Privacy of Data Outsourced to a Cloud for Selected Readers through Client-Side Encryption

Sushil Jajodia
George Mason University
Fairfax, VA
Jajodia@GMU.edu

Witold Litwin
Dauphine University
Paris, France,
Witold.Litwin@Dauphine.fr

Thomas Schwarz
Universidad Católica de Uruguay
Montevideo, Uruguay
TSchwarz@UCU.edu.uy

ABSTRACT

We propose a scheme using client-side encryption with symmetric keys for the privacy of data outsourced to the cloud for selected readers. The scheme is safe under the most popular "honest, but curious" model. Readers get the keys from access grants or have them cached. LH* files store cloud data and metadata. Diffie-Hellman scheme authenticates clients. Every client can read any data, but only a grantee decrypts the content. Access to data is usually the fastest possible that is two messages and the decryption, regardless of the cloud scale up. Data or grant creation or update costs are also constant with a few messages and fast processing. All these features serve our main goal: the search speed and scalability yet unmatched to our best knowledge. The scheme is finally intentionally very simple.

Categories and Subject Descriptors

C.2.4 [Cloud Computing]

General Terms

Algorithms, Performance, Security

Keywords

Scalable Distributed Data Structure

1. INTRODUCTION

The ever growing volume of data to store makes clouds attractive to data owners. Outsourcing to the cloud is often potentially more useful than caring about the private hardware. Usually, a data owner disseminates the cloud data to selected readers only. The update and access granting privileges remain with the owner. A course secretariat may distribute a class material to the students, letting each to read it, copy, print out… But, no student can modify the original and only a class member should get any documents. A student may depart; the secretariat should revoke any of her/his access rights. An employee usually produces documents to be read, perhaps copied or printed, but not modified, by other selected employees or customers. Only these persons should have the access, an unauthorized one could end up at Wikileaks. A "friend" in a social network often wishes to post information only for the "friends" and only for them. The saga of the Facebook Quit Day showed how strong this need became. Etc.

WPES'11, October 17, 2011, Chicago, Illinois, USA.

In this context, the cloud privacy (information assurance, safety, security…), is a paramount requirement, [2], [4]. No cloud storage intruder (adversary…) or unauthorized client should see the outsourced data. A natural approach is the client (owner) side encryption, despite the industrial push towards the server-side one, [7]. Symmetric encryption, e.g. the AES 256, appears the choice. The read grant should then somehow pass the encryption key to the grantee. To keep the key and grant (meta)data private, an obvious way is to have them at the grantor and grantees nodes only. The downside is potential for loss. A parade is the backup of the keys in the cloud. The client or a trusted administrator may recover them. The backup should be secure under a common threat model. The algorithms should be scalable and distributed, staying efficient under the data deluge.

We propose the scheme termed CSCP, for Client Share for Cloud Privacy, aiming at these goals. The owner encrypts every data record with unique key, using a symmetric encryption. Clients cache the keys, but all keys are also in the cloud. Each key in the cloud is hidden in a *public share* (PS) produced by the owner. Each public share belongs to a two-share secret. The secret is the key. The other share is the *client share* (CS), every CS being specific to the owner and each selected reader. Exceptionally, e.g., when the client privacy goes awry, a "super-client" termed *Admin* may rescue.

The CSCP client data and metadata records form three *catalogs*. These are physically the LH* files, [12], [11]. Such files have records structured into (primary key, (data) value) pairs. Records are stored according to the scalable distributed linear hash over the primary key. The CSCP performance announced in the Abstract are due to LH* efficiency, together with the speed of the encryption scheme, as well as of DH calculation and of one-way hash as we explain later. The cloud read access is usually the fastest possible, i.e., time for two messages per record plus the symmetric decryption. More costly CSCP cloud updates should be less frequent in practice. The most costly ones, i.e., the grant revoke and the client delete or takeover should be the least frequent.

We optimized CSCP for the read access and scalability under rapid data growth to practically any size, e.g., Exabyte. The result is yet unmatched to our best knowledge. The scheme has also a rather negligible storage overhead. It is finally intentionally simple. Besides the known tools, one needs only three files for tabular catalogs with a few attributes.

Space limitations force us to present CSCP only in short. Details and variants trading selected features for other properties, useful for selected applications are in [8]. Section 2 presents the CSCP architecture. Section 3 discusses the data manipulation. Section 4 analyzes the safety. Section 5 overviews the scheme's

performance. Section 6 deals with the related work. Section 7 briefly concludes and presents the future work.

2. CSCP ARCHITECTURE

2.1 Gross Architecture

A CSCP cloud is *public*, *private* or *hybrid*. A public cloud, e.g. Amazon or Azure, offers storage and related services, for rent at one or more *data centers*. A private cloud is for the internal use, e.g., Google or Yahoo. A hybrid cloud mixes both.

Every cloud is a collection of *server nodes* (servers) storing data manipulated by *client* nodes (clients). The clients are autonomous and initiate operations. They expect the cloud servers always available. In contrast, a client can be sometimes off-line. The cloud storage can be dominantly disk, flash or RAM. The latter constitutes a *RAM* Cloud. These are up to now only in private clouds, e.g., SDDS_2000 or Gemfire, [3], [6], [11], [13]. Depending on the cloud, the messaging time greatly varies. It is 3-5s per message at least for a typical public cloud, through a dozen of milliseconds for a private one, to under 100 microseconds for a RAM Cloud. A *CSCP* cloud can be any of these.

Every CSCP cloud has a specific client termed *Admin*. *Admin* provides exclusive trusted capabilities we introduce progressively. These are of exceptional use, avoiding to Admin being a hot-spot. They let Admin to manipulate the data of any client. Admin can be in also the LH* file coordinator. By usual analogy, Admin is a CSCP "super-client". Any other client is a *normal* one.

Every CSCP client *X* may create a data record. The client uses symmetric encryption on the new record and inserts the encrypted record, say *D*, in the cloud. It selects the server according to LH* addressing rules. *X* is then also the sole *owner* of *D*. Later, the ownership can move to another (unique) client and so on. Only the owner (i) has update, rekeying, and deletion rights on *D*, (ii) may disseminate *D*.

To disseminate *D* to another client *Y*, the owner *X* grants *Y* the *read* access to *D*. The grant hides the encryption key. Among normal clients, only *X* and the grantee *Y* may unhide it. Every other normal client can also read *D*. It will face however only the encrypted result. The grantee *Y* can make a fully owned copy of *D* and disseminate it, unless the application interface limits copying. The *grantor X* can finally revoke any of its grants, e.g. to *Y*. *Y* cannot then decrypt *D* anymore.

An owner *X* of data *D* can issue a grant to a reader representing in fact a group of users (a *g-client*). The g-client disseminates transparently for *X* a fully owned copy of *D* to these users, as we discuss more in [8]. The groups can be static or dynamic.

For all data and metadata, CSCP uses three catalogs, Section 2.5. These are scalable distributed linear hash (LH*) files with records structured as usual (key, value) pairs [12]. The value can be a large field. The key is a primary key, *p-key,* in the typical sense, i.e., without semantics for good hash. P-key can be derived from a single or multiple subfields in the value field, through cryptographically strong one-way hash, e.g., SHA 256.

2.2 Client Identity

Every CSCP client has a (unique) public *customary* identity (CID), e.g., an email address. Every client also carries a (unique) public *trusted* (certified) ID, we call TID. The client carries finally a (unique) secret *private* ID, (PID). TID and PID obey the *Diffie-Hellman scheme with published numbers*, (DH), [10]. If *X* is a CID, we usually note PID as *x*. *X*'s TID is then equal to g^x, where *g* is some *base*. The calculation is done in a finite (strong prime) field, e.g., with 512b numbers. The field and *g* are the common knowledge.

All cloud servers share a PID *s* and publish the corresponding TID g^s. Basically, Admin sets up any PID and recovers it if need occurs. A variant in [8] allows however the client to change the PID, even hiding the new one to Admin. Next, two clients or client and the cloud communicate all their data within their CSCP *session*. Each session has an ID (SID) formed from the client PIDs and TIDs as the Diffie-Hellman key. If *x* and *y* are thus PIDs, then the SID is g^{xy}. Each partner can efficiently calculate the SID from its own PID and the TID of the other party, as $(g^y)^x = (g^x)^y = g^{xy}$. , Every SID is unique to two parties. Also, the common belief in the security field states that only the two parties can calculate the SID unless one of the PIDs or entire SID of course, has leaked out to an intruder. Finally, it is believed that no one can determine the PID *x* or *y* from the SID g^{xy}. If it happened, it would be a breakthrough solving the *discrete logarithm* problem. Hence, in particular, despite common SID, a partner cannot calculate the PID of the other party, to form a SID of another session, especially with the cloud. All this is our rationale for use of the DH numbers as trusted IDs.

2.3 Client Authentication

A client interacts with the cloud through two types of queries (operations) presented in Section 3. A *usual* query is an LH* manipulation, i.e., a record (p-key based) search, insert, update, delete or (entire file) scan. The record update only affects the non-key data. The p-key and any record ID remain the same. A *signed* query also invokes LH* operations, but the cloud only executes it if the client authenticates successfully. For every client *X,* the signature is the tuple (*T, S*) that should carry the pair (g^x, g^{xs}). Successful authentication explores the both parties' capability to calculate the SID and the quality of the requestor for the operation. It is successful only if the server receiving the signature calculates that $T^s = S$. According to DH scheme and for all the reasons in Section 2.2 above, the server trusts the client then to be *X*. Next, the server may check whether *T* matches the TID in appropriate catalog. The matching means that *X* has the claimed capability.

Which server performs the authentication depends on the query. For a CSCP operation involving only a p-key based LH* operation, we recall that an LH* client sometimes commits an addressing error, sending the operation to an outdated address. Such error leads to one or at most two hops towards the correct server (bucket), see Section 2.5 below. For these operations, the correct server authenticates the client A CSCP query may also involve an LH* scan. The client sends (maps) this one only to the buckets known to the client image, at most. These may be not all the buckets in the file. The buckets getting the scan map it therefore through hops to ultimately every bucket. A CSCP scan is authenticated only by the servers that get it from the client. Since all the servers have the same cloud PID *s*, all fit the (unique) SID formed by the client, using g^s.

2.4 Data Encryption and Storage

The owner encrypts every value in a record with a new key using a strong encryption scheme. The LH* p-key is not encrypted as without semantics. If the actual data record ID carries information, e.g., as does a passport number, a one-way hash should convert it to the p-key.

2.5 Catalogs

Conceptually, each of three CSCP catalogs in the cloud is a relational table. We use the RDB conventions for catalogs' structure. However, CSCP manipulates here the catalogs as the (physical) LH* files only.

We store (client) data records in the *data catalog* D (DID, OID, ED). DID is the data record p-key in the relational sense(tuple ID). It is one-way hashed if needed to form the actual LH* p-key. OID is the TID of the owner. ED contains the encrypted data.

Each grant forms a record of *grant catalog* G (RID, DID, PS). RID is the grantee's (reader's) TID. DID identifies the granted record. Together they form the p-key of a record in G. PS is the *public share*. It is the publicly available share of the 2-share secret encoding an encryption key. The other share is called *client share* (CS). CS results from the SID of the grantor and the grantee and from some one-way hash discussed later. Only these partners among the normal clients know the SID hence the CS. As we demonstrate soon, only these partners among the normal clients may decode the secret, i.e., the key from the PS.

Finally CSCP uses the *Client* catalog C (CID, TID). Here, each client has one public record. Admin maintains C. It inserts client data upon the registration, the latter being beyond CSCP scheme.

All three catalogs being LH* files, all data are in buckets, one per cloud server. As the catalogs grow, they spread over more buckets. This results from the LH* splits occurring when buckets overflow. The LH* file *coordinator* chooses the bucket to split through the linear though also circular move of the *split pointer*. The data in the splitting bucket are rehashed so that half on the average migrates to the new bucket. This one receives the cloud PID *s* from the splitting bucket.

A client reads, inserts, updates, or deletes a record in an LH* file using the p-key based address calculation. Each client carries for this purpose the private *image* of the *file state*, consisting of selected parameters of the actual file. An image can become outdated, since clients are not aware of splits synchronously. The request may go to an *incorrect* bucket. The server forwards it towards the correct one. As we said, the hop can repeat at most once. This efficiency is unique to LH* when compared to DHT or consistent hashing schemes. If hop(s) occurred, an *image adjustment message* (IAM) updates the image so that the same error cannot repeat.

As we signaled, a client can also initiate a scan query *Q* that should reach each bucket in the file. For instance, a scan may broadcast over G query *Q* requesting to delete every grant over a given record, since the latter was deleted itself from D. The client sends a scan in fact only towards some or all the buckets in the image. These may map *Q* further so that every bucket yet unknown to the client gets it once and only once. The the servers may send results to aggregate or ack (reduce) individually to the client. Alternatively, they may aggregate them through other servers. The client gets fewer messages, even only one. In both cases, some *deterministic termination* protocol makes the delivery reliable if needed. All this makes the LH* scans *elastic* in the current cloud terminology. Unlike the yet typical Map/Reduce scan implementations, i.e., GFS and Hadoop based.

3. CSCP DATA MANIPULATION

3.1 Data Record Create

Through this query, the client inserts the data record. When client *X* creates a new record *D*, it is of the form (*d*,*r*) with p-key *d* and record contents *r*. *X* invents an encryption key *e* and encrypts *r* with *e* to obtain $e\{r\}$. *X* caches *e* and creates a *self-grant* over *D* in G. The self-grant backups up *e* , in case *X* loses it and has no other grantee over *D*. To create a self-grant, *X* calculates the self-SID g^{xx}. It then calculates $C = h(d, g^{xx})$ with a one-way hash function *h*. *C* is the self-CS. Next, *X* calculates the PS, say *P*, as $e \oplus C$. Note that *C* depends on DID. This is crucial for the scheme safety, Section 4. Finally, *X* sends to D the signed (i) insert of encrypted $D = (d, e\{r\})$ and (ii) request for the insert of the self-grant record (g^x, d, P) into G on behalf of *X*. The correct D server authenticates the client, by searching C in particular for g^x, and eventually performs the insert. Then, it resends the signed self-grant insert to G. If hops occur, *X* get IAMs.

3.2 Grant Record Create

This signed operation inserts a new (grant) record into G. If client *X* wants to grant to reader *Y* the access to record *D* with DID = *d*, *X* needs to provide *Y* with encryption key *e*. *X* uses the PS for this purpose. *X* starts by retrieving *e* from the cache or from the self-grant. For the latter, say *g*, with PS *P*, *X* recalculates the self-CS *C*, reads *P* and calculates *e* as $C \oplus P$. The correctness of this calculation when *X* performs it is easy to see. In contrast, we recall, no other normal client can calculate *C* hence may unhide *e*.

Next, *X* accesses to *Y*'s TID g^y, cached or in C catalog. Then, *X* forms the CS $C' = h(g^{xy}, d)$ and the PS $P' = C' \oplus e$. Finally, *X* sends the signed query, say *Q*, requesting the insert of the grant record $G(g^y, d, P')$. When the correct G bucket gets *Q*, it first authenticates *X*. If so, the server reads *D* and matches the TID against the OID in *D*. If those match as well it means that *X* is the owner of *D*, so *X* is allowed to insert *G*.

3.3 Data Record Read

Through this query, the client reads from D the record with DID provided as the p-key for the LH* record search. It is an unsigned operation. Every client can search and retrieve any record. However, as we already heavily insisted, only a grantee for the record profits from the retrieval.

3.4 Grant Record Read

This unsigned operation extracts the encryption key *e* of data record *D* with DID *d* owned by some client *X*. The reader, say *Z*, searching for the grant record (g^y, d, P) searches for it using the p-key (g^y, d). *Z* gets g^x from catalog C or from *D* in D or has it cached. *Z* calculates then the CS $C = h(d, g^{xz})$ and attempts to unhide the secret by calculating $e' = C \oplus P$. It is easily seen that $e' = e$ iff $Z = Y$.

3.5 Data Record Update

Through this signed operation, the owner *X* alters the data value. *X* first reads the data record *D* from D, retrieving *e* from the cache or the self-grant. *X* then decrypts the data, updates them, and re-encrypts. *X* finally issues a signed update to D that differs from a normal LH* one only by the authentication. The owner does not change the encryption key what would make the record unreadable to any grantee. The cloud cannot enforce the coherence of data and of PSs in grants.

3.6 Data Record Delete

This is a signed query deleting data record D with given DID d and all the related grants. The correct server (i) authenticates the owner, (ii) deletes D and (iii) issues the signed scan delete of every record X in G where X.DID = d.

3.7 Grant Revoke

In overview, this signed operation first deletes the requested grant, identified by its p-key consisting of RID, say y, & of DID, say d, of the data record, say D, encrypted with key e. Next, it rekeys D to new key e' and changes the PS in every other grant over D to reflect e'. The rationale is that the client could have e cached, hence without rekeying would continue to be able to decrypt D after the revocation. We recall that the grant has the form (g^y, d, P) with $P = C \oplus e$. To actualize P to P' hiding e', one may XOR P with the symmetric difference $\Delta = e \oplus e'$. Then $P' = P \oplus \Delta$. Only the owner may calculate Δ but the grant servers may calculate P'.

In more detail, owner X removing grant record R for client Y over D, sends the operation with Δ. These arrive to the correct server of R. The server deletes R and issues a signed scan for all other grants with DID = d. The scan requests every server with such a grant, say (g^y, d, P) to *Δ-update* it to $(g^y, d, P \oplus \Delta)$.

A reader familiar with the DH scheme could observe that SID is the exchanged encryption key. Hence, one could use it directly for hiding the key in the PS. This would be however a naïve design. It would be up to the client to carry every PS actualization. Letting the cloud to do most of the rekeying work through Δ-updates, should be usually by far more efficient, [8]. Having Δ-updates motivated therefore our CS design discussed in previous sections. The rationale appears to hold for any "competitor" to DH for our goal scheme, e.g., RSA. Nevertheless, deeper study of such directions remains a future work.

3.8 Client Delete

Client Delete removes the given client with all its data and metadata from the cloud. This signed operation is for Admin only. It deletes all the owned records and invalidates all cached keys of the client. To delete thus client X: (1) Admin deletes all records with CID X from C; (2) Admin deletes every data records D in D belonging to X, i.e., with OID = g^x, saved if needed from C, (3) Admin deletes every grant R issued by X, i.e., where for every D from step (2), R.DID = D.DID. Finally, (4) Admin deletes every grant R to X by some owner Y, i.e., where R.RID = X. For each R, Admin rekeys the granted data record D and Δ-updates every related PS. Admin can do it since it knows PID y of every data owner Y. The enumerated operations use scans. Details are in [8] since lengthy.

3.9 Client Takeover

This is another operation reserved to Admin. It applies, e.g., to a departed client with all the data and capabilities to preserve for another client. Admin uses it also for the client node theft, where the thief could learn the PID. The takeover brings thus to client Y all the data and capabilities formerly belonging to another client X. To change ownership of data, each data record is rekeyed and updated as belonging now to Y. This triggers changes to all grants with X as grantor, creating new PSs. Finally, all grants with X as a grantee are Δ-updated. Details are in [8].

4. CSCP SAFETY

4.1 Threat Model

Our threat model considers adversaries which are clients or intruders to the servers. We use the popular "honest, but curious" model. In particular, messages among servers are trusted. Next, an intruder might attempt to decrypt stored data, but refrains from any malicious data or software modifications. Thus, the intruder may use some legitimate cloud insider's rights to access a server, e.g., to execute a storage dump. Finally, if the intruder happens to be also a cloud client, then s/he only attempts as the client the operations aiming at the data decryption.

On the client side, a client X who is not a reader for data record D may try to read it. Likewise, a reader of D may attempt to update D or delete it. Or, the reader may attempt to update, create or delete a grant on it. Finally a non-grantee may attempt these operations as well.

Next, as usual, the network between the client and the cloud does not allow snooping at the transiting data. Then, no client releases maliciously the PID or any key used to any other client. A client not being also a cloud intruder may attempt any not granted but potentially legal manipulation, like we just discussed. The client and cloud intruder, succeeding to get the cloud PID s, does not attempt however to use a fake SID, despite the capability to form any SID in such a case. The reason is that it would not let the intruder to decrypt any data anyhow, as we show below. The only result could be a blind modification of some data or metadata. Hence, it would be a "dishonest" outcome beyond the model.

As discussed, a client site may fall into illegitimate hands. The illegitimate client may use any apparently legitimate capability, as long as the client is not deleted or taken over by one with a different TID.

4.2 Cloud and Client Side Safety

At the cloud side, according to our threat model, the intruder, say X, who is not a client, only sees encrypted data and CSCP metadata. The use of a strong encryption scheme makes up to date impossible to decrypt any record directly. Neither X may decode any PS. Each CS is by its construction a very large pseudo random number. By properties of the secret-sharing X cannot decrypt the key from the PSs. X needs the PID of the attacked PS. But there are no PIDs at a cloud server. Also no one up to date broke the well-managed DH scheme so to find PID, say x from the public w^x, hence known also to the cloud intruder. Thus the stored data are safe against the cloud side intrusion under our threat model.

At the client side, the safety means first that (i) a grantee, reader or owner, of record D may always decrypt D, (ii) no normal client other than the grantee may decrypt D. Also (iii) a reader of D cannot update or delete D, nor create, update or revoke a grant on D. Next, (iv) between the normal clients, D gets updated, deleted only by its owner. Likewise, a grant on D gets created, updated or delete only by D owner as well. Finally, (v), an illegitimate client C loses any capability to manipulate D, once C revoke or takeover occurred.

For (i) the sections defining the algorithmic of the various operations under the scheme should make clear that any grantee of D can always decrypt D through cached key or PS of the grant. The latter possibility remains for D rekeyed any number of times.

For (ii), observe first that the CSCP algorithmic of trusted IDs guarantees that no client can pretend to be another for ID-secured operation. More precisely, in addition to the general properties of the DH calculus, including its robustness, it cannot happen that client *Z* (i) makes client *X* believing that *X* is in session with some *Y*, while the session involves *X* and *Z*, and (ii) makes *Y* similarly believing in the session with *X*. This is known as the "monkey in the middle" attack. The reason is that only Admin issues the PIDs and posts the TIDs in C, being a trusted party. The attack could in contrast occur if clients negotiated the SIDs themselves. See the DH literature for details. The consequences of such an attack if it could succeed are obvious.

Next, for (ii), let *Y* be some client attempting to decrypt *D*. *Y* can be either a former grantee of *D* or not. In both cases *Y* can get *D* from the cloud. To decrypt *D*, *Y* needs its key, say *E*, in the cache or has to calculate it from some PS. If *Y* is not a former grantee, it cannot have *E* cached. *Y* may nevertheless have key *E'* of another record for which *Y* is/was a grantee in the cache. Whatever *E'* is, we have $E \neq E'$ and it is impossible in practice to infer *E* from *E'*. To decode the PS in some grant over *D* is therefore the only remaining possibility for *Y*. The read access to G is not ID-secured, hence *Y* can get any grant over *D*, and any grant record in general. This by issuing an unsecured search in G, using the adequate publicly known CID, e.g., of friend known to be a grantee of *D*, and DID of *D*. As for the cloud intruder, the brute force decoding is however hopeless, at least till now.

Another way *Y* could try out, is to find whether *Y* and *D* owner, say *C*, are both grantees over some record *D'*. Then, *Y* knows key *E'* of *D'* and may always calculate the CS in *C*'s grant over *D'*. If this CS was by any chances the same for *D*, e.g. it would be simply the SID of the session between *Y* and *C*, then *Y* would be able to calculate *E*, without *C*'s PID.

Impeaching such an attack is among rationales for our CS design. The *h* function calculation makes CS in practice unique for every DID and SID, i.e., every data record and session. Next, *h* is one-way to render up to now impossible to determine the SID. Hence, whatever are *D* and *D'*, knowing CS for *D'* would not let *Y* to determine CS used for *D* and decrypt *D* in consequence.

If in turn *Y* is a former grantee of *D*, *Y* may have *E* cached. *Y* is a former grantee iff the grantor, or the grantor's takeover or revoke has revoked the grant. The grantor revoke would delete *D*, hence, trivially; *Y* would not be able to decrypt it. Revoking the *Y* grant in both other cases would trigger the rekeying with $E' \neq E$ and impossible to infer from *E* at present, whatever *E* could be. No *E* would let thus *Y* to decrypt *D* anymore.

Finally for (ii), let *Y* be also the cloud intruder. *Y* could then see some DID and the cloud PID *s*. By definition, a DID is a random value unrelated to its record data value. Hence, no DID is of use for *Y*. With *s* in contrast, *Y* could alter the actual TID g^y into a fake one. More precisely, *Y* could choose g^z of any client *Z* and generate fake SID g^{zs} used by *Z* for an ID-secured operation. But, no CS uses such a SID, only a SID of a session between the grantor and the grantee, not between a grantee and the cloud. Hence a fake SID would not let the intruder to decrypt any data neither.

Next, for (iii), a reader *Y* attempting to write or delete *D*, or to create, update, or delete a grant on *D* has to issue the ID-secured query using the SID g^{ys}. Any server receiving the query directly from *Y* or through LH* forwarding would match *Y*'s TID g^y to *D*'s owner TID g^c in stored *D*. *Y* cannot enter a fake TID into a SID at present, as no one succeeded breaking the DH scheme yet. The matching must be negative and the server would not execute the query.

For (iv), if $Y = C$, then *Y* may do all the enumerated updates through their (tedious) related already discussed algorithmic details. Otherwise, if *Y* is not a cloud intruder, the ID verification will block any such attack. Finally, if *Y* is also the cloud intruder, *Y* could enter g^{cs} and pass the ID check. As we discussed, this would not allow however *Y* to decrypt *D*. Hence, it would only let *Y* to perform blind updates creating havoc in the system. Under our threat model, *Y* refrains of any such attacks, as "dishonest", i.e., malicious ones. A big stick motivating such honest behavior is that a cloud attacker and client is usually a cloud insider so among the first to get investigated, as a likely culprit.

Finally, for (v), it is again the already discussed correctness of tedious details of the client revoke and takeover operations that proves this facet of CSCP safety.

5. PERFORMANCE ANALYSIS

CSCP performance is analyzed in detail in [8], with respect to messaging, storage and processing on various types of clouds. Here we overview the messaging costs that are the basic ones. The processing costs due to DH calculations, data encryption/decryption and CS and PS computations appeared relatively secondary. The analysis is based on the well-known LH* performance figures [12].

As usual, we measure the messaging performance through the number of messages and rounds per operation. The data record search cost *S* with cached key is $S_{\min} = 2$ messages at best, in $R_{\min} = 2$ rounds. At worst, there are two LH* hops hence $S_{\max} = R_{\max} = 4$. If the key has to be also brought from the cloud, we have the costs, say *S*' and *R*', of $S' = 4$ and $R' = 2$ at best, and $S'_{\max} = 8$ in $R' = 4$ rounds. The reason is that the client searches D and G in parallel.

The LH* hops should be rare. Next, the key should be typically brought from the cloud infrequently. The average search costs should thus be close to minimal ones, i.e., $S_{avg} = R_{avg} = 2+\varepsilon$. E.g., if a data record is on the average read about ten times, $S_{avg} = R_{avg} \cong 2.2$. Detailed ε values remain to get determined. They depend on bucket sizes of D and of G, data record and grant creation rates etc. These features determine the frequency of hops for LH* files.

The minimal cost of a data record insert, always with its self-grant, is $I_{\min} = R_{\min} = 2$. The maximal one is $I_{\max} = \underline{R}_{\max} = 6$. Hops are rare hence we should have $I_{avg} \cong I_{\min}$ and $R_{avg} \cong R_{\min}$. All these most frequent CSCP costs are constant, i.e., independent of the file size involved. They offer thus the best scalability. The remaining infrequent costs are less straightforward to evaluate, but should remain practical. The search response time appears from 100μs for a RAMCloud to 6-9s for a public one. The RAMCloud speed is due to absence of DH calculations that costs at least a few ms. Notice that some see RAMClouds among dominant trends of the future cloud business. See [8] for details.

6. RELATED WORK

Cloud data privacy (information assurance, safety…), including trusted identity management, is a paramount goal for the cloud industry, [2], [4]. Data privacy through the access privileges is among classical goals of data management with countless proposals, e.g., [9]. The use of the data encryption for this goal is

more recent. Distributed file system research studied it since the Andrew FMS. More recently, it appeared practical for the new data outsourcing needs, [5]. Combination with the ID management, especially using the DH scheme, proved useful for the groupware. P2P research also has recently experimented with, [1]. We are nevertheless aware of only a few cloud specific proposals. None appears fully competitive to ours.

One recent proposal of the access control through the data owner side encryption is [16]. The scheme is for the non-cloud applications. Like CSCP, it uses the symmetric encryption and the key encoding through one way hash, into so-called tokens. It provides the same granting capability as CSCP, i.e., the read-only non-transitive grants. The grant revoke similarly includes the rekeying. The grants, client and records are however managed through a dedicated graph structure, where the edges represent the grants. The graph usually requires less storage than our tabular catalogs, perhaps several times less. The grant manipulation can be also faster through the edges instead of our scans. However, there is no method known for making the graph used easily scalable and distributed. Unlike the tabular, hash or range partitioned, structures, already in cloud infrastructures. Finally, in [16], the secure client identification is not a concern. Neither DH nor any other trusted ID system is in the scheme.

In contrast, the DH system is within popular MS groupware SharePoint, [14], derived from Groove. A group uses a unique *shared space*, initiated by *Admin*. Each shared space has its unique symmetric encryption key and a shared key, the *group* ID, for signing the messages and for the (shared) group client ID. If any client leaves a group, all stored data are rekeyed. Just like in CSCP. CSCP corresponds further to the *mutually suspicious* mode in SharePoint, where each message is signed with the individual ID of the owner. SharePoint also have a simplified *trusted* mode, intended to offset the cost of the DH calculations.

SharePoint is the probably most used of many groupware tools. All use a central server. None is yet really cloud oriented. Also, we could not locate any study of SharePoint performance. We can nevertheless expect that the average time of CSCP data record read, for practically any data collection size, could be at worst about that of a single client-to-client SharePoint message in trusted mode for the limited to TB size SharePoint can apparently manage at present.

With respect to the DH calculation, there is a huge literature, as well as for the encryption. See any search engine or [8] for some pointers to. Finally, the CSCP goal of client-side encryption is shared by the LH*$_{RE}$ and Clasas schemes, [7], [15].

7. CONCLUSION

CSCP appears practical for cloud privacy, when data owner disseminates the data to selected readers. It is safe under the most popular cloud threat model. It also provides the read access about as fast as possible and of constant cost regardless of data file scalability. Other CSCP operations have higher costs, but are less frequent and scale well as well. All this makes the scheme the best choice at present for its intended goal. The future work on the scheme should include the simulations, prototyping and experiments. One may also add capabilities to support additional threats or expand CSCP current capabilities.

8. ACKNOWLEDGMENTS

The work of Sushil Jajodia was partially supported by the National Science Foundation under grants CCF-1037987 and CT-20013A.

REFERENCES

[1] Bonifati, A. Liu, R., Wang, H., W. Distributed and Secure Access Control in P2P Databases. DBSec 2010.

[2] Cloud Security Alliance. Security Guidance for Critical Areas of Focus in Cloud Computing. https://cloudsecurityalliance.org/csaguide.pdf

[3] Diene, A. W., Litwin. W. Performance Measurements of RP*:A Scalable Distributed Data Structure For Range Partitioning. 2000 Intl. Conf. on Information Society in the 21st Century. Aizu City, Japan, 2000

[4] ENISA Cloud Computing Risk Assessment. http://www.enisa.europa.eu/act/res/other-areas/

[5] Foresti, S. Preserving Privacy in Data Outsourcing. Spinger, 2011

[6] http://www.gemstone.com/products/gemfire.

[7] Jajodia, S., Litwin, W. & Schwarz, Th. LH*^{RE}: A Scalable Distributed Data Structure with Recoverable Encryption. IEEE-CLOUD 2010.

[8] Jajodia, S., Litwin, W. & Schwarz, Th. Privacy of Data Outsourcing to a Cloud for Selected Readers. Res. Rep. Lamsade, Feb. 2011.

[9] Jajodia, S., Smarati, P., Sapino, M. L., Subrahmanian, V. S. Flexible support for multiple access control policies. ACM-TODS, 26(2), 2001.

[10] Kaufman, Ch., Perlman, R., Speciner, M. Network Security: Private Communication in a Public World. (2nd Ed. Prentice Hall, 2002

[11] Litwin, W. Moussa, R., Schwarz, Th. LH*$_{RS}$ A Highly-Available Scalable Distributed Data Structure. ACM TODS, 10, 2005.

[12] Litwin, W., Neimat, M-A., Schneider, D. LH* - A Scalable Distributed Data Structure. ACM TODS. 12, 1996.

[13] Ousterhout, J. & al. The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM. ACM SIGOPS Operating Systems Review, 43 4, 2010.

[14] Sharepoint 2010. http://sharepoint.microsoft.com/en-us/pages/default.aspx

[15] Schwarz, Th., Long, D. Clasas: A Key-Store for the Cloud, MASCOTS 2010

[16] Vimercati, S., Foresti, S., Jajodia, S., Paraboschi, S., Samarati, P**.** Encryption policies for regulating access to outsourced data. ACM TODS, 35,2, 2010.