Chapter 4
ReDS: A System for Revision-Secure Data Storage

Tobias Pöppke and Dirk Achenbach

Abstract The CLOUDwerker project seeks to develop a tool that allows for the collaborative creation of documents in a cloud environment. This neccessitates a secure, non-repudiable document storage layer. We introduce ReDS, a software backend that stores encrypted documents in the cloud. The system also guarantees the non-repudiability of changes, makes older revisions of files accessible and has access control. Our architecture makes use of a trusted master server to store encryption keys and perform authentication and authorization. We implemented ReDS using Python and several open-source components. ReDS is open source and available for download.

4.1 Introduction

Cloud computing has developed from a hype to a widely used technology and business model. Seemingly infinite resources and the possibility to adapt the use of resources to a changing demand make cloud technology attractive even to small and medium enterprises (SMEs). The general fear of industrial espionage and a growing uncertainty regarding the privacy of outsourced data [1] make the technology difficult to adopt, however. Since SMEs are usually not capable of operating their own data centers, public cloud solutions seem practical. Therefore, there is a demand for technological solutions to increase data security and privacy in cloud scenarios.

Builders and trade contractors often work out joint contracts to provide a combined offer which includes the work of several contractors. Cloud storage systems provides a facility to share documents for cooperation. For an efficient workflow perfectly interacting systems are necessary. The CLOUDwerker project aims at providing a solution for this problem. The solution will employ cloud technology in

Tobias Pöppke · Dirk Achenbach
Karlsruhe Institute of Technology
e-mail: {dirk.achenbach,tobias.poeppke}@student.kit.edu

© Springer International Publishing Switzerland 2014

H. Krcmar et al. (eds.), *Trusted Cloud Computing*,
DOI 10.1007/978-3-319-12718-7_4

order to guarantee a high availability and scalability. The partners in the CLOUDwerker project are CAS Software AG, 1&1 Internet AG, Haufe-Lexware GmbH & Co.

KG, Fraunhofer IOA and the Karlsruhe Institute of Technology (KIT). A secure document storage system is an important part of the tool. The documents that are jointly drafted by the users of the system need to be stored in a reliable data store. Public cloud storage solutions usually offer a high level of availability. The CLOUDwerker document storage system must guarantee confidentiality against the storage provider, however. Also, the storage backend must offer versioning, non-repudiation of changes and access control for the users of the system.

Our Contribution We introduce the ReDS system, a document storage system for public cloud stores. It offers the recovery of earlier versions of documents, maintains an append-only revision history, and ensures that modifications to documents are non-repudiable. Further, it ensures the confidentiality, integrity and authenticity of the stored documents. It also offers an access control mechanism to simplify user management. We implemented ReDS using open-source components like Mercurial, Cement and S3QL. We released the sourcecode for ReDS itself under the Apache License (see Section 3).

4.1.1 Related Work

Feldman et al. proposed the SPORC [3] system for collaborative work with untrusted cloud storage. It uses an untrusted cloud server for computation. SPORC uses digital signatures for non-repudiation, but does not support the recovery of earlier revisions of a document. Depot by Mahajan et al. [6] focuses more on the aspect of availability than SPORC and our system. As in SPORC there are no revisions of documents available. Wingu [2] by Chasseur et al. is a system to coordinate concurrent access to cloud storage. Our systems overall design is similar to the one introduced in Wingu, but with the focus on information security. Cryptonite [5] by Kumbhare et al., like our system, uses well-known security techniques to implement a secure data storage on a public cloud. Cryptonite also does not support revision control for documents, however.

4.2 ReDS

In this section we give an overview of the design of ReDS. ReDS employs a master server, which coordinates the access to the documents. The master itself is structured using a layered approach with the connection to the cloud storage as the first layer. The second layer consists of the revision-control system and in the third layer, access control mechanisms are implemented. These layers are also dicussed in this section.

4.2.1 Overview

To meet the requirements for the secure document storage service, we use a master component, similar to the approach by Chasseur and Terrell [2]. The master coordinates the access to the cloud storage and the overall system is therefore designed as a client-server system. This architecture makes it easy to coordinate concurrent access to the documents, which is the primary concern of the contribution by Chasseur and Terrell.

To ensure the confidentiality of the documents, we propose the use of symmetric-key encryption on all data that is stored in the cloud storage. For the purpose of encoding and decoding, the master chooses and stores the used symmetric key locally. This neccessitates the premise that the master is to be trusted regarding the confidentiality of the documents. The clients authenticate themselves to the master by using public-key cryptography. Therefore the master has to have access to the public keys of all authorized clients. Public-key cryptography is also used to digitally sign all documents. The master verifies the signature by using either a public-key infrastructure or by storing the used public-keys of the clients. Refer to 14.1 for a graphical overview of the system.

Fig.1. Overview of the system structure. Clients connect to the cloud store through the master server. The master server stores the symmetric key for data encryption and the public keys of the clients.

The master component is designed using a layered approach with three layers. Therefore the implementation of any layer can be replaced without affecting other layers. The lowest layer has the task to make the cloud storage available to the upper layers. This layer is also responsible for the encryption of the data that is transmitted to the cloud storage provider.

The second layer implements the revision-control system as well as the verification mechanisms for the used digital signatures. By verifying the digital signatures, the integrity and authenticity of the documents is also verified. Furthermore, the version control system also has to coordinate concurrent access to the documents. Figure 2 shows the layers and the security goals they achieve.

Figure 14.2 Overview of the layers in the master and the security goals they achieve. The layered architecture ensures the modularity of our approach.

The clients store their respective private keys for authentication and digital signatures. This implies that the clients are responsible for signing documents they wish to store in the cloud store. The master will refuse to store documents that have no valid signature. The master communicates with the clients over secure communication channels to mitigate the possibility of attacks on the communication. With the proposed design it is possible to combine well-known security mechanisms and concepts and a cloud infrastructure. We now describe the particular layers in detail.

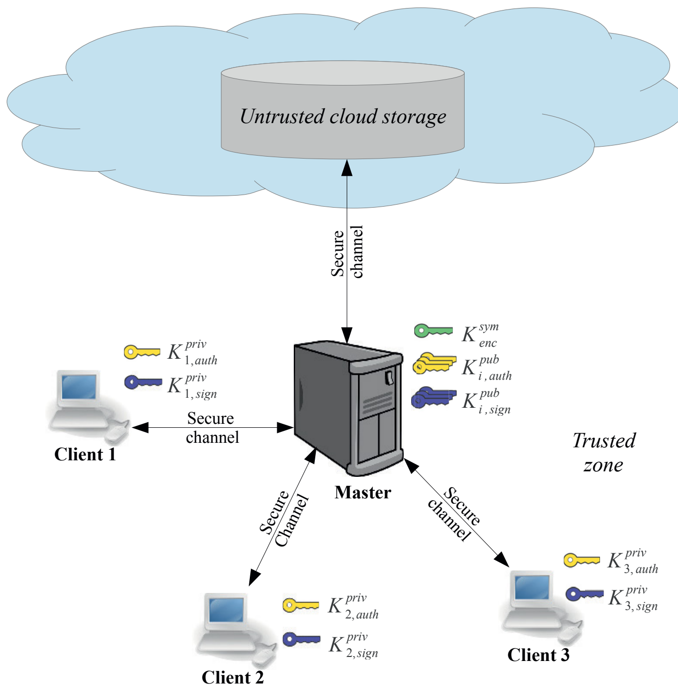

Fig. 4.1: Overview of the system structure. Clients connect to the cloud store through the master server. The master server stores the symmetric key for data encryption and the public keys of the clients.

4.2.2 Storage Connection

To connect the cloud storage with the system, we propose the use of well-known and tested security mechanisms to combine them into a cryptographic filesystem. We use symmetric-key encryption to protect the confidentiality of the data. The symmetric key is stored on the master. With a centrally stored key, it is not necessary to distribute the symmetric key to the clients like in other systems. Therefore it is not necessary to use concepts like broadcast encryption [FN94], which has to be used in systems like SPORC or Cryptonite.

In our system, the master knows the symmetric key of the filesystem encryption and therefore has access to the unencrypted data. Therefore, the master has to be trusted to protect the confidentiality of the data. However, the master has not to be trusted with regard to the integrity of the data, because of the used digital signatures. To check the integrity and authenticity of the data stored in the cloud storage, the cryptographic filesystem uses message authentication codes (MACs). They are used

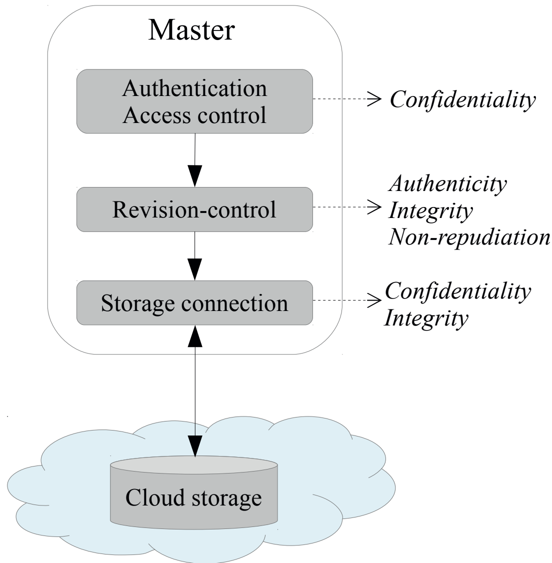

Fig. 4.2: Overview of the layers in the master and the security goals they achieve. The layered architecture ensures the modularity of our approach.

to check that the data in the cloud storage system was not altered. After checking the integrity of the data received from the cloud storage, the master mounts the cryptographic filesystem locally to make it available to the revision-control system.

4.2.3 Revision Control

The revision control system is responsible for tracking every modification of the status of stored documents and making sure that the modification is not repudiatable afterwards. To this end, we combine digital signatures with the concept of hash-trees, which were introduced by Merkle [7]. Hash-trees are used in popular distributed version control systems like git or Mercurial to store the history of a repository and to check its integrity.

Distributed version control systems are also capable of efficiently handling concurrent access on the data of a repository. Therefore we propose the use of a distributed version control system to fulfill the tasks of the revision-control layer in the master. This decision makes it also possible for the clients to work offline.

Additionally to this, the revision-control layer demands a digital signature from the client on any revision that should be uploaded to the storage. With the digital sig-

nature on the head-revision, the client also signs the entire history of the repository. This is a property of the hash-trees that are used in the distributed version-control system—the hash of a node recursively includes the hashes of its children. Therefore every revision prior to the signed revision can be traced to the user who digitally signed the head-revision.

By using digital signatures on the revisions, it is also possible to verify the integrity and authenticity of the revision and the included documents.

Additionally it is not possible for someone with access to the master to forge documents or modify them. An attacker could not forge the documents, because they would have to be signed to go unnoticed by the clients. Therefore the attacker would have to sign the respective revisions and would be tracable as the author.

The master could dispute the reception of a revision, because using a digital signature only provides a proof of origin. In a fair non-repudiation protocol, there should also be a proof of receipt from the master [9].

4.2.4 Access Control and Authentication

One important aspect of data confidentiality is the restriction of access to the data. Because we use a central master in our system, it is possible to use well-established methods to control access to the data. We propose the use of role-based access control (RBAC [4]). With RBAC it is possible to restrict the access of a client so that it is only possible to access the documents and repositories for which it is authorized to do so. We propose RBAC because of its flexibility and wide acceptance. Before authorization can occur, the client must be authenticated. To authenticate a client, we propose the use of an asymmetric key exchange protocol. This not only authenticates the client to the master but can also be used to create a secure communication channel between the two. To this purpose, the master stores the public-keys of the clients.

4.3 Implementation

In this section we discuss the implementation of the design given in Chapter 2. The ReDS master was implemented in Python. Python makes it easy to rapidly develop functionality by providing existing modules for many common tasks. It is also possible to call external programs with little effort. This capability is used a lot in the implementation of the master. Python uses an interpreter for the application to run, which is slower than native code. This could become an issue in systems with many users. The implementation of the master consists of approximately 2000 source lines of code. As a framework for the master we used the command-line application framework Cement. Cement allows the use of extensions and plug-ins and handles the configuration files. We use this functionality by only defining the inter-

faces of the three layers in the core of the application. The actual implementation of the functionality is then done inside an extension which is easily exchangeable.

The implementation of the master component of ReDS is licensed under the Apache License, Version 2.0. It is available through the Python Package Index or Bitbucket. For our implementation we rely on the S3QL file system. S3QL is an cryptographic file system designed for the use of online storage. It supports Google Storage, Amazon S3 and Open Stack, among others. With an existing Python 2.7 installation it can be easily downloaded and installed by executing `pip install redsmaster`.

In the remainder of this section we describe how the connection to the storage, the revision control and the access control components are implemented. The section closes with a description of the clients that can connect to the master.

4.3.1 Storage Connection

The interface of the storage connection component is defined in the `IEncryptedFS` class. In our implementation we used the cryptographic filesystem S3QL as the underlying filesystem. This filesystem has several advantages over other cryptographic filesystems like EncFS . S3QL was specifically designed to access cloud storage and is therefore optimised to produce only minimal network traffic. To accomplish this, it stores the filesystem metadata in a database that is cached locally. Therefore operations such as moving files or renaming them only require updating the database. The database is then periodically written back in encrypted form to the cloud storage or upon unmounting [8].

Another advantage of S3QL is the transparent data de-duplication and data compression to reduce the used storage space. Additionally, the filesystem only creates blocks of data in the cloud storage. No information of the content of those blocks is visible to an adversary with access to the cloud storage. EncFS in contrast mirrors the exact directory structure and only encrypts the files themselves so that an attacker could gain valuable information about the stored data. Moreover, most cryptographic filesystems are not designed to work with cloud storage. One downside with S3QL is that it does not support multiple mounts of the same cloud storage, which makes it impossible to run multiple instances of the master component.

S3QL uses SHA256-HMAC as the message authentication code to ensure the integrity of the data. The computed hash of the data that is to be uploaded is then encrypted (using AES-256) together with the data and uploaded to the cloud storage provider. The used symmetric key is a random master key, which itself is encrypted using a passphrase. That makes it easier to change the passphrase, because only the master key has to be reencrypted. For transport layer security, S3QL uses SSL/TLS, if supported by the storage provider.

We implemented the interface to S3QL as a wrapper around the console interface of S3QL in the class `S3QLHandler`. Because S3QL is implemented as a FUSE

filesystem, it can be mounted without requiring special rights. But this also limits the supported operating systems for the master to the ones which support FUSE.

4.3.2 Revision Control

In the class IVersionControl the interface for the revision control component is defined. Our implementation in the `HgHandler` class wraps the Mercurial distributed version control system to manage the documents. In addition to using Mercurial, the implemented revision control component ensures that all repositories used by ReDS use the Commitsigs extension for Mercurial. The extension makes it possible to embed and verify digital signatures in revisions uploaded to the master. In the case of a non-existing or invalid digital signature, the master refuses to accept the changes. The supported systems to digitally sign revisions with the extension are GnuPG and X.509 certificates. Therefore it is possible to use existing public key infrastructures such as the Web of Trust with ReDS. The interface between the storage connection layer and the revision control layer in our implementation is given through filesystem operations, because S3QL mounts the cloud storage locally. Mercurial can then access the data through the mounted filesystem.

4.3.3 Access Control and Authentication

Access control is performed by our implementation of the core RBAC model [4] which is implemented in the `RedsAccessManager` class. To store the access control policies a SQLite database is used. Because the database consists of a single file, the database can be stored in the cloud storage using the cryptographic filesystem. In this way the access control policies are available independently of the specific master used. Five basic rights are defined to control access to the data:

repo.read The user has read access to a specific repository.
repo.write The user has write access to a specific repository. This also includes read access.
repo.create The user has the right to create new repositories.
admin.repos The user has full access rights to all repositories, including creation and deletion.
admin.usermanagement The user can change the access rights of users.

All admin operations are granted on the root directory, that means the tuple `(admin.repos)` describes the access rights of a repository administrator. For all other operations the path to the repository has to be given, for example `(repo.read, myrepo)`.

In the database used to store the access control policies, the public keys to authenticate the clients are also stored, as well as the hashes of defined pass-

words. The authentication is done by our implementation of an SSH server in the `SSHServerManager` class. This implementation is based on the SSH implementation Paramiko and also provides a secure communication channel to the clients.

4.3.4 Clients

Because the master is implemented using Mercurial to manage the documents, it is possible for the clients to use a Mercurial client to access the documents. The client can use the master like any Mercurial server. Our server enforces access control and does not accept unsigned revisions or revisions with invalid signatures, however. To generate digital signatures which the master accepts, it is necessary for the client to use the Commitsigs extension. But this is only needed if a revision is to be uploaded to the master. A Mercurial client can read the repositories which he has access to, even without this extension.

4.4 Summary and Outlook

We presented the ReDS system, a revision-secure document store. ReDS is intended for the use in conjunction with a cloud data store. The architecture of the system relies on a master server to store encryption keys, authenticate users and ensure consistency. We implemented ReDS in Python using several open source frameworks. Our implementation is open source itself and available for download.

Our document storage system is a backend. Because it behaves like a server of the Mercurial revision control system, any Mercurial client can be used to interact with it. To make the system more user friendly however, there is a need for a web frontend.

The implementation of our design performed well in preliminary tests. We plan to perform usability tests with a larger group of users in the near future.

References

1. U.S., British intelligence mining data from nine U.S. Internet companies in broad secret program. Washington Post (2013). http://www.washingtonpost.com/investigations/us-intelligence-mining-data-from-nine-us-internet-companies-in-broad-secret-program/2013/06/06/3a0c0da8-cebf-11e2-8845-d970ccb04497_story.html
2. Chasseur, C., Terrell, A.: Wingu - a synchronization layer for safe concurrent access to cloud storage (2010)
3. Feldman, A.J., Zeller, W.P., Freedman, M.J., Felten, E.W.: Sporc: Group collaboration using untrusted cloud resources. OSDI, Oct (2010)

4. Ferraiolo, D.F., Sandhu, R., Gavrila, S., Kuhn, D.R., Chandramouli, R.: Proposed nist standard for role-based access control. ACM Transactions on Information and System Security (TISSEC) **4**(3), 224–274 (2001)
5. Kumbhare, A., Simmhan, Y., Prasanna, V.: Cryptonite: A secure and performant data repository on public clouds. In: Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on, pp. 510–517 (2012). DOI 10.1109/CLOUD.2012.109
6. Mahajan, P., Setty, S., Lee, S., Clement, A., Alvisi, L., Dahlin, M., Walfish, M.: Depot: Cloud storage with minimal trust. ACM Trans. Comput. Syst. **29**(4), 12:1–12:38 (2011). DOI 10.1145/2063509.2063512. URL http://doi.acm.org/10.1145/2063509.2063512
7. Merkle, R.C.: A certified digital signature. In: G. Brassard (ed.) Advances in Cryptology - CRYPTO 89 Proceedings, *Lecture Notes in Computer Science*, vol. 435, pp. 218–238. Springer New York (1990). DOI 10.1007/0-387-34805-0_21. URL http://dx.doi.org/10.1007/0-387-34805-0_21
8. Rath, N.: http://www.rath.org/s3ql-docs/impl_details.html (access: 07/19/2013)
9. Zhou, J., Gollman, D.: A fair non-repudiation protocol. In: Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on, pp. 55–61 (May). DOI 10.1109/SECPRI.1996.502669