing cloud storage systems, including Google Drive [1], Dropbox [2], iCloud [3], SugarSync [4], SkyDrive [5], and Box [6], do not support mutual nonrepudiation SLAs. One way to recover from unwanted changes to files on a server is based on replicating the data on multiple servers [15], [16], [17], [18], [19], [20]. For example, Plutus is a cryptographic storage system that enables secure file sharing without relying on trusting the file servers [15]. All data are stored in an encrypted form and the distribution of all keys is handled in a decentralized manner. Client devices read and write data to each replica. A client device can recover a corrupted file by contacting a sufficient number of replicas. However, such systems cannot support any kind of POV.

The challenge-response protocol and proofs of retrievability (POR) were proposed to perform remote integrity checking for file systems [21], [22]. In these methods the verifier sends a request to the server periodically for it to compute a checksum on a file (specified as a request parameter) and return the result to the verifier. The verifier then compares the returned result with a locally stored reference checksum for the same file. Several POR-based remote integrity checking protocols were recently proposed to allow a third-party auditor to check the data integrity on the remote server [23], [24]. The auditor must be a trusted organization (e.g., the government) that can provide unbiased auditing results for both data owners and cloud servers. Since the auditing is performed by the auditor, many servers must be installed when there is a large number of users. Individual files or blocks of files are usually randomly selected for auditing, and hence it cannot be guaranteed that all the files have been audited. Moreover, the third-party auditor cannot detect violations of write serializability and read freshness.

Some systems consider the cloud storage to be inherently untrustworthy and so provide a way to detect violations of integrity, write serializability, and read freshness without the need for data replication or a trusted third party. SiRiUS is a secure file system that is implemented with a layered design over insecure networks [25]. Files stored on the file server are kept in two parts: one part contains the file meta data (md-file) and the other the encrypted data file (d-file). SiRiUS uses a hash tree to guarantee the freshness of the md-file. The client device of a user generates a hash tree consisting of all his/her md-files. The freshness guarantees in SiRiUS only apply to the md-files, and so a roll-back attack that replaces the newest version of the d-file with an older version is possible. SUNDR is a network file system designed to store data securely on untrusted servers [26]. SUNDR makes it possible for client devices to detect any attempts at unauthorized file modification by malicious server operators or users. The protocol used by SUNDR implements a property called fork consistency, which guarantees that client devices can detect any integrity or consistency failures as long as they can see each other's file modifications. Client devices maintain a list of the file versions in the storage server. When user U performs a file-system operation, his/her client device acquires the global lock and downloads the latest version list for each user and group. These version lists can be used to detect violations. SUNDR deals with the violation detect problem using so-called forking semantics, which has also been employed in other studies [27], [28], [29]. The above-described solutions guarantee integrity, and the addition of some extra out-of-band communication among the client devices can also achieve a related notion of consistency. The Venus system eliminates this communication problem by delaying establishing the consistency of operations until some time after they have finished [9]. Generally speaking, all the above-described schemes only support the detection of violation, and cannot offer a way to prove the occurrence of violations to a third party.

Iris is a cloud file system that supports real-time integrity checking. It ensures that the customer can always verify if the latest version of the data is retrieved and thus prevents roll-back attacks that revert the file system to a previous state [13]. Iris is designed to be applied in an enterprise. A centralized and trusted portal residing within the enterprise trust boundary acts as an intermediary for all communication between enterprise client devices and the cloud. The portal caches data that have recently been accessed by enterprise users and the hash tree of the users' files. The portal employs a semaphore to ensure mutual accesses for the cached hash tree, and this could be a bottleneck of the system. Although it can perform dynamic POR, Iris cannot support POV.

Yumerefendi and Chase presented a framework, CATS, for accountability for network storage [11]. CATS servers can supply cryptographic proofs that their read responses resulted from valid write operations. Every CATS request and response carries a digital signature that uniquely authenticates its sender and proves the integrity of the message. To support write serializability and read freshness, the server accepts the write request only if at the time of the write the version stamp of the object given by the writer is equal to current. CloudProof can support epoch-based POV [30]. Users can not only detect violations of integrity, write serializability, and read freshness, they can also prove the occurrence of these violations to a third party. With signed messages and chain hashing, the CloudProof architecture can provide the nonrepudiation and write-serializability properties. Freshness is guaranteed by periodically auditing data. However, client devices need to exchange the latest user-side attestation if they access the same account. Otherwise, client devices must keep all the user-side attestations.

In our previous work, we propose a practical epoch-based POV scheme called C&L scheme [12]. We first show that the chain-hashing scheme cannot resist roll-back attack from service provider unless client devices keep all the attestations or there exists a way to broadcast the last attestation to all the client devices. In C&L scheme the client devices that may access files in a single account do not have to exchange attestations during their file operations, and each client device only has to keep the last attestation it received. We also present how to apply the hash tree to clean up the accumulated attestations. We conducted numerous experiments to examine the overhead of the proposed scheme, and the results therefrom demonstrate that the proposed epoch-based POV scheme is practical.

V. Conclusion

Mutual nonrepudiation between users and the service provider is crucial in cloud storage. POV schemes are formal trust mechanisms that allow the user to prove the occurrence of violation of specified properties to a third party according to accumulated attestations. POV schemes are critical to enabling security guarantees in SLAs, wherein users pay for a desired level of security and are assured they will receive a certain compensation in the event of the service provided by the service provider being inadequate in a defined way. In this paper we have defined and explored POV for cloud storage systems, and propose a real-time scheme. A client device only needs to store a partial hash tree in the real-time POV scheme, with the scheme proposing solutions for how to synchronize the cached partial hash trees of multiple client devices. The presented experimental results have demonstrated the feasibility of the proposed scheme.

The service providers of cloud storage can use the scheme proposed in this paper to provide a mutual nonrepudiation guarantee in their SLAs. The scheme has the overhead of exchanging response and reply-response messages between client devices and the service provider, and a compromise is that only files contained within certain specified directories will benefit from the protection provided by the POV scheme.

Acknowledgment

The corresponding author is Gwan-Hwan Hwang and this work was supported in part by NSC of Taiwan under Grant NSC100-2219-E-003-001 and NSC101-2219-E-003-001.

References

[1] "Google Drive," https://drive.google.com/start#home.

[2] "Dropbox," https://www.dropbox.com/home.

[3] "iCloud," https://www.icloud.com/.

[4] "SugarSync," https://www.sugarsync.com/.

[5] "Microsoft SkyDrive," http://skydrive.live.com/.

[6] "Box," http://www.box.net.

[7] S. Kamara and K. Lauter, "Cryptographic cloud storage," *Financial Cryptography and Data Security*, ser. Lecture Notes in Computer Science. Springer Berlin/Heidelberg, 2010, vol. 6054, pp. 136–149.

[8] J. Feng, Y. Chen, D. Summerville, W.S. Ku, and Z. Su., "Enhancing Cloud Storage Security Against Roll-back Attacks with a New Fair Multi-party Non-repudiation Protocol," *IEEE Consumer Communications and Networking Conference* (CCNC), 2011.

[9] A. Shraer, I. Keidar, C. Cachin, Y. Michalevsky, A. Cidon, and D. Shaket., "Venus: Verification for untrusted cloud storage," *ACM CCSW* 2010, pp. 19-30.

[10] "Amazon S3 Service Level Agreement," http://aws.amazon.com/s3-sla/.

[11] Aydan R. Yumerefendi and Jeffrey S. Chase, "Strong Accountability for Network Storage," *ACM Transactions on Storage*, Vol. 3, No. 3, October 2007.

[12] Gwan-Hwan Hwang, Jenn-Zjone Peng, and Wei-Sian Huang, "A Mutual Nonrepudiation Protocol for Cloud Storage with Interchangeable Accesses of a Single Account from Multiple Devices," *The 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (IEEE TrustCom-2013)*, Melbourne, Australia, 16-18 July.

[13] E. Stefanov, M. van Dijk, A. Oprea, and A. Juels, "Iris: A scalable cloud file system with efficient integrity checks," *The 28th Annual Computer Security Applications Conference (ACSAC 2012)*. ACM, 2012.

[14] R. C. Merkle. "A Digital Signature Based on a Conventional Encryption Function," *Proc. Conf. Theory and Applications of Cryptographic Techniques on Advances in Cryptology (CRYPTO '87)*, 1987.

[15] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu, "Plutus: Scalable Secure File Sharing on Untrusted Storage," in *USENIX FAST* (2003).

[16] A. Adya, W.J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J.R. Douceur, J. Howell, J.R. Lorch, M. Theimer, and R. Wattenhofer, "Farsite: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment," *Proc. Fifth Symp. Operating System Design and Implementation (OSDI)*, pp. 1-14, 2002.

[17] G. Ganger, P. Khosla, M. Bakkaloglu, M. Bigrigg, G. Goodson, S. Oguz, V. Pandurangan, C. Soules, J. Strunk, and J.Wylie, "Survivable storage systems," *DARPA Information Survivability Conference & Exposition II, 2001. DISCEX'01. Proceedings*, vol. 2. IEEE, 2001, pp. 184–195.

[18] A. Rowstron and P. Druschel, "Storage Management and Caching in PAST, a Large-Scale, Persistent Peer-to-Peer Storage Utility," *SIGOPS Operating Systems Rev*., vol. 35, no. 5, pp. 188-201, 2001.

[19] J. Strunk, G. Goodson, M. Scheinholtz, C. Soules, and G. Ganger, "Self-Securing Storage: Protecting Data in Compromised Systems," *Proc. Fourth Symp. Operating Systems Design and Implementation*, Oct. 2000.

[20] A. Bessani, M. Correia, B. Quaresma, F. Andr´e, and P. Sousa, "Depsky: Dependable and secure storage in a cloud-ofclouds," *EuroSys'11*, 2011.

[21] Y. Deswarte, J.-J. Quisquater, and A. Saidane, "Remote integrity checking," *Proc. Conference on Integrity and Internal Control in Information Systems (IICIS'03)*, November 2003.

[22] A. Juels and B. S. Kaliski, "PORs: Proofs of retrievability for large files," *Proc. of ACM CCS*, 2007.

[23] Y. Zhu, G. Ahn, H. Hu, S. Yau, H. An, and S. Chen, "Dynamic Audit Services for Outsourced Storages in Clouds," *IEEE Transactions on Services Computing*, vol. PP, no. 99, p. 1, 2011.

[24] Kan Yang and Xiaohua Jia, "An Efficient and Secure Dynamic Auditing Protocol for Data Storage in Cloud Computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 9, 2013.

[25] E. Goh, H. Shacham, N. Modadugu and D. Boneh. "Sirius: securing remote untrusted storage," *Proceedings of NDSS*. 2003. pp. 131- 145.

[26] J. Li, M. Krohn, D. Mazi`eres, and D. Shasha, "Secure untrusted data repository (SUNDR)," *Proceedings of the Symposium on Operating Systems Design and Implementation*, 2004.

[27] C. Cachin, A. Shelat, and A. Shraer. "Efficient fork-linearizable access to untrusted shared memory," *Proc. 26th ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 129–138, 2007.

[28] M. Majuntke, D. Dobre, M. Serafini, and N. Suri, "Abortable fork-linearizable storage," in T. F. Abdelzaher, M. Raynal, and N. Santoro, editors, *Proc. 13th Conference on Principles of Distributed Systems (OPODIS)*, vol 5923, pp. 255–269, 2009.

[29] C. Cachin and M. Geisler, "Integrity protection for revision control," in M. Abdalla and D. Pointcheval, editors," *Proc. Applied Cryptography and Network Security (ACNS)*, vol 5536, pp. 382–399, 2009.

[30] R. A. Popa and J. R. Lorch. "Enabling Security in Cloud Storage SLAs with CloudProof," *USENIX Annual Technical Conference (USENIX), 2011.*