Storekeeper: A Security-Enhanced Cloud Storage Aggregation Service

Sancha Pereira, André Alves, Nuno Santos, Ricardo Chaves
INESC-ID / Instituto Superior Técnico, Universidade de Lisboa
Lisbon, Portugal
Email: {*sancha.pereira,andre.pimenta.alves,nuno.m.santos,ricardo.chaves*}*@tecnico.ulisboa.pt*

***Abstract*—Cloud storage services are currently a commodity that allows users to store data persistently, access the data from everywhere, and share it with friends or co-workers. However, due to the proliferation of cloud storage accounts and lack of interoperability between cloud services, managing and sharing cloud-hosted files is a nightmare for many users. To address this problem, specialized cloud aggregator systems emerged that provide users a global view of all files in their accounts and enable file sharing between users from different clouds. Such systems, however, have limited security: not only they fail to provide end-to-end privacy from cloud providers, but they require users to grant full access privileges to individual cloud storage accounts. In this paper, we present Storekeeper, a privacy-preserving cloud aggregation service that enables file sharing on multi-user multi-cloud storage platforms while preserving data confidentiality from cloud providers and from the cloud aggregator service. To provide this property, Storekeeper decentralizes most of the cloud aggregation logic to the client side enabling security sensitive functions to be performed only on the trusted client endpoints. This decentralization brings new challenges related with file update propagation, access control, user authentication, and key management that are addressed by Storekeeper. This is provided at a low cost (7% on average) when compared with the underlining cloud providers.**

I. Introduction

Over the past years, cloud storage has become an invaluable resource for consumers. For many users, cloud services such as Dropbox, Google Drive, or Microsoft OneDrive constitute essential platforms for storing personal documents, photos, videos, etc., and for sharing such files with friends. Moreover, driven by the fact that many companies have limited storage resources, personal cloud accounts tend to be adopted also for professional purposes, e.g. to store work-related files, collaborate with co-workers, or back up data.

However, in the current state-of-affairs, managing and sharing cloud-hosted files can be very cumbersome for users. To save costs, many users use only the amount of free space provided by default per cloud storage account. Since this space is relatively small (tens of gigabytes), in order to accommodate growing storage demands, users end up signing up for multiple cloud accounts in different cloud providers (e.g., Dropbox and Google Drive) or even within the same service under a different username. As a result, users' files tend to become scattered across multiple accounts, making file maintenance and navigation quite involved.

Moreover, given the lack of interoperability between cloud services, sharing files between cloud providers is not directly permitted. For example, a Dropbox user is not allowed to share a file and edit it jointly with a Google Drive user. This limitation is particularly troublesome whenever collaboration is required between co-workers, friends, or family members. As a result, users tend to create cloud accounts on the same cloud service so that they can access common files using the internal file sharing mechanisms of the service. Therefore, even the users that can afford to pay for larger storage space are normally forced to maintain multiple accounts for collaboration purposes.

To address this file management hurdle, a few services have recently emerged to enable cloud storage aggregation. Cloudfogger [1] and Odrive [2], for example, implement a *multi-user multi-cloud* file sharing layer which exposes to the users a unified view of all files located in their individual accounts and enables seamless file sharing across cloud accounts, for example between a Google Drive user and a Dropbox user. However, existing cloud aggregators fail to provide satisfactory security. Although systems like Cloudfogger provide end-to-end file encryption at the client side, in all these systems the cloud aggregator has full access to users' cloud storage accounts. Such privileges are required so that the cloud aggregator can push file updates to cloud-backed storage. As a result, users incur security risks by entrusting access credentials of their cloud storage accounts to the cloud aggregator.

In spite of extensive research on cloud storage security, the problem of enabling secure cloud storage aggregation has been overlooked. Systems like BlueSky [3] focus on securing single-user single-cloud platforms, i.e., a single user account located on a single cloud service. Security is normally achieved by interposing an encryption layer between the user's client and the cloud provider. SPORC [4] focuses also on a single cloud service, but allow for secure file sharing between users (i.e., multi-user single-cloud platforms). Multiple clouds are handled by systems such as DepSky [5] and SCFS [6]. Such systems combine multiple clouds into a conceptually unique cloud called cloud-of-clouds (CoC) in order to maintain replicated file copies within a single administrative domain. A CoC can be used by a single user (or by a set of users) to maintain multiple file replicas in different cloud backends *as if it were a single*

cloud; users are oblivious to the actual file locations. In contrast, in cloud storage aggregators, users do not share a common set of cloud accounts. Instead, each user must always retain the control over its own pool of cloud accounts which means that, as opposed to CoC systems, it is necessary to securely support file sharing across independent cloud-backed stores managed by different users.

This paper presents the design, implementation, and evaluation of Storekeeper, a distributed system that provides a secure cloud aggregation service for multi-user multi-cloud storage platforms. Storekeeper comprises a client application (akin to a Dropbox client) to be installed on the users' computers and a centralized cloud aggregator server. Storekeeper users sign up to the server through the client and register their individual cloud storage accounts from Dropbox or Google Drive. Through a unified file namespace, users can seamlessly browse files across different cloud accounts and share files with other users.

A key technical contribution of Storekeeper is its novel design, which provides end-to-end data confidentiality without entrusting sensitive user account credentials to the cloud aggregator server. The security-sensitive logic is displaced to the clients' trusted endpoints so that the server has no writing privileges on any of the users' cloud service accounts.

When compared with a typical centralized cloud aggregator, Storekeeper's decentralized architecture introduces new challenges, stemming mostly from the lack of interoperability and heterogeneity of cloud services. Storekeeper includes mechanisms to handle user identities from different cloud providers, and provide key management and storage solutions that are easy to use by the users. Secondly, our system includes adequate security model and enforcement mechanisms that masks the diversity of file permission models across cloud providers. Thirdly, Storekeeper enables secure read and write operations to shared files located in different cloud accounts. To this end, Storekeeper incorporates techniques to securely propagate file updates without introducing privilege escalation vulnerabilities against potentially malicious users.

From the performed evaluation, we found that Storekeeper adds little performance overheads (6 to 10%), which are mostly due to inevitable system interactions or file encryption operations. We envision Storekeeper's cloud aggregator to be deployed internally by companies in order to enable file sharing between their employees without considerable investment.

II. Design

A. Goals, Threat Model, and Design Principles

Goals: Our central goal is to design a secure cloud storage aggregation service. A user is expected to register pre-existing cloud accounts in the service, and the service will provide a unified view of the user's files stored in those cloud accounts. A user must be allowed to share files with other Storekeeper users, and possibly revoke access to them in the future. File sharing must not require that users have individual accounts on the same cloud service. The aggregation service must not require privileged access to users' cloud accounts, and provide end-to-end data confidentiality. Note that, in this work, we are not focusing on preserving integrity or availability of files.

Threat model: We consider cloud providers to be potentially malicious with respect to violation of data confidentiality, but are required to provide data storage resources and enable access to that same data. The cloud aggregator (and its administrators) is assumed to be *honest but curious*, meaning that it can passively listen to all exchanged messages and try to learn sensitive information, but follow the system protocols and do not launch active attacks. Note that we do not prevent intentional modification of data or meta-data by the cloud aggregator or cloud providers that may lead to service unavailability or to lack of data integrity. Nevertheless, we assume that the communication channels are insecure. They can be actively eavesdropped or manipulated by external malicious agents. We do not consider side-channel attacks or social engineering attacks (e.g., to obtain users' passwords), and assume that cryptographic algorithms are sound.

Design principles: We follow three main design principles: *P1* and *P2* ensure cloud space isolation between users; *P3* preserves usability.

- *P1: Users must be restricted to write on their accounts only.* By ensuring that users can only store content in their own accounts, we prevent abuses from other users willing to "free ride" on the cloud storage space owned by other users.
- *P2: Users must have their files physically located on their accounts.* In a cloud storage aggregation system, cloud accounts must be managed independently per user. Therefore, it is important to ensure that users keep control of their files, even if they share it with others.
- *P3: Users must not need to maintain persistent state at the client side.* By keeping all persistent state on the server side, the service will be more convenient for users and more robust to data loss than if the user is required, for example, to manage private keys.

B. System Overview

This section presents an overview of Storekeeper, a security-enhanced cloud storage aggregator that follows the design principles stated above. Storekeeper consists of two main components: a client application and the Storekeeper Directory Server (SDS). The client is an application that runs on the users' devices and serves as an interface to the system. Similarly to the Dropbox client application, the Storekeeper client maintains a local cache of the user files persistently stored on cloud-backed *stores*. Stores represent

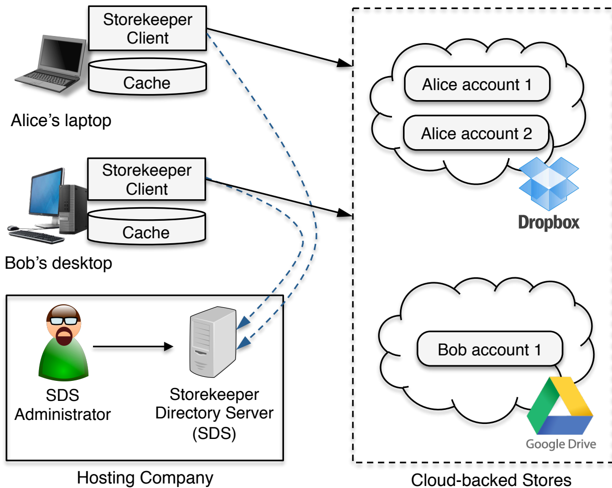

Figure 1. System overview.

cloud accounts hosted by cloud services such as Dropbox or Google Drive. These cloud accounts are provided by the user. The SDS is the heart of Storekeeper. This component runs on a dedicated server and manages the meta-data associated with users, files, and stores. Files themselves are not stored in the SDS, but on stores provided by users.

Figure 1 illustrates the architecture of the system using a simple deployment scenario. Alice and Bob are faculty members of a university, which runs Storekeeper in its premises in order to foster internal collaboration. A dedicated administrator is responsible for managing the SDS server and registering users in the system. Alice and Bob can log into the system using a username and a secret password, and register their personal cloud accounts. In this usage scenario, Alice has two accounts in Dropbox and Bob has one account in Google Drive. Storekeeper will interpret them as stores allowing Alice to see a unified view of files in accounts 1 and 2, and Bob to see all files from account 1. This unified view, seen by each user, is named *workspace*. Each user can thus share files with each other, independently of whether or not they have accounts on the same cloud provider. Access to cloud stores is performed at the client side only, ensuring that users retain exclusive control of their accounts. Files are encrypted at the client endpoint.

C. A Global File Namespace

We now describe the design of Storekeeper in more detail starting with its file namespace. In a cloud aggregator service like Storekeeper, it is necessary to define how files physically dispersed across various cloud-backed stores are presented to the user under a uniform file naming scheme. Figure 2 helps illustrate the file naming organization of Storekeeper considering the usage scenario introduced in Section II-B. It represents (1) aggregated cloud files (workspace) seen by Alice and Bob mounted on the local filesystem, on the left, (2) the actual location of these files on the users' cloud accounts (stores) on the right, and (3) mapping between

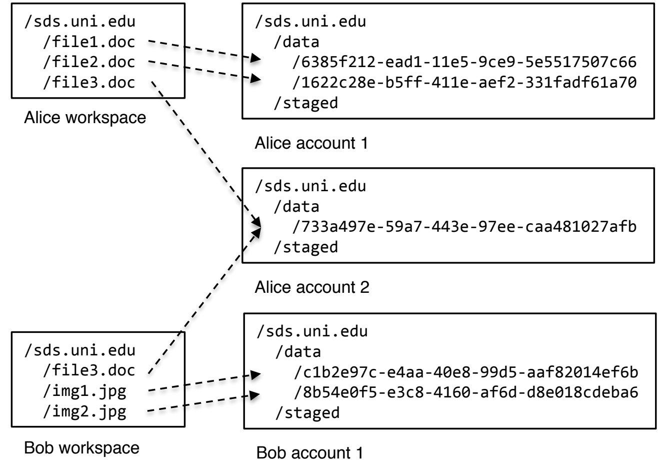

Figure 2. File name mapping in Storekeeper.

workspace file names and file locations, represented by the arrows.

Since there can be multiple independent instances of the Storekeeper service, every SDS server defines a unique *domain name* to avoid name collision. The domain name is set up by the SDS administrator and may correspond to the DNS name of the SDS server (e.g., sds.uni.edu). The Storekeeper domain name will give the name to both (1) a root folder on the user's workspace under which the user files of that domain will be mounted, and (2) a root folder on each cloud store in which user files and pending updates are persistently saved. Files in the cloud stores are given globally unique *file identifiers* (FIDs) and accessible to clients via a service-dependent URL (the arrows in Figure 2). If a file is shared with a user whose workspace contains a file with a similar name, Storekeeper resolves this naming conflict by adding the prefix "shared-" to the shared file.

D. Securing User Credentials

Storekeeper depends on specific user credentials that must be properly secured: access tokens and user keys.

a. Access tokens: In order to access a cloud store through the respective API, a typical third-party application (including the Storekeeper client) needs to authenticate itself towards the cloud service by providing a specific credential named *access token* (AT). Access tokens preclude the need for the user to interactively input username and password. Since access tokens allow for unrestricted access to users' stores, Storekeeper needs to maintain them securely.

b. User keys: To provide end-to-end confidentiality, files must be encrypted at the client side with a symmetric key – a *file encryption key* (KF) – before storing them on the cloud. For this, a symmetric *file encryption key* (KF) is used, as further discussed in Sections II-G and II-H. The considered approach to protect this key, while assuring that the file owner alone can access it, is to use a public-key pair that represents a *user key* (KU) and use this key to

encrypt KF (KF_{KU^+}) (or actually the read key (KR), as discussed in Section II-H). By ensuring that the user's private key (KU^-) remains private, access to KF is restricted.

To avoid burdening the user with the responsibility to maintain user credentials, Storekeeper allows to store access tokens and user key in encrypted form using a symmetric *access key* (KA). Encrypted credentials are stored at the SDS as a tuple that contains: username u, encrypted user's private key, and list of encrypted access tokens AT_s, one for each store s (from 0 to n) that the user has registered in the server, resulting in:

$$u,\ \{u, KU^-\}_{KA},\ [\{u, AT_{s0}\}_{KA}, ..., \{u, AT_{sn}\}_{KA}]$$

To properly protect these credentials, KA must be a cryptographically secure password (or key) and known only to the user. Access tokens are encrypted and sent to the SDS whenever the user adds a new cloud store. For added security, the user may store this tuple at another location of his choice. To access these credentials, the user must first authenticate himself towards the SDS by having username and password established with the SDS for a secure login. With this method, users only need to know (i.e. memorize) username, login password, and access password (or key).

E. Consistency Semantics

To ensure that users retain control of their files (principle P2), files have a fixed location on a cloud store that belongs to the owner of the file. We name *home store* the cloud store where a given file is located. As writes are performed to clients' local file copies, inconsistencies may arise between local copies and the primary replica in home store.

To resolve inconsistencies, we adopt *eventual consistency*. Eventual consistency favours file availability and is already familiar to users, being adopted by Dropbox, Google Drive, and other cloud services. To implement eventual consistency, each file has a *home version number* (v_h) which is centrally tracked by the SDS. For each local replica, the client keeps track of (1) a file's *local version number* (v_l), which corresponds to v_h of the downloaded version, and (2) a *dirty flag*, which is set to 1 if the user modifies the file. Local file modifications are put in a queue to be propagated to the SDS. Periodically, clients synchronize the local cache with home stores. In case of a write conflict, the local file copy is put in quarantine in a special directory to be manually resolved by the user, such as in Dropbox and other cloud services. From a user's perspective, Storekeeper implements a *read-your-own-writes*, i.e., local writes performed by the user are visible by local reads.

F. Access Permissions

Existing cloud storage services allow file sharing, but normally implement different access permission restrictions for internal and for external users: internal users have local accounts in the cloud service, external users have not. External users have access to files via URL and tend to have less privileges than internal users. For example, in Dropbox and Google Docs, external users can access individual files or directories with read permissions only; writes are not allowed. Specificities in access permission semantics can also be observed across cloud services. In Dropbox, for instance, internal users can share files with read permissions, but write permissions can be granted to directories only. Google Drive, on the other hand, allows internal users to share files or directories in both read-only or write modes. In Storekeeper, we need to accommodate this heterogeneity. Since directories can be seen as special files, we focus primarily on file access permissions.

Operation	**R-Permission**	**W-Permission**	**S-Permission**
read	yes	yes	yes
create	no	yes	yes
update	no	yes	yes
delete	no	yes	yes
chperm	no	no	yes

Table I
STOREKEEPER'S ACCESS CONTROL MATRIX.

In Storekeeper, every file has a file *owner*. The file owner has full access privileges over a file, which include: read, write (i.e., modify or delete), and changing access permissions. Modifying access permissions entails granting or revoking access privileges to a certain Storekeeper user – the *grantee*. A grantee can be given one of three *access permissions*: *read* (R), *write* (W), and *share* (S). These permissions are cumulative: R allows a grantee to read a file, W allows to read and write a file (i.e., modify or delete it), and S allows to read, write, and share a file with other users. Note, however, that the privileges of a file owner can never be restricted. Internally, Storekeeper will use ACLs to enforce these permissions and mask the underlying mechanisms of each cloud service.

Table I shows, on each row, the file operations supported by Storekeeper and the respective authorization result yielded to a grantee based on his access permissions (columns 2-4). File read operations are allowed under R, W, or S permissions. Creating, updating, or deleting a file are permitted with W or S permissions. Changing file permissions (chperm) is allowed with S permission only. Next sections present the algorithms for file operations.

G. Basic Algorithm for File Operations

To describe the Storekeeper file operations, we start by presenting a simple algorithm that illustrates the basic workings of read and write operations and then revisit it in the next sections to address emerging challenges. Read and write operations are the most fundamental ones. For simplicity, consider that write operations comprise: create, update, and delete (see Table I). To guide our explanation, we take the use case depicted in Figure 2 in which Alice and Bob are Storekeeper users in domain sds.uni.edu and

share file file3.doc (f). Alice is the owner of this file and Bob has W privileges (i.e., he can read and write the file).

Our goal is to allow Alice and Bob to read / write file f while providing end-to-end data confidentiality. To achieve this property, when Alice creates the file, the local client must first encrypt the file with a file key (KF) and then uploads the resulting ciphertext to the file's home store. The file key is randomly generated and is specific to that file. To protect the file key, the client encrypts KF with the public part of Alice's user key (KU_A^+) and send it to the SDS. To read the file in the future, Alice's client downloads the encrypted file from the home store, and fetches from the SDS the encrypted credentials: KF and private part of Alice's user key (KU_A^-). Next, based on the access key (KA), decrypts KU_A^-, which in turn uses to decrypt the file key KF. From KF, the client decrypts the file contents. Writes can be performed by re-encrypting the new file version with KF and uploading it onto the home store. Since both file and file keys are encrypted, neither SDS nor cloud provider can read the file contents.

To allow Bob to read or write the file, the file key KF can be securely shared with Bob, which can be done by encrypting KF with the public part of Bob's user key (KU_B^+). When Alice, the owner of the file, grants W permissions to Bob, the SDS not only updates the file's ACL with W, but also receives from Alice's client a file access credential: $\{KF\}_{KU_B^+}$. When Bob reads / writes this file, the SDS forwards this credential to Bob's client, allowing KF to be decrypted and the operation to proceed.

H. Permission Enforcement and Revocation

Permission enforcement is based on a combination of access control checks performed at the SDS and cryptographic protocols. Considering the basic scenario described in the section above, the SDS data structures that control access to file f (file3.doc) can be represented by the tuple P_{f_0}:

$$P_{f_0} : (\mathsf{alice}, \{KF\}_{KU_A^+}), [(\mathsf{bob}, W, \{KF\}_{KU_B^+})]$$

P_{f_0} implements an ACL in which Alice is the owner of the file, and Bob has writing privileges to the file. In order to decrypt the file for read operations and re-encrypt the file for writes, the file key KF is encrypted with the public part of their respective user keys (see Section II-D). P_{f_0} is verified every time a user performs a file operation to f; without adequate permissions, the operation is refused by the SDS. Nevertheless, even if the SDS accidentally sends a file to a wrong user, that user will not be able to decrypt it to read.

However, this simple approach is insecure when revoking reading privileges. Since Bob holds KF and file URL, it will still be possible to download and decrypt the file after Bob is excluded from the ACL. A commonly used approach to handle revocation is to re-encrypt the file whenever the revocation operation takes place using a new file key. The downside of this approach is the performance overhead introduced by re-encryption.

To avoid file re-encryption, we employ three techniques, enumerated below. To illustrate each step, we show the new tuple state when: both Alice and Bob belong to the readset (P'_{f_1}), Bob's permissions were revoked (P''_{f_1}), and Alice submits new update (P'''_{f_1}). Changes are underlined.

a. Readers have access to a read key: Instead of granting readers direct access to the file key KF, they are given access to an intermediate symmetric key that is shared between authorized readers. This key – named *read key* (KR) – is then encrypted with the readers' public key and added into the ACL. The read key is then used for encrypting the file key KF, which effectively encrypts the file contents.

$$P'_{f_1} : \underline{\{KF\}_{KR}}, (\mathsf{alice}, \{\underline{KR}\}_{KU_A^+}), [(\mathsf{bob}, W, \{\underline{KR}\}_{KU_B^+})]$$

b. Revocation generates a new read key: Every time revocation occurs, a new read key KR' is generated and the ACL updated such that the new read set has access to the new read key. This means that KR' must be encrypted with the readers' public keys and the ACL updated accordingly.

$$P''_{f_1} : \{KF\}_{\underline{KR'}}, (\mathsf{alice}, \{\underline{KR'}\}_{KU_A^+}), []$$

c. Every write generates new file key: Every time a writer submits a file update, instead of encrypting the file with the same file key KF, the writer generates a new file key KF', encrypts the file with KF', and replaces the old KF with the new KF', which must be encrypted with the read key so that readers can continue reading the file.

$$P'''_{f_1} : \{\underline{KF'}\}_{KR'}, (\mathsf{alice}, \{KR'\}_{KU_A^+}), []$$

With this method, a revoked reader will not be able to see any future writes, since the file key and read key will become inaccessible to the revoked user. As a result, revocation is achieved without re-encrypting the file.

I. Staging Space

The basic algorithm described in Section II-G skips another important question which is how to enable clients to retrieve or update files homed on cloud stores of other users. As mentioned previously (Section II-F), existing cloud services put important restrictions in the way files can be shared. For example, external users are allowed to read files via a URL, but cannot perform writes to such files. In some other cases, internal users are restricted to sharing directories in write-mode, not single files. Our goal is therefore to provide transparent interoperability between cloud stores such that files can be seamlessly shared at a file-level granularity while ensuring isolation between user accounts.

To achieve this goal, our approach is to entirely prevent direct *cross-writes*, i.e., writes on other users' stores; only *cross-reads* will be allowed. Figure 3 illustrates these concepts for a simple scenario. Bob's client will be able

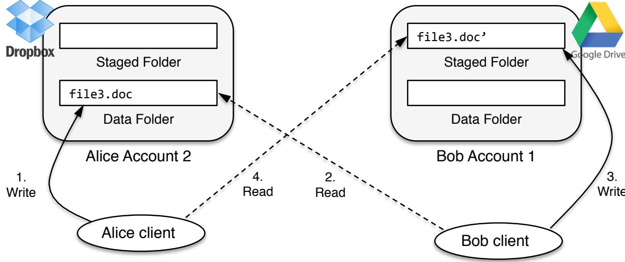

Figure 3. Example scenario to illustrate the use of the staging space. Read / write operations are represented in dashed / solid lines. First, Alice client writes the file in her account (1). Then, Bob fetches the file from the file's home account using a read-only URL (2). Bob updates the file and places the file temporarily in the staging space (3). Finally, Alice retrieves the updated file version from Bob's staging space using a read-only URL (4).

to read Alice's file based on an external URL. Whenever Bob has pending writes, rather than submitting them to Alice's account via a direct URL, the new file version must temporarily be placed in a reserved area which we call *staging space*. Staging space is nothing but a dedicated folder in Bob's cloud stores where pending updates to foreign files can be temporarily saved (see Figure 2). To let Alice access the new file version, Bob's staged file is set to be readable and a corresponding read-only URL is given to Alice so that the file can be pulled from Bob's account.

With staging, however, a few negative side effects emerge:

a. Dangling pointers: A file is always expected to be found via a URL. With staging, the URL must be updated to the file's new location: in Figure 3, after step 3 the file will reside in Bob's staging space. However, if Bob removes his account from Storekeeper or deletes the staged file from his account, the URL becomes invalid (a dangling pointer) and the file inaccessible.

b. Lost updates: If Bob removes the cloud store, the file updates performed by Bob will be lost because updates were staged in a Bob's store that is no longer available.

c. Free riding: File staging can also lead to situations of potential abuse between users. For example, suppose that the latest file version is hosted in Bob's staged space. If Alice revokes Bob's permissions to access the file, Bob can no longer access the file but the file will still be provisioned by Bob. As a result, part of Bob's storage space will be consumed by someone else without bringing direct benefits to Bob. Alice would be "free riding" on Bob's storage space.

J. File Homing

To overcome the negative side effects of staging, Storekeeper uses a technique which essentially consists of relocating staged files back to their home stores; we call it *file homing*. Taking Figure 3 as example, file homing (a) copies the file from Bob's staged space back to Alice's, and (b) updates the file URL in the SDS to point to the home store file version. Thus, if Bob's cloud store becomes inaccessible, file homing prevents dangling pointers and lost updates because the file has been relocated to Alice's store. By the same reasoning, if Bob's permissions are revoked, free riding is prevented because the file is now served from Alice's own storage capacity.

However, there are practical challenges to implementing this technique. File homing should ideally be executed by the time Bob submits his file update. Since only Alice's client has write access to her accounts, chances are that when Alice's client triggers file homing, Bob's store has been removed from the system. As a result, the file URL in SDS would be invalid (dangling pointer) and the submitted update lost. To handle this we adopt three techniques:

a. Increase frequency of file homing events. To avoid consuming too many resources at the client, staging checks are piggybacked in periodic interactions with the SDS that the client already performs in two occasions: when refreshing the local cache, and when submitting writes.

b. Serve stale version from home store. If the staged store becomes available a (stale) home version can be returned instead. Rather than maintaining a single file URL and version number pair, the SDS will now maintain two pairs: one for the home store and other for the staged store. If the staged store is removed before file homing takes place, the home version will be returned instead. This technique does not entirely prevent lost updates, but limits them to the updates submitted since the last file homing event.

c. Perform garbage collection. As a result of staging and file homing, replicas will be left on staging directories, taking up precious storage space. To claim such space, Storekeeper has a simple garbage collection mechanism which periodically deletes stale files from staging folders.

K. Scalability and Fault Tolerance

A potential bottleneck to the scalability of the system can be caused by the centralized nature of Storekeeper's SDS. The throughput of the system in terms of number of requests served per unit of time will be limited by the capacity of the SDS server. Similarly, the maximum of users, cloud stores, and files will be bound by the memory and disk available on the server to store the meta-data state of the SDS.

As for fault tolerance, the SDS constitutes the most critical component in the system. SDS failures may affect service availability and data durability. To improve availability, it is possible to deploy multiple SDS servers operating in master-slave configuration so that a slave server can gracefully replace a master server in case of failure. To assure SDS meta-data durability (e.g., upon HW failures), replication can be implemented using RAID, backups, or both.

III. Implementation

We implemented the Storekeeper system, which consists of two components: the *Storekeeper Daemon* (SD) and the *Storekeeper Directory Server* (SDS). The SD is a standalone

process running on the user's device and is responsible for the implementation of the client-side logic. Written in Java, the SD provides an API to a local command line tool that allows the user to input management commands to Storekeeper (e.g., add a new account, login to the service, etc.), and a periodic monitoring service that synchronizes the local workspace with the cloud-backed stores. To interact with the cloud stores, the SD uses service-specific API libraries. Currently, the SD supports integration with two cloud services: Dropbox (API version 1) and Google Drive (API version 2). SD's modular design allows for easy support of additional cloud services.

The SDS is also implemented in Java and receives commands from the SD. Communication between SD and SDS takes place over SSL. By using SSL, we ensure that the SDS is properly authenticated, which is important for users to validate the identity of the Storekeeper domain provider. SSL also provides integrity- and confidentiality-protected message exchage capability between SD and SDS. Over SSL channels, SD and SDS exchange protocol-specific commands encoded in JSON. At the SDS side, persistence of meta-data is achieved by serializing it into local XML files. The SDS provides a command line interface that allows the local administrator to manage users of the system. For space constraints we omit further details about the SDS internal data structures and distributed protocols.

To perform cryptographic operations, we used the Java library provided by the JCA framework. We use AES cipher and generate 256-bit symmetric encryption keys randomly. For asymmetric encryption, we use 1024-bit RSA keys randomly generated. SHA1 is used for hashing.

IV. Evaluation

This section presents the performance evaluation of Storekeeper. This evaluation was accomplished using micro-benchmarks to measure the performance of each of its core operations, namely: read, write, delete, file sharing, and file access revocation. To illustrate the multi-cloud support of Storekeeper, tests are also performed using two distinct cloud providers and multiple accounts from each one.

A. Methodology

Experiments were performed using Amazon EC2 virtual machines for hosting the SDS and clients. The SDS was deployed on a standalone virtual machine. Each virtual machine has a 1-core Intel Xeon Family 2.5 GHz machine with 1GiB memory and 8 GiB SSD, and runs Ubuntu Server 14.04 LTS. For the supporting cloud storage, we used standard accounts from Google Drive and Dropbox.

To evaluate the impact of the data length in the data dependent operations (read, write, and delete), metrics were obtained for varying file sizes, particularly: 1KB, 10KB, 100KB, 1MB, 10MB, and 100MB. For operations whose performance varies according to the amount of users sharing a given file (file sharing and file access revocation), experimental results were obtained for a variable amount of users, varying between 1 and 100 users. These results were obtained between 15th and 20th of April, 2016. The presented figures illustrate the average value of the execution of each micro-kernel 100 times. To depict their statistical relevance the standard deviation is also depicted.

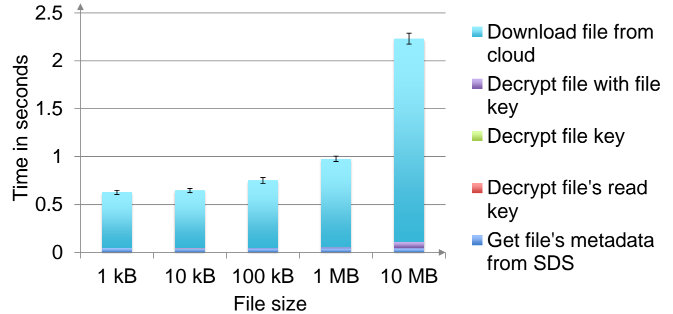

Figure 4. Read performance using GoogleDrive.

The following starts by illustrating the performance and impact of the proposed solution for the supported operations considering Google Drive as the cloud provider. To conclude, comparative results for Storekeeper using Google Drive and Dropbox are presented. This will illustrate the performance variations according to the provider, the operation, and the file size.

B. Performance of Read

Typically, the most frequently used operation in this type of systems is the file read operation. In Storekeeper, for each file read, the total execution time (client side) can be broken down into the time taken for each of the five sub-steps of the read operation, particularly: (1) retrieve the file's metadata from the SDS; (2) using the users private key, to obtain the file key; (3) obtain the file key using the deciphered read key; (4) download the file from the cloud storage provider; and finally (5) decipher the obtained file. Since the performance of reads depends on the file size, time results were obtained for different file sizes. Note that, files $\geq$ 100MB cannot be directly read from Google Drive using the interface herein considered. As such, these are herein not depicted.

From the results depicted on Figure 4, it can be seen that, as expected, the larger the file the longer it takes to download. However, this increase is not linear, since the download of a 10MB file requires 2.2s (achieving a throughput of 4.4MB/s) while for a 1kB file it requires 0,6s (achieving a throughput of 1.6kB/s). Meaning that the download of 10MB files is 2800 times higher.

The Storekeeper system imposes a constant delay overhead, caused by sub-steps (1), (2), and (3). The bulk of the delay is imposed by the actual file download from the cloud storage provider. As seen from these results, the file decryption also imposes a varying delay in the operation, but is between less than 0.1% to 3% of the total time, for

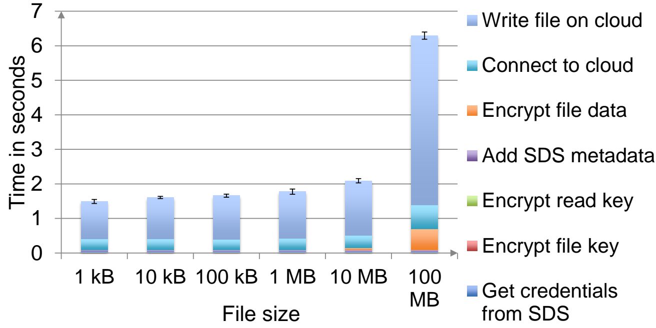

Figure 5. File create performance using GoogleDrive.

1kB to 10MB files, respectively. Actually, apart from the actual file decryption, the cost imposed by Storekeeper is independent from the file size, requiring 50ms. Most of this time is imposed by the reading of the meta data from the SDS. In the worst case, namely for very small files such as the 1kB file, the overhead imposed by the proposed system is of 8%. For larger files, equal or above 100kB, this overhead slightly reduces to 7%.

C. Performance of Write

The other common operation is file writing. In Storekeeper, the write operation is used for both creating a new file and updating an existing file.

Identically to reads, the write operation, in particular file create, is divided into several sub-steps: (1) get credentials from SDS, (2) encrypt the file key with the read key, (3) encrypt the file with file key, (4) encrypt the read key with the user's public-key, (5) connect to cloud storage provider, (6) upload the file to the cloud storage provider, and (7) add a new file entry on SDS. Once more we start by analyzing the performance of this operation, for files varying from 1kB to 100MB. The results for file creation are depicted in Figure 5.

Identically to the read operation, the write operation delay depends on the file size. For smaller files, between 1kB and 10kB, the proportional overhead of Storekeeper is also in the order of 7%, mostly imposed by the interaction with the SDS. Each interaction with the SDS (step 1 and 7), to complete the write operation, takes about 50ms, in a total average time of 100ms. Identical results are obtained for 1MB and 10MB files, but the encryption of the file itself starts to become more significant with a cost between 0.5% to 3% (8.5ms and 63ms) , respectively of the total operation time. For significantly larger files (100MB), the cost of interacting with the SDS becomes negligible, but the cost of encrypting the file becomes more relevant, about 10% of the total operation time. Overall, for files this big, the proposed solution imposes a cost of 10%, caused mostly by the file encryption. For a 10MB, the write operation requires 1.6s (achieving a throughput of 5.2MB/s) when using Google Drive directly, against the 2s (4.8MB/s) when using Storekeeper, thus being 8% slower when using Storekeeper. When considering 1kB files, a write delay of 1.5s is imposed against 1.4s when just using Google Drive, being the proposed system 7% slower. This added delay is mostly due to the communication with the SDS, when considering small files; and due to the encryption of the file, for bigger files. For middle size files (of 1MB) the proposed system imposes a 6% additional delay.

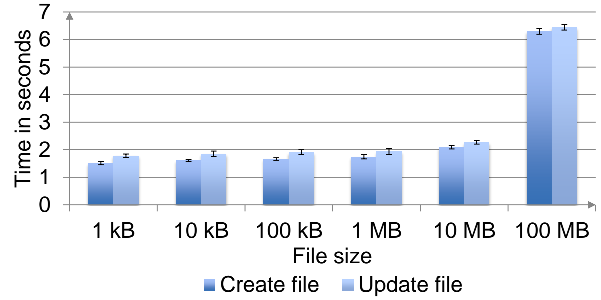

Figure 6. File create and update times in Google Drive.

Steps	Values [ms]	%
Get cloud credentials from SDS	48	5
Remove file's metadata from SDS	50	5
Delete file locally	0,9	0,1
Connect to cloud	290	30
Delete file from cloud	560	59
Total	947	100

Table II
EXECUTION TIMES IN MILLISECONDS OF A DELETE OPERATION FOR A 1KB FILE STORED ON GOOGLE DRIVE.

The above results depict the create operation. While very similar, the update operation is slightly different, requiring an additional step, namely to interact with the SDS to lookup the existing file's metadata in particular to obtain its read key. This would imply an extra cost of 50 ms to interact with the SDS. However, the obtained results, depicted in Figure 6, suggest that the update operation has higher cost (between 267ms to 150ms). This larger delay is due to the data write in the cloud backend. We suspect that this is due to an additional cost in the provider itself when updating a file, since it also has to look for the existing file, which does not happen on the creation of the new file.

D. Performance of Delete

Another important operation to evaluate is delete. To delete a file, both remotely and locally, the following sub-steps are needed: (1) get credentials from SDS; (2) request the SDS to delete the file's metadata; (3) delete the file locally, (4) connect to cloud provider; (5) delete the file on cloud provider. To evaluate the performance of the file delete operation, which should not significantly depend on the file size, we first evaluated the delay imposed for a 1KB file.

The obtained results, depicted in Table II, suggest that the largest performance impact is imposed by the interaction with the cloud storage provider, imposing 87% of the delay, about 1s. The two interactions with the SDS requires about 100ms (50ms + 50ms) on average. For larger files the delay increase in this operation is minimum and imposed by the

Steps	Values (ms)	%
Get the file's read key from SDS	57	33
Get the user's public key from SDS	53	31
Decrypt file's read key	1,3	0.7
Encrypt file's read key with user's public key	0,8	0.4
Add user to file's ACL on SDS	61	35
Total	173	100

Table III
EXECUTION TIME OF THE SHARE OPERATION BROKEN UP INTO ITS SUBCOMPONENTS (IN MILLISECONDS).

increase delay of the Delete file from cloud and Delete file locally operations. For 10MB and 1MB files an average of 0.991s and 1.024s are required, respectively.

E. Performance of Share

In Storekeeper, access control to a file is made by encrypting the file and controlling with whom this key is shared. To share a file with another user, the share operation is used, composed of the sub-steps: (1) get the file's read key from SDS; (2) get the public key of the targeted user from the SDS; (3) decipher the file's read key (with user's private key); (4) cipher the read key with targeted users public key; and (5) add that user to the file's ACL on the SDS.

The detailed results obtained for this operation are depicted in Table III. These results suggest a average time of about 170ms. This delay is imposed by the 3 accesses to the SDS. Note that, the SDS interaction of step 5 takes a bit more than the typical SDS interaction. This can be justified by the fact that in this step the user's (actually the file read key encrypted with its public key) is added to the file's ACL which varies in size depending on the users in this list. These results are completely independent of the cloud storage provider.

F. Performance of Revoke

The performance of revoke depends on the size of the file's ACL. To study the performance evolution of the revoke operation we executed micro-benchmarks varying the ACL size of a file up to 100 users. Figure 7 presents the results of this experiment. The x-axis shows how many users remain with access to the file after revoking the access rights of a given user. We can see that the minimum execution time of revoke is 168ms and it happens when the final ACL length is 1, which means that only the file owner can access the file. Just like the share operation, this time is dominated by the SDS accesses, which take 98,9% of the total time.

As the ACL length becomes larger, the total execution time of revoke is more or less constant between two discontinuities, one around 60 and 65 users and the other around 95 and 100 users. Starting with discontinuity 1, based on the values there is a difference of 40ms between ACL's size, that corresponds to an increase of 23%. This increase is caused by the third step of obtaining the public keys of all users that remain in the ACL from SDS, such step increases 93% varying from 41,9ms to 81ms. The increase in the third step

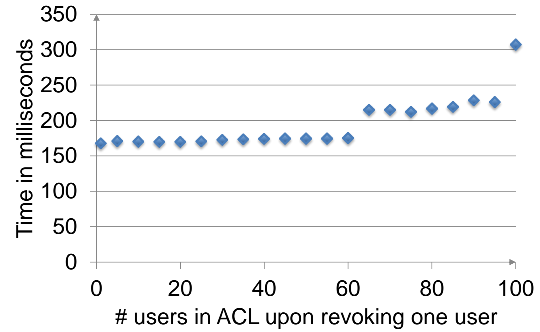

Figure 7. Evolution of revoke execution time with increasing ACL size.

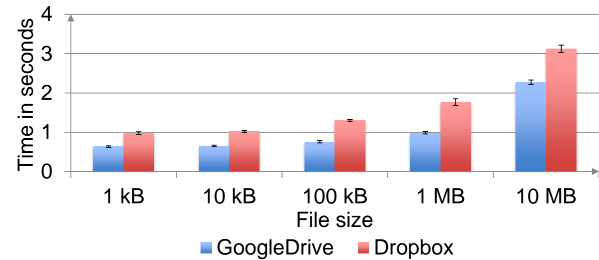

Figure 8. Performance of reads in different clouds.

appears to be due to the addition of one more public key in the SDS's response that forces the SSL socket to perform another round-trip to the SDS. Looking at discontinuity 2, there is a difference of 64ms between 99 and 100 users, which corresponds to an increase of 26%. This increase results from two contributions: first by an increase of 119% while ciphering the new read key with each retrieved public key, and second by an increase of 37% in sending the new read key and updated ACL to the SDS.

G. Multi-cloud Performance

In order to evaluate the achievable metrics, and to demonstrate the support of Storekeper to multiple cloud providers, experimental results were also obtained using Dropbox.

Figure 8 depicts, side by side, the read performance considering both these cloud providers. These results show that the read delay is consistently higher when using Dropbox. Note that the delay imposed by the proposed solution is independent from the cloud provider.

As for the write performance, we compare the file update time using Dropbox and Google Drive. In our tests, we execute the update operation rather create, since updates are more frequent and hence more representative of the performance experienced by Storekeeper's users.

The obtained results, depicted in Figure 9 reveal that for small files, between 1kB and 100kB, Dropbox provides better results. However for larger files above 1MB, Google Drive actually outperforms Dropbox. This is particularly relevant for very large files. For 100MB files the difference is very significantly, with Google presenting a delay of 6.5s (throughput of 15.5MB/s), while with Dropbox presents a delay of 60.8s (throughput of 1.64MB/s). It is our belief

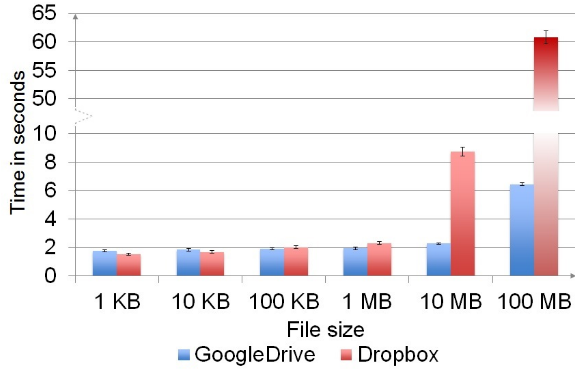

Figure 9. Performance of updates in different clouds.

that this performance difference is completely due to the cloud backends, where Google probably parallelizes data writing while Dropbox serializes it. In either cases the delay imposed by Storekeeper is always the same. This observation may lead to an automated decision to store small files on Dropbox and larger ones in Google Drive. This option is not yet implemented in the presented version of Storekeeper.

V. Related Work

Over the past years, extensive research has been carried out on cloud storage security, with a particular focus on the single cloud setting. BlueSky [3], for instance, aims to replace local storage servers with a proxy, providing the illusion of a single traditional server backing up data to the cloud. BlueSky is mostly tailored to enterprise environments and provides end-to-end confidentiality protection from cloud providers by encrypting file system data before shiping it to the cloud. SPORC [4] also interposes an encryption layer between the user's client and the cloud provider, and additionally allow for secure file sharing between users. Some other systems follow on SPORC's footsteps, but focusing specifically on access control [7], [8], confidentiality protection [9], [10], or fault tolerance [11]. However, both BlueSky and SPORC lack the mechanisms to enable secure exchange of files between multi-clouds, which is required in a cloud aggregation system.

Multiple clouds have been handled by systems such as DepSky [5] and SCFS [6], but these systems are conceptually different from cloud storage aggregators. DepSky and SCFS combine multiple clouds into a conceptually unique cloud called cloud-of-clouds (CoC). A CoC can then be used by a single user (or by a set of users) to automatically maintain multiple file replicas in different cloud backends. The goal of a CoC is mostly to increase fault tolerance or file availability. Users are oblivious to the actual location of each file. In contrast, in cloud storage aggregators, users do not share a common set of cloud accounts. Instead, each user manages an individual pool of cloud accounts where her files are located, and can freely share files with other users. As a result, the technical challenges faced by such systems are orthogonal to those addressed by Storekeeper.

VI. Conclusions

This paper presented Storekeeper, a privacy-preserving cloud aggregation service that enables file sharing on multi-user multi-cloud storage platforms while preserving data confidentiality from cloud providers and the cloud aggregator service. Storekeeper is further motivated by lack of interoperability between cloud services, which makes managing and sharing cloud-hosted files a nightmare for many users. Storekeeper aims to fill this gap. To build Storekeeper, in order to overcome heterogeneity issues between cloud services, it was necessary to devise adequate (1) file naming scheme, (2) user credentials, (3) consistency semantics, (4) access permission model and (5) file operation protocols. Storekeeper contributes to the cloud storage landscape with an original design that provides solution to all these issues.

Acknowledgments: This work was partially supported by the EC through project H2020-645342 (reTHINK), and by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013 (INESC-ID).

References

[1] "CloudFogger," http://www.cloudfogger.com/.

[2] "Odrive," http://www.odrive.com/.

[3] M. Vrable, S. Savage, and G. M. Voelker, "BlueSky: A Cloud-Backed File System for the Enterprise," in *Proc. of FAST*, 2012.

[4] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten, "SPORC: Group Collaboration using Untrusted Cloud Resources," in *Proc. of OSDI*, 2010.

[5] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa, "DepSky: Dependable and Secure Storage in a Cloud-of-Clouds," *ACM TOS*, 2013.

[6] A. Bessani, R. Mendes, T. Oliveira, N. Neves, M. Correia, M. Pasin, and P. Verissimo, "SCFS: a shared cloud-backed file system," in *Proc. of USENIX ATC*, 2014.

[7] S. Yu, C. Wang, K. Ren, and W. Lou, "Achieving secure, scalable, and fine-grained data access control in cloud computing," in *Proc. of INFOCOM*, 2010.

[8] S. Zarandioon, D. D. Yao, and V. Ganapathy, "K2C: Cryptographic Cloud Storage with Lazy Revocation and Anonymous Access," in *Security and Privacy in Communication Networks*. Springer, 2012, pp. 59–76.

[9] G. Zhao, C. Rong, J. Li, F. Zhang, and Y. Tang, "Trusted data sharing over untrusted cloud storage providers," in *Proc. of CloudCom*, 2010.

[10] H. Xiong, X. Zhang, D. Yao, X. Wu, and Y. Wen, "Towards End-to-End Secure Content Storage and Delivery with Public Cloud," in *Proc. of CODASPY*. ACM, 2012.

[11] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish, "Depot: Cloud storage with minimal trust," *TOCS*, vol. 29, no. 4, p. 12, 2011.