KEY MANAGEMENT SERVICE: ENABLING SECURE SHARING AND DELETING OF DOCUMENTS ON PUBLIC CLOUDS

Surya Nepal, Carsten Friedrich, Catherine Wise, Richard O. Sinnott*, Julian Jang-Jaccard†, Shiping Chen
CSIRO Data61, Australia
*Department of Computing and Information Systems, University of Melbourne, Australia
†The Centre for Information Technology, Massey University, New Zealand
{FirstName.LastName}@csiro.au, rsinnott@unimelb.edu.au, jjang-jaccard@massey.ac.nz

Abstract
The primary focus of existing secure cloud storage solutions have been on securing data both in motion and at rest. These storage solutions mostly focus on three essential properties: confidentiality, integrity and availability. However, modern enterprise applications demand data can be shared within or across organizations. The challenge is how to securely share data in public clouds using federated identities without increasing data movement and computation costs. Furthermore, the consumer should be able to delete their data in the cloud in the context of collaboration without leaving any traces behind. This problem has been addressed in recent times by utilizing or developing new data encryption techniques such as identity-based encryption, attribute-based encryption and proxy-re-encryption. However, these techniques suffer from scalability and flexibility problems when dealing with big data and support for dynamic and federated access control. This paper presents a novel architecture and corresponding protocols to provide secure sharing and deletion of documents on public cloud services: CloudDocs. This system uses AES for data encryption to achieve scalability, supports identity-based access control rules using private-public key pairs to provide flexibility, and uses independent key management services to support secure deletion, whereby the data is irrecoverable once the keys are destroyed. The key management service also supports dynamic and federated access control.

Keywords: cloud storage, data storage, public clouds, data sharing, data deletion, key management

1. INTRODUCTION

Cloud storage services have become popular for not only individuals to store their digital life, but also organizations to store business data for a range of reasons: their scalability, the high quality of service, ease of use, and significantly reduced cost (Armbrust et al., 2010). Cloud storage services rely on an outsourced business model, where data is accessed over the Internet and the data owners do not typically have full control on where they are stored, how they are stored, who are managing them, etc. (Buyya, et al., 2009). Thus, the cloud storage services are inherently vulnerable to threats from both local and remote adversaries (Kaufman, 2009).

In recent times, many widely published reports on data breaches at cloud services such as Apple’s iCloud, Expedia Inc.’s Trip Advisor, and Sony Corp’s online entertainment services have occurred. These breaches have not only exposed the threats and vulnerabilities of such services, but also have raised awareness among cloud users about the problems with storing their data using web services which are opaque. This problem has been studied by a number of researchers in the last few years and resulted in a number of solutions such as HPISecure (Saleh & Meinel, 2013), Twin Clouds (Bugiel et al., 2011), SecCSIE (Seiger et al., 2011), and SecCloud (Wei et al., 2010). We have also contributed in this space by developing one of the earliest secure data storage systems for cloud services, called TrustStore, and reported it in (Yao et al., 2009). These systems were developed with the intention of providing data confidentiality, integrity and availability (CIA) – three key features of data security. However, one of the requirements of individuals as well as organizations in the connected world is to share data with others in a controlled manner. This requirement has driven the development of collaborative systems, but it raises the question of how to securely share data that is stored in cloud services in the context of collaboration among individuals from different organizations using their own identities.

Information assurance while sharing data in collaborative environment is achieved through access control mechanisms (Tolone et al., 2005) [8]. However, their cloud-based application is limited as the data resides outside the defense perimeter (i.e., organizational boundaries).

Public key infrastructure (PKI) is another popular mechanism that can support secure data sharing in the cloud environment (Solo et al., 1999). In PKI, the data is encrypted using the public key of another user with whom the data is being shared so that only the user with the associated private key can decrypt the data. One of the key problems with this approach is that this solution is not scalable, as this technique requires all the data to be
encrypted with the public key of each user. To overcome this limitation, a number of encryption techniques have been developed (Ateniese et al., 2006; Bethencourt et al., 2007; Canetti & Hohenberger, 2007; Goyal et al., 2006; Liu et al., 2014; Sahai & Waters, 2005). We can classify them into three general categories: attribute-based encryption, proxy re-encryption and a combination of two.

Blaze et al. (1998) introduced the proxy re-encryption technique that enables data encrypted using one user’s public key to be transformed in such a way that it can be decrypted using another user’s private key. This removes the burden for users to know the public key of other collaborators with whom the data is being shared. This functionality is moved to the proxy. Though this technique overcomes the need to encrypt data with the public keys of all other users, there is no underlying access control mechanism associated with the data. To overcome this limitation, attribute-based encryption techniques have been proposed (Sahai & Waters, 2005; Shamir, 1985).

In attribute-based encryption techniques, the encrypted data can only be accessed by those users whose keys match the attributes. Access control policies are defined by attributes. Two different approaches have been used to define access policies: (a) attaching policies with the encrypted data (Goyal et al., 2006) and (b) attaching policies with the encryption keys (Bethencourt et al., 2007). However, attribute-based encryption techniques incur heavy computational burdens and as such they do not scale well when supporting fine-grained access control. Yu et al. (2010) have proposed a new technique to overcome this problem. It combines proxy re-encryption and attribute-based encryption.

The above observations led us to believe that existing solutions either (a) focus solely on securing the data in the cloud (e.g., HPISecure, SecCloud, TrustStore, etc.) or (b) devising a new data encryption technique to incorporate access control mechanism for data sharing (e.g., proxy re-encryption, attribute based encryption, etc.). In our previous paper, we propose a novel way of introducing an access control mechanism to an existing secure cloud data storage solution as part of the key management service. We demonstrated our solution by building the CloudDocs application (Wise et al., 2015), utilizing a secure data storage system called TrustStore (Yao et al., 2010). Our approach supports scalability and flexibility as (a) the data is encrypted using the Advanced Encryption Standard (AES) so that it scales well to a high volume of data without having a significant increase in the computation cost, and (b) the key management service supports the flexibility required by defining and managing access control along with users and keys. This paper is an extension of CloudDocs (Wise et al., 2015). In this paper, we extend the key management service to support federated access control through the Australian Access Federation (AAF) for CloudDocs. In addition, we also demonstrate how the key management service, in conjunction with integrity management, service can support policy-based secure deletion – an essential capability for cloud storage services.

The rest of the paper is structured as follows. We briefly provide the summary of our secure cloud storage system TrustStore in Section 2. This is followed by the detailed description of the secure data sharing application CloudDocs, built over TrustStore in Section 3. Section 4 describes two key aspects of key management service: AAF support and support for secure data deletion. In Section 5, we provide the experimental results and evaluation. Section 6 provides a brief summary of related work. The final section draws conclusions and outlines potential future areas of work.

2. BACKGROUND: TRUSTSTORE

Figure 1 shows the TrustStore architecture. It is one of the earliest secure cloud storage systems developed to provide confidentiality, integrity and availability (CIA) to data stored in the cloud (Yao et al., 2010). It’s key features can be summarized as follow:

- It provides an overlay storage service under existing cloud storage services. Hence, it supports multiple storage services from different cloud vendors. Thus it is possible to develop hybrid, redundant virtual file systems using replication for high data availability.
- It decouples keys from the encrypted data using an independent key management service that
can be deployed in a (semi)-trusted environment.
- It provides an independent integrity management service (Nepal et al., 2011) that supports both online and offline integrity verification.
- It supports different APIs so that a variety of applications can be developed and deployed in different platforms.

TrustStore has 3 main layers: an application layer, the TrustStore client, the TrustStore server-side services, and the cloud storage services. It is built following the principles of service-oriented architecture. Hence, the services in each layer are developed and deployed independently interacting through a standardized API. Such loose coupling of services enables flexible development and deployment of different applications utilizing TrustStore. We describe each layer and their core components and functionalities.

2.1 CLOUD STORAGE SERVICES

The cloud storage services layer interfaces with a collection of cloud storage services from heterogeneous service providers such as Amazon, Azure and Australian Cloud resources such as the National eResearch Collaboration Tools and Resources (NeCTAR) Research Cloud. In this layer, we abstract the underlying storage infrastructure and provide a uniform interface for the rest of the client operations; for example, getting/putting data from/to the storage infrastructure, supporting authentication to the storage infrastructure, etc. This abstraction helps higher layers to access the storage infrastructure without needing to be concerned about specific technologies, topologies and their location. Due to its support for multiple cloud services from different vendors concurrently, hybrid cloud data security solutions such as Twin Cloud (Bugiel et al., 2011) can be implemented in TrustStore without many changes. For example, a Twin Cloud solution can be implemented by storing sensitive data in a private cloud (e.g., the CSIRO internal cloud) and non-sensitive data in the public cloud (e.g., Amazon S3). By default, TrustStore treats all data as sensitive data, and stores them in encrypted form on all cloud services. This multi-cloud support also enables the high availability requirement for a secure system.

2.2 TRUSTSTORE SERVERS

The TrustStore Servers are implemented as services following the principles of service-oriented architecture. These services operate in a trusted environment. Two key services are designed to support two key characteristics of data security: confidentiality and integrity.

Key Management Service (KMS) is the key component of the server-side software. It manages (a) the symmetric keys, (b) the public/private key pairs of users, (c) user accounts (d) data sharing through an access control mechanism and (f) federated identity based access. The first function supports the data confidentiality property of data security. The later three functions are added into TrustStore to support secure data sharing, which was the main focus of CloudDocs (Wise et al., 2015). The last part supports the federated access control, which is the main extension to the CloudDocs and focus of this paper.

Integrity Management Service (IMS) is used to verify the integrity of the data and thus supports the data integrity property of the security requirements. The IMS collects and stores hash values of all data before they are sent and stored in the cloud. The hash values are then subsequently used to verify integrity of the data when it is retrieved. IMS can support both passive as well as active verification of the integrity of the data stored in the cloud. Stored hash values in IMS can also be used to implement the Proof of Erasability (PoE), which we describe later.

The three fundamental aspects of security are provided by three independent services: KMS (data confidentiality), IMS (data integrity) and the replication support of the client. It is not necessary that TrustStore provides server-side services, i.e. they can be provided by trusted third parties. For instance, data can be encrypted to be stored in multiple storage services, or multiple KMS instances could be established in more or less secure environments for the protection higher or lower security data. If employees of an organization are using it, the IT department of the organization can provide the trusted services to all employees, rather than individual employees having his/her own services.

While each entity is subject to fraud and deception with certain probabilities, the chance that multiple entities collude with each other and tamper a given user’s data without being detected is very low.

2.3 TRUSTSTORE CLIENT

The TrustStore Client serves three purposes: (a) it provides a user interface, (b) it interacts with the TrustStore server layer, and (c) it provides those security functionalities which need to be deployed in the most trusted environment of all, that is, local data partitioning, replication, key generation, and encryption/decryption.

TrustStore has a number of client applications on different platforms, to support different use cases. The current implementation of TrustStore has five different clients: WebStart (Java), Android (Java), iPad (Objective-C), Windows (C Sharp), and cross platform command line (Python). The Windows client lets the user mount and unmount a store as if it were a disk drive in Windows; the Python client is used as a cross platform command line client as well as for TruXy (Trusted Galaxy Workflows in the Cloud), and the WebStart application is used as a cross-platform FTP-like client.

The cryptographic libraries provide the ability to encrypt and decrypt file fragments, produce and verify message digests, and generate keys. It uses a variety of standard,
third party libraries containing cryptographic algorithms such as AES, RSA, CMS and SHA. The basic functionality of the TrustStore client can be described as follows.

When a user uploads a file, the file is first fragmented into a number of pieces. A fragmentation map is then created. The fragmented data and the fragmentation map are encrypted separately by the encryption/decryption module. During download, the individual fragments are merged back together using fragmentation map to generate the complete file.

Each fragment is individually encrypted with its own randomly generated key and initialisation vector. The encrypted fragments are uploaded to the storage providers and the keys are stored with KMS. During download, the fragments are downloaded from the first available storage provider and decrypted with the corresponding keys, which are retrieved from the KMS.

2.4 APPLICATION LAYER

The Application Layer allows a variety of enterprise applications from different domains to transparently use the underlying Cloud storage infrastructure for their specific purposes via the TrustStore clients. Depending on the choice of implementation, clients can either fully or partially rely on TrustStore to access data in the cloud. In the former case, they can use TrustStore as a Cloud storage broker to fully administrate users’ data. In the latter case, they may utilize only certain parts of its functionality; for instance, just the services to manage encryption keys for data confidentiality. TrustStore also provides an effective way of overcoming some common cloud storage service restrictions, such as limited file sizes.

3. CLOUDDOCS

The confidentiality and integrity aspects of TrustStore were reported in (Yao et al., 2010) and (Nepal et al., 2011), respectively. KMS provides the confidentiality by managing symmetric keys. In CloudDocs (Wise et al., 2015), we extended KMS to support secure data sharing by (a) managing public/private keys, (b) managing user accounts and (c) supporting an access control mechanism for all keys and metadata. In this paper, we further extend CloudDocs to support federated access and secure deletion. Before explaining the CloudDocs protocols and extensions, we provide the motivation behind its development.

3.1 MOTIVATION

TrustStore provides secure cloud storage services for different application domains. We have developed an application called TruXy, a trusted integration of TrustStore and Galaxy. The motivation for this work comes from the post-genomic life sciences (Sinnott et al., 2013; Stell & Sinnott, 2013). In this model, highly sensitive data such as human genomic data is provisioned, stored, and analysed through on the Cloud to identify disease-causing variants. The detailed analysis of these data sets requires complex workflows to be defined and developed that explicitly address the many privacy and security concerns that exist regarding such data. The TruXy system (Nepal et al., 2015), was developed to tackle these issues focusing on workflow environment (Galaxy). The need for secure data sharing and collaboration is a common requirement. Hence, we developed a generic application for data sharing utilizing TrustStore, called CloudDocs. In the following, we explain the protocols in the CloudDocs system and describe the process for how it works.

3.2 USER REGISTRATION

A new user registers using CloudDocs by providing a new username and corresponding password. The client then generates a public-private key pair and registers the user with KMS. The user authentication with KMS in CloudDocs is managed via an OAuth2 flow using the resource-owner password credentials grant, so the flow of data in the registration process can be described as follows:

1. User → Client : Username, Password

2. Client → KMS : Kpub, Kpriv ← generateKeyPair()

3. Client → KMS : Username, Kpub, EncPassword(Kpriv), Hash(Password, Salt)

4. KMS → Client : OAuth Key, RefreshToken

The public key in step 2 is stored using the X.509v3 RSA certificate. Similarly, the private key is encrypted with a password using PKCS#8 EncryptedPrivateKeyInfo. The hash function used is PBKDF2 SHA256. All communications between client and KMS (steps 3, and 4) are done over SSL.

As the user’s password is used to encrypt the private key, the password is needed to access any data stored in CloudDocs. This password is never sent to any of the services, including the KMS. This means that CloudDocs provides a high level of security, as the data cannot be decrypted even when all services (KMS, IMS and cloud storage) in the system collude with each other or are compromised.

3.3 USER SIGN-IN

Once the user is registered they can sign in with the CloudDocs application. The sign in process starts with the client requesting the user’s username and password. The
client then sends the username to KMS and retrieves the login salt. The client then applies a hash function to the password and salt as described in step 2 of the registration process above. The salted, hashed password is then sent to KMS as part of the OAuth2 password flow login. In return, KMS sends an OAuth Key (as described by the OAuth 2 protocol), which is then used by the client to request the user’s encrypted private key. The private key can then be decrypted in the client using the (un-hashed) password. All public keys are available to all users. The data flow in the process is described as follows:

1. User → Client : Username, Password
2. Client → KMS : Username
3. KMS → Client : Salt
4. Client → KMS : Hash(Password, Salt)
5. KMS → Client : Kpub, Oauth Key, Refresh Token
6. Client → KMS : OAuth Key, RequestPrivateKey
7. KMS → Client : EncPassword(Kpriv)
8. Client → KMS : DecPassword(EncPassword(Kpriv))

It is important to note that in the process described above, the private keys are not recoverable if the password is lost, since they are encrypted using the password. It can only be changed if the user knows the existing password, in order to first decrypt the private key. The discussion of recovering lost passwords is beyond the scope of the paper and has not been implemented in the current version of CloudDocs. Alternative mechanisms will be explored in future works.

3.4 Creating a Store

The next step in CloudDocs is to create a store—a virtual file system—utilizing the cloud storage services provided by different vendors. The process of creating store contains two steps: (a) setting up the store identity, and (b) setting up the profile.

Setting up the Store Identity.

The client first sends a request to KMS using the OAuth2 protocol, containing the metadata for the store. KMS records this information, and returns the same store with the addition of a unique identifier. The metadata contains the following items:

Store Name. The store name is supplied by the user. This is not required or guaranteed to be unique in any way, for example a store can be named “CloudDocs project”. If the user does not supply this name, the store is only identifiable using its unique identity. However, it is always recommended that the user supply a store name for their own ease of use, and a client may choose to enforce this. The store name also helps to search and browse the stores when the user has a large number of them.

Access Control List (ACL). This is a list of usernames and their access levels. When the store is created for the first time, none of these have to be set, with the exception that the creating user must be the Owner. The possible values in the access control list are further explained in the next section.

Codename. A unique identifier of the file representing the base of the directory tree in the cloud storage. If the cloud storage has not been initialised, this will be empty.

Initialization Vector. The initialisation vector used when encrypting the base of the directory tree. If the cloud storage has not been set up, this will be empty.

Profile File Name. The name of the file on KMS that holds the IMS and cloud storage credentials. If the cloud storage has not been set up yet, this will be empty.

Setting up the Profile.

The client then needs to set up the store’s profile file, and the directory tree in the cloud. For the profile, the client must generate a username, password, and a public-private key pair to use with the IMS. The public key and password are sent in a POST request (over a secure connection) to IMS, which responds with its public key, which should be added to the profile. The profile also contains the credentials for all cloud providers to be used, and the bucket (also known as a workspace) name. Different clients will support different cloud providers, depending on the availability of suitable libraries.

The protocol describing the creation of a store is described as follows:

1. User → Client : StoreName, ACL
2. Client : IV, Codename, ProfileName
3. Client → KMS : StoreName, ACL, Codename, IV, ProfileName
4. KMS → Client : ProfileID, StoreName, ACL, Codename, IV, ProfileName
5. Client ← IMS : IMS Credentials
6. User → Client : Cloud Credentials
7. Client : ProfileID (IMS Credential, Cloud Credential)
8. Client → KMS : ProfileID, EncKpub(Profile)
3.5 Access Control List

The access control in CloudDocs is provided at the store level. Some cloud services support access control at the individual bucket level, while others only support it at the account level. In order to support heterogeneous cloud service providers, we have kept the access control at the store level. We plan to study fine-grained access control, such as at the file level in our future work.

The access control list is defined as a triplet as follows:

- **User**: refers to a user of CloudDocs, who can upload, share and consume the data stored in CloudDocs.
- **Profile**: is a record of the credentials required to access the data stored in a particular store.
- **Role**: refers to a set of types of CloudDocs user, who have different permissions to access and use the data.

Their relationships are shown in Figure 2. Four roles are defined in CloudDocs: Administrator, Owner, Author and Reader. A Reader can only read the data contents within a given store. An Author can read all data contents, and also add/update/delete data. An Administrator can read and write data, and is also able to grant and remove roles from other users (including Administrator), whilst an Administrator is able to delete the entire store. There is only one Owner per store, and they have the same privileges as an Administrator. The only difference is that their role is permanent; no user (including themselves) may revoke their access rights. This is because it is assumed that they own the cloud account, and are therefore responsible for any costs for using the cloud service.

3.6 Uploading File to the Store

Users can upload a file by sending it to the client (such as by drag-and-drop, or specifying the path). The client then breaks the file into multiple fragments to create a Fragment Map (list of fragments). When creating the fragment map, the client must also create AES-128 keys (Key Map), and initialization vectors (IVs). The fragments are then encrypted, and the keys and IVs are sent to KMS. For each fragment, a SHA-256 hash is calculated, and the client stores this hash with IMS. The client then uploads the encrypted fragments to the cloud, using the credentials stored in the profile. The fragments should be uploaded to all available cloud providers.

The data flow in the steps described above can be summarised as follows:

1. User → Client : File
2. Client : FragmentMap ← Fragment(File)  
   KeyMap ← GenerateKeys(FragmentMap)  
   IVs ← GenerateIVs()  
   EncryptedFragments ← EncryptFragment(FragmentMap, KeyMap)
3. Client → KMS : FragmentMap, KeyMap, ProfileID, oAuth Key
4. KMS → Client : ACK
5. Client : HashedFragmentMap ← Hash(FragmentMap)
6. Client → IMS : HashedFragmentMap, KeyMap, ProfileID
7. IMS → Client : ACK
8. Client → Cloud : EncryptedFragments
9. Cloud → Client : ACK
10. Client → User : ACK

3.7 Getting Access to the Store

Once a user has created a store, they may share that store with any other users. To do this, the user first views the store in the store list interface, and chooses ‘edit’. The list shows the stores that the user has permission to access. The user can only edit those stores they have Administrator or Owner access to. Following this, the user is able to add new users with access levels described above.
When the user has modified the access of others to a store, the client will request from KMS the public keys of all users who have been granted access. The client must then encrypt the profile using each key, using SMIME encryption, and store the file on KMS for the other users to retrieve. The client can then request KMS update the store metadata.

When a user chooses to open a store from the store list, the client requests the store credentials from KMS. These will have been encrypted with the user’s public key. The client then decrypts the profile using the user’s private key. Once in possession of the decrypted profile, the client can access the file system representation on the cloud storage providers.

The above steps in the protocol can be summarised as follows:

1. **User → Client**: StoreListRequest
2. **Client → KMS**: StoreListRequest, oAuth Key
3. **KMS → Client**: StoreList
4. **Client → User**: StoreList
5. **User**: Select store
6. **User → Client**: Modified ACL
7. **Client → KMS**: ProfileID, Modified ACL, oAuth Key
8. **KMS → Client**: Public Keys
9. **Client**: EncryptedProfile ← EncKpub1(Profile), ...., EncKpubn(Profile)
10. **Client → KMS**: EncryptedProfile

### 3.8 ACCESSING FILE FROM THE STORE

The user accesses the files in the store (virtual file system) in two steps as follows:

#### Opening a Store.

The user first signs in to the client as explained above. Once the user is signed in, the user then selects the store and clicks “open” to open the store. The client then sends the profile name of the store to KMS. In return, KMS sends the profile file. The client then decrypts the profile, which contains all vital meta-information about the store with the user’s private key. The protocol is shown below.

1. **User → Client**: OpenStore
2. **Client → KMS**: ProfileID
3. **KMS → Client**: Profile

### 4. Downloading a File.

The user may select a file from the list of files in the store to open it. To support this, the client needs to download the fragments and re-assemble the file. It navigates through the file system to find the codenames for the fragments in question. The client then requests the decryption keys from KMS. Using the credentials from the profile, it downloads the fragments from any one of the storage providers (if they are unavailable from any provider for any reason, the next one should be used). Once the fragments are downloaded, their hashes are calculated and their integrity is verified with IMS.

The steps above can be summarized as follows:

1. **User → Client**: File
2. **Client → Cloud**: FragmentMap
3. **Cloud → Client**: EncryptedFragments
4. **Client → KMS**: oAuth Key, ProfileID, FragmentMap
5. **KMS → Client**: KeyMap
6. **Client → IMS**: oAuth Key, ProfileID, FragmentMap
7. **IMS → Client**: HashedFragmentMap
8. **Client**: FragmentMap ← \{DecryptFragment(EncryptedFragments, KeyMap)\}
9. **Client → User**: File

### 3.9 EXAMPLE SCENARIO: HOW IT WORKS TOGETHER?

We have implemented the CloudDocs application using TrustStore. Figure 3 shows some screenshots from CloudDocs. In the following, we explain a simple scenario to describe the way CloudDocs works.

In this scenario we assume that the user has already purchased storage space on a number of commercial cloud storage services such as Amazon S3 or RackSpace. The user would like to use these services to securely store sensitive data from his/her desktop computer and share with other collaborators.
In order to do so, the user installs CloudDocs to the local machine, runs the application, and registers as a user with KMS. Once the registration is complete, the user can sign in to the system using the login windows as shown in Figure 3 (a).

When the user has successfully signed in to the system, they can access the store management window as shown in Figure 3 (b). The user can create a new store by clicking on “new” and providing credentials to access cloud services supporting this newly created store. Figure 3(b) shows the list of existing stores. Similarly, the user can edit the ACL by clicking on the “edit” button. The user interface for editing the ACL for the selected store is shown in Figure 3 (c). The user can open the store, and browse the files in the store as shown in Figure 3 (d). The appearance of the file browser is similar to a split-view file explorer. It supports standard operations, such as drag-and-drop.

When the user wants to access the stored files, they start CloudDocs, sign in, and select the appropriate store. CloudDocs shows the directory tree of the stored files again. By double-clicking on a file, the user can download and open it.

A user can share the files in the store with another user. For example, three institutes (A, B, C) may be working on a collaborative project that requires them to share some data. While Institute A and B produce the data they wish to share, Institute C only needs to use the data, without modifying it. To support the above data sharing requirement, Institute A can create a store, called “Shared Resources”, and be assigned as the data owner. They can then add Institute B as an author and Institute C as a reader. Later on, if Institute C ends its participation in the project, Institute A can revoke Institute C’s privileges by removing C from access to the
store without need for re-encrypting the shared data in CloudDocs.

The process of decrypting and assembling the fragments using the encryption key is encapsulated in the TrustStore driver/applications, executed in memory at runtime, and transmitted over a secure connection. It is therefore unlikely a user would be able to capture and store the encryption key locally for unauthorized usage later on without building specialized software to do so. A dishonest (or forgetful) user may retain access to any downloaded copies of the data after access has been revoked. This is beyond the scope of this paper and can only be addressed using traditional methods, such as contracts and/or laws.

4. **AUSTRALIAN ACCESS FEDERATION (AAF) AND SECURE DELETION SUPPORT**

This section focuses on the two key features that have been added to the CloudDocs (Wise et al., 2015) including support for Australian Access Federation (AAF) and secure deletion. This is achieved by extending the key management service. We explain these two features in the following sub-sections.

4.1 **AAF Based Single Sign-On**

In single sign-on, the authorization code is obtained by using an authorization server, which in our implementation is offered by AAF, as an intermediary between the client and resource owner. Instead of requesting authorization directly from the resource owner, the client directs the authentication requests to the AAF, which in turn directs authentication responses back to the client with the authorization code. Before obtaining the authorization code, the AAF authenticates the resource owner and obtains authorization. Because the resource owner only authenticates the authorization server, the resource owner’s credentials are never shared with the client. The authorization code based on the AAF approach provides a few important security benefits:

The use of a federated identity allows the user to create only single account. The clients do not need to create multiple service accounts to access different services. In our implementation, CSIRO employee account is used as a single account to access different services such as Galaxy and other services.

The different user authentication mechanisms based on our previous the username/password-based and new AAF-based single sign-on are depicted in Figure 4.

Since the access token is transmitted directly to the client without passing it through the resource owner, any potential risk in exposing it to others including the resource owner is prevented. This narrows down the security attack surface.

The different user authentication mechanisms based on our previous the username/password-based and new AAF-based single sign-on are depicted in Figure 4.

4.2 **INTEGRATION WITH AAF RAPID CONNECT**

In our implementation, we use AAF Rapid Connect to get access to the authentication server provided by AAF. Traditionally an authentication service involves having to install a Shibboleth SP locally and use Security Assertion Markup Language (SAML) to exchange user authentication and authorization data with the Shibboleth identity provider (IdP) server that sits remotely. However, Rapid Connect hides all these complexities by providing an easy to use platform that natively integrates into many commonly used development languages. With the Rapid Connect solution, a service provider can be written with minimal integration code and without having to install a local Shibboleth SP. The service provider sends a simple https POST request to
Rapid Connect, which translates it into SAML assertions, verifies the request with the underlying Shibboleth service, then sends back an authorization code as a JSON Web Token (JWT).

The following is the standard flow of integrating with AAF Rapid Connect.

To start with Rapid Connect, the service (e.g., TrustStore Client) that is accessed via the user needs to be registered with Rapid Connect which asks for the service URL (e.g., where the authentication request comes from) and the callback URL (e.g., where the authentication responses are received) along with a secret that is used between the registering service and Rapid Connect service.

<table>
<thead>
<tr>
<th>Service</th>
<th>URL: <a href="https://tstest-kms.it.csiro.au/kmscolab_3_0">https://tstest-kms.it.csiro.au/kmscolab_3_0</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>Callback</td>
<td>URL: <a href="https://tstest-kms.it.csiro.au/kmscolab_3_0/aaf">https://tstest-kms.it.csiro.au/kmscolab_3_0/aaf</a></td>
</tr>
<tr>
<td>Secret</td>
<td>xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx</td>
</tr>
</tbody>
</table>

Once the service registration is successful, the Rapid Connect provides a unique URL specific to the registered service. This unique URL is presented as a link on the authentication page within the TrustStore GUI Client for the user to click on.

```java
public AAFAuthentication parseJWT(String tokenString) throws AuthenticationException {
    try {
        // decode jwt with the secret used during the registration
        Jwt jwt = JwtHelper.decodeAndVerify(tokenString,
                                             new MacSigner(jwtSecret.getBytes()));

        // gets JWT claims
        String claims = jwt.getClaims();
        ObjectMapper mapper = new ObjectMapper();

        // run a security check (code omitted for brevity purpose):
        (1)The token is sent via a legitimate AAF servers.
        (2)The token has yet expired.
        (3)The token is fresh and is not a replay attack

        // Register the user with the authorization token
        boolean didAdd = registerAAFUser(token.attributes);

        return new AAFAuthentication(token.attributes, token, didAdd);
    } catch (IOException e) {
        throw new AuthenticationServiceException(e.getLocalizedMessage());
    }
}
```

4.3 HOW IT WORKS?

To access AAF-based single sign-on, the user runs an executable jar which starts up a dialogue box, as shown in Figure 5, that directs the user to the authentication page.

```
try {
    // get AAF authorization token
    AAFJWT token = mapper.readValue(claims, AAFJWT.class);

    // run a security check (code omitted for brevity purpose):
    (1)The token is sent via a legitimate AAF servers.
    (2)The token has yet expired.
    (3)The token is fresh and is not a replay attack

    // Register the user with the authorization token
    boolean didAdd = registerAAFUser(token.attributes);

    return new AAFAuthentication(token.attributes, token, true);
} catch (IOException e) {
    throw new AuthenticationServiceException(e.getLocalizedMessage());
}
```

![Figure 5. User authentication methods dialogue box.](image)

The user authenticates to their institution’s Identity Provider via their usual means (e.g., CSIRO employee account). If the user’s IDP is known in advance, it can be specified by appending its entityID to the unique Rapid Connect URL. If no entityID is specified, the user will select their Identity Provider from a list of providers at the AAF Discovery Service. We implemented our service to select CSIRO’s IDP from the list.

The AAF Rapid Connect validates the user’s identity information and generates a unique and signed JWT for TrustStore client using the secret defined during the service registration. The generated JWT is sent via HTTP to the callback endpoint defined during the service registration.

```
try {
    // get AAF authorization token
    AAFJWT token = mapper.readValue(claims, AAFJWT.class);

    // run a security check (code omitted for brevity purpose):
    (1)The token is sent via a legitimate AAF servers.
    (2)The token has yet expired.
    (3)The token is fresh and is not a replay attack

    // Register the user with the authorization token
    boolean didAdd = registerAAFUser(token.attributes);

    return new AAFAuthentication(token.attributes, token, true);
} catch (IOException e) {
    throw new AuthenticationServiceException(e.getLocalizedMessage());
}
```

```
4.3 HOW IT WORKS?

To access AAF-based single sign-on, the user runs an executable jar which starts up a dialogue box, as shown in Figure 5, that directs the user to the authentication web page.

![Figure 5. User authentication methods dialogue box.](image)

The user clicks on “Australian Access Federation” image log which takes the user to the list of authorized identity providers. The user chooses an identity provider where his/her authentication credential would be recognized from the list.

The identity provider then asks the user to provide the details of the authentication credential, such as CSIRO employee account as depicted in Figure 6.
Figure 6. CSIRO identity provider page.

If the identity provider approves the authentication credential, an authorization token is issued to the user as shown in Figure 6. The user can then use this authorization token by copying the code from the web page and pasting it onto the pop-up dialogue box. If the authorization token is approved, the user now can access to the TrustStore to access, use and manage the data.

Figure 7. Authorization token page.

4.4 Secure Deletion

To increase availability of data, cloud services often replicate data in servers located in many geographic places. Though public auditing and verification has been used to check and maintain the integrity of these replicated data, they do not deal with the deletion of the data at termination of cloud use — a common security requirement, especially with sensitive data. NIST has recognised the sanitization of data as one of the key computer security issues and developed guidelines for media sanitization (Kissel et al., 2014). The issue is even more complicated in cloud storage services as the data owner typically does not have a control over the data and associated storage media.

Tang et al. (2010) proposed a policy based file assured deletion technique, called FADE. The basic idea behind the scheme is to decouple the keys and encrypted data. The key management system works independently. In this model, secure deletion is achieved by deleting the keys as the encrypted data becomes irrecoverable without keys. Policies can be attached to the keys. For example, a specific time as in Vanish system (Geambasu et al., 2009) and a policy graph in (Cuchin et al., 2013).

Motivated from integrity checking techniques such as Provable Data Possession (PDP) and Proof of Retrievability (PoR), Paul and Saxena (2010) proposed a technique scheme, called Proof of Erasability (PoE). The PoE based technique compare the hash of the data and still relies on the honesty of the storage service provider. The technique is useful to use along with legally binding Service Level Agreements (SLAs) between data owners and cloud service providers.

Our key management service works in a similar manner to FADE whereby the keys are decoupled from the encrypted data. The policies are defined for the keys. Once the keys are deleted, the data becomes irrecoverable. Furthermore, our integrity management service (IMS) keeps the hash values of data to provide independent auditing of the data stored in the cloud. The hash values stored in IMS can be used to implement PoE.

5. Experimental Results

We have undertaken extensive benchmarks to ascertain the scalability of the system in the context of the NeCTAR funded endocrine genomics virtual laboratory (endoVL) project that has focus on supporting genomic research into a range of disorders (Sinnott et al, 2013). We have used varying files sizes, up to 100GB, and various CloudDocs client implementations. The results are shown in Figure 8 and Figure 9. While there is a significant overhead for small files, for large files CloudDocs performance converges close to the maximum throughput available at the used storage provider (NeCTAR). The overhead for small files is the result of a deliberate design decision, which allows more efficient random access to small-medium sized files. Note, that non-CloudDocs performance results for files larger than 5GB cannot be obtained due to file size restrictions at the cloud storage provider.

Figure 8. Upload Speed Benchmark Results for NeCTAR Cloud
In addition to the NeCTAR research cloud, we have experimented with Amazon cloud in the US. Figure 10 shows the results. The results clearly indicate the variation of time for upload. The variation occurs due to factors beyond the control of TrustStore. Figure 11 and 12 shows the upload and download speeds for various file sizes on private (CSIRO Research Cloud), community (NeCTAR Cloud) and public (Amazon) clouds using Java and Python clients. The results clearly show that the underlying security platform can handle large amounts of data with little overhead.

6. RELATED WORK

What do we mean by data security in the cloud? A number of properties from secure and dependable systems, database management systems and distributed systems have
been used to define security. These properties include confidentiality, integrity, availability, reliability, efficient retrieval, data sharing, data searching, data consistency, safety and liveness (Subashini & Kavitha, 2011).

Many secure cloud storage systems are based on encryption of data that is stored in the cloud, such as in HPISecure (Saleh & Meinel, 2013). Another approach is to separate confidential data from non-confidential data and store them in different cloud services, a trusted cloud and a commodity cloud (e.g. Twin Clouds (Bugiel et al., 2011). A similar approach is taken by the SecCSIE (Seiger et al., 2011), which provides a Cloud storage integration solution at the network perimeter level where the data is encrypted before leaving the local network. The local network is equivalent to the trusted cloud in Twin Clouds. This means that any adversaries within the trusted network can have access to the unencrypted data.

In terms of integrity, third-party auditing is the most commonly used approach to check data integrity. SecCloud (Wei et al., 2010) provides a solution based on auditing where the auditor ensures that the cloud providers behave as expected. The availability of data can be increased by using multiple cloud service providers and replicating data amongst those providers as has been realized in DepSky (Bessani et al., 2013). HAIL is another cloud storage system that supports high availability independent of individual storage services (Bowers et al., 2009).

These systems focus on providing confidentiality, integrity and availability of data stored in cloud storage systems. However, these systems cannot be used in the context of collaboration, where the data needs to be shared with multiple parties. This collaboration aspect is the focus of our paper. There are a number of techniques that can be applied to secure data sharing on the cloud. We briefly capture them as follows:

**Access Control:** The most basic approach to provide controlled sharing of information is to use access control mechanisms. Different types of access control mechanisms have been developed for collaboration. A survey of access control in collaborative environments can be found in (Tolone et al., 2005). These mechanisms cannot be applied directly if the collaboration resources are stored on public clouds.

**Public Key Infrastructure (PKI):** Another approach to data sharing is based on encryption technologies, such as the use of public-private key pairs (Solo et al., 1999). However, this is not scalable in cloud environments. To address this problem, a new breed of approaches combining access control mechanisms and data encryption technologies has been proposed. In the following, we summarize some techniques that have been proposed to combine data security provided through both user authorization and data encryption, that are suitable for cloud environments.

**Identity-Based Encryption (IBE):** Shamir presented a cryptographic scheme to enable any pair of users to communicate securely and to verify each other's signatures without (a) exchanging private or public keys, (b) keeping key directories, or (c) using the services of a third party (Shamir, 1985). This scheme is ideal for sharing information among a closed group of people (e.g., within an organization). The idea is based on the public key cryptosystem, where the public keys are generated using attributes (e.g., company name, IP address, etc.) and individual users have the corresponding private keys.

**Attribute-Based Encryption (ABE):** One of the limitations of earlier IBE scheme is that they use string-based attributes. In order to overcome this limitation, Sahai and Waters presented a Fuzzy IBE, where the attributes can take values from a domain other than strings (Sahai & Waters, 2005). They proposed two applications for Fuzzy IBE: IBE that uses biometric identities, and ABE. In an ABE system, a user’s keys and ciphertexts are labelled with sets of descriptive attributes and a particular key can decrypt a particular ciphertext only if there is a match between the attributes of the ciphertext and the user’s key. Sahai and Waters’ cryptosystem allows for decryption when at least k attributes overlap between a ciphertext and a private key.

While this primitive was shown to be useful for error-tolerant encryption with biometrics, the lack of expressibility limits its applicability to larger systems.

**Key-Policy ABE (KP-ABE):** To address the limitation of ABE, Goyal et al. proposed a much richer type of attribute-based encryption cryptosystem, where each ciphertext is labeled by the person who encrypts it with a set of descriptive attributes (Goyal et al., 2006). The private key is associated with an access structure that defines what types of ciphertexts the key can decrypt, so KP-ABE can support fine-grained access control.

**Ciphertext-policy attribute-based encryption (CP-ABE):** KP-ABE used attributes to describe the encrypted data and built policies into user keys. However these methods are susceptible to collusion attacks. In order to overcome such attacks, Bethercourt et al. proposed a scheme where system attributes are used to describe a user’s credentials, and a party encrypting data determines a policy for who can decrypt (Bethercourt et al., 2007). This makes CP-ABE conceptually closer to traditional access control methods such as Role-Based Access Control (RBAC).

**Proxy Re-Encryption (PRE):** Another common approach used in secure data sharing using cryptography system is the widely known proxy re-encryption approach introduced by Blaze et al. as atomic proxy cryptography (Blaze et al., 1998). The basic idea is that two parties publish a proxy key that allows an untrusted intermediary to convert ciphertexts encrypted for the first party directly into ciphertexts that can be decrypted by the second party. Variations of PRE techniques have been proposed in the literature (Ateniiese et al., 2006; Canetti & Hohenberger, 2007).

**ABE & PRE:** Existing solutions based on ABE and PRE introduce a heavy computation overhead on the data owner for key distribution and hence do not scale well when fine-grained data access control is desired. To address this
problem, a combination of ABE and ARE schemes have been proposed in the literature in order to take the benefits of both schemes: fine-grained access control on encrypted data and scalable user revocation such as (Yu et al., 2010) and TimePre (Liu et al., 2014).

The techniques described above can be categorized as follows: (a) re-encryption, (b) data sharing encryption algorithm, and (c) role-based access control. There are a number of key problems with these approaches. All the data may need to be re-encrypted in order to revoke access for users. A full reliance on PKI infrastructure means there is no protection and management for the private keys. Alternatively there is full reliance on the access control mechanism. In our solution, we provide a unique mechanism for data sharing where the PKI infrastructure is used for data sharing, but the private keys are stored centrally protected with a password. The public key is used to provide role-based access control. In our approach, the key management service acts as a policy protection and enforcement authority. Key features of our scheme can be summarized as follows: (a) the data is encrypted using standard AES symmetric keys so that the computation cost is cheaper than most of the systems/techniques described above that use asymmetric encryption techniques, (b) the policy is defined independent of the data so that the changes in the policy can be easily managed and is scalable, (c) the policies are defined using public keys, but the private keys are protected using user passwords. Hence, without knowing a user password, the data cannot be accessed.

**Fully Homomorphic Encryption (FHE):** This a type of encryption that allows functions to be computed directly on encrypted data. This technique has gained popularity in cloud environments because it avoids costly operations where the data owner has to download the whole data and run the decryption operation before analysing any data. Fully homomorphic encryption allows any computation function (i.e., comprising multiple operations such as multiplication and addition) to be performed over the encrypted data and (encrypted) intermediate results, without the need to decrypt data between the computing steps. Moreover, a single entity in possession of the encryption (e.g., a cloud node) can perform this computation, without the need for interaction with the data owner or other entities. Gentry introduced the first fully homomorphic encryption scheme in 2009 (Gentry, 2009). This was a revolutionary cryptographic achievement, but the scheme was far too inefficient for any practical use especially due its running time. Since 2009, several works have improved Gentry’s technique, significantly reducing the running time. Liu et al developed a new homomorphic encryption with order-preserving indexing schemes that provides a better performance (Liu & Wang, 2013; Liu, 2014; Liu, 2015). With these schemes, the data is encrypted and indexed before they are put in the database. These new schemes also allow direct SQL queries to be conducted over encrypted data in the cloud. The data owner only needs to decrypt the returned query results. At the heart of this approach is a query proxy, which translates SQL queries between the database (client) applications and encrypted data stored in the Cloud. Compared to other similar techniques, this homomorphic encryption approach works on infinite sums of values and ensures the correctness of aggregate query results no matter the size of the data.

Though the processing time has improved by many researchers, homomorphic encryption has other limitations. For instance, homomorphic encryption requires that all recipients have the same key to encrypt the inputs and decrypt the results, which may be difficult to arrange if they belong to different organizations (Liu, 2014). Solutions are required to support multiple organisations that do not necessarily have the same key but still wish to operate on encrypted data collaboratively.

For some applications, using a technique such as homomorphic encryption can be applicable but for others it may not be possible to encrypt the whole data as it severely limits the ability of users to selectively share their encrypted data at a fine-grained level.

**Verifiable Computation (VC):** This is another type of encryption that sparked much interest in dealing with data in cloud by allowing the data owner to check the integrity of the computation. In a verifiable computation scheme, the data owner gives the data along with a specification of the computation to some (usually more powerful) entity (i.e., usually called a prover). The prover then outputs the result of the specified computation, along with a “proof” that the output is in fact correct. The “proof” typically takes the form of one of two cryptographic objects: probabilistically checkable proofs (PCPs) (Sudan, 2009), and succinct non-interactive arguments of knowledge (SNARKs) (Bitansky et al., 2012). The rationale behind verifiable computation is that that it must be easier and less costly to verify the proof than to perform the computation. Recent implementations of VC techniques include Pinocchio (Parno et al., 2013) and SNARKs for C (Bitansky et al., 2012). While these protocols have made proof verification nearly practical, the cost to generate a proof remains a significant barrier to practicality. Indeed, most applications are constrained to small instances, since proof generation costs 3-6 orders of magnitude more than the original computation. More improved techniques that support more flexible and efficient provers have been raised (Fiore et al., 2014; Canetti et al., 2014)

7. **Conclusions**

We have presented an extension to a secure scalable document sharing application using public cloud infrastructure, called CloudDocs, which supports a secure cloud storage system. The application was developed by extending the TrustStore key management service, which is designed to manage only data encryption keys in TrustStore, as follows: (a) it manages users and their secret (i.e., private
key), and (b) it manages the access control over data by defining access control rules. The security of data in the key management service is achieved by protecting the profile with the user secret, and the user’s secret with their password. We have described the protocols and illustrated how the system works. In this paper, we further extended the CloudDocs to support federated access control and secure deletion of data using the key management service and integrity management service.

The current implementation of CloudDocs has a number of limitations that we plan to address in future including, but not limited to: (a) secure password management system including the feasibility of password recovery, (b) fine-grained access control mechanisms so that access control can be defined at the file level rather than the store level, (c) application of proxy re-encryption and attribute-based encryption techniques for managing credentials and access rules stored in the key management service. In future, we plan to implement and validate the proof of erasability (PoE) as a key feature of the integrity management service.

8. ACKNOWLEDGMENT

We wish to acknowledge the NeCTAR project and the research cloud infrastructure that was used here. NeCTAR also funded the endoVL project that provided many of the scenarios and case studies necessitating the secure storage solutions outlined here.

9. REFERENCES


**Authors**

**Dr Surya Nepal** is a Principal Research Scientist at CSIRO Data61. His main research interest is in the development and implementation of technologies in the area of distributed systems and social networks, with a specific focus on security, privacy and trust. He obtained his BE from National Institute of Technology (NIT) Surat, India; ME from Asian Institute of Technology (AIT), Thailand; and PhD from RMIT University, Australia. At CSIRO, he undertook research in the area of multimedia databases, web services and service-oriented architectures, social networks, security, privacy and trust in collaborative environments, cloud systems and big data. He has more than 150 peer-reviewed publications to his credit.

**Dr Carsten Friedrich** is a Research Team Leader in the CSIRO Data61. Carsten received his undergraduate degree in Computer Science from the University of Passau in Germany, and his PhD in Computer Science from the University of Sydney. His PhD thesis won the “Most outstanding PhD thesis in the field of Computer Science in Australasia” award in 2003. In recent years, Carsten has focused his work on building and commercializing research based software for the Capital Markets CRC, various start-up companies, and CSIRO. His current work focuses on Cloud Computing and Cyber Security.
Catherine Wise used to be a CSIRO software engineer working on projects researching Cloud Computing, geographic data aggregation, and web user experience optimisation. She has a Bachelor of Mathematics (Hons) from the University of Wollongong.

Professor Richard O. Sinnott is Director of eResearch at the University of Melbourne and holds a Professorial role in Applied Computer Systems. He was formerly technical director of the National e-Science Centre, UK; director of e-Science at the University of Glasgow. He has a PhD in Distributed Systems; a Master of Science in Software Engineering and a Bachelor of Science in Theoretical Physics (Hons). He has published over 250 peer-reviewed papers in conferences/journals across a wide range of computing science areas with specific focus over the last ten years in supporting communities demanding finer-grained access control (security).

Dr Julian Jang-Jaccard is an Associate Professor at Massey University. Julian received her PhD in database transaction (University of Sydney); a Master of Information Engineering (University of Sydney) and a Bachelor of Computer Science (University of Western Sydney). Her research interests span from database, cloud computing, mobile systems, cybersecurity, and big data analytics with a specific focus on security and privacy. She has many years of industrial strength application development experience and has been trained for an entrepreneurship skill. She has published numerous peer-reviewed papers in respectable conferences and journals across a wide range of computing science and engineering communities.

Dr. Shiping Chen is a principal research scientist at CSIRO Data61. He also holds an adjunct associate professor title with the University of Sydney through teaching and supervising PhD/Master students. He has been working on distributed systems for over 20 years with focus on performance and security. He has published over 100 research papers in these research areas. He is actively involved in computing research community through publications, journal editorships and conference PC services, including WWW, EDOC, ICSOC and IEEE ICWS/SCC/CLOUD. His current research interests include secure data storage & sharing and secure multi-party collaboration. He is a senior member of the IEEE.