SafeCloud

Initial Secure File System

Deliverable 2.4

Project reference no. 653884

February 2017

European Commission

Horizon 2020
European Union funding
for Research & Innovation

Document information

Scheduled delivery	01.03.2017
Actual delivery	01.03.2017
Version	1.0
Responsible partner	INESC-ID

Dissemination level

Public

Revision history

Date	Editor	Status	Version	Changes
15.01.2017	M. Pardal	Draft	0.0	Initial TOC
20.01.2017	M. Pardal	Draft	0.1	Extended with INESC-ID components
28.01.2017	S. H. Totakura	Draft	0.2	Extended with sKnock
31.01.2017	H. Niedermayer	Draft	0.3	Added certificate management
03.02.2017	S. H. Totakura	Draft	0.4	Format compliance
23.02.2017	D. R. Matos	Draft	0.5	Improved format compliance
28.02.2017	M. Correia	Final	1.0	Reviewers comments incorporated

Contributors

M. Correia (INESC-ID)
M. Pardal (INESC-ID)
F. Apolinário (INESC-ID)
E. A. Silva (INESC-ID)
D. Matos (INESC-ID)
L. Rodrigues (INESC-ID)
A. R. Silva (INESC-ID)
J. D. Pereira (INESC-ID)
J. C. Monteiro (INESC-ID)

Internal reviewers

S. H. Totakura (TUM) D. Burihabwa (UniNE) L. Yazdanov (C&H)

Acknowledgments

This project is partially funded by the European Commission Horizon 2020 work programme under grant agreement no. 653884

More information

Additional information and public deliverables of SafeCloud can be found at:
`http://www.safecloud-project.eu/`

Contents

List of Figures

List of Tables

1 Executive Summary

File storage is one of the most successful use cases for cloud computing. Services like Dropbox, Google Drive, Amazon S3, Microsoft OneDrive, and Apple iCloud Drive are widely used worldwide to store both personal and work files. Therefore, it makes sense to provide the security assurances of SafeCloud to files stored in such services.

Some of these cloud storage services provide a web interface, often a RESTful interface, but this is not the usual interface for file storage. In fact, usually users access files through the interface provided by the operating system (e.g., through a windows interface or a command line console) or through applications that process these files (e.g., a text editor, spreadsheet editor, etc.). Several of the cloud storage services, e.g., Dropbox and Google Drive, recognise that this integration with the operating system is the natural way of accessing files, so they provide client software that provides such integration. Specifically, from the point of view of the user, the cloud storage space appears as if it is yet another folder in his disk. This integration is possible by mimicking a (local) file system by providing a Portable Operating System Interface (POSIX) file system interface, which is the standard in most current operating systems (e.g., Linux, Mac OS X, and Windows).

The objective of WP2's Task T2.3 is to provide a *secure cloud-backed file system*, more exactly a file system that stores files in clouds and provides SafeCloud's high degree of protection from strong adversaries. This deliverable presents the initial design of this file system. The final design will be presented in deliverable D2.8.

The SafeCloud filesystem – *SafeCloud-FS* – provides a POSIX interface and stores files in a set of clouds – in a *cloud-of-clouds* – in such a way that the files *integrity* and *availability* are guaranteed even if some clouds are compromised. These clouds may provide the SafeCloud block storage abstraction. Moreover, the file system stores both the files and their metadata (e.g., file name, modification date, and directory) encrypted for *confidentiality* and *privacy*. Keeping metadata encrypted is particularly challenging as this data must be accessed by the cloud (e.g., to return a file with a certain name), so the file system has to resort to homomorphic encryption to support some operations without decrypting the data [Gen09]. Finally, the file system allows a mechanism to audit if the files are actually stored and not modified in the clouds without the need to download the files, which may be costly. This mechanism provides a second layer of *integrity* and *availability*. Communication may be done over SafeCloud's middleware (WP1) for higher security also at that level.

This deliverable is organized as follows:

- Section 2 presents the business requirements for the SafeCloud filesystem.
- Section 3 describes the overall design of the file system and how it provides the main security properties by leveraging the notion of cloud-of-clouds.
- Section 4 details the coordination service used in the file system – HomomorphicSpace – and the library in which it is based – MorphicLib.
- Section 5 presents SafeAudit, which is the service that allows verifying if files stored in the cloud have not been modified.

- Section 6 concludes the deliverable.

2 Business Requirements

This section describes the business requirements for SafeCloud's secure cloud-backed file system.

2.1 Cloud storage market

The cloud storage market is strongly expanding. A recent market study estimates a growth from 23.76 billion US dollars in 2016 to 74.94 billion in 2021 at a CAGR[1] of 25.8% [Mar16]. According to that report, "Factors such as high demand for hybrid cloud storage, growing need for enterprise mobility, and need for easy implementation of cloud storage solutions are fuelling the growth of the cloud storage market." The main solutions in that market are (i) primary storage, (ii) disaster recovery and backup storage, (iii) cloud storage gateways, and (iv) data archival. Another market study, estimates the market to reach 67 billion dollars by 2022 [Res16].

There is a clear demand for cloud storage to be integrated with the operating system, providing a POSIX *file system* interface (or close to it). A first argument to substantiate this claim is that major cloud storage services are providing a client that provides exactly that integration. More precisely, these clients allow files stored at the cloud to be created, listed, deleted, and edited as if they were in a folder/directory in the local file system of the computer. Examples are the Dropbox desktop application[2], Microsoft OneDrive sync client[3], Google Drive's client[4], and iCloud Drive's client (that comes with Mac OS X and iOS). There is also a vast offer of open clients for Amazon S3 and others, e.g., DragonDisk S3 Client[5], S3 Browser[6], and CloudBerry Explorer[7].

A second argument is the increasing adoption of *cloud storage gateways* (CSGs). A CSG is a network appliance or a server that companies can install to mediate the access to cloud storage. From the point of view of the computer accessing the storage, the CSG appears to be a block storage device (communicating, e.g., using iSCSI) or a distributed file system (e.g., NFS or AFS), so the access is made similarly to what happens with such systems and is fully integrated with the operating system. This is interesting because it makes configuring the access to cloud storage similar to configuring access to classical distributed file systems. According to the first study mentioned, the CSG market will be the solution that will grow faster in the 2016-21 period, i.e., will have the highest CAGR among the four solutions mentioned above [Mar16].

[1] Compound Annual Growth Rate
[2] https://www.dropbox.com/en/help/65
[3] https://support.office.com/en-us/article/Get-started-with-the-new-OneDrive-sync-client-in-Windows-615391c4-2bd3-4aae-a42a-858262e42a49
[4] https://www.google.com/drive/download/
[5] https://aws.amazon.com/customerapps/3929
[6] http://s3browser.com
[7] https://www.cloudberrylab.com/explorer/amazon-s3.aspx

2.2 Cloud storage security market

The cloud security market is also expanding. Recent studies estimate that it will grow to 8.71 billion US dollars in 2019 [Mar14] and to 11.8 billion in 2022 [Tra16]. The security solutions considered in these studies are security controls, so storage and file systems do not appear. However, some of these controls have the primary goal of protecting data stored in the cloud, e.g., encryption and database security.

The security of data stored in the cloud is a major concern since the beginning of the boom of the cloud. In an interesting early roundtable on cloud security, held between major cloud players, all assured the audience that data was safe in their clouds [GHR+10]. However, at a point a participant in the roundtable replied to a question about security and trust in the cloud that "there are some things that will never go into [the cloud], for example, our SAP back end."

There is much data in the Internet on companies' concerns about privacy, confidentiality, integrity, and availability of data stored in the cloud, which shows that these concerns are still present among cloud adopters. A piece of news stated that "In the three years since Snowden's initial leak, Apple, Google, Microsoft, Facebook and Yahoo [all major cloud providers except Apple] have become some of the biggest advocates of consumer privacy. They've beefed up encryption and other safeguards in their products and services" [Hau16]. These companies took such measures "to protect business", meaning that they understand the importance of providing services that guarantee these properties. A similar conclusion may be taken from the coalition of 40 large tech companies created around Apple to block the FBI's access to private data stored in iPhones [Che16].

In a recent report, Microsoft provided a list of "What customers want from cloud providers" [Mic15]. The top 2 desires are "Secure our data" and "Keep our data private". The two top threats listed by the Cloud Security Alliance in their report "The Notorious Nine – Cloud Computing Top Threats in 2013" are also related to cloud storage: "Data breaches" and "Data loss" [Clo13]. The first is a threat against privacy and confidentiality, whereas the second is a threat against integrity and availability. This shows how one of the biggest cloud providers – Microsoft – and the major consortium on cloud security worldwide – CSA – consider data confidentiality/privacy/integrity to have a huge demand from the market.

2.3 Limitations of industry-standard solutions

The above-mentioned Microsoft report on cloud security includes a section on "Data Protection" that provides a glimpse of the security mechanisms provided by Microsoft Windows Azure [Mic15] (Google also has an interesting report on the matter but it provides less detail [Goo17]). These mechanisms correspond to the current industry-standard solutions:

1. *Data isolation* – data from different clients is logically separated, e.g., using virtualization schemes;

2. *Protecting data at rest* – stored data is protected using encryption;

3. *Protecting data in transit* – data is protected while being transferred over the network;

4. *Encryption* – customers can encrypt their data for protection;

5. *Data redundancy* – data may be replicated for redundancy and disaster recovery;

6. *Data destruction* – data should become unreadable when the customer requests its deletion or leaves the service.

There are different ways to implement these services. Common mechanisms include disk encryption, file system encryption, hash-based message authentication codes, digital signatures, the SSL/TLS protocol, and passive replication.

Nevertheless, these solutions have a set of limitations:

1. If an external attacker or a malicious insider access an account in a cloud provider, they may corrupt or delete user files, making them unusable;

2. An outage in a cloud provider may let files temporarily unavailable;

3. Even if files are encrypted, meta-data like file names and creation dates is not;

4. Checking if data is stored uncorrupted in a cloud may require downloading the whole files.

The SafeCloud secure cloud-backed file system addresses these limitations of current solutions.

3 Secure file system design

SafeCloud's secure file system – SafeCloud-FS – is based on some of the design principles of the Shared Cloud-backed File System (SCFS) [BMO+14]. In fact, SafeCloud-FS can be consider to be an extension – or a enhancement – of SCFS.

SafeCloud-FS is a distributed, POSIX-compliant, distributed file system that guarantees data confidentiality, integrity, and availability. It allows users to store files in a cloud or a set of clouds (a *cloud-of-clouds*) with the usual consistency of a file system, atomic consistency or linearizability [HW90], even if weak consistency storage cloud services are used. This is important as public clouds normally provide only eventual consistency [Vog09].

To use the file system, users *mount* it on a folder of their computer or device, and the SafeCloud-FS client-side library synchronizes files with the cloud storage services. SCFS supports data sharing among several users, automatically propagating users' modifications between them.

In SafeCloud-FS files are stored on several clouds using the DepSky software library [BCQ+13]. DepSky provides an API for uploading and operating with a set clouds, while enforcing fault tolerance, lock-in resilience, confidentiality, and integrity as long as the clouds affected with the aforementioned problems do not reach the majority of the cloud set.

3.1 Features

The main features of SafeCloud-FS are the following:

- It stores every file in a set of clouds, forming a *cloud-of-clouds*;
- It provides a *POSIX interface*, so files are manipulated using the standard functions, e.g., open, read, write, chmod, mkdir, flush, fsync, link, rmdir, symlink, chown, etc.;
- Similarly to local file systems, each file has an owner, but may be *shared* with other users that may also read and modify it;
- It provides *controlled sharing*, in the sense that it provides access control mechanisms that allow controlling who can use each file;
- It provides a *pay-per-ownership cost model*, meaning that each user pays for the storage of his files;
- It *runs mostly at the client* and does not require a cloud storage gateway (CSG);
- It uses *unmodified storage clouds* for storing the files;
- It provides strong consistency by leveraging a *consistency anchor*, which in implemented using a *coordination service*;
- It is *modular* in the sense that the service is composed by a set of parts that work

together but can be exchanged by others with similar functionality – coordination service, verifier, storage clouds;

- It uses *caching* extensively in order to provide a performance as close as possible to a local file system and to reduce monetary costs;

- It provides *consistency-on-close semantics*, i.e., when a user closes a file, all updates he did become observable to the rest of the users, and it provides *locks* to avoid write-write conflicts;

- It allows doing *integrity verification* of the files stored in the individual clouds without downloading them.

SafeCloud-FS provides the following security and dependability properties:

- *Availability* – files continue to be usable even if some clouds stop working (the other clouds are still there);

- *Integrity* – files continue to be usable even if some clouds corrupt them (the files are still at the other clouds);

- *Disaster-tolerance* – files continue to be usable even if some clouds suffer disasters such as earthquakes and floods (files may be stored in clouds geographically far apart);

- *Confidentiality (from clouds)* – neither files not their metadata can be read by external intruders or malicious insiders (they are encrypted);

- *Confidentiality/integrity (from users)* – files cannot be read or modified by unauthorized users (there is access control).

The initial version of SafeCloud-FS, the one that is described in this deliverable, essentially adds two mechanisms to SCFS:

- *Encryption of file metadata* – file metadata such as names, directories, and timestamps may be private. Although encrypting metadata may seem as a trivial extension of file encryption, this is not the case. In fact it involves using homomorphic encryption because metadata must be searched and it is impractical to download and decrypt all metadata before accessing files. This encryption is supported by an homomorphic encryption library (MorphicLib) and an homomorphic tuple space (HomomorphicSpace) that are presented in Chapter 4.

- *Integrity verification mechanism* – if files are deleted or corrupted in a cloud, either due to accidental or intentional reasons, the degree of redundancy becomes lower and files become more vulnerable to other issues. SCFS allows doing this verification but it requires downloading the files and checking a signature, which is both slow and expensive (downloading files from a cloud has a cost). The integrity verification mechanism (SafeAudit) is presented in Chapter 5.

3.2 Architecture

Figure 3.1 presents the overall architecture of SafeCloud-FS. This architecture has mainly three parts: storage clouds, computing clouds, and clients.

In SafeCloud-FS, files are envisaged to be stored in public *storage cloud* services, such as Windows Azure, Google Files, rackspace, and Amazon S3. These services are not modified, i.e., there is no SafeCloud-FS code running in that part of the system. Alternatively, any device that provides the *SafeCloud block storage abstraction* can be used. This part is shown in the bottom-right of the figure.

SafeCloud-FS needs some code to run in the cloud, so it also resorts to *computing cloud* services like Windows Azure or Amazon EC2 (top of the figure). SafeCloud-FS runs two components in those services. First, it permanently runs a coordination service called HomomorphicSpace replicated in several of these services in order to support locks, access control, and storing file metadata. Second, when a user requires file integrity verification, they run a verifier (auditor).

The rest of the logic of SafeCloud-FS is implemented at the clients: *FS Client* in the figure (left). The clients do mainly four tasks. First, they manage file caching, which is extremely important from the point of view of performance and cost. Second, they access the storage clouds to read and write files. Third, they access the coordination service for reading and writing file metadata, and to access the files in a controlled way (locks, access control). Four, they launch and access verifiers in the computing clouds to do integrity verification.

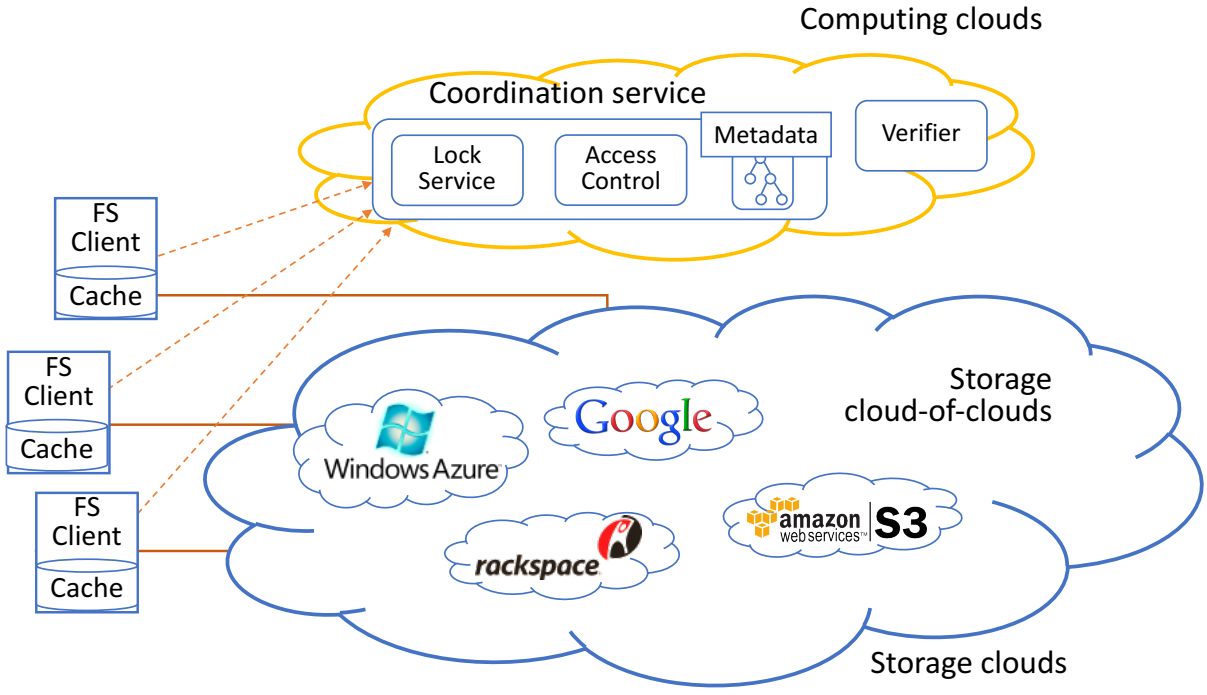

Figure 3.1: SafeCloud-FS architecture.

3.3 Functions

In this section we describe how five important POSIX functions are implemented in SafeCloud-FS: open, read, write, flush, and close.

Open Before a file is read or written, it must be opened using function *open*. This function involves three main steps:

1. Access the coordination service to read the file metadata;
2. If the file is being opened for writing, access the coordination service to create a lock for the file and wait for the lock to be granted;
3. Access the storage cloud-of-clouds to read the file to the local cache.

The reading of the file in the last step is done using DepSky's read protocol (see Figure 3.2). In this protocol, the client accesses all storage clouds and gets the storage metadata of the file stored from a majority of them. Then it reads the file from one of the clouds that has the highest version of the file. If there is some problem with the file (e.g., the signature does not match the file or the cloud does not provide it), the file is read from another storage cloud.

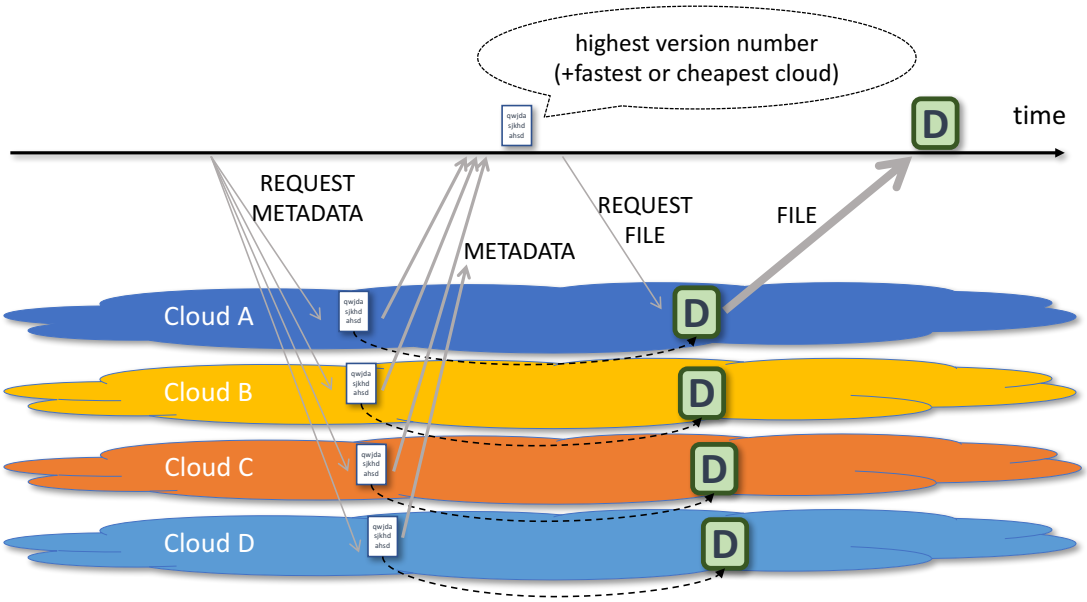

Figure 3.2: DepSky read protocol.

Read and write As mentioned above, when the file is opened it is downloaded and stored in the local cache. Reads and writes are done in the version of the file stored locally, therefore they do not involve interactions with the computing clouds or the storage clouds.

A concern may be raised about the fact that writes done locally will not become visible to

other users accessing the same file. However, this is not a problem, but a direct consequence of the consistency-on-close semantics provided by SafeCloud-FS.

Flush and close Flushing and closing a file involve pushing it from the local cache to the cloud. The main steps are:

1. Write the file to the storage cloud-of-clouds;
2. Access the coordination service to update its metadata (e.g., the version);
3. If the operation is *close*, access the coordination service to unlock the file.

The writing of the file to the cloud is done using DepSky's write protocol (see Figure 3.3). The client essentially uploads the file to all clouds, then writes the storage metadata.

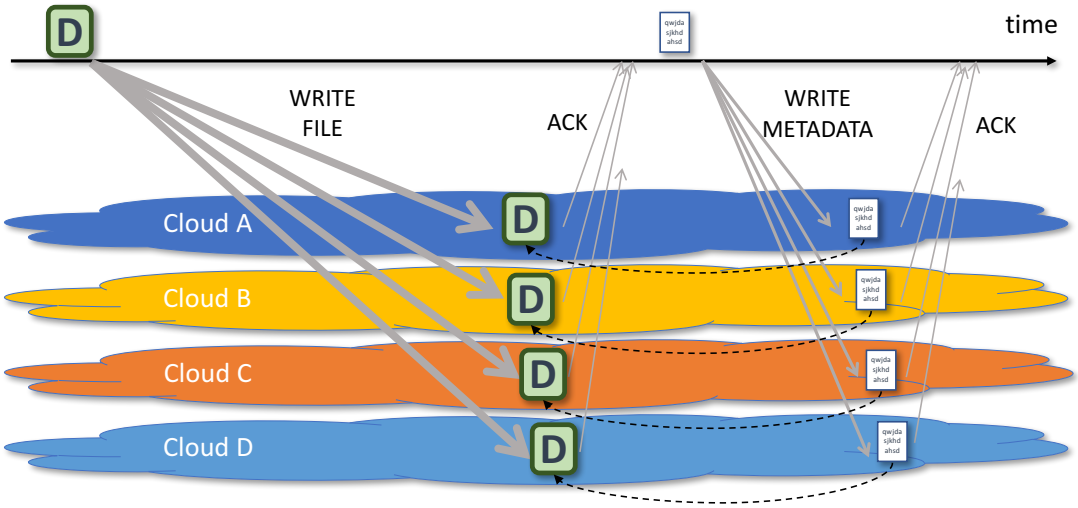

Figure 3.3: DepSky write protocol.

3.4 Summary

This section presented the overall design of SafeCloud-FS: its features, the properties it enforces, the architecture, and the operation of its main functions. All these functions are similar to SCFS'. The main difference is that metadata is stored encrypted in the coordination service, which is the topic of the following section. Later, Section 5 presents the integrity verification scheme.

4 Homomorphic coordination service

This sections presents HomomorphicSpace, a coordination service that provides a *tuple space* abstraction [Gel85].

The HomomorphicSpace is based on a new library of homomorphic functions that we designed called MorphicLib, so we present it first (Section 4.1). The HomomorphicSpace is an extension of the DepSpace coordination service, so we introduce that system afterwards (Section 4.2). The rest of the section presents HomomorphicSpace itself.

4.1 MorphicLib

MorphicLib is a novel library of partial homomorphic cryptographic functions written in Java and providing a Java API. MorphicLib was not developed from scratch, but based on existing source code whenever possible. The objective was both to simplify the task and to avoid introducing bugs, which tend to appear due to the complexity of cryptographic code. This library can be used both at the client-side to encrypt and decrypt data, and at the server-side to do operations over encrypted data.

The code of the library is organized in classes, one per *homomorphic property*. One crucial difference between *partial homomorphic encryption* (PHE) and *fully homomorphic encryption* (FHE) is that in PHE data has to be encrypted taking into account the kind of operation that will be supported over the encrypted data. With FHE, on the contrary, arbitrary computation is possible over encrypted data (at a huge cost, in terms of performance). As we opted for PHE for efficiency (FHE is extremely slow), for each homomorphic operation we have four kinds of functions (or methods):

- Key generation function, typically used at client-side;
- Encryption function, typically used at client-side;
- Decryption function, typically used at client-side;
- Homomorphic operation functions, which allow doing operations over encrypted data, typically used at the server-side.

Next we explain the implementation of the functions for each homomorphic property. Information about the properties of the PHE algorithm, the operations supported, and the classes are in Table 4.1. Figure 4.1 shows a summary of the library API.

Table 4.1: MorphicLib's main classes

Property	Homomorphic Operations	Class	Input Data Types
Random	None (strong cryptanalisys resistance)	HomoRand	Strings, Byte Arrays
Deterministic	Equality an inequality comparisons	HomoDet	Strings, Byte Arrays
Searchable	Keyword search in text	HomoSearch	Strings
Order preserving	Less, greater, equality comparisons	HomoOpeInt	32 bit Integers
Sum	Add encrypted values	HomoAdd	BigInteger, String
Multiplication	Multiply encrypted values	HomoMult	BigInteger, String

```
1  public class HomoRand {
2      public static SecretKey generateKey()
3      public static byte[] encrypt(SecretKey key, byte[] IV, byte[] plaintext)
4      public static byte[] decrypt(SecretKey key, byte[] IV, byte[] ciphertext)
5  }
6  public class HomoDet {
7      public static SecretKey generateKey()
8      public static byte[] encrypt(SecretKey key, byte[] plaintext)
9      public static byte[] decrypt(SecretKey key, byte[] ciphertext)
10     public static boolean compare(byte[] op1, byte[] op2)
11         throws UnsupportedEncodingException
12 }
13 public class HomoOpeInt {
14     public static SecretKey generateKey()
15     public long encrypt(SecretKey key, int plaintext)
16     public int decrypt(SecretKey key, long ciphertext)
17 }
18 public class HomoSearch {
19     public static byte[] wordDigest(SecretKey key, String word)
20     public static SecretKey generateKey()
21     public static String encrypt(SecretKey key, String plaintext)
22     public static String decrypt(SecretKey key, String ciphertext)
23     public static boolean searchAll(String words, String ciphertext)
24 }
25 public class HomoAdd {
26     public static PaillierKey generateKey()
27     public static BigInteger encrypt(BigInteger m, PaillierKey pk)
28         throws Exception
29     public static BigInteger decrypt(BigInteger c, PaillierKey pk)
30     public static BigInteger sum(BigInteger a, BigInteger b, BigInteger
           nsquare)
31     public static BigInteger dif(BigInteger a, BigInteger b, BigInteger
           nsquare)
32     public static BigInteger mult(BigInteger a, int prod, BigInteger nsquare)
33
34 }
35 public class HomoMult {
36     public static KeyPair generateKey()
37     public static BigInteger encrypt(RSAKey key, BigInteger value)
38     public static BigInteger decrypt(RSAKey key, BigInteger ciphertext)
39     public static BigInteger multiply(BigInteger op1, BigInteger op2,
           RSAPublicKey publicKey)
41 }
```

Figure 4.1: MorphicLib API (summary)

Random – Class HomoRand The cryptographic Random scheme is not homomorphic, but was included in the library for completeness. This scheme, is called Random because every time a given value is encrypted, it gives a different cyphertext. In fact, it is not an homomorphic encryption system, but can be used in a general homomorphic aware application precisely when no homomorphic property is required for certain data. In this case, Random is more secure than any of the homomorphic encryption schemes as it is not vulnerable to a chosen plaintext attack [KL07].

For this scheme we have used the Advanced Encryption Standard (AES) implementation of the javax.crypto package with CBC mode and PKCS #5 padding. This algorithm is recommended for legacy and future use by ENISA [ENI14].

What gives this scheme the randomness property (same cleartext producing different ciphertexts) is the use of a random Initialization Vector (IV).

Deterministic – Class HomoDet In order to make possible equality comparison operations we need deterministic encryption, i.e., encryption in which the same plaintext originates always the same ciphertext. The deterministic scheme is essentially the same as the random encryption scheme, except that the IV takes a fixed value. In order to avoid that plaintexts with the same beginning have the same beginning on the correspondent ciphertext, we make a second encryption with the blocks in the reverse order, with the same IV. This form of encryption is weaker than the random scheme, but necessary for equality and inequality determinations [ENI14, PRZB11]. Needless to say, in this encryption system an attacker will be able to notice if two equal ciphertexts correspond to the same plaintext. Otherwise, this encryption scheme is as strong as AES encryption.

Searchable – Class HomoSearch The searchable scheme aims to produce a ciphertext that allows searching for words within it, without having to decrypt it. The trivial option would be to encrypt the text word by word with a deterministic encryption system. However, this approach would provide too much information to an attacker: frequency of words, position of the words in the text, and size of the words. To avoid those drawbacks we have built a scheme closely following the solution used in CryptDB [PRZB11]. The encryption for this scheme was implemented with the following sequence of steps:

1. It builds a list of distinct words found in the text (hides the frequency);
2. It encrypts each word with deterministic encryption;
3. It obtains a SHA 256 (also recommended by ENISA [ENI14]) hash of each encrypted word (hides the size of words);
4. It orders the obtained list randomly (hides the position in the text)
5. The text to be searched is encrypted with the random scheme and the list of hashes is attached.

Searching for keywords in text consists in:

- The client encrypts and hashes the keyword(s) to be searched;
- The server searches for these hashes in the list and returns the encrypted text if there is a match.

To decrypt the text the list of hashes is not necessary.

Order Preserving – Class HomoOpeInt Order preserving encryption aims to allow comparisons of encrypted values such as *greater than*, *less than*, and *greater or equal to*. We implemented this scheme by supporting the encryption of 32-bit signed integers (Java's *int* primitive type). Encryption maps each value into a positive number in the range [0, MaxLong/2]. The algorithm implemented was the one described by Boldyreva et al. [BCLO09]. The implementation was based on CryptDB's C++ implementation obtained in GitHub [P+15]. A challenge of the implementation was to find a reverse hypergeometric pseudo-random variate generator method, as CryptDB's code was too complex. Instead we used a Java implementation of the algorithm described in [KS85] available at GitHub [Der13].

Sum – Class HomoAdd As partial homomorphic scheme for the sum operation, we used the Paillier cryptosystem [KL07]. In order to be able to work with numbers as large as necessary, we decided to use as inputs big integers, namely Java's *BigInteger* class. For the implementation of Paillier we have adapted the Java code authored by Hassan found in the web [Has09].

The Paillier cryptosystem is an asymmetric scheme with the following two keys: public key – the pair (n, g); private key – the pair (λ, μ). The parameters n, g, λ, and μ are generated from two big prime numbers p and q. The parameter $n = p.q$, is part of the public key. So, the security of the system is based on the fact that an attacker cannot find p and q factorizing n. This is the same problem used by RSA, so the length of n, two times the length of p and q, should follow the recommendations for RSA, and have at least 2048 bits [ENI14].

This scheme also supports *multiplication* of encrypted values by constants. For that purpose, we raise the encrypted value to the constant (for a sufficiently large n):

$$Enc(a + b \bmod n) = Enc(a).Enc(b) \bmod n^2$$

$$Enc(k.m \bmod n) = Enc(m)^k \bmod n^2$$

Note that in PHE the operations performed with the encrypted data do not have to be the same that would be executed with plaintext. Those operations just need to produce the desired result, i.e., the result obtained must be the encryption of the result that would be obtained executing the original operation over the plaintext. This is the case with Paillier, in which to obtain the encryption of a sum, a product is made. The same way, the multiplication by a constant is determined by rising the encrypted value to that constant.

Multiplication – Class HomoMult For multiplication we used RSA, again with big integers. We used the standard Java functions in *javax.crypto* for encryption, decryption, and key generation. No padding is used to guarantee the homomorphic property.

We implemented encryption functions accepting inputs of the types *BigInteger* or *String* (containing an integer).

Two aspects should be noted:

1. In this way of using RSA both keys must be kept secret, otherwise chosen plaintext attacks would be possible;

2. The partial homomorphism for multiplication is valid for the modular multiplication. As the RSA keys have more than one thousand bits, that means that we can comfortably work with 32 bit integers or even 64 bit long integers. Actually we can work with BigIntegers of hundreds of bits provided that the multiplications do not exceed the value of the module used in the encryption.

4.2 DepSpace

DepSpace is a fault- and intrusion-tolerant *tuple space* service [BACF08]. Architecturally it is client-server system implemented in Java (see Figure 4.2). The server-side is replicated in order to tolerate arbitrary faults. The client-side is a library that can be called by applications that use the service. Clients communicate with the servers using a Byzantine fault-tolerant total order broadcast protocol called BFT-Smart. The most recent version supports extensions to the service [DBB+15]. A stable prototype is available online.[1]

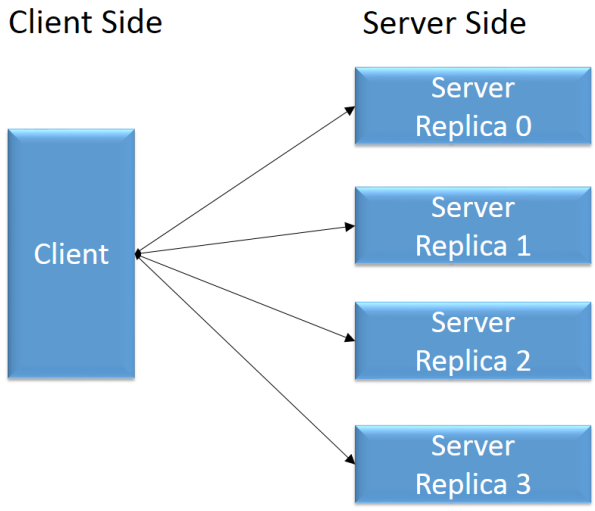

Figure 4.2: DepSpace architecture with 4 server replicas

The service provides the abstraction of tuple spaces. A tuple space can be understood as a shared memory that stores *tuples*, i.e., sequences of *fields* (data items) such as (1, 2, a). Tuples are accessed using *templates*. Templates are special tuples in which some fields have values and others have undefined values, e.g., wildcards meaning any value ("*"). A template *matches* any tuple of the space that has the same number of fields, in which the values in the same position are identical, and the undefined values match in some sense.

[1] `https://github.com/bft-smart/depspace`

For example, the template (1, *, a, *), matches the tuples (1, 2, a, b) and (1, 7, a, 14), but neither (1, 2, b, 4), where the 3rd field does not match, or (1, 2, a, b, 5), where the number of fields are different.

DepSpace supports a set of commands, issued by clients and executed by the servers. Here we consider the following commands:

- *out tuple* – inserts a tuple in the space;
- *inp template* – reads and removes from the space a tuple that matches the template;
- *rdp template* – reads but does not remove from the space a tuple that matches the template;
- *inAll template* – reads and removes from the space all tuples that match the template;
- *rdAll template* – reads but does not remove from the space all tuples that match the template.

DepSpace does not support homomorphic operations. However, it allows fields to be encrypted and basic equality matching by storing a hash jointly with the encrypted field. This solution however is vulnerable to trivial brute force and dictionary attacks. It does support the definition of access control policies using its policy-enforcement mechanism.

4.3 HomomorphicSpace

This section presents our homomorphic tuple space service.

4.3.1 Threat model

The threat model we consider for HomomorphicSpace is similar to the threat model for DepSpace except for one, crucial, difference: we consider that any server (or any cloud that contains the server) may be adversarial and try to read the content of the tuples it stores. We consider that all tuples whose fields' confidentiality has to be preserved are encrypted using homomorphic encryption, preventing malicious servers from doing such an attack.

Similarly to DepSpace, adversaries may compromise up to f out of $3f + 1$ servers and stop them or modify their behavior arbitrarily. This is tolerated using replication and the BFT-Smart protocol. Network messages may also be tampered with by the adversary, but the system tolerates this by using secure channels.

4.3.2 Commands

HomomorphicSpace extends DepSpace to allow commands over tuples with encrypted data items. More precisely in comparison with DepSpace, HomomorphicSpace:

- Supports the original match operations over encrypted data;
- Extends matching beyond the equality and wildcards with more complex matches, i.e., inequality, order comparisons (lower, greater), and keyword presence in a text, all over encrypted data;
- Allows addition and multiplication off encrypted fields.

Besides values and wildcards ("*"), HomomorphicSpace's *templates* can include the fields shown in Table 4.2.

Table 4.2: Fields that may be used in tuples in HomomorphicSpace, besides values and wildcards.

Field	Meaning
% $word_1$...$word_n$	matches a textual field containing all the words indicated
$> val$	matches a numeric field containing a value greater than *val*
$>= val$	matches a numeric field containing a value greater or equal to *val*
$< val$	matches a numeric field containing a value lower than *val*
$<= val$	matches a numeric field containing a value lower or equal to *val*

HomomorphicSpace adds three commands to those provided by DepSpace (Section 4.2): *crypt*, *rdSum* and *rdProd*.

The first is *crypt id template* and aims to define a *tuple encryption type*. The command takes as input an identifier (id) for the type it will create, and a template with the homomorphic operation desired for each of the fields, which will determine the homomorphic property. For example, if the template contains for a given field the operation "=", the system infers that the encryption to be used for that field is deterministic, which is the strongest that allows that operation. If no operation is indicated, the field will not be encrypted. The complete list of interpreted operations is:

- $=, <>$ – determinist encryption (notice that $<>$ means "different from")
- $>, >=, <, <=$ – order preserving encryption
- % - searchable encryption
- $+$ – Paillier
- & – RSA
- . – random encryption
- other value – no encryption

The second command is *rdSum template*. This command starts by collecting all the tuples that match the template similarly to *rdAll*, then sums the (encrypted) fields with $+$ in the template. The function returns a single tuple with the result.

The third command is *rdProd template*, which works similarly to *rdSum* but does multipli-

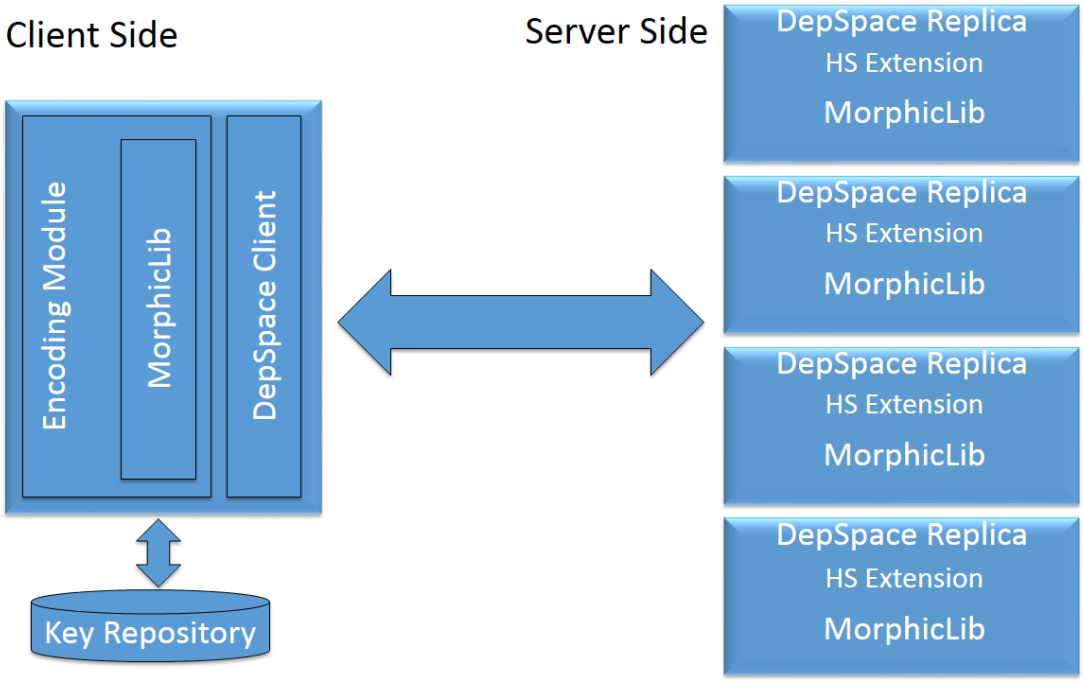

Figure 4.3: HomomorphicSpace architecture

cations instead of sum.

This scheme allows a single type of encryption per field (unlike, e.g., CryptDB). However, with the tuple data structure this is not a restriction. For instance, for tuples with a single numeric field, two operations like equality and sum can be supported by transforming that field in two and using the tuple encryption type ($=$, $+$).

4.3.3 Architecture and functioning

Architecturaly the HomomorphicSpace is similar to DepSpace, with a client-side and a server-side. Figure 4.3 represents the system with 4 replicas, i.e., with $f = 1$. From the confidentiality point of view, the server-side is untrusted and the client-side trusted.

The server-side of the system is mostly DepSpace code with the server-side of the MorphicLib and with extensions to process the homomorphic operations. The client-side includes MorphicLib's and DepSpace's client-side libraries. The main functions of the client are to encrypt tuples and send them to the tuple space, and to decrypt them before they are delivered to the application. When a tuple is encrypted, the encryption keys are stored in a *key repository* (a folder with one file per key). Next we describe both sides in more detail.

Client side. When the *crypt* command is issued (i.e., that method is called), the library generates keys for every field of the tuple for which homomorphic properties are desired. These keys are stored jointly with the tuple encryption type (id and template) in the key repository.

All the other commands (*out*, *inp*, etc.) include an id that the library uses to retrieve the

corresponding tuple encryption type and keys from the repository. If the operation indicated in a field is not compatible with the encryption defined with the *crypt* command, the command returns an error.

The library uses the DepSpace client library to send to the servers the command and the fields. If the command is an *out*, the fields are encrypted with the scheme defined in the tuple encryption type and the keys previously stored. If the command involves reading tuples it contains the operation and encrypted values. Note that each field of each id has its own key (or key pair for RSA), but the same field for the same id is always encrypted with the same key.

When the library receives a reply from the servers, it does the opposite, i.e., it decrypts the encrypted fields using the corresponding schemes and keys.

Server side. The server-side handles different commands in different ways. The *out* command is executed the same way as in DepSpace. The fields may be encrypted but they come encrypted from the client so the tuple is stored unmodified. The *inp* and *rdp* commands were modified using DepSpace's extension mechanism in order to support the $=$, $<>$, $>$, $>=$, $<$, $<=$, and text search operations over encrypted data, returning one of the matching tuples. The *rdall* and *inall* commands work similarly, as rdp and inp, but return all matching tuples. The *rdSum* and *rdProd* commands are implemented as a modification of the original *rdAll* command that returns a single tuple with the relevant fields respectively added or multiplied.

4.4 Summary

This chapter presented HomomorphicSpace, a coordination service capable of storing and processing encrypted data. The service can search encrypted data with matching operators like $=, <>, >, >=, <, <=$, find text based on keywords, and execute sums and multiplications. All those functions are performed without any decryption of the encrypted data.

5 Integrity verification service

This section presents SafeAudit, a software library that improves the Shacham-Waters (SW) integrity verification scheme [SW08] and adapts it for use with commercial clouds and SafeCloud-FS. SafeAudit improves the original SW scheme and provides: overall performance increase by carefully selecting *pairing-friendly elliptic curves* [BN05] for SW scheme parametrization; and a storage cost decrease of 50% in relation to the original scheme using *point compression* [Lyn07].

Nowadays data owners resort to integrity control mechanisms based on *cryptographic hashes* [Mer79, ErJ01] to verify the integrity of data they store in clouds: *digital signatures* for *collaborative storage*, where data is shared among several cloud users; and MACs (Message Authentication Codes) for *private storage*, where data is used by a single cloud user. To do so, users have some personal key: an asymmetric private/public key pair for digital signatures or a symmetric key for MACs. A user stores data either with a signature or a MAC, obtained respectively with the user's private key or symmetric key. Whenever the user wants to guarantee that the integrity of the data is preserved, the user: downloads the data and the corresponding signature/MAC from the cloud; then verifies if the data matches the signature/MAC. If they match, the data integrity is verified.

Notwithstanding the effectiveness of these mechanisms, they require downloading all the data to be verified from the cloud. Therefore, when users are only interested in verifying the integrity of the data, not reading it, each verification requires an unnecessary download that implies a potentially large bandwidth consumption, delay and monetary costs (downloads have a cost in most cloud storage services).

In order to reduce delay and bandwidth consumption some works evolved these integrity mechanisms [AKK09, WWRL10, WLL15, BJO09, WRLL10, dC14, SW08]. On the contrary to the previously-mentioned mechanisms (both MACs and signatures), their evolution are *homomorphic*, i.e., the integrity control structures they produce have the same structure/-format as the signed data (details later). These mechanisms provide *verifiability* (data integrity can be verified using proofs) and *unforgeability* (unauthorized modifications to proofs, data or control structures are always detected) without the need of downloading the data to be verified. The mechanisms fall in two categories: homomorphic digital signatures, that provide *public verifiability* (anyone can perform the integrity verification); and homomorphic message authentication codes, that provide *private verifiability* (only the user that possesses the secret key can verify integrity). Therefore, independently of the size of the data to be verified, integrity verification with these mechanism requires downloading a small proof, with the associated low communication delay and negligible cost.

5.1 SafeAudit

SafeAudit leverages *homomorphic digital signatures* for integrity control of the stored data, and the computation resources of commercial clouds infrastructures for executing code and generate compact integrity proofs based on the data and signatures present in the cloud storage. By requesting and verifying those small proofs, cloud-backed applications

can perform storage integrity control without being constrained with network bandwidth limitations or downloading large quantities of data.

5.1.1 Threat model and assumptions

SafeAudit was designed under a threat model where attackers have full permissions to access the storage cloud and perform any operation on the users' data, particularly the operations that compromise integrity: write and delete. Under this scenario an attacker can be: an external entity that managed to bypass the cloud's access control mechanisms and has obtained remote root access to one or more cloud storage machines; or an internal entity who is trusted by the cloud and authorized to have physical access to the machine (e.g., a cloud's employee), has obtained control of one or more storage machines and, moved by malicious intent, performs several operations that compromise integrity of the stored data. Also it is assumed that all the attackers fingerprints have been erased and that the cloud either has no knowledge of the attack or is hiding it from the user and auditor.

Since the purpose of SafeAudit is to detect cloud integrity attacks, this software library is based on the assumption that the only way the attackers can compromise the users' data is by attacking the cloud. This assumption was made to isolate the threat model from problems related with network or identity spoofing attacks, which are outside of the scope of this section. To do so, the threat model assumes that all communication between entities is authenticated and secure at all times (e.g., all entities communicate through *HTTPS* and use certificates signed by certificate authority trusted by all entities) and that neither the user or auditor suffer Byzantine faults, i.e. users and auditors are not malicious and their machine do not respond arbitrarily to the other entities' requests.

5.1.2 Preliminaries

SafeAudit is built on top of *multiplicative cyclic groups* and uses several pairing based cryptographic techniques, BLS homomorphic digital signatures [Lyn07] and the Shacham-Waters (SW) integrity verification scheme [SW08]. In this section some mathematical background is provided and the aforementioned cryptographic techniques will be summarized for better understanding of the remaining of the section.

5.1.2.1 Multiplicative cyclic group

A *cyclic group* is composed by members that are generated by a single *group generator* element g. In a *multiplicative cyclic group* every member is generated by raising the generator g with integers belonging to Z (the set of positive and negative integers). Multiplicative cyclic groups can be finite of infinite. The infinite ones are generated by raising with unbounded integers from Z. The finite ones of order n are generated by raising g with a bounded set of integers belonging to Z that are modulo of p (also called group order p). For better understanding consider the example illustrated in Figure 5.1, where a multiplicative cyclic group of order 6 and generator $g = 2$ is represented. The multiplicative group is composed of six members $[g^0 = 1, g^1 = 2, g^2 = 4, g^3 = 8, g^4 = 16, g^5 = 32]$, and

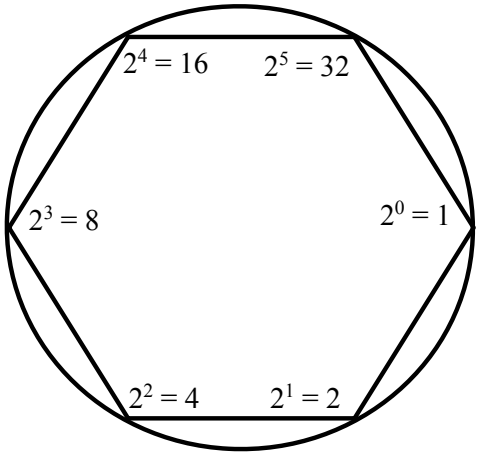

Figure 5.1: A multiplicative cyclic group representation of order 6 with 2 as its generator .

linear operations over members of the group are mapped as follows:

- $g^x = g^{xmod6}$, for example $g^6 = g^0 = 1$ and $g^7 = g^1 = 2$
- $g^x \times g^y = g^{(x+y)mod6}$, for example $g^1 \times g^2 = g^3 = 8$ and $g^7 \times g^8 = 8$

Due to their modular nature, the finite multiplicative cyclic groups can represent large numbers of unbounded size into finite group elements. SafeAudit relies on this technique to represent data and signatures of unbounded sizes into small sized group elements and uses them for creating compact proofs.

5.1.2.2 Pairing-based cryptography

In SafeAudit all the cryptographic techniques are built using pairing-based cryptography in order to preserve homomorphism in all operations. In this type of cryptography, each cryptographic function uses a pairing e (also called bilinear map) to convert a multiplicative cyclic group (G) of prime order p, generated with the number g, into another multiplicative cyclic group (G_T) of the same prime order (p), i.e., $e : G \times G \rightarrow G_T$. By using the pairing, the following properties are ensured: *Computabillity:* there exists an efficient algorithm to compute the pairing; *Bilinearity:* for all u, v belonging to G, a, b belonging to Z_p and pairing $e : G \times G \rightarrow G_T$, it is guaranteed that $e(u^a, v^b) = e(u, v)^{ab}$.

5.1.2.3 BLS signature scheme

In order to provide integrity control of a file SafeAudit uses the *BLS signature scheme* [BB04] for constructing digital signatures over pairing based cryptography. To do so, integrity control assumes the following steps:

- *Setup:* Choose two distinct multiplicative cyclic groups G and G_T of order p, and a generator g for G and generate pairing $e : G \times G \rightarrow G_T$.
- *Key Generation:* Using g compute an asymmetric secret/public key pair $sk \in Z_p$ and

$pk \in G$. First compute sk, by selecting a random number that belongs to Z_p and then generate pk as g^{sk}.

- *Sign:* Sign the data $d \in Z_p$ using the secret key sk belonging to Z_p and by computing the signature $\theta = d^{sk}$ belonging to G.

- *Verify:* Using the public key $pk \in G$, the pairing e and the generator g, verify the signature $\theta \in G$ of the data $d \in Z_p$ by testing the following hypothesis: $e(\theta, g) = e(d, pk)$. If the hypothesis verifies the integrity is assured.

5.1.2.4 Homomorphic verifiable integrity proofs

The use of BLS signatures ensures the homomorphic property for integrity verification and consequently allows the construction of homomorphic verification schemes, where data and signatures are aggregated using additions and multiplications into compact verifiable proofs. This is done because if each file and signatures can be divided into blocks of a given size (e.g., 128 bits) and these blocks can be mapped into multiplicative cyclic groups with *order* $= 2^{size}$ (e.g., 128 bits will generate group $0...128$), multiplications and additions will always produce elements of the same group. Thus, files and signatures of unbounded size can be aggregated into compact structures of the multiplicative cyclic group (e.g., a file with 10^6 bits is divided into 128 bits blocks mapped to multiplicative cyclic group and multiplied each block, and therefore producing 128 bit aggregation structure that represents the 10^6 bits file). In SafeAudit the SW integrity verification scheme [SW08] is used in order to provide homomorphic generation and verification of compact integrity proofs. To do so, under this scheme, integrity control assumes the following steps:

- *Setup:* Choose two distinct multiplicative cyclic groups G and G_T of order p, and a generator g for G and generate the pairing $e : G \times G \rightarrow G_T$.

- *Key Generation:* Using e and g, compute: a signature parameter w, by selecting a random number that belongs to G; and an asymmetric secret/public key pair $sk \in Z_p$ and $pk \in G$. First compute sk, by selecting a random number that belongs to Z_p and then generate pk as g^{sk}.

- *Sign block:* Given a block with the identifier $id \in Z$ and the corresponding block's data $d_{id} \in Z_p$, a hash function that maps $H : Z \rightarrow Z_p$, the secret key $sk \in Z_p$, and the signature parameter w, compute the signature $\theta_{id} = (H(id) \times w^{d_{id}})^{sk} \in G$.

- *Proof Generation:* Given a collection of block ids $id_1...id_n \in Z$, the corresponding data $d_1...d_n \in Z_p$ and numerical challenge vector of random numbers $chal_1...chal_n \in Z_p$, compute the integrity proof:
  $\alpha = \sum_{i=1}^{n} d_i \times chal_i \in Z_p$ and $\beta = \prod_{i=1}^{n} \theta_i^{chal_i} \in G$.

- *Proof Verification:* given the proof (α and β), the ids $i...n$, the public key $pk \in G$, the signature $\theta \in G$, the pairing e, the generator g, and the signature parameter w, verify by applying pairing that:
  $e(\beta, g) = e(\prod_{i=1}^{n} H(id_i) \times w^{\alpha}, pk)$
  If verification is positive integrity is assured.

5.1.3 SafeAudit's interaction protocol

In order to preserve the integrity of the data stored on the cloud using SafeAudit, the entities involved (cloud, user and auditor) need to follow the SafeAudit's interaction protocol. As will be explained in the rest of this subsection, the interaction protocol is divided into four parts: set up (Section 5.1.3.1), store data (Section 5.1.3.2), request and verify of integrity proof (Section 5.1.3.3), and proof generation (Section 5.1.3.4).

5.1.3.1 Setup

In order to setup integrity verification with SafeAudit the users and the auditor must perform the following interaction protocol steps before storing any data in the cloud:

- The user and the auditor exchange knowledge. The auditor provides two files[1] to the user for setting-up pairing-based-cryptography: the '.param' file with all the secure public initialization parameters needed for configuring cyclic groups G, G_T and the pairing for mapping $G \times G \rightarrow G_T$; and the '.g' file with generator g of the cyclic group G. The user provides information to the auditor about the amount of money the user wants to pay for audit, the time when each audit should be performed (e.g., daily, weekly, ...) and which data is the most critical to be verified.

- The user generates his secret/public asymmetric key pair and the signature parameter (w) for signing and verifying data under the SW scheme, using respectively the SafeAudit's key and random number generators (further explained in Section 5.2).

- The user shares the public key and w with auditor and stores w on the cloud.

- The user configures the cloud for listening to the auditor's requests and for responding to them, with the execution of SafeAudit's proof generator service (further explained in Section 5.2).

After these steps are performed users can now store their data in the cloud.

5.1.3.2 Store data

When the user stores data in the cloud, all data must be divided into blocks belonging to Z_p and signed. The SafeAudit's signature generator (further explained in Section 5.2) automates these tasks and produces a signature equivalent to the SW sign block step (as described in Section 5.1.2.4). To do so, the client provides as input, for the signature generator, the data and its id (e.g., the file content of the 'data.txt' file is used as the data and the id is the filename 'data.txt'), alongside with the pairing cryptography parameters ('.param' and '.g' files), secret key ('.sk'), and the signature parameter ('.w'), and obtains the signature of all the data blocks.

After the signature of the data is obtained, the user stores both the data and signature in

[1] *Data structures* would be probably a better expression than *files*, but we believe the word *file* is easier to understand and in our implementation they are indeed files.

the cloud.

5.1.3.3 Requesting and verifying integrity proofs

In SafeAudit's iteration protocol, the auditor is responsible for integrity verification. To do so, whenever the auditor wants to obtain integrity proofs of a dataset stored on the cloud, he must perform the following steps:

- Select a dataset composed of x data elements (vector $[0, ..., x]$), so that the cost of obtaining the proof for the x elements is, at most, the price the user wants to pay for the audit.
- Generate a random challenge (number belonging to Z_p) for each of the x data elements chosen, using the SafeAudit's random number generator.
- Issue the integrity proof request to the cloud specifying: identifiers vector ($[id_0, ..., id_x]$) and the corresponding challenge vector ($[chal_0, ..., chal_x]$).
- Upon receiving a response from the cloud, with the requested integrity proof, the auditor verifies it using the SafeAudit's proof verifier (further explained in Section 5.2). The auditor provides the public key pk and the signature parameter w, alongside with the ids and challenges used on the integrity request; and obtains the integrity verification result. Using SafeAudit's proof verifier for performing the verification test corresponds to the proof verification step of the SW scheme.

5.1.3.4 Generating integrity proofs

Whenever the cloud receives an integrity proof request of a given dataset, the cloud performs the following steps:

1. Fetch all dataset's data and signatures from the storage cloud corresponding to the ids specified.
2. Fetch from the storage cloud, the pairing cryptography parameters ('.param' and '.g'), and the signature parameter ('.w'), of the user requested.
3. Generate integrity proof, composed of: the aggregation of signatures provided (β); and the aggregation of data provided (α), by using SafeAudit's proof generator (further explained in 5.2.5). The generator receives data, setup parameters ('.g' and '.param'), signatures, challenges, pairing cryptography parameters and the random initialization parameter related to the dataset; and produces the α and β. This step corresponds to the proof generation step of the SW scheme.
4. Respond to the requester with the integrity proof (α and β).

5.1.4 SW signature size optimization

The size of the block signatures produced by SafeAudit is equal to the size of the multiplicative cyclic group G stipulated by the auditor (e.g., if G is equal to 128 bytes then the block signatures are also 128 bytes). Also, the size of the group G is determined by the elliptic curve selected for its initialization and are always larger than the integers used for its generation Z_p. For example, when a type A elliptic curve [Lyn07] is used for the generation of multiplicative cyclic groups, with the recommended sizes where G and G_T are 128 bytes and Z_p is 20 bytes, the signatures produced are 6.4 times bigger than the original file, raising the storage costs also to 6.4 times higher.

Since SafeAudit is intended to be used in commercial clouds, this library strives to lower the monetary costs paid for integrity verification. To do so, the size of the signatures used needs to be optimized to the minimum possible for reducing the additional monetary storage costs paid for storing them on the cloud. To do so, two optimizations were made on the original SW scheme.

The first optimization is the selection of the pairing curve that produces the shortest multiplicative cyclic groups, which is the pairing-friendly elliptic curves of prime order [BN05] (also named type F curves and described in [BB04]), as recommended by both BLS and SW authors in [SW08] and [Lyn07]. This optimization allows the creation of multiplicative cyclic groups G that are twice the size of the original data Z_p, and thus produces signatures twice the size of data.

The second optimization is to incorporate a signature compression in SafeAudit using the *point compression* technique described in [Lyn07]. This optimization comes from the fact that the multiplicative cyclic group G, where the signature belongs, is a two coordinate point (x, y) where y is one of the possible results of applying the elliptic curve function selected for pairing initialization. Due to this fact the y coordinate of the signature can be computed solely based on the x coordinate, the elliptic function, and a one bit value indicating which of the possible values to select. Thus, the y coordinate can be completely discarded, and the signature is compressed always by half of the original size and represented by its x coordinate and the one bit value necessary to recompute the y coordinate. This optimization allows signatures to have half of the expected size of applying the signature step of SW scheme and in the best case where type F elliptic curves [BN05] are used are the same size of the original data.

With these two optimizations SafeAudit is able to produce signatures the same size of the original data, which is the lowest possible using an homomorphic signature scheme.

5.2 SafeAudit's implementation

In order to simplify integration with the users' cloud-backed applications, commercial clouds, and auditors, the SafeAudit software library is composed of several components, each one implementing a task of the SafeAudit interaction protocol. The *Pairing Generator* component, allows auditors to generate all the setup parameters, required to initialize pairing-based-cryptography. The *Key Generator* component allows users to generate their

asymmetric secret/public key pair and signature parameter (w). The *Signature Generator* component allows users to sign their data. The *Random Generator* component allows entities to generate random numbers belonging to any field of their choosing (Z_p, G or G_T). The *Proof Generator* component allows clouds to generate integrity proofs. The *Proof Verifier* component allows auditors to verify the proofs obtained from the cloud. The rest of the section explains each component.

5.2.1 Pairing generator

This component allows auditors to construct setup parameters ('.param' and '.g') for initializing pairing based cryptography, according to the their security specification.

Auditors provide as input the type of pairing curve to be used for pairing generation ($type = A|B|C|D|E|F$)[2], and the parameters needed for initializing the curves.

The Pairing Generator outputs: a specifier file ('.param') detailing all the information about the multiplicative cyclic groups G and G_T, the integer range of the Z integers used for generating elements, and the pairing specifications for mapping G to G_T; and the generator file '.g'containing the absolute value of the element used for generating the multiplicative group G.

5.2.2 Key generator

The Key Generator component allows users to generate their own asymmetric key pair and signature parameter according to the security information provided by the auditor. The generated keys can be further used for the BLS and SW schemes.

The generator works as follows: the user inputs the setup parameters provided by the auditor '.param' and '.g'; the component initializes the pairing ; generates the secret key by selecting a random number belonging to Z_p; generates the public key by computing g^{sk}; generate the signature parameter by selecting a random number belonging to $G$and returns the keys and w to the user .

5.2.3 Signature generator

The Signature Generator component allows clients to sign their data using the signing step of SW scheme and compute the digital signatures.

In the SW scheme, the data to be signed is assumed to have fixed sizes and belongs to Z_p. To support data sizes bigger than original data, users have to divide the data in blocks that belong to Z_p, and sign each block individually. In order to automate data division into Z_p data blocks and sign each of them with the SW scheme, Signature Generator supports two signing modes: the Sign Block mode, for signing individual data blocks in Z_p; and the Sign Data mode, that converts all the input data to one or several blocks $\in Z_p$, signs each block

[2] See Section 4 of [Lyn07] for more information about the pairing curves and their selection

using Sign Block component, and returns the concatenation of all generated signatures from the blocks.

5.2.3.1 Sign block

The Sign Block mode works as follows: the user inputs the setup parameters provided by the auditor '.param' and '.g', the block d, id of the block id_d, the secret key sk and the signature parameter w; the component initializes the pairing; hashes the id to Z_p; multiplies the id's hash with w^d; signs the multiplication with the user's secret key; and returns the signature of the block.

5.2.3.2 Sign data

The Sign Data mode works as follows: the user inputs the setup parameters provided by the auditor '.param'and '.g', the data d, id of the data id_d, the secret key sk and the signature parameter w; the component initializes the pairing; the component divides the data into a vector of blocks that belong to Z_p; signs each block individually with an unique id; concatenates the blocks' signatures into one; and returns the concatenation.

5.2.4 Random number generator

This component allows generation of random numbers belonging to any of Z_p, G or G_T fields. To do so, this generator receives as inputs the desired field, the pairing '.param'and the '.g' and outputs the random number.

5.2.5 Proof generator

The Proof Generator component allows clouds to generate integrity proofs with the information they have stored whenever an auditor requests them. To do so, the algorithm first initializes pairing with the setup parameters , calculates alpha and beta based on the data's blocks present in the dataset.

5.2.6 Proof verification

The Proof Verifier component allows users to verify integrity proofs, using the SW proof verification step. To do so, the algorithm first initializes pairing with the setup parameters ('.param' and '.g'); applies g pairing to beta, multiplies all ids present in the proof with w^{alpha}, applies public key pairing to the id and alpha multiplication and verifies if both pairings obtained match. If so, the data integrity is preserved.

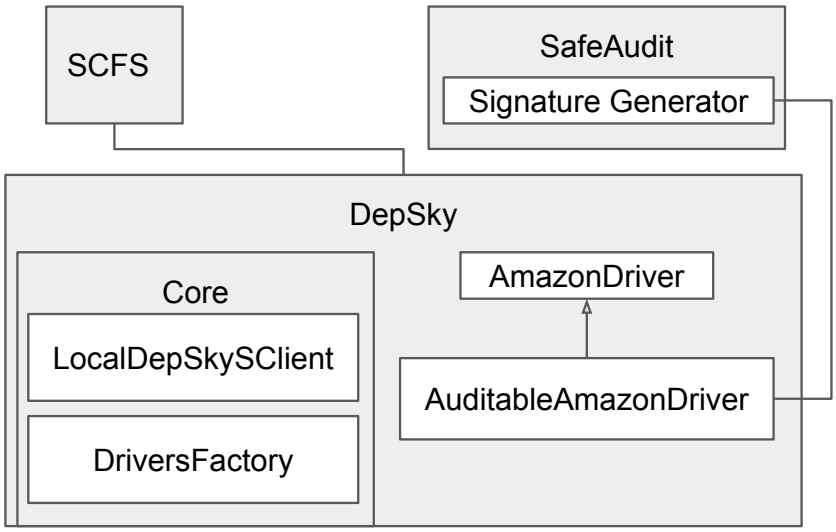

Figure 5.2: Components modified for integrating SAFEAUDIT in DepSky.

5.3 Integration in SafeCloud-FS

The SafeAudit library code was developed in Java and all the pairing-based cryptographic mechanisms used in SafeAudit were implemented using the Java Pairing-Based Cryptography Library (JPBC) [DI11], that implements Multi-linear Maps and all the operations required by these mechanisms for manipulating those maps.

At the client-side, SafeAudit is integrated with DepSky components. More specifically, in DepSky the logic for communicating with different commercial clouds is implemented in subcomponents called *cloud drivers*. In Depsky these components read and write files in the storage clouds. With SafeAudit, *auditable cloud drivers* sign files using the SafeAudit's signature generator and the signature is also stored on the cloud. As seen in Figure 5.2, for integrating these new drivers, DepSky suffered changes in two packages: core and drivers. Code was added to the core package of DepSky, in the DepSky's initialization function (present on *LocalDepSkySClient.java*) and to the DepSky's driver constructor function (present on *DriversFactory.java*).

To use SAFEAUDIT, SCFS has to be configured with what we call *auditable cloud drivers*, which implement our system's logic. For instance, to use Amazon S3 as cloud storage, instead of using the original (non-auditable) driver amazon-s3, the corresponding auditable driver *auditable-amazon-s3* shall be used. Users can choose which drivers to use, by modifying the configuration file with the name of the desired drivers. The DepSky's initialization function automatically reads the user's secret key, the setup parameters (.param and .g) and the signature parameters (.w) provided by the auditor; and uses the initialization function of DepSky driver for initializing the driver with that information. Regarding the driver package, the auditable drivers extend the non-auditable drivers. Whenever data is uploaded to the commercial cloud using the auditable driver, data is signed by using SafeAudit's sign data component and then stored both signature and data on the commercial cloud by invoking the superclass' non-auditable driver upload data function.

To run the verifier in the clouds the use of a service like AWS Lambda may be used[3]. By using services like Lambda instead of the more traditional alternative of keeping an EC2 virtual machine running, users only pay for computing resources during the code execution, not paying for idle time. Therefore, users can leave the system always on and ready to execute the auditors proof generation request without additional monetary costs. This is not possible in computing services like EC2 since monetary costs are charged from the moment the machine boots until it is completely shut down.

5.4 Summary

In this section, the SafeAudit software library was proposed. This software library was designed to be easily integrated with the current remote storage solutions, including solutions that store data with cloud-backed applications on commercial clouds, and automate all the tasks involved in storage integrity control, including signature generation and verification.

[3] https://aws.amazon.com/lambda/

6 Conclusion

This deliverable presents SafeCloud's cloud-backed secure file system, SafeCloud-FS. This system is based on SCFS, a modular cloud-backed file system.

SafeCloud-FS contains two new features. First, it stores metadata encrypted, hiding from the clouds information such as the names of the files, the directory tree, and their timestamps (time of creation and of the last change).

Second, it allows verifying the availability and integrity of the files stored at the cloud without downloading it.

These new mechanisms are based on 3 new components: the MorphicLib library, the HomomorphicSpace coordination service, and the SafeAudit service.

Bibliography

[AKK09] Giuseppe Ateniese, Seny Kamara, and Jonathan Katz. Proofs of storage from homomorphic identification protocols. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 319–333. Springer, 2009.

[BACF08] A. N. Bessani, E. P. Alchieri, M. Correia, and J. S. Fraga. DepSpace: a Byzantine fault-tolerant coordination service. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Systems Conference*, pages 163–176, April 2008.

[BB04] Dan Boneh and Xavier Boyen. Short signatures without random oracles. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 56–73. Springer, 2004.

[BCLO09] Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O'Neill. Order-preserving symmetric encryption. In *Proceedings of the 28th Annual International Conference on Advances in Cryptology: The Theory and Applications of Cryptographic Techniques*, pages 224–241, 2009.

[BCQ+13] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. Depsky: dependable and secure storage in a cloud-of-clouds. *ACM Transactions on Storage (TOS)*, 9(4):12, 2013.

[BJO09] Kevin D Bowers, Ari Juels, and Alina Oprea. Hail: a high-availability and integrity layer for cloud storage. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 187–198. ACM, 2009.

[BMO+14] Alysson Bessani, Ricardo Mendes, Tiago Oliveira, Nuno Neves, Miguel Correia, Marcelo Pasin, and Paulo Verissimo. Scfs: a shared cloud-backed file system. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 169–180, 2014.

[BN05] Paulo SLM Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. In *International Workshop on Selected Areas in Cryptography*, pages 319–331. Springer, 2005.

[Che16] Roger Cheng. Tech industry rallies around Apple in its iPhone fight with FBI. CNET. https://www.cnet.com/news/iphone-fight-against-fbi-tech-groups-industry-leaders-throw-support-behind-apple/, March 2016.

[Clo13] Cloud Security Alliance. The notorious nine: Cloud computing top threats in 2013, February 2013.

[DBB+15] Tobias Distler, Christopher Bahn, Alysson Bessani, Frank Fischer, and Flavio Junqueira. Extensible distributed coordination. In *Proceedings of the 10th ACM SIGOPS/EuroSys European Systems Conference*, pages 10:1–10:16, 2015.

[dC14] Nuno Tiago Ferreira de Carvalho. A practical validation of homomorphic message authentication schemes. Master's thesis, University of Minho, 2014.

[Der13] Masih H. Derkani. Hypergeometric.java. `https://github.com/masih/sina/blob/master/src/main/java/DistLib/hypergeometric.java`, 2013.

[DI11] Angelo De Caro and Vincenzo Iovino. jPBC: Java pairing based cryptography. In *Proceedings of the 16th IEEE Symposium on Computers and Communications, 2011*, pages 850–855. IEEE, 2011.

[ENI14] ENISA. Algorithms, key size and parameters report – 2014, November 2014.

[ErJ01] D Eastlake 3rd and Paul Jones. US secure hash algorithm 1 (SHA1). Technical report, 2001. Accessed: 2016-05-29.

[Gel85] David Gelernter. Generative communication in Linda. *ACM Transactions on Programing Languages and Systems*, 7(1):80–112, January 1985.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing*, pages 169–178, 2009.

[GHR+10] E. Grosse, J. Howie, J. Ransome, J. Reavis, and S. Schmidt. Cloud computing roundtable. *IEEE Security Privacy*, 8(6):17–23, 2010.

[Goo17] Google. Google infrastructure security design overview – google cloud whitepaper, January 2017.

[Has09] Omar Hasan. Paillier.java. `http://liris.cnrs.fr/~ohasan/pprs/paillierdemo/Paillier.java`, 2009.

[Hau16] Laura Hautala. The Snowden effect: Privacy is good for business. CNET. https://www.cnet.com/news/the-snowden-effect-privacy-is-good-for-business-nsa-data-collection/, June 2016.

[HW90] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[KL07] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography: Principles and Protocols*. Chapman & Hall/CRC, 2007.

[KS85] V. Kachitvichyanukul and B. Schmeiser. Computer generation of hypergeometric random variates. *Statistical Computation and Simulation*, 22:127–145, 1985.

[Lyn07] Ben Lynn. *On the implementation of pairing-based cryptosystems*. PhD thesis, Stanford University, 2007.

[Mar14] Markets and Markets. Cloud security market (cloud IAM/IDAASS, DLP, web security, email security, cloud IDS/IPS, SIEM, encryption services, BCDR, network security, cloud database security, virtualization security) - global advancements, forecasts & analysis (2014–2019), 2014.

[Mar16] Markets and Markets. Cloud storage market by solution (primary storage, disaster recovery & backup storage, cloud storage gateway & data archiving), service, deployment model (public, private & hybrid), organization size, vertical & region – global forecast to 2021, 2016.

[Mer79] Ralph Charles Merkle. Secrecy, authentication, and public key systems. 1979.

[Mic15] Microsoft. Trusted cloud: Microsoft Azure security, privacy, and compliance. http://download.microsoft.com/download/1/6/0/160216AA-8445-480B-B60F-5C8EC8067FCA/WindowsAzure-SecurityPrivacyCompliance.pdf, April 2015.

[P+15] Raluca Ada Popa et al. CryptDB webpage. `https://css.csail.mit.edu/cryptdb/`, 2015.

[PRZB11] Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, pages 85–100, 2011.

[Res16] Research and Markets. Cloud storage market insights, opportunity analysis, market shares and forecast 2016–2022, July 2016.

[SW08] Hovav Shacham and Brent Waters. Compact proofs of retrievability. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 90–107. Springer, 2008.

[Tra16] Transparency Market Research. Cloud security market - global industry analysis, size, share, growth, trends and forecast 2014–2022, 2016.

[Vog09] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[WLL15] Boyang Wang, Baochun Li, and Hui Li. Panda: public auditing for shared data with efficient user revocation in the cloud. *IEEE Transactions on services computing*, 8(1):92–106, 2015.

[WRLL10] Cong Wang, Kui Ren, Wenjing Lou, and Jin Li. Toward publicly auditable secure cloud data storage services. *IEEE network*, 24(4):19–24, 2010.

[WWRL10] Cong Wang, Qian Wang, Kui Ren, and Wenjing Lou. Privacy-preserving public auditing for data storage security in cloud computing. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9. Ieee, 2010.