PAPER *Special Section on Information and Communication System Security — Against Cyberattacks —*

Efficient Data Possession Auditing for Real-World Cloud Storage Environments

Da XIAO[†a)], **Lvyin YANG**[†], ***Nonmembers***, **Chuanyi LIU**[††], ***Member***, **Bin SUN**[†], ***and* Shihui ZHENG**[†], ***Nonmembers***

SUMMARY Provable Data Possession (PDP) schemes enable users to efficiently check the integrity of their data in the cloud. Support for massive and dynamic sets of data and adaptability to third-party auditing are two key factors that affect the practicality of existing PDP schemes. We propose a secure and efficient PDP system called IDPA-MF-PDP, by exploiting the characteristics of real-world cloud storage environments. The cost of auditing massive and dynamic sets of data is dramatically reduced by utilizing a multiple-file PDP scheme (MF-PDP), based on the data update patterns of cloud storage. Deployment and operational costs of third-party auditing and information leakage risks are reduced by an auditing framework based on integrated data possession auditors (DPAs), instantiated by trusted hardware and tamper-evident audit logs. The interaction protocols between the user, the cloud server, and the DPA integrate MF-PDP with the auditing framework. Analytical and experimental results demonstrate that IDPA-MF-PDP provides the same level of security as the original PDP scheme while reducing computation and communication overhead on the DPA, from linear the size of data to near constant. The performance of the system is bounded by disk I/O capacity.

key words: *provable data possession, third-party audit, tamper-evident audit logs, trusted hardware*

1. Introduction

Cloud storage enables users to store large amounts of data at low costs on remote hardware, hosted by Storage Service Providers (SSPs). Despite its benefits and popularity, this new data hosting paradigm introduces additional security challenges [16]. Users are frequently concerned that their outsourced data may become damaged or lost for various reasons; they may also be concerned that SSPs might attempt to conceal data loss incidents caused by administrative errors in order to preserve their reputations, or discard data that are rarely accessed.

As a potential solution to the above issues, Provable Data Possession (PDP) [1] enables users to verify if their outsourced data is being kept intact, without downloading all of their data from the cloud. Though numerous PDP schemes have been proposed in recent years [2]–[6], [12]–[14], there is still considerable room of improvement to apply these theoretical schemes to actual cloud storage environments. Two issues seriously affect the practicality of PDP schemes. First, data stored in the cloud is massive and dynamic in nature, making the original PDP scheme [1] for a single static file unsuitable. Later schemes that support dynamic data updates [2]–[6] often introduce complicated authenticated data structures, which add significant overhead to the schemes. Moreover, when naively extending these single-file schemes to apply to a large number of files, which is a common scenario in cloud storage, the overhead becomes prohibitively larger. Second, to alleviate the burden on users, a number of schemes advocate the use of Trusted Third Parties (TTPs) to perform data possession auditing on behalf of users [5], [12]–[14]. However, they do not explicitly propose how to implement such a TTP in actual cloud storage environments. Conventionally, TTPs are instantiated by independent organizations. This introduces the problem of potential sensitive information leakage to the TTP [5], as well as increased system deployment and operational costs. Alternatively, the TTP can be integrated with the cloud storage service. However, it may be difficult to convince users that the audit results produced in such a setting are trustworthy.

In this paper, we improve the practicality of data possession auditing by tackling the above two issues from a different perspective. While most existing schemes are built upon generalized assumptions about usage patterns and security model of an idealized outsourced storage model, we demonstrate that it is possible to construct more efficient solutions by identifying and leveraging some important characteristics of real-world cloud storage environments and applications. We make contributions in the following aspects:

1) To efficiently audit massive and dynamic sets of data, we identify the specific data update pattern for cloud storage and propose a group-based update model. Based on this observation, we define a new PDP model called Multiple-File PDP (MF-PDP) and construct a secure and efficient MF-PDP scheme. The overhead of auditing massive and dynamic sets of data is reduced by checking a group of files aggregately.

2) To address concerns about third-party auditing, we adopt the semi-trusted SSP assumption and propose a data possession auditing framework based on integrated TTPs, which are tamper-resistant hardware devices bundled with cloud servers that act as data possession auditors (DPAs). The semi-trusted SSP interacts with the DPA to produce

Manuscript received May 26, 2014.
Manuscript revised September 30, 2014.
Manuscript publicized December 4, 2014.

[†]The authors are with School of Computer Science, Beijing University of Posts and Telecommunications, Beijing, 100876, China.
[††]The author is with School of Software Engineering, Beijing University of Posts and Telecommunications, Beijing, 100876, China.
a) E-mail: xiaoda99@gmail.com

DOI: 10.1587/transinf.2014ICP0016

tamper-evident logs, providing trustworthy audit results to users. In this manner, third-party auditing costs are reduced, and the risk of information leakage to auditors is minimized.

3) To integrate MF-PDP with the auditing framework, we design the interaction protocols between the user, the SSP and the DPA, which constitutes IDPA-MF-PDP. The protocols impose minimal processing and storage requirements on the DPA, making hardware-based implementation feasible.

4) We have implemented a prototype IDPA-MF-PDP system, and have conducted a theoretical analysis and experimental evaluation of its security and efficiency. Results demonstrate that IDPA-MF-PDP provides the same level of security as the original PDP scheme, while reducing computational and communication overhead on the DPA from linear in the size of the file group to near constant. The tamper-evident audit log provides a trustworthy record of audit results. The performance of the system is bounded by disk I/O, rather than cryptographic computation.

2. Multiple-File Prove Data Possession

2.1 Data Update Model

We observed that, in contrast to data update patterns for traditional storage systems (e.g., a file system), file objects written to cloud are rarely updated. For example, in backup and archiving, which are currently the dominant cloud storage applications, the data written to cloud are typically file system snapshots or archival objects, which are static in nature. The data operation interfaces provided by primary cloud storage providers also reflect this pattern. For example, Amazon S3 provides a simple object interface that supports read, write, list, and delete operations for an entire object [7]. In addition, delete operations for cloud data are relatively rare events and tend to occur in a batch manner, such as when an archive of documents exceeds its retention period.

Based on the above observations, we propose a data update model for cloud storage using file groups as the basic update units. It consists of three update operations: 1) *create_group*: an empty file group is created; 2) *add_file*: a new file is added to the group, after which it cannot be modified or removed; 3) *remove_group*: the file group is removed, along with all of the files in the group. Note that update operations for an existing file can be indirectly supported by adding a new version of the file while retaining the old version, similar to the versioning feature in the S3 interface [8].

2.2 MF-PDP Scheme

A MF-PDP scheme is a collection of five polynomial-time algorithms:

KeyGen() $\rightarrow (pk, sk)$ is a probabilistic key generation algorithm run by both the user and the DPA. It produces a pair of matching public and secret keys. We use (pk, sk) to denote the user's key pair.

Add$(sk, F, \alpha, GID) \rightarrow (F', M, \alpha')$ is an algorithm run by the user to add a file to a file group. It encodes the file, generates the verification metadata for it and updates a persistent state, which is maintained for each file group by the DPA. It accepts as inputs the user's secret key sk, a file F, the persistent state α and the group identifier GID, and produces the encoded file F', the verification metadata M and the updated persistent state α'.

Challenge$(\alpha) \rightarrow chal$ is a probabilistic algorithm run by the DPA to generate a challenge of the current file group. It accepts as input the persistent state α, and produces a challenge $chal$.

Prove$(pk, chal, F', M, \alpha) \rightarrow P$ is run by the SSP to generate a proof for a challenge. It accepts as input the user's public key pk, the challenge $chal$, a group of file blocks F', their corresponding verification metadata M, and the persistent state α. It produces a proof P.

```
KeyGen()→(pk, sk)
  1. Generate pk= (N, e, g) and sk= (N, d, g) such that (N, e) and (N, d) are matching RSA public and secret keys, g is a generator of $QR_N$.
  2. Output (pk, sk).
Add(sk, F, α, GID) →(F', M, α')
  1. Let (N, d, g) = sk and (B) = α.
  2. Split F into t blocks F[1], …, F[t], each L bits long.
  3. For i ∈[1, t]:
     a) Compute a homomorphic authenticator for block F[i] where B+i is it's virtual block index:
          $A[i] = (H(GID \,||\, B+i) \cdot g^{F[i]})^d \bmod N$
  4. Update index base: B'= B+ t.
  5. Output (F'= F, M= (A[1], … , A[t]), α'= (B')).
Challenge(α)→chal
  1. Generate random challenge keys k.
  2. Determine the number of sampled blocks c from α and target security level.
  3. Output chal= (c, k).
Prove(pk, chal, F', M, α)→P
  1. Let (N, e, g) = pk and (c, k) = chal.
  2. For j ∈[1, c]:
     a) Compute the virtual index of the $i^{th}$ sampled block:
          $i_j = \pi[B]_k(j)$ .
     b) Locate the sampled block and its authenticator.
  3. Compute $A = A[i_1] \cdot ... \cdot A[i_c] \bmod N$ .
  4. Compute $F = F[i_1] + ... + F[i_c]$ .
  5. Output P= (A, F).
Verify(pk, chal, P, α) →{0, 1}
  1. Let (N, e, g) = pk, (c, k) = chal, (A, F) = P and (B) = α.
  2. For j ∈[1, c]:
     a) Compute $i_j = \pi[B]_k(j)$ .
  3. Compute $\tau_1 = A^e \bmod N$ .
  4. Compute $\tau_2 = H(GID \,||\, i_1) \cdot ... \cdot H(GID \,||\, i_c) \cdot g^F \bmod N$ .
  5. If $\tau_1 == \tau_2$ then output 1; otherwise output 0.
```

Fig. 1 Algorithms of MF-PDP scheme.

Verify$(pk, chal, P, \alpha) \rightarrow \{0, 1\}$ is run by the DPA to validate a proof. It accepts as input the user's public key pk, the challenge $chal$, the proof P and the persistent state α. It returns a '1' bit if the verification succeeds, and '0' otherwise.

A naive MF-PDP scheme can be obtained by checking each file separately, using some single-file PDP scheme (e.g. [1]). Obviously, the complexity of such a scheme is $O(n)$, where n is the number of files in the group. Here we describe a more efficient scheme that aggregates challenges proposed in [11]. The primary idea is to consider a file group as a single virtual file; new files are added to the group by appending them to the end of the virtual file. By checking the virtual file in one challenge, all files in the group are checked. Every block in the group has a virtual index, i.e., its index in the virtual file, which is used to compute homomorphic authenticators.

Our MF-PDP scheme is described in Fig. 1, which is a simplified version of the scheme proposed in [11]. Similar to single-file PDP schemes, in the challenge phase of MF-PDP, the DPA asks the SSP for proof of c file blocks designated by virtual indices. SSP returns an aggregate block and an aggregate authenticator for these blocks, which are verified by the DPA. The c blocks are randomly distributed among all the blocks in the group. We utilize two cryptographic primitives: a cryptographic hash function H for generating authenticators, and a family of pseudo-random permutations (PRP) π[T] for generating the random virtual indices of the sampled blocks. The persistent state α comprises the size of the current file group, denoted by B, and is initialized to zero when the group is created. Refer to [11] for more details.

3. Auditing Framework with Integrated DPA

As an auditing framework-independent and concrete PDP scheme, MF-PDP enhances the efficiency of checking a group of files. To further reduce the overhead of the entire audit workflow and alleviate the burden on users, we propose an auditing framework based on integrated DPAs. This section describes the framework itself. The interaction protocols that integrate MF-PDP with the framework will be discussed in the following section.

3.1 Security Model and Auditing Framework

Our framework's security model has three roles: User, Service Storage Provider (SSP), and Data Possession Auditor (DPA). The user is trusted. The SSP is semi-trusted, i.e., it will comply with the protocols most of the time, unless it attempts to conceal data corruption. Similar to the economically rational adversary SSP model [17], this reflects the fact that an SSP's main incentive is not to corrupt users' data, but to avoid reputational and economic loss. The DPA, as the trusted entity, executes data possession auditing independently and will not conspire with any malicious party.

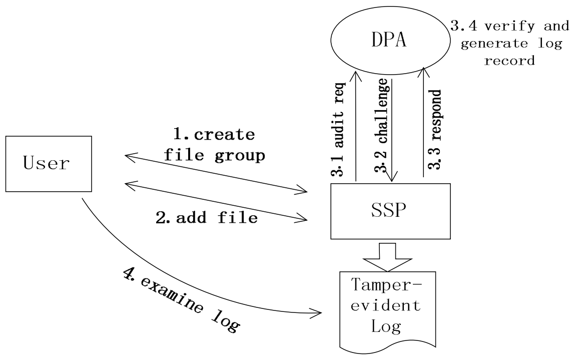

Fig. 2 Interaction process in the auditing framework.

The user only trusts the audit results generated by the DPA.

To reduce the deployment and operational costs of third party auditing, and minimize information leakage risks, the DPA is integrated with cloud servers in our auditing framework. The DPA executes possession auditing by periodically interacting with the SSP, and generates logs that record audit results. The logs are stored by the SSP and can be verified by any party holding the DPA's public key. Users can remain off-line most of the time, and check the audit results by examining the log whenever necessary. The interactions between the user, SSP and DPA are displayed in Fig. 2. (Step 1 and Step 2 also require interactions between the SSP and the DPA, which are not displayed to maintain clarity. Steps 2 and 3.1 through 3.4 can be interwoven and executed repeatedly.) We emphasize that the integrated DPA is a passively responding entity, unlike a conventional TTP. Auditing operations are initiated by the SSP and audit results are stored by the SSP. This reduces the processing and storage requirements for the DPA, making hardware-based implementation feasible.

Since the DPA is integrated with the untrusted cloud servers, it is imperative to specify how to bootstrap the trust of users on DPAs. In our design, an independent third party (called DPA provider) provides DPAs to different SSPs. The DPA provider maintains a mapping between SSPs and theirs DPAs' public keys. When a user uses an SSP's service, he retrieves the public keys of DPAs on this SSP from the DPA provider. As the user does not get DPA's public key from the SSP, the case that the SSP gives a forged DPA public key to the user is avoided. The above is only a high-level description of the DPA deployment process. More detailed design is omitted due to space limitations. Note that the DPA provider does not take part in the interaction process of auditing and thus will not become a bottleneck.

3.2 Implementation of DPA

We propose implementing the DPA with a piece of tamper-resistant, trusted hardware having the following properties: 1) No entities can rewrite its state and pre-loaded programs, or access any of its secret key information; 2) When a physical attack is detected, it will destroy itself by resetting the memory area containing all critical secret data. Existing secure coprocessor products, such as the IBM 4764 [15], can satisfy the requirements listed above. Because the DPA is attached to the cloud servers, e.g., in the form of a PCI card, the SSP should guarantee that the DPA will be functioning properly and will not be attacked in either a physical or logical manner. If the audit log becomes unverifiable because of the damage to the DPA, the SSP will be held accountable.

In real-world cloud storage environments, there are typically a large number of file groups owned by a large number of users. Because of the limited processing capacity of a single DPA, multiple DPAs should work collaboratively to audit a large number of file groups. These DPAs could form a cluster or a virtual cloud [18] in the SSP's intranet. The simplicity of the functionality and internal state of the DPA, discussed in Sect. 4, ensures the scalability of the DPA cluster, e.g., each DPA may be responsible for file groups owned by a subset of users. The detailed design of multiple DPAs is beyond the scope of this paper.

3.3 Tamper-Evident Audit Logs

Audit logs, which are structured on-disk data, are generated by the DPA to record audit results (Fig. 3). A Log Entry (LE) records the result of one audit operation. LEs for one file group are linked into a list. By consulting the list, a user can check the audit history of that file group. An LE contains five fields: *result* is the result of this audit operation, equaling '1' for intactness and '0' otherwise; *time* is the timestamp indicating when the LE was generated; thus, it ensures the freshness and non-reproducibility of an LE; *eid* is the unique identifier of an LE in the log; *prev_eid* is the *eid* of the last LE auditing the same file group, and is used to link the list; *sig* is the RSA signature generated by the DPA using its secret key *dsk* based on the aforementioned four fields.

Every file group has an Entry reference (ER), which contains five fields: *UID* is the identifier of the user and *GID* is the identifier of the file group; *time* is the timestamp indicating when the ER was created or modified; *sig* is an RSA signature generated by the DPA using *dsk* based on the aforementioned four fields. ER has the same *eid* as the last LE of the group. If the file group has not been audited yet, i.e., there is no LE, the ER's *eid* is NULL. An ER is generated when the file group is created, and acts as the head node of the list.

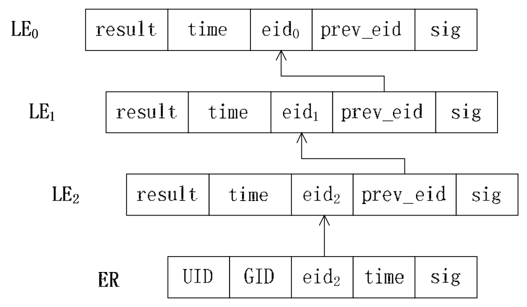

Fig. 3 Audit log of a file group.

An example of the audit log of a file group is shown in Fig. 3. Notice that when multiple file groups coexist, their LEs may be physically stored in a interwoven way, though logically separable by the *prev_eid* field.

As the tamper-evident audit logs are signed by the DPA and can be publicly verified, they can be seen as impartial evidence. Thus, neither the User nor the SSP can cheat in this audit framework for its own sake.

4. Interaction Protocols

Assume that User has obtained DPA's public key from the DPA provider, as described in Sect. 3.1. In order to integrate MF-PDP with the audit framework, in this section we describe the interaction protocols between User, DPA and SSP. It encompasses the registration step in DPA, the procedures of challenge-response in MF-PDP and trustworthy audit results creation and retention, forming the complete IDPA-MF-PDP system. The key design principle is to minimize the computation and storage requirement on DPA while ensuring security. To achieve this goal, special attention is paid to the fact that the integrated DPA, different from a conventional TTP, is a passively responding device who has direct communication channel only with the SSP.

In all the protocols, (pk, sk) stands for the user's key pair generated by method MF-PDP.KeyGen (see Sect. 2.2), and (dpk, dsk) for the DPA's key pair. *CHECK*() is run by DPA to verify the validity of the inputs. It is unlikely that *CHECK*() fails. If it does, *CHECK*() terminates the interaction by returning an error message to SSP. Then SSP can correct the mistake and resend a request to DPA to continue the protocol. $SIGN\_MSG_{sk}(M)$ is a function run by DPA to produce a signed message from M using private key sk and $VERIFY\_MSG_{pk}(M)$ is used to verify a singed M using public key pk.

4.1 Protocol for Registering User to DPA

As the initialization step of the interaction, Fig. 4 explains

```
USER:
    MF-PDP.KeyGen() → (pk,sk)
USER → SSP:
    pk
SSP:
    UID ← UUID()
    write(UID, pk)
SSP → DPA:
    (UID,pk)
DPA:
    store(UID,pk)
    userInfo ← $SIGN\_MSG_{dsk}(UID, pk)$
DPA → SSP → USER:
    userInfo
USER:
    (UID,pk) ← $VERIFY\_MSG_{dpk}(userInfo)$
    write(UID)
```

Fig. 4 Protocol for registering User to DPA.

```
USER→SSP:
    CREATE_GROUP_REQUEST()
SSP→DPA:
    CREATE_GROUP_REQUEST(UID)
DPA:
    CHECK(UID)
    GID←UUID()
    α[UID,GID]←0,   ER.uid←UID
    ER.gid←GID,   ER.time←NULL
    ER.eid←NULL,   ER.sig←NULL
    signed_msg←$SIGN\_MSG_{dsk}$(GID,α[UID,GID])
DPA→SSP:
    signed_msg, ER
SSP:
    write(GID,α,ER)
SSP→USER:
    signed_msg
USER:
    (GID,α)←$VERIFY\_MSG_{dpk}$(signed_msg)
    write(GID,α)
```

Fig. 5 Protocol for creating a file group.

how DPA gets new users' information. User first generates its key pair and sends the public key *pk* to SSP. SSP generates a *UID* for this User, and passes *UID* and *pk* to DPA for storage. DPA generates a signed message *userInfo* using its secret key *dsk* on (*UID*, *pk*), and passes *userInfo* to User through SSP. By verifying *userInfo*, User can know whether DPA has saved and matched its information or not, and save UID locally for future use.

4.2 Protocol for Creating a File Group

Figure 5 shows the interactions of creating a file group. Assume DPA has a list of UIDs for which it is responsible. DPA preserves a persistent state α for each file group owned by these users.

User sends a request to SSP asking for creating a file group. SSP passes this request and User's *UID* to DPA. DPA first verifies if the *UID* exists in his list, then generates a *GID* for the new group and sets its α to 0. After that, DPA generates an *ER* for this file group and initializes its fields. DPA returns *ER* to SSP along with *signed_msg* produced by method *SIGN_MSG* on fields GID and α. SSP extracts *GID*, α and *ER* from *signed_msg* for storage and passes it to User. User verifies *signed_msg* and saves *GID* and α locally.

4.3 Protocol for Adding a File to a File Group

Figure 6 shows the interactions of add file operation. User invokes MF-PDP.Add method, and sends the encoded file *F*', the homomorphic authenticator *M*, the number of blocks *t* and *GID* of the target group to SSP, requesting for adding a file. SSP passes *UID*, *GID* and *t* to DPA. On receiving the request, DPA updates the file group's persistent state α and returns a signed message of *GID* and the modified α to SSP. SSP saves *F*', *M* and α, and returns the signed message to User for verification of the successful execution of the operation by DPA.

```
USER:
    MF-PDP.Add(sk,F,α,GID)→(F',M,α')
USER→SSP:
    ADD_REQUEST(F',M,t,GID)
SSP→DPA:
    (UID,GID,t)
DPA:
    CHECK(UID,GID)
    B=α[UID,GID]      α[UID,GID]=B+t
    signed_msg←$SIGN\_MSG_{dsk}$(GID,α[UID,GID])
DPA→SSP:
    signed_msg
SSP:
    write(F',M,α')
SSP→USER:
    signed_msg
USER:
    (GID,α)←$VERIFY\_MSG_{dpk}$(signed_msg)
```

Fig. 6 Protocol for adding a file to a file group.

```
USER→SSP:
    DELETE_REQUEST(GID)
SSP→DPA:
    GROUP_DELETE_REQUEST(UID,GID)
DPA:
    CHECK(UID,GID)
    DELETE(UID,GID)
    signed_msg←$SIGN\_MSG_{dsk}$(GID,DELETED)
DPA→SSP:
    signed_msg
SSP:
    DELETE_GROUP(GID)
SSP→USER:
    signed_msg
USER:
    (GID,DELETED) ←$VERIFY\_MSG_{dpk}$(signed_msg)
```

Fig. 7 Protocol for deleting a file group.

4.4 Protocol for Deleting a File Group

Figure 7 shows the interactions of deleting a file group. User sends the target *GID* and a request for deletion to SSP. SSP passes this request to DPA. DPA deletes the information about the group and returns a signed message of GID and a flag of deletion to SSP. SSP then deletes all the data about this file group and returns the signed message to User for verification.

4.5 Protocol for Auditing a File Group

Figure 8 shows the interactions of auditing a file group. Audit process is launched by SSP periodically. The time interval between two consecutive audits is called an *epoch*. On receiving an audit request, DPA first checks *UID* and *GID*, then sends a challenge *chal* to SSP. SSP calculates a proof *P* and returns it together with the *ER* of this file group. DPA verifies *ER* and generates a new *LE*, then renews the fields of *ER* and resigns it. If it is the first time that this file group is audited, the current *ER*.*eid* is *NULL*, so the *prev_id* field of this LE is also *NULL*. If verification of *ER* fails, DPA terminates the interaction without generating any *LE*. Finally, DPA sends the new *LE* and the renewed *ER* to SSP for storage. We emphasize that though it is expected that SSP should interact with DPA to generate an *LE* in each *epoch*, in our protocols it is not an error if no *LE* is generated in some *epoch*. The audit log only faithfully represents the audit history. It should be judged by User whether SSP has behaved improperly or not.

4.6 Protocol for Checking the Audit Log

Figure 9 shows the interactions of checking the audit log by User. User sends the file group's *GID* to SSP requesting a log list *l* (with *ER*), or a section of it corresponding to a specific time period. Upon receiving *l* and *ER*, User first verifies the *ER*, then verifies each *LE* along the log list until the *prev_id* field of the last *LE* points to *NULL* or the *time* field reaches the boundary of the time period. Note that this protocol only ensures the trustworthiness of the audit log. To know if the file group has undergone any corruption during the period, User needs to examine the result field of each log entry.

```
SSP→DPA:
    AUDIT_REQUEST(UID,GID)
DPA:
    CHECK(UID,GID)
    chal←MF-PDP.Challenge(α[UID,GID])
DPA→SSP:   chal
SSP:
    MF-PDP.Prove(pk,chal,F',M ,α[UID,GID])→P
SSP→DPA: (P,ER)
DPA:
    VERIFY_MSG_dpk(ER)
    result←MF-PDP.Verify(pk,chal,P,α[UID,GID])
    LE.result←result   LE.eid←UUID()
    LE.time←GET_CUR_TIME()
    LE.prev_eid←ER.eid
    LE.sig←SIGN_dsk(LE.result,LE.time,LE.eid,LE.prev_eid)
    ER.eid←LE.eid   ER.time←LE.time
    ER.sig←SIGN_dsk(ER.uid,ER,gid,ER.eid,ER.time)
DPA→SSP: (ER,LE)
SSP: write(ER,LE)
```

Fig. 8 Protocol for auditing a file group.

```
USER→SSP:
    CHECK_LOG_REQUEST(GID, timeperiod)
SSP→USER: l, ER
USER:
    VERIFY_MSG_dpk(ER)
    eid = ER.eid
    while eid ≠ NULL and l[eid].time in timeperiod do
        VERIFY_MSG_dpk(l[eid])
        eid = l[eid].prev_eid
    end while
```

Fig. 9 Protocol for checking the audit log.

5. Security and Complexity Analysis

5.1 Security Analysis

We decompose data possession auditing with IDPA-MF-PDP into two parts: audit result generation and audit result retention, and use three theorems to characterize its security properties. We assume that the DPA is trusted, i.e., no motivated parties can modify the critical secret data in the DPA, such as the secret key, the internal clock, or the preloaded programs (see Sect. 3.2). We formulate the security of IDPA-MF-PDP as three theorems, the proofs of which are given in Appendix.

THEOREM 1. *Suppose the DPA correctly holds the persistent state of a file group. Given the same number of corrupted blocks to detect and the same number of blocks to sample from the group, the detection probability of* MF-PDP *asymptotically equals that of* PDP *when the total number of blocks to check possession is sufficiently large, compared to the number of sampled blocks.*

THEOREM 2. *Throughout the life cycle of a file group consisting of normal interactions defined in Sect.* 4*, DPA correctly holds the persistent state of the group.*

Theorem 1 and Theorem 2 ensure the correctness of the result generated during each audit. The equation $P \approx ce/t$ derived in the proof of Theorem 1 implies that to achieve a fixed detection probability P for a fixed number of corrupted blocks e, the number of sampled blocks c is asymptotically proportional to the file group's size t. Figure 10 illustrates how c grows with t under different e values for a detection probability of 0.99.

Before stating Theorem 3, we first provide three definitions.

DEFINITION 1 (Verifiability of Entry Reference). *ER is verifiable iff*:

1) *Its signature can be verified by the DPA's public key*

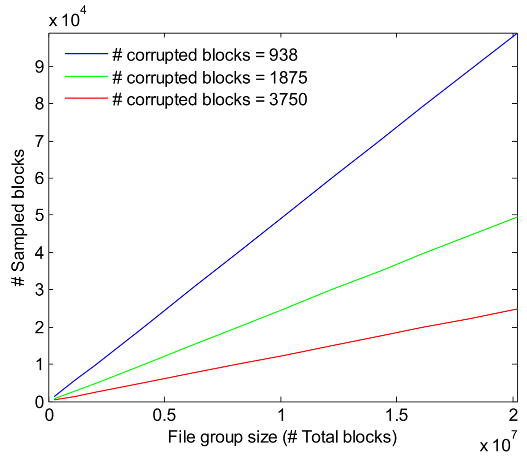

Fig. 10 Number of sampled blocks under different number of total blocks and different number of corrupted blocks (e) for detection probability of 0.99.

dpk;

2) *ER.time falls within the latest epoch.*

DEFINITION 2 (Verifiability of Log Entry). *LE is verifiable iff the LE's signature can be verified by the DPA's public key dpk.*

DEFINITION 3 (The Consistency of audit history). *The audit history h of file group G is consistent iff*:

1) *ER is verifiable*;

2) *If ER.eid is not NULL, the LE pointed to by ER.eid must exist and be verifiable*;

3) *For any LE in the LE chain, if LE.prev_id is not NULL, the LE it points to must exist and be verifiable*.

THEOREM 3. *The audit history of a file group can only make transitions between consistent states by means of interaction between the* DPA *and* SSP, *according to the protocol described in Sect.* 4.6.

Theorem 3 ensures the trustworthiness of the audit log. In practice, a user can conveniently check the consistency of a file group's whole or recent audit history as stated in Sect. 4.6.

5.2 Complexities

In this section, we analytically quantify the complexities of I/O, computation, communication and storage overhead of IDPA-MF-PDP. The complexities are quantified against the number of files n in a file group, irrespective of the size of each file.

I/O Overhead The SSP's I/O overhead is caused by reading sampled blocks from the disk. As analyzed in Sect. 5.1, for a fixed number of corrupted blocks to detect and fixed detection probability, SSP is required to read c sampled blocks, where c is proportional to the file group's size. The I/O complexity is $O(n)$.

Computation Overhead In the Prove algorithm, the SSP computes $A = A[i_1] \cdot \ldots \cdot A[i_c] \bmod N$, which contains c multiplications. The complexity is $O(n)$. In the Verify algorithm, the DPA computes $\tau_1 = A^e \bmod N$ and $\tau_2 = H(GID \| i_1) \cdot \ldots \cdot H(GID \| i_c) \cdot g^F \bmod N$ which contains c multiplications and 2 exponentiations. The computational overhead is dominated by exponentiations; therefore, the complexity is $O(1)$. Note that although the Prove algorithm has greater complexity than Verify, its absolute overhead is substantially smaller. This is displayed in the experimental results in Sect. 6.2.

Communication Overhead The communication overhead is the number of bits sent between the SSP and the DPA in the audit protocol. DPA sends a challenge $chal = (c, k)$ to the SSP, then the SSP sends a proof $P = (A, F)$ with the current ER to the DPA; finally, the DPA sends the new ER and LE to the SSP. The lengths of all these messages are independent of the file group size. The complexity is $O(1)$.

DPA storage overhead The DPA storage overhead is the size of the keys and states that are stored locally by the DPA. In IDPA-MF-PDP, the DPA stores the RSA key pairs (n, e, d) and the persistent state $\alpha = (B)$. The complexity is $O(1)$.

Table 1 Overheads of PDP, HMAC-PDP and IDPA-MF-PDP.

Scheme	PDP	HMAC-PDP	IDPA-MF-PDP
Server I/O (# blocks accessed)	$O(n)$	$O(n)$	$O(n)$
Server comp. (#multiplications.)	$O(n)$	$O(1)$	$O(n)$
DPA comp. (#exponentiations)	$O(n)$	$O(n)$	$O(1)$
Comm. (bits)	$O(n)$	$O(n)$	$O(1)$
DPA storage (bits)	$O(n)$	$O(1)$	$O(1)$

Table 1 summarizes the asymptotic complexities of IDPA-MF-PDP including their formations, and compares them with two baselines. One is the naive MF-PDP scheme based on RSA single-file PDP scheme [1] (denoted as PDP). The other is a light-weight keyed hash based MF-PDP scheme (denoted as HMAC-PDP) constructed as follows: The authenticators in HMAC-PDP are computed as keyed hashes (e.g. HMAC-SHA1) over the concatenation of block index and block content. In prove phase, the sampled blocks are returned by the SSP along with their authenticators. They are verified by recomputing the keyed hashes and comparing with the returned ones. The hash key is shared by the user and the verifier, which can be instantiated either by a traditional TTP or by an integrated DPA. The other aspects of the scheme are the same as those in Fig. 1. Though a major drawback of such a hash based MF-PDP scheme is the lack of public verifiability, it has the advantage of avoiding any expensive exponential computation. So we include it as a very strong baseline for evaluating auditing efficiency. As shown in the table, the reduction in computation and storage overhead on the DPA from $O(n)$ to $O(1)$ is crucial for the implementation of the integrated DPA using trusted hardware.

Note that the complexities of PDP are also quantified against n files. A simple PDP has less complexities of I/O, computation, and communication compared to IDPA-MF-PDP in checking a single file. Although it is unnecessary for PDP scheme to check multiple files, most users may like to check the integrity of all files which he/she owns, so Table 1 is an appropriate comparison for real-world applications.

6. Implementation and Experimental Evaluation

6.1 Implementation and Experimental Setup

We have implemented a prototype system of IDPA-MF-PDP to test its efficiency. For comparison, we have also implemented the single-file-based naive MF-PDP scheme (PDP) and the keyed hash based MF-PDP scheme described in Sect. 5.2 (HMAC-PDP). For both baselines, third party auditing is instantiated either by the integrated DPA auditing framework or by the traditional TTP, the first of which is denoted by an "IDPA-" prefix. This results in a total of four baseline schemes, denoted respectively as PDP, IDPA-PDP, HMAC-PDP and IDPA-HMAC-PDP. The code was written in C on Linux. All cryptographic operations use the crypto library of OpenSSL version 0.9.8o (OpenSSL, 1998) [9].

All experiments were conducted on three servers with the same configuration: Intel Xeon Core 4 processor

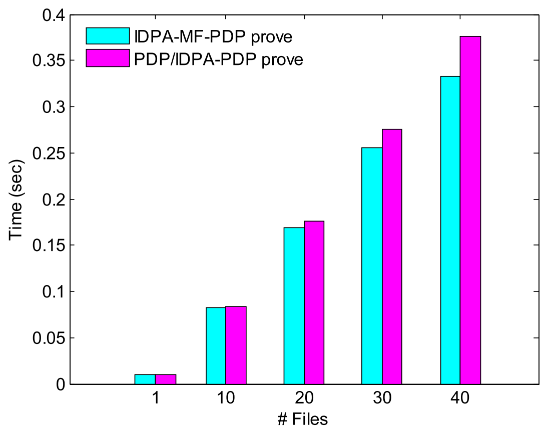

Fig. 11 Prove time of IDPA-MF-PDP and PDP/IDPA-PDP.

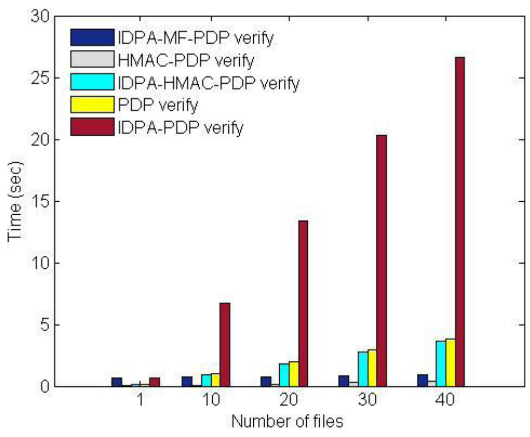

Fig. 12 Verify time of IDPA-MF-PDP, HMAC-PDP, IDPA-HMAC-PDP, PDP and IDPA-PDP.

running at 2.27 GHz, 8GB RAM, and a 10000 RPM 146GB SAS drive with an 16MB buffer. The three servers act as the SSP, the integrated DPA and the traditional TTP. Since mainstream secure coprocessors have weaker compute capability than server CPUs, we use a slowdown factor on the DPA server to simulate the performance of a real integrated DPA. IBM 4764 can generate 1024-bit RSA key pairs at a speed of 2.16 per second [14]. By comparing this with the results obtained on our server (14.92/s), we set the slowdown factor to be 2.16/14.92 = 0.145.

The file sizes are randomly distributed among the interval [0.5GB, 1GB]. The block size is 4KB. For IDPA-MF-PDP, the number of corrupted blocks e to detect is a fixed number regardless of the file group size. The number is set to be 1% of the average file size, i.e. 0.75GB/4KB * 1% = 1875. The number of sampled blocks c is chosen so as to detect corruption of e blocks with fixed detection probability of 0.99. All reported results represent the mean of 5 trials. As the results varied little across trials, the confidence intervals are not presented.

6.2 Results

We tested the performance of auditing a file group (Sect. 4.5) as it is the most frequently executed operation. The time of an audit operation consists of four parts: I/O time of reading sampled blocks from the disk by the SSP, computation time of generating a proof by the SSP, computation time of verifying a proof by the DPA, and communication time between the SSP and the DPA.

Figure 11 shows the computation time of prove time of IDPA-MF-PDP, PDP and IDPA-PDP. The computation time of prove is determined by the number of sampled blocks. So for both IDPA-MF-PDP and PDP/IDPA-PDP the prove time show linear growth with the number of files. Nevertheless, the prove time of IDPA-MF-PDP is still a little shorter than that of PDP and IDPA-PDP due to only one modulus operation. For HMAC-PDP and IDPA-HMAC-PDP, the SSP does not do any computation when generating a proof, leading to a prove time of zero which is not shown in the figure.

Table 2 Overheads of IDPA-MF-PDP, IDPA-HMAC-PDP, IDPA-PDP and PDP.

Overhead	IDPA-MF-PDP (sec)	IDPA-HMAC -PDP (sec)	IDPA-PDP (sec)
Prove I/O	248 (99.16%)	247	247
Prove Comp.	0.82 (0.33%)	0	0.80
Verify	1.27 (0.51%)	8.97	66.9
Communication	0.0003 (0%)	16.31	0.033
Log Generation	0.003 (0.0012%)	0.3	0.3
Total	250.1 (100%)	272.6	315.0

Figure 12 shows the verify time of the five schemes. The verify time of IDPA-MF-PDP remains almost constant while those of the other four grow linearly as the number of files increases. When there are 40 files in the group, compared with IDPA-MF-PDP's verify time, IDPA-PDP's time is 29.5 times longer. This is because IDPA-MF-DPA needs to do 2 exponentiations in verify phase while PDP and IDPA-PDP needs to do $2n$ exponentiations, where n is the number of files. Compared with IDPA-MF-PDP, IDPA-HMAC-PDP's verify time is about 4 times longer than that of IDPA-MF-PDP when audit 40 files. Although IDPA-HMAC-PDP does not do any exponentiation computation in verify, the number of keyed hashes of blocks it needs to check grows linearly with the number of files. The reduced verify computation on DPA is crucial for trusted hardware-based implementation.

Table 2 summarizes the overall overhead and its compositions of IDPA-MF-PDP compared with IDPA-HMAC-PDP and IDPA-PDP when there are 100 files to audit. To measure communication overhead, we limit the bandwidth between the SSP and DPA server to 100Mbps. As shown in the table, the verify, communication and log generation overheads of IDPA-MF-PDP are much lower than those of the other two. This result is remarkable especially when considering that IDPA-MF-PDP uses RSA exponential computation while IDPA-HMAC-PDP uses very light-weight keyed hash functions. We can also see that the I/O time constitutes most (99.16%) of the overall overhead of IDPA-MF-PDP, indicating that the performance of the system is I/O bounded rather than computation bounded.

Although IDPA-HMAC-PDP has zero computation time in Prove compared to IDPA-MF-PDP, its total prove cost (I/O plus computation) is still close.

7. Related Works

Formal definitions for Provable Data Possession (PDP) and Proof of Retrievability (POR) schemes were first provided by Ateniese et al. [1] and Juels and Kaliski [10]. Ateniese et al. [1] define the PDP model and propose efficient constructions of the scheme. In their schemes, random sampling is used to reduce the overhead of verification. RSA-based homomorphic tags are utilized for verifying the sampled blocks aggregately, thus public verifiability is also provided. Homomorphic authenticators and random sampling are also building blocks of our scheme.

The original PDP scheme [1] only supports checking of a static file. In their subsequent work, Ateniese et al. proposed a dynamic version of their PDP scheme. Their idea is to predefine all future challenges during setup and store pre-computed proofs as metadata. As a result, the number of challenges a verifier can perform is fixed. Furthermore, each update requires re-creating all the remaining challenges, which is problematic for large files. Wang et al. considered dynamic data storage in distributed systems [4]. The proposed scheme can determine the data correctness and locate possible errors. In another work, Wang et al. constructed a scheme that integrates data dynamics with public verifiability by TTP [5]. Fully dynamic data operations, especially block insertions, are supported by manipulating the Merkle hash tree constructed for block tag authentication. Erway et al. proposed another fully dynamic PDP construction [3]. They extended the PDP model of Ateniese et al. [1] to support provable updates to stored files using rank-based authenticated skip lists. However, the efficiency of update operations remains in question. Li et al. proposed the support of block insertions using SN-BN tables [6], which match the logical indices of blocks with their actual positions. Though the support for insertion in these dynamic PDP schemes is much like the add operation on file groups in our scheme, we emphasize that the overhead of preserving such complicated data structures is nonnegligible. Additionally, these data structures may introduce new problems to PDP schemes, such as the balance of the Merkle hash tree adopted in [5].

All the aforementioned dynamic PDP schemes are designed mainly for single file updating and have high updating and verification overhead, especially when naively extended to deal with multiple files ($O(n)$). Only [3] considered the problem of verifying multiple files in an extension. The verification metadata is organized into a tree structure, mimicking the file system hierarchy. The complexities of updating and checking are both $O(d \cdot \log_2 n)$, where d is the depth of the tree and n is the maximum number of leaves in each skip list. The MF-PDP scheme proposed in this paper leverages the specific data update pattern of cloud storage. While being simple, it significantly reduces the overhead of multiple-file checking ($O(1)$).

Based on the work of Wang et al. [5], a few researchers have considered the practical issues of deploying third-party-auditing-based PDP schemes in real-world cloud storage environments, and proposed solutions. Their works are mostly concentrated on multiple-SSPs support and scalability. Zhu et al. [12] considered the situation where multiple SSPs cooperate to provide storage service. They constructed a cooperative PDP scheme based on a multiprover zero-knowledge proof system. The dynamic PDP scheme proposed by Yang et al. [13] supports batch auditing for multiple owners and multiple SSPs. The key idea of using batch auditing to increase efficiency is similar to MF-PDP, but they did not consider batch auditing for multiple files. Wang et al. [14] constructed an identity-based PDP scheme for distributed multi-cloud storage, to increase the scalability of the system.

All these third party auditing schemes assume the existence of a TTP, without mentioning explicitly how to implement and deploy such a TTP in a secure and cost-effective manner. In contrast, we propose a concrete implementation of TTP with an integrated DPA, and an auditing framework based on it. The framework is general and can be easily integrated with various PDP schemes (e.g. MF-PDP).

8. Conclusion

In this article, we analyze the key factors that affect the practicality of existing PDP schemes and provide a solution based on characteristics of real-world cloud storage environments. Theoretical analysis shows that IDPA-MF-PDP has the same security property as the original PDP scheme, and the audit results recorded in the tamper-evident audit logs are trustworthy. IDPA-MF-PDP is secure. Complexity analysis and experiments demonstrate that the communication and computation overhead on the DPA is reduced from linear in the size of the data to near constant, and its performance is bounded by disk I/O. IDPA-MF-PDP is efficient. Our work opens the door to the practical application of PDP schemes in real-world cloud storage environments.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (61202082, 61472048, 61402058), the Beijing Natural Science Foundation (4152038), the China Postdoctoral Science Foundation funded project (2014M561826).

References

[1] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song, "Provable data possession at untrusted stores," ACM CCS'07, pp.598–609, VA, USA, Oct. 2007.

[2] G. Ateniese, RD. Pietro, LV. Mancini, and G. Tsudik, "Scalable and efficient provable data possession," Conference Securecomm08 Fourth International Conference On Security on Privacy for communication Networks, Article no. 9, Istanbul, Turkey, Sept. 2008.

[3] C. Erway, A. Kupcu, C. Papamanthou, and R. Tamassia, "Dynamic provable data possession," ACM CCS'09, pp.213–222, IL, USA, Nov. 2009.
[4] C. Wang, Q. Wang, K. Ren, and W. Lou, "Ensuring data storage security in cloud computing," 17th International Workshop on Quality of Service (IWQoS 2009), pp.1–9, South Carolina, USA, July 2009.
[5] Q. Wang, C. Wang, K. Ren, W. Lou, and J. Li, "Enabling public verifiability and data dynamics for storage security in cloud computing," IEEE Trans. Parallel Distrib. Syst., vol.22, no.5, pp.847–859, 2011.
[6] C. Li, Y. Chen, P. Tan, and G. Yang, "An efficient provable data possession scheme with data dynamics," 2012 International Conference on Computer Science and Service System, pp.706–710, Nanjing, China, Aug. 2012.
[7] Amazon S3 Versioning. Available from http://doc.s3.amazonaws.com/betadesign/Versioning.html.
[8] Amazon Simple Storage Service. Available from http://aws.amazon.com/s3/
[9] Openssl Crypto Library. Available from http://www.openssl.org/
[10] A. Juels and B. Kaliski, "PORs: Proofs of retrievability for large files," ACM CCS'07, pp.584–597, VA, USA, Oct. 2007.
[11] D. Xiao, W. Yao, C. Wu, J. Liu, and Y. Yang, "Multiple-file remote data checking for cloud storage," Computers & Security, vol.31, no.2, pp.192–205, 2012.
[12] Y. Zhu, H. Hu, H. Ahn, and M. Yu, "Cooperative provable data possession for integrity verification in multi-cloud storage," IEEE Trans. Parallel Distrib. Syst., vol.23, no.12, pp.2231–2244, 2011.
[13] K. Yang and X. Jia, "An efficient and secure dynamic auditing protocol for data storage in cloud computing," IEEE Trans. Parallel Distrib. Syst., vol.24, no.9, pp.1717–1726, 2013.
[14] H. Wang, "Identity-based distributed provable data possession in multi-cloud storage," IEEE Trans. Services Computing, DOI:10.1109/TSC.2014.1. 2014.
[15] IBM 4764 PCI-X Cryptographic Coprocessor. http://www-03.ibm.com/security/cryptocards/pcixcc/overperformance.shtml
[16] R. Shaikh and M. Sasikumar, "Security issues in cloud computing: A survey," Int. J. Computer Applications, vol.44, no.19, pp.4–10, 2012.
[17] M. van Dijk, A. Juels, A. Oprea, R.L. Rivest, E. Stefanov, and N. Triandopoulos, "Hourglass schemes: how to prove that cloud files are encrypted," ACM CCS, North Carolina, USA, Oct. 2012.
[18] D. Liu, J. Lee, J. Jang, S. Nepal, and J. Zic, "A new cloud architecture of virtual trusted platform modules," IEICE Trans. Inf. & Syst., vol.E95-D, no.6, pp.1577–1589, June 2012.

Appendix

In this section, we give the proofs of the theorems stated in Sect. 5.1.

A.1 Proof of Theorem 1

Suppose the file group consists of n files and a total of t blocks, of which e blocks are corrupted. File i has t_i blocks, of which e_i are corrupted. $t = \sum_{i=1}^{n} t_i$, $e = \sum_{i=1}^{n} e_i$. To detect corruption, PDP samples c_i blocks for file i, while MF-PDP samples c blocks for the file group. $c = \sum_{i=1}^{n} c_i$. Suppose the number of sampled blocks is proportional to the size of the file, i.e., $c_i = wt_i$, $c = wt$, where w is a constant. According to the large t assumption, we have $w \to 0$. For both PDP and MF-PDP, the SSP can prove its possession of the sampled blocks by responding to the challenge with a proof that passes the Verify algorithm. This security property is already proved by [1].

As the DPA holds the correct persistent state of the file group, i.e., its size t, the c blocks sampled by the DPA are randomly distributed among the t blocks. MF-PDP detection fails iff none of the c sampled blocks fall on the e corrupted blocks, the probability of which is given by $P = 1 - \left(1 - \frac{e}{t}\right) \cdot \left(1 - \frac{e}{t-1}\right) \cdot \ldots \cdot \left(1 - \frac{e}{t-c+1}\right)$. Under the large t assumption, the detection probability of MF-PDP is $P = 1 - \left(1 - \frac{e}{t}\right) \cdot \ldots \cdot \left(1 - \frac{e}{t-c+1}\right) \approx 1 - \left(1 - \frac{e}{t}\right)^c \approx \frac{ce}{t} = we$. The first approximately equal follows from the large t assumption, and the second approximately equal follows from the binomial expansion formula and $w \to 0$. Similarly, the detection probability for PDP to detect the corruption of file i is $P_i = 1 - \left(1 - \frac{e_i}{t_i}\right)^{c_i} \approx \frac{c_i e_i}{t_i} = we_i$. PDP detection fails iff all n single file detection fails, the probability of which is given by $\prod_{i=1}^{n}(1 - P_i)$. Thus, the overall detection probability of PDP is $P' = 1 - \prod_{i=1}^{n}(1 - P_i) \approx 1 - \prod_{i=1}^{n}(1 - we_i) \approx w\sum_{i=1}^{n} e_i = we = P$.□

A.2 Proof of Theorem 2

In IDPA-MF-PDP, the DPA holds a persistent state α for each file group. Because the DPA is trusted hardware, α can be modified only when the DPA interacts with the SSP. When creating a file group, α is initialized to 0 by the DPA. When a file is added to a file group, both User and DPA will modify α locally, then the DPA sends the signed α to the user for comparison. Because the user always has the correct α, if the two α are equal, then the current α in the DPA is correct. Thus, as long as the interactions are executed properly, the α preserved by the DPA is always correct.□

A.3 Proof of Theorem 3

It is obvious that when the SSP interacts properly with the DPA according to the protocol in Sect. 4.4, the generated audit history h of file group G will be confined within consistent states. We now demonstrate that an adversary cannot change a consistent h into another consistent state by manipulating h without interacting with the DPA.

Assume the entry reference of h in state s_1 is designated by ER_1. We analyze three possible attacks for manipulating h in turn: (1) The adversary attempts to modify ER or use a saved old version of it. Because the adversary cannot forge the ER's signature without knowing the DPA's secret key, modification of any field in the ER or replacing it with an old one will make it unverifiable. This in turn will violate the 1st condition of Definition 3, making h inconsistent; (2) If h is non-empty and the adversary attempts to modify or discard the first LE of h, the 2nd condition of Definition 3 will be violated; (3) If the number of LEs in h is larger than 1 and the adversary attempts to modify or discard any LE of h except for the first one, the 3rd condition of Definition 3 will be violated, making h inconsistent. In summary, any manipulation with any part of h will result in its departure from the consistent state.□

Da Xiao received his BS and Ph.D. degrees in Computer Science and technology from Tsinghua University, China in 2003 and 2009, respectively. He is currently an assistant professor in School of Computer Science, Beijing University of Posts and Telecommunications, China. His research interests include cloud storage, storage security, distributed storage and file systems, disaster backup and recovery.

Lvyin Yang received her BS degrees in software engineering at Hebei Normal University, China in 2013. She is currently a Master student majored in computer science and technology in School of Computer Science, Beijing University of Posts and Telecommunications, China. Her research interests include cloud storage security.

Chuanyi Liu received his BS and Ph.D. degrees in Computer Science and technology from Tsinghua University, China in 2003 and 2009, respectively. He spent one year as a visiting scholar at the Digital Technology Center of the University of Minnesota. He is currently an associate professor in School of Software Engineering, Beijing University of Posts and Telecommunications, China. His research interests include computer architecture, cloud computing and cloud security, file and storage systems, information security and data protection. He is a member of IEICE.

Bin Sun received her BS, MS and Ph.D. degrees in Communication Engineering from Beijing University of Posts and Telecommunications, China in 1989, 1992 and 2010, respectively. She is currently an associate professor in School of Computer Science, Beijing University of Posts and Telecommunications, China. Her research interests include computer networks and network security.

Shihui Zheng received her BS and Ph.D. degrees in Mathematics from Shandong University, China in 2001 and 2006, respectively. From 2006 to 2008, she was in Beijing University of Posts and Telecommunications, China as a postdoc research fellow. She is currently an assistant professor in School of Computer Science, Beijing University of Posts and Telecommunications, China. Her research interests include cryptography schemes design and analysis.