A Security Architecture for Cloud Storage Combining Proofs of Retrievability and Fairness

Aiiad Albeshri*†, Colin Boyd* and Juan Gonzalez Nieto*
*Information Security Institute, Queensland University of Technology, Brisbane, Australia.
{c.boyd, j.gonzaleznieto}@qut.edu.au, a.albeshri@student.qut.edu.au
†Faculty of Computing and IT, King Abdulaziz University, Jeddah, Saudi Arabia

Abstract—**We investigate existing cloud storage schemes and identify limitations in each one based on the security services that they provide. We then propose a new cloud storage architecture that extends CloudProof of Popa et al. to provide availability assurance. This is accomplished by incorporating a proof of storage protocol. As a result, we obtain the first secure storage cloud computing scheme that furnishes all three properties of availability, fairness and freshness.**

Keywords- Cloud Computing; Cloud Storage; Cloud Security.

I. Introduction

Cloud computing is essentially a large-scale distributed and virtual machine computing infrastructure. This new paradigm delivers a large pool of virtual and dynamically scalable resources, including computational power, storage, hardware platforms and applications, which are made available via Internet technologies. There are many advantages for private and public organisations that decide to migrate all or some of their information services to the cloud computing environment. Examples of these benefits include increased flexibility and budgetary savings through minimisation of hardware and software investments [7], [8], [15]. However, while the benefits of adopting cloud computing are clear, there are also associated critical security and privacy risks that result from placing data off-premises. Indeed, it has been observed that data owners who outsource their data to the cloud also tend to outsource control over their data [8].

Consumers have the option to trade the privacy of their data for the convenience of software services (e.g., web based email and calendars). However, this is generally not applicable in the case of government organisations and commercial enterprises [15]. Such organisations will not consider cloud computing as a viable solution for their ICT needs, unless they can be assured that their data will be protected at least to the same degree that in-house computing offers currently. Yet, none of today's storage service providers in the cloud (e.g., Amazon Simple Storage Service (S3) [2] and Google's BigTable [12]) guarantee any security in their service level agreements. Moreover, there have been already security breach incidents in cloud based services, such as the corruption of Amazon S3, due to an internal failure caused by mismatching files with customers' hashes [1].

This paper focuses on designing a secure storage architecture for cloud computing. As discussed below, important security requirements that a cloud storage architecture should satisfy are confidentiality, integrity, availability, fairness (or non-repudiation) and data freshness. Examination of the literature shows that there is no single complete proposal that provides assurance for all of these security properties. Also, some existing secure cloud storage schemes are designed only for static/archival data and are not suitable for dynamic data.

Proof of storage (POS) protocols are a key component in most secure cloud storage proposals in the literature. A POS is an interactive cryptographic protocol that is executed between clients and storage providers in order to prove to the clients that their data has not been modified or (partially) deleted by the providers [15]. The POS protocol will be executed every time a client wants to verify the integrity of the stored data. A key property of POS protocols is that the size of the information exchanged between client and server is very small and may even be independent of the size of stored data [8].

We investigated different types of existing cloud storage schemes and identified limitations in each one of them based on the security services that they provide. We identified a scheme by Popa *et al.* [18], called CloudProof, as one satisfying the majority of the security requirements. However, it does not provide assurance on data availability, i.e., it does not guarantee that the entire data is indeed stored by the cloud provider. Our goal then is to provide a cloud storage architecture that extends CloudProof in order to provide availability assurance, by incorporating a proof of storage protocol.

The rest of this paper is organised as follows: the second section elucidates the set of security properties that a secure cloud storage application must fulfill; the third section provides an analysis of existing secure cloud storage proposals from the literature; the fourth section introduces the proposed architecture; finally, in the fifth section, the paper draws some conclusions and points at future work.

II. Security requirements

We consider a cloud storage scenario where there are four kinds of parties involved: the data owner, the cloud provider, clients and an optional third party auditor (TPA). The data owner pays for the cloud storage service and sets the access control policies. The cloud provider offers the data storage service for a fee. Clients request and use the data from the cloud. In the cloud environment we assume that there is no mutual trust between parties. Thus, several security properties need to be assured when storing the data in the cloud, as discussed in many related works (e.g., [16], [22]).

a) Data Confidentiality: ensures that only authorised clients with the appropriate rights and privileges can access the stored information. The most effective way to ensure the confidentiality of the client's data is by using encryption, even though the cloud provider may still be able to predict some information based on monitoring the access patterns of clients [18]. Most existing secure storage proposals provide data confidentiality by allowing clients to encrypt their data before sending it to the cloud. However, critical issues such as key management may be problematic, especially when we have a multiple user scenario.

b) Data Integrity: ensures that the stored data has not been inappropriately modified (whether accidentally or deliberately). Data integrity becomes more challenging when adopting cloud computing where cloud customers outsource their data and have no (or very limited) control over their stored data from being modified by the storage service provider. Thus, cloud customers are aiming to detect any unauthorized modification of their data by the cloud storage provider.

c) Data Availability: ensures that users are able to obtain their data from the cloud provider when they need it. Cloud customers want to be sure that their data is always available at the cloud storage. To this end, a number of proof of storage protocols have been devised that allow the cloud provider to prove to clients that their entire data is being stored, which implies that the data has not been deleted or modified. Section III discusses some of these schemes.

d) Public Verifiability: means that service providers allow a TPA to perform periodical availability verifications on behalf of their customers. In cloud computing environments, customers may need to allow a TPA to verify the integrity of the dynamic data stored in the cloud storage [21]. Public verifiability allows the cloud customers (or their TPA) to challenge the cloud server for correctness of stored data. In fact, security requirements can be inter-related. For instance, when a TPA is delegated to perform verification, the confidentiality may be compromised. However, this issue could be resolved by utilising a verification protocol that allows TPA to verify without knowing the stored data [21].

e) Freshness: ensures that the retrieved data is fresh, i.e., it contains the last updates to the data. This is very important in shared and dynamic environments where multiple clients may simultaneously update data. Cloud customers need to ensure that the retrieved data is the latest version. To the best of our knowledge, CloudProof [18] is the only cloud storage scheme that addresses freshness.

f) Fairness: or non-repudiation ensures that a dishonest party cannot accuse an honest party of manipulating its data [24]. If a dispute arises between a client and storage provider regarding whether the correct data is stored then it may be necessary to invoke a judge to decide who is right. Fairness will typically be implemented by using digital signatures. Clients may want to have a signature from the provider acknowledging what data is stored. Providers may want signatures from clients whenever the stored data is altered, with deletion being an important special case.

III. Proof of Storage Schemes (POS)

Cloud storage schemes can be categorised into two types, static and dynamic. In static schemes, clients store their data and never change or update it. In dynamic schemes clients can update the stored data. In the following two subsections, we review existing proposals for POS protocols. Table I lists the schemes reviewed and indicates the security requirements that are satisfied by them. The entry with the dagger (†) indicates that the property is only partially satisfied. It can be seen that no single proposal encompasses all security requirements identified in Section II. The security requirements in the table are Confidentiality (C), Integrity (I), Availability (A), Public Verifiability (PV), Freshness (Fr) and Fairness (Fa).

Table I
Overview of the prominent proof of storage (POS) schemes.

POS Scheme	C	I	A	PV	Fr	Fa	Type
Proof of Retrievability (POR) [13]	✓	✓	✓	✓	✗	✗	Static
Provable Data Possession (PDP)[3]	✓	✓	✓	✓	✗	✗	Static
Compact POR [19]	✓	✓	✓	✓	✗	✗	Static
Tahoe [23]	✓	✓	✗	✗	✗	✗	Static
HAIL [5]	✗	✗	✓	✓	✗	✗	Static
POR (experimental test) [6]	✓	✓	✓	✓	✗	✗	Static
Framework for POR protocols [9]	✓	✓	✓	✓	✗	✗	Static
POS from HIP [4]	✓	✓	✓	✓	✗	✗	Static
DPDP [10]	✓	✓	✓	✗	✗	✗	Dynamic
POR with public verifiability [21]	✓	✓	✓	✓	✗	✗	Dynamic
Depot [17]	✗	✓	✓	✓	✗	✗	Dynamic
Wang *et al.* [20]	✗	✓	✓	✓	✗	✗	Dynamic
CloudProof [18]	✓	✓	✗	✓	✓	✓	Dynamic
Fair and Dynamic POR [24]	✓	✓	✓	✗	✗	✗†	Dynamic

A. POS for Static Data

There are several POS schemes that support storage of static data. Juels and Kaliski [13] introduced proof of retrievability (POR). In POR the *Encode* algorithm firstly encrypts all the data. Additionally, a number of random-valued blocks (sentinels) are inserted at randomly chosen positions within the encrypted data. Finally, an error correction code is applied to the resulting new data. Clients *challenge* the service provider by identifying the positions of a subset of sentinels and asking the service provider to retrieve the requested values. The *VerifyProof* process works because, with high probability, if the service provider modifies any portions of the data, the modification will include some of the sentinels and will therefore be detected. If the damage is so small that it does not affect any sentinel, then it can be reversed using error correction.

POR [13] only allows a limited number of executions of the *Challenge* algorithm (for the whole data). The verification capability of POR is limited by the number of precomputed sentinels embedded into the encoded file. This is improved by the scheme of Shacham and Waters [19], which enables an unlimited number of queries and requires less communication overhead. In this scheme, in addition to encoding each file block, the client appends a special type of authenticator to each block. The encoded blocks and authenticators are stored on the server. The verifier challenges the service provider by sending a set of randomly selected block indexes. The response from the service provider is a compact proof that combines the challenge blocks and authenticators and which can be validated very efficiently by the verifier. Likewise, Bowers *et al.* [6], Ateniese *et al.* [4] and Dodis *et al.* [9] provided POR schemes which provide probabilistic assurance that a remotely stored file remains intact.

Table I lists other prominent POS examples. All POS schemes mentioned above were designed to deal with static or archival data only and are not suitable for dynamic environments. The efficiency of these schemes is mainly based on the preprocessing of the data before sending it to remote storage. Any modification to the data requires re-encoding the whole data file, so it has associated a significant computation and communication overhead.

B. POS for Dynamic Data

It is natural that clients may want to update their files while they are in storage without having to resubmit the whole data set to the server. Therefore, it is desirable to offer an option to update files in such a way that the proof of storage for the whole data still applies. POS for dynamic data is more challenging than static data. There are several dynamic POS schemes. Erway *et al.* [10] introduced what they called "Dynamic Provable Data Possession" or DPDP, which extends the static PDP [3]. Their approach uses a variant of authenticated dictionaries, which allows insertion and deletion of blocks within the data structure. A limitation of DPDP is that it does not allow for public verifiability of the stored data; in addition it does not consider data freshness or fairness. Wang *et al.* [21] improve on DPDP by adding public verifiability, thus allowing a TPA to verify the integrity of the dynamic data storage. Now the authenticated data structure employed is the classic Merkle Hash Tree (MHT). Still, data freshness and fairness are not considered.

Popa *et al.* [18] introduced CloudProof, which provides fairness by allowing customers to detect and prove cloud misbehaviour. This is achieved by means of digitally signed attestations. Each request and response for reading (get) and writing (put) data is associated with an attestation. This attestation will be used as proof of any misbehaviour from both sides. CloudProof [18] is the only POS scheme that provides assurance of data freshness by using hash chains. For each put and get attestation, the hash chain is computed over the hash of the data in the current attestation and the hash value of the previous attestation. More details are provided in Section IV.

In addition, CloudProof emphasises the importance of "fairness". If the cloud misbehaves, for example it deletes some user blocks, then the owner has the ability to prove to a judge that the cloud was at fault. At the same time, if the owner claims falsely that a file was deleted, the cloud can prove to the judge that the owner asked for this to be done.

It should be noted that fairness in CloudProof does not extend to the meaning normally expected in protocols for fair exchange. In particular, Feng *et al.* [11] have pointed out that a provider could omit sending its signature once it has received the signature of the client on an update. Consequently the provider has an "advantage" in the sense that it can prove to a judge that the client asked for an update but the client cannot provide any evidence that the provider received the update request. Arguably this advantage has limited consequences because the client can retain the update details pending the receipt of the provider's signature. If the provider does not send the signature then this is inconvenient for the client but he can recover from it; meanwhile, the client can seek other remedies. In any case, ensuring fairness in the stronger sense that neither party ever gets an advantage can only be achieved in general using an online trusted third party which is likely to be too costly to justify.

Zheng and Xu [24] have a rather different definition of fairness for their dynamic scheme. They require only that clients are not able to find two different files which both will satisfy the update protocol. The idea is that a malicious client can then produce a different file from that which the server can produce and claim that the server altered the file without authority. Zheng and Xu do not require that the update protocol outputs a publicly verifiable signature so a judge can only verify this fact by interacting with the client using public information. In addition, they do not consider the situation where a server does maliciously alter the file - for example deletes it. In this case, the client may no longer have anything to input to the verification equation.

In fact, the security model for CloudProof is quite weak. Auditing is only done on a probabilistic basis to save on processing. The data owner (or TPA) assigns to each block some probability of being audited, so an audit need not check every block. Thus, for parts that are rarely touched by users, this means that it could be a long time before it is noticed if something has been deleted. Whether or not a block will be audited is known to any user who has access to it, but is hidden from the cloud. Blocks which are not audited can be changed at will (or deleted) by the cloud. Popa *et al.* [18] state that "We do not try to prevent against users informing the cloud of when a block should be audited (and thus, the cloud misbehaves only when a block is not to be audited)". This seems too optimistic - if even a single user can be corrupted by the cloud, then the cloud can delete all the blocks to which that user has access without any chance of detection. it is clear therefore that CloudProof does not provide the availability assurance. However, as seen in Table I, it is the scheme that provides the most security services. In the next section, we extend CloudProof to provide availability of the whole stored data. We do so by combining CloudProof with the dynamic POR of Wang *et al.* [21].

IV. Proposed Architecture

We now describe a new architecture which combines the idea of CloudProof [18] and Dynamic Proofs Of Retrievability (DPOR) [21] as it provides data availability for dynamic data along with must of other security requirements. The proposed

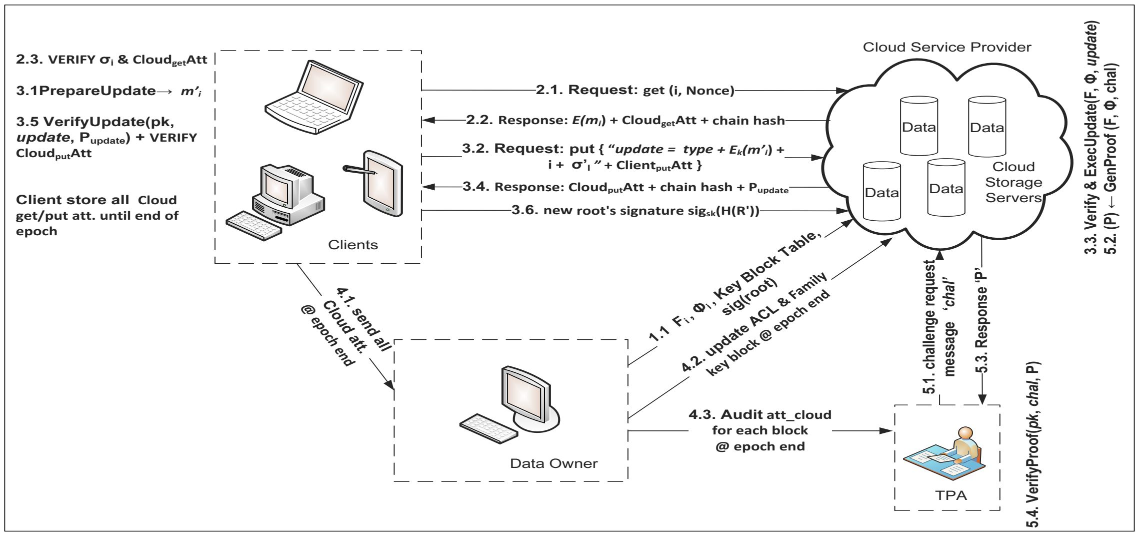
2.3. VERIFY σ_i & $Cloud_{get}Att$
3.1PrepareUpdate→ m'_i
3.5 VerifyUpdate(pk, *update*, P_{update}) + VERIFY $Cloud_{put}Att$
Client store all Cloud get/put att. until end of epoch
Clients
2.1. Request: get (i, Nonce)
2.2. Response: $E(m_i)$ + $Cloud_{get}Att$ + chain hash
3.2. Request: put { "update = type + $E_k(m'_i)$ + i + σ'_i" + $Client_{put}Att$ }
3.4. Response: $Cloud_{put}Att$ + chain hash + P_{update}
3.6. new root's signature $sig_{sk}(H(R'))$
Cloud Service Provider
Data
Data
Data
Data
Cloud Storage Servers
3.3. Verify & ExecUpdate(F, Φ, *update*)
5.2. (P) ← GenProof (F, Φ, chal)
4.1. send all Cloud att. @ epoch end
1.1 F_i, Φ_i, Key Block Table, sig(root)
4.2. update ACL & Family key block @ epoch end
5.1. challenge request message '*chal*'
5.3. Response 'P'
Data Owner
4.3. Audit att_cloud for each block @ epoch end
TPA
5.4. VerifyProof(*pk*, *chal*, P)

Figure 1. Proposed Architecture

POS architecture tackles the limitations of both schemes. Thus, the limitation of CloudProof of being unable to check data availability at the whole data set level is overcome by employing DPOR.

DPOR consists of the following protocols/algorithms:

1) *KeyGen*: is a randomized algorithm that is used to generate cryptographic key material.
2) *Encode*: is used for encoding data before sending it to the remote storage.
3) *GenProof*: the service responds to the client's challenge request by generating a proof which is sent to the verifier.
4) *VerifyProof*: upon receiving the proof from the service provider, the client executes this protocol to verify the validity of the proof.
5) *ExecUpdate*: this protocol is used in dynamic schemes and is executed by the cloud provider. This protocol may include a proof by the service provider of the successful update of the data, so that the customer can verify the update process.
6) *VerifyUpdate*: this is executed by the client in order to verify the proof sent by the service provider after an update.

As in CloudProof, we consider different time periods or *epochs*. At end of each epoch the data owner or TPA performs a verification process to assure that the cloud storage possesses its data. In this way we obtain a design that satisfies all the desirable properties discussed in Section II. Figure IV describes the proposed architecture and identifies its parties and the different protocols that are executed between them.

g) Key Management: we assume that the data owner will divide the plaintext data file into blocks $F'' = \{m''_1, m''_2, ..., m''_n\}$. Each data block is assigned to an ACL (set of users and groups) and blocks with similar ACL are grouped in a single block family. In addition, for each block family there is a family key block that contains a secret (signing) key sk (known only to clients with write access in the ACL), read access key k (known only to clients with read access in the ACL), public (verification) key pk (known to all clients and the cloud provider), version of pk and k keys, block version, and signature of the owner. The data owner will create the family key block table in which each row in this table corresponds to an ACL (Fig. 2). The data owner maintains the key production while the key distribution process is offloaded to the cloud service provider but in a verifiable way. The key distribution process involves two cryptographic tools; **broadcast encryption** E_F which is used to encrypt the secret key ($E_F(sk)$) and read access key ($E_F(k)$). $E_F(k)$ guarantees that only allowed clients and groups in the ACL's read set can decrypt the key and use it to decrypt the blocks in the corresponding family. sk is used to generate update signatures for blocks. $E_F(sk)$ guarantees that only users and groups in the ACL's write set can decrypt the key and use it to generate update signatures for blocks in the corresponding family. The **key rotation** scheme is another cryptographic tool which is used to generate a sequence of keys using an initial key with a secret master key [14]. Thus, only the owner of the secret master key can produce the next key in the sequence. Also, by using key rotation, the updated key allows computing of old keys. Thus, there is no need to re-encrypt all encrypted data blocks [18]. The data owner will keep the family key block table and every time there is a change of membership, the data owner will re-encrypt the key and update the family key block table.

h) Pre-Storage Processing: the data owner encodes each block in the data file F'' using Reed-Solomon error correction $F' = encode_{RS}(F'')$. Then, each block in F' is encrypted using the corresponding k of that block family; $F = E_k(F') = \{m_1, m_2, ..., m_n\}$. The data owner creates a Merkle Hash Tree (MHT) for each block family. The MHT is constructed as a binary tree that consists of a root R and leaf nodes which are an ordered set of hashes of the family data blocks $H(m_i)$. MHT is used to authenticate the values of the data blocks. As in DPOR [21], the leaf nodes are treated in the left-to-right sequence thus, any data block (node) can be uniquely identified by following this sequence up to the root (Fig. 4).

DPOR [21] uses BLS or RSA in such a way that multiple signatures verification can be done very efficiently. Thus, for each block family F, the data owner runs the signature generatour algorithm $(\Phi,\ sig_{sk}(H(R))) \longleftarrow SigGen(sk, F)$ which takes the signing key of the family (sk) and the encrypted block family F and generates the signature set for this family $\Phi = \{\sigma_1, \sigma_2, ..., \sigma_n\}$; where $\sigma_i \leftarrow (H(m_i) \cdot u^{m_i})^{sk}$ for each family block m_i; $u \leftarrow G$ is a random element choosed by the data owner. In addition, a signature of the root that associated MHT is generated $sig_{sk}(H(R))$. Then, each block m_i will be associated with its signature σ_i and some metadata such as block version and version of k and pk; $b_i = \{m_i || blockVer || kVer || pkVer || \sigma_i\}$ (Fig. 2). Finally, the data owner sends to the cloud storage the block family $\{b_1, b_2, ..., b_n\}$, its signature set Φ, the family key block table, and the root signature of this block family $sig_{sk}(H(R))$ (Message 1.1 of Fig. IV).

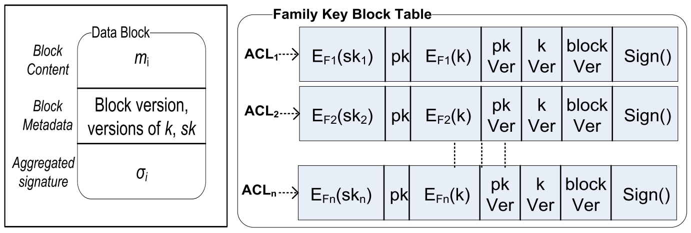

Figure 2. Data block and family key block table sent to the cloud

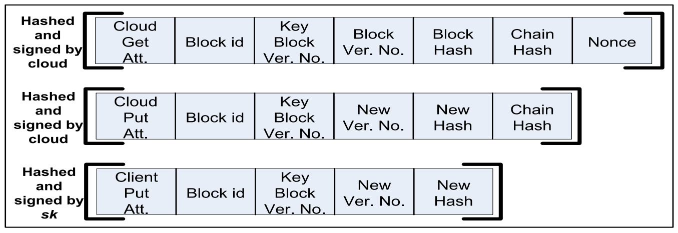

Figure 3. Attestations of Popa *et al.* [18]

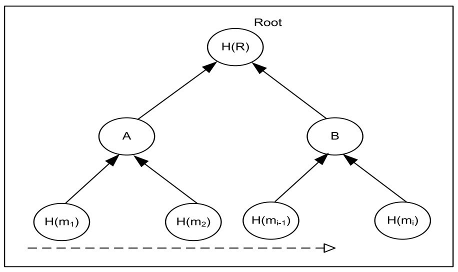

Figure 4. Merkle Hash Tree

i) Attestations: As in CloudProof [18] we build a hash chain from all data changes and requires signing from both parties on all updates. Thus, any misbehaviour could be detected and proved by exchanging attestations for each request or response between data owner, clients and cloud provider. The structure of exchanged attestations includes metadata such as the block version and current hash which are used to maintain the write-serialisability (when each client placing an update is aware of the latest committed update to the same block) and the hash chain value which is used for freshness (Fig. 3). The hash chain is computed over the hash of the data in the current attestation and the chain hash of the previous attestation. Thus it is a sequence of hashes which contains current attestation and all history of attestations of a specific block as follows: *chain hash* = *hash*(*data, previous hash chain value*). Thus, if the sequence of attestations is broken this means there is a violation of freshness property. In addition, during each epoch clients need to locally store all received attestations and forward them to the data owner for auditing purposes at end of each epoch (Fig. IV). For simplicity, in our proposal we assume that all data blocks will be audited, however, in practice a probabilistic approach as in CloudProof would be advantageous.

j) Get block: in the get (read) request for a specific data block, clients need to send to the cloud provider the block index (i) for that block and a random nonce (Message 2.1 of Fig. IV). The cloud provider will verify the client by checking the ACL and make sure that only clients with *read/access permission* (of the block) can gain access to this block. If the client is authorised then it will respond by sending the requested block (b_i) with its signature (σ_i), the cloud get attestation $Cloud_{get}Att$ and signature of the attestation $Sign(Cloud_{get}Att)$ (Message 2.2 of Fig. IV). The client will verify the retrieved attestation and make sure that it was computed over the data in the block and the nonce. Also, the client will verify the integrity signature (σ_i) of the received block. Clients need to locally store these attestations and their signatures and forward them at the end of each epoch for auditing purposes.

k) Put block: suppose the client wants to update a specific block (m_i) into (m'_i). First, the client needs to generate the corresponding signature σ'_i. Also, the client prepares the update (put) request message $update = (type,\ i,\ m'_i,\ \sigma'_i)$; where $type$ denotes the type of update (Modify M, Insert I or Delete D). In addition, the client will use sk to compute its put attestation $(Client_{put}Att)$ and sign it $sign_{sk}(Client_{put}Att)$. Then client sends $update$ message, $Client_{put}Att$ and $sign_{sk}(Client_{put}Att)$ to the cloud servers (Message 3.2 of Fig. IV). On the cloud side, cloud provider will verify the client by checking the ACL and make sure that only clients with *write permission* (of the block) can update this block. In addition, cloud provider will verify the client's attestation. If the client is authorised then it runs $(F',\ \Phi',\ P_{update}) \leftarrow ExecUpdate(F,\ \Phi,\ update)$ which replaces the block m_i with m'_i and generates the new block family F'; and replaces the signature$\sigma_i$with σ'_i and generates new signature set of the family Φ'; and updates the $H(m_i)$ with $H(m'_i)$ in the MHT and generates the new root R' (in MHT scheme as a new block added into or deleted from a file these new nodes are added to MHT as described in DPOR [21] and the tree is rearranged according to this update). The cloud responds to the update request by sending a proof for the successful update $(P_{update} = \{\Omega_i,\ H(m_i),\ sig_{sk}(H(R)),\ R'\}$; where Ω_i is used for authentication of m_i). Also, the cloud constructs the put attestation $(Cloud_{put}Att)$ and signs it $sign_{sk}(Cloud_{put}Att)$ and send them to the client (Messages 3.3 and 3.4 of Fig. IV). In addition, the cloud provider will store the received client attestations to be used if any misbehaviour detected. The client verifies the cloud put attestation and check the chain hash. Also, client verify the received update proof by running this algorithm: $\{(TRUE,\ sig_{sk}(H(R'))),\ FALSE\} \leftarrow VerifyUpdate(pk,\ update,\ P_{update})$ which takes pk, the old root's signature $sig_{sk}(H(R))$, the update message request ($update$), and the received proof (P_{update}). If verification succeeds, it generates the new root's signature $sig_{sk}(H(R'))$ for the new root $R'$and send it back to the cloud (Messages 3.5 and 3.6 of Fig. IV). In addition, client need to store all received cloud put attestation ($Cloud_{put}Att$) and forward them to the data owner for auditing purposes.

l) Auditing: the auditing process is carried out at the end of each epoch and consists of two parts. In the first part the attestations produced within the given epoch are verified as per CloudProof. In the second part, the integrity of the whole data set as in DPOR [21]. For each family block the TPA picks random c-element subset $I = s_1, ..., s_c$. For each $i \in I$, the TPA selects a random element $v_i \leftarrow Z$. Then TPA sends the message $chal$ which identifies which blocks to be checked ($chal = \{(i, v_i)\}_{s_1 \leq i \leq s_c}$). When the cloud provider receives the $chal$ message, prover will compute: 1. $\mu = \sum_{i=s_1}^{s_c} v_i m_i \in Z$; and 2. $\sigma = \prod_{i=s_1}^{s_c} \sigma_i^{v_i} \in G$. The prover runs $P \leftarrow GenProof(F,\ \Phi,\ ,chal)$ algorithm to generate the proof of integrity $P = \{\mu, \sigma, \{H(m_i), \Omega_i\}_{s_1 \leq i \leq s_c}, sig_{sk}(H(R))\}$; where Ω_i is the node siblings on the path from the leave i to the root *R* in the MHT. The verifier will verify the received proof by running this algorithm $\{TRUE, FALSE\} \leftarrow VerifyProof(pk,\ chal,\ P)$. This way we are able to check data availability at the whole file level.

V. Conclusion and Future Work

We have investigated the different type of existing cloud storage schemes and identified limitations in each one of them based on the security services that they provide. We have then introduced a cloud storage architecture that extends CloudProof in order to provide availability assurance. This is accomplished by incorporating a proof of storage protocol such as DPOR. The proposed POS architecture overcomes the weaknesses of both schemes. In this way we obtain a design that satisfies all the identified desirable security properties.

Both schemes are considered secure and work efficiently individually and it is reasonable to assume that they should work in a secure and an efficient way when combined. However, it may be interesting to perform a detail performance and security analysis of the proposed architecture.

References

[1] Amazon S3 availability event: July 20, 2008. http://status.aws.amazon.com/s3-20080720.html. [retrieved: Feb, 2012].

[2] Amazon Web Services. Amazon simple storage service FAQs, Mar 2011. Available at: http://aws.amazon.com/s3/faqs. [retrieved: Dec, 2011].

[3] Giuseppe Ateniese, Randal Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary Peterson, and Dawn Song. Provable data possession at untrusted stores. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 598–609, New York, NY, USA, 2007. ACM.

[4] Giuseppe Ateniese, Seny Kamara, and Jonathan Katz. Proofs of storage from homomorphic identification protocols. In *Proceedings of the 15th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*, ASIACRYPT '09, pages 319–333, Berlin, Heidelberg, 2009. Springer-Verlag.

[5] K.D. Bowers, A. Juels, and A. Oprea. HAIL: A high-availability and integrity layer for cloud storage. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 187–198. ACM, 2009.

[6] Kevin D. Bowers, Ari Juels, and Alina Oprea. Proofs of retrievability: theory and implementation. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, CCSW '09, pages 43–54, New York, NY, USA, 2009. ACM.

[7] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25(6):599–616, 2009.

[8] R. Chow, P. Golle, M. Jakobsson, E. Shi, J. Staddon, R. Masuoka, and J. Molina. Controlling data in the cloud: outsourcing computation without outsourcing control. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 85–90. ACM, 2009.

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

[10] Chris Erway, Alptekin Küpçü, Charalampos Papamanthou, and Roberto Tamassia. Dynamic provable data possession. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 213–222, New York, NY, USA, 2009. ACM.

[11] 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. In *The 8th IEEE Consumer Communications & Networking Conference*, 2010.

[12] Google. Security and privacy FAQs, Mar 2011. Available at: http://aws.amazon.com/s3/faqs. [retrieved: Jan, 2012].

[13] Ari Juels and Burton S. Kaliski, Jr. PORs: proofs of retrievability for large files. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 584–597, New York, NY, USA, 2007. ACM.

[14] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 29–42, 2003.

[15] Seny Kamara and Kristin Lauter. Cryptographic cloud storage. In Radu Sion, Reza Curtmola, Sven Dietrich, Aggelos Kiayias, Josep Miret, Kazue Sako, and Francesc Sebé, editors, *Financial Cryptography and Data Security*, volume 6054 of *Lecture Notes in Computer Science*, pages 136–149. Springer Berlin / Heidelberg, 2010.

[16] R.L. Krutz and R.D. Vines. *Cloud Security: A Comprehensive Guide to Secure Cloud Computing*. Wiley, 2010.

[17] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. In *Proc. OSDI*, 2010.

[18] R.A. Popa, J.R. Lorch, D. Molnar, H.J. Wang, and L. Zhuang. Enabling security in cloud storage slas with cloudproof. *Microsoft TechReport MSR-TR-2010*, 46:1–12, 2010.

[19] Hovav Shacham and Brent Waters. Compact proofs of retrievability. In *Proceedings of the 14th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*, ASIACRYPT '08, pages 90–107, Berlin, Heidelberg, 2008. Springer-Verlag.

[20] Cong Wang, Qian Wang, Kui Ren, and Wenjing Lou. Ensuring data storage security in cloud computing. In *Cloud Computing*, pages 1 –9, july 2009.

[21] Qian Wang, Cong Wang, Jin Li, Kui Ren, and Wenjing Lou. Enabling public verifiability and data dynamics for storage security in cloud computing. In *Proceedings of the 14th European conference on Research in computer security*, ESORICS'09, pages 355–370, Berlin, Heidelberg, 2009. Springer-Verlag.

[22] M.E. Whitman and H.J. Mattord. *Principles of Information Security*. Course Technology Ptr, 3rd edition, 2009.

[23] Z. Wilcox-O'Hearn and B. Warner. Tahoe: the least-authority filesystem. In *Proceedings of the 4th ACM international workshop on Storage security and survivability*, pages 21–26. ACM, 2008.

[24] Q. Zheng and S. Xu. Fair and dynamic proofs of retrievability. In *Proceedings of the first ACM conference on Data and application security and privacy*, pages 237–248. ACM, 2011.