SafeAudit

A Software Library for Efficient Data Integrity Verification on Commercial Clouds

Filipe Miguel Marcos Apolinário

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Miguel Filipe Leitão Pardal
Prof. Miguel Nuno Dias Alves Pupo Correia

Examination Committee

Chairperson: Professor António Manuel Ferreira Rito da Silva
Supervisor: Professor Miguel Filipe Leitão Pardal
Member of the Committee: Professor Alysson Neves Bessani

November 2016

Acknowledgments

I would like to thank my advisors Miguel Correia and Miguel Pardal for guiding me throughout the development of this thesis.

I would also, like to thank Karan Balu, Inês Araujo, André Joaquim, and Diogo Raposo for their support and for the feedback provided during the development of this work.

Finally, I would like to thank all my family and friends for their support during the development of this thesis.

This work was supported by the European Commission through project H2020-653884 (SafeCloud) and by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013 (INESC-ID).

Abstract

The most common integrity verification methods for data stored in remote storage use digital signatures or message authentication codes. These control structures allow checking that data in remote storage has not been modified. However, as the data size grows, these methods have scalability problems including high bandwidth consumption and latency, because all the data has to be downloaded to perform the check.

This dissertation explores homomorphic authentication with digital signatures to provide integrity verification of data stored in commercial cloud services without having to retrieve all of it and proposes the SafeAudit software library. SafeAudit automates integrity verification on cloud storage services and supports integration with cloud-backed applications. The experimental results show that integrity verification with SafeAudit requires low network bandwidth consumption and has low latencies, at the expense of higher computational load on the verifiers. This combination produces significant savings in monetary cost for cloud use while assuring the desired data integrity.

Keywords: Cloud storage, Integrity verification, Homomorphic authentication, Homomorphic signatures, Homomorphic encryption, Security

Resumo

Os métodos mais comuns de verificação da integridade de dados armazenados em servidores remotos usam assinaturas digitais e códigos de autenticação de mensagem. Estas estruturas de controlo permitem comprovar que os dados dos servidores remotos não foram modificados. No entanto, devido ao facto de todos os dados terem de ser descarregados para realizar a verificação, estes métodos têm problemas de escalabilidade quando o tamanho dos dados aumenta, incluindo consumo excessivo da largura de banda e latência.

Esta dissertação explora autenticação homomórfica com assinaturas digitais e apresenta a biblioteca de *software* SAFEAUDIT, de modo a realizar verificação de integridade de dados armazenados em serviços de nuvem comerciais sem ter que descarregar toda a informação. SAFEAUDIT automatiza a verificação de integridade das nuvens de armazenamento e suporta integração com aplicações que utilizam a nuvem para guardar dados. Os resultados experimentais mostram que utilização de SAFEAUDIT para a verificação de integridade requer um consumo baixo de largura de banda e latências diminutas através de um aumento na carga de computação exigida ao verificador. Esta combinação produz uma poupança significativa de custos monetários em nuvens comerciais e garante a verificação eficaz da integridade dos dados.

Palavras-Chave: Armazenamento em nuvem, Verificação de integridade, Autenticação homomórfica, Cifra homomórfica, Assinaturas homomórficas, Segurança

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Nowadays commercial cloud storage solutions are becoming widely adopted for storing data with acceptable monetary costs. In those solutions, users store their data in remote data-centers, and benefit from on-demand scalable growth of resources and worldwide accessibility [1]. The data-centers are managed by the cloud provider who is payed by its users for the resources offered. This allows users to achieve optimal storage monetary costs, only paying for actual resource usage with no additional management and allocation charges. Despite these benefits, users inevitably lose control of the fate of their data and it is up to the cloud provider to ensure storage security. If the cloud providers fail to protect the data-center, users may become vulnerable to attacks caused by access control breaches, which may compromise the users' data integrity and make them temporarily or eternally unavailable [2].

According to recent studies [3–5] there is a growth tendency of integrity attacks in public infrastructures, such as hospitals and other public enterprises, which are subjected to these attacks in order to pay to the attackers for the recovery of that information using ransomware and other blackmailing mechanisms. Integrity attacks may be classified into two categories: *external*, where an attacker penetrates the storage infrastructure and performs unauthorized modifications to the stored data; and *internal*, where an authorized individual (e.g., a current or former employee) accesses and performs unauthorized modifications to the data (e.g., to destroy the reputation of the service or to blackmail clients).

In order to minimize the impact of both internal and external attacks on the cloud storage, users need mechanisms for detecting unauthorized modifications performed on the storage infrastructure. Nowadays data owners resort to integrity control mechanisms to protect their outsourced storage, such as digital signatures for *collaborative storage environments* (where data is shared among several cloud users), and message authentication codes (MAC) for *private storage environments* (where data is used by a single cloud user). To do so, the users have some personal authentication metadata (asymmetric private/public key pair for digital signatures or a symmetric key for MAC) and before storing the data in the cloud: they create a short digest of the data (using a digest function, e.g., SHA-1 [6]); then use the authentication metadata to sign the digest (private key for digital signatures or the symmetric key for MAC); and finally store the data and the corresponding signed digest in the cloud. Whenever the user wants to guarantee that the integrity of the data is kept in the cloud, the user: downloads the data and the corresponding signed digest from the cloud; then verifies that the downloaded data's digest matches the signed digest using their personal authentication metadata (public key for digital signatures or symmetric key for MAC). If they both match, the data's integrity is preserved. If not, the data has suffered unauthorized changes and has integrity problems.

Notwithstanding the effectiveness of these mechanisms for detecting unauthorized modifications, they require downloading all the data from the cloud to verify its integrity. This implies large bandwidth consumption and massive read costs on huge data sets and leads to scalability problems for obtaining periodical integrity assurances on all the stored data. To better understand this scenario, consider a user with 1000 files stored on Amazon AWS [7] cloud located in Ireland, each with 1GB of size. If the user wants to weekly check the integrity of every file, each week the user has to read 1TB from the cloud. In this scenario the user is subjected to network latency costs for downloading 1TB and an additional charge of 900US$ for that download[1] [8].

In order to reduce the users' monetary charges and allow clients to obtain integrity assurances with low and constant bandwidth consumption costs, several works [2, 9–14] propose using new versions of these integrity mechanisms that, contrary to their previous versions, are *homomorphic*, i.e., the integrity control structures they produce have the same structure as the signed data. Due to homomorphism, these new mechanisms allow data and control structures to be aggregated by the remote storage cloud into a small sized proof that contains one aggregation of the data and another for the corresponding control structures, and provide *verifiability* (integrity of all the signed data can be verified using the proof) and *unforgeability* (any unauthorized modification to the proofs, data or control structures is always detected). These new versions are divided into two categories: the homomorphic digital signatures, that provide *public verifiability* (anyone can perform the integrity verification); and the homomorphic message authentication codes, that provide *private verifiability* (only the person who possesses the the secret key can verify it). To better understand the benefits of these homomorphic mechanisms for integrity verification in cloud storages, consider the previous example, where the user has 1000 files stored on Amazon AWS, each with 1GB of size. If the user uses an homomorphic mechanism (e.g., homomorphic digital signatures with a 40 byte public key) for verifying the integrity of all the files in the storage, the user would only have to download a small proof (e.g., 60 bytes) from the cloud and verify with his authentication metadata (e.g., the 40 byte public key) that the aggregation of the files match the aggregation of the integrity control structures. If they both match, integrity is preserved. If they do not match, some of the files have been tampered. Thus, as the data grows, integrity verification under these mechanisms always provides low and constant bandwidth consumption costs and almost no monetary charges for downloading the proof (e.g., downloading 60 bytes of data costs is negligible).

Contrary to prior works that explore the potential of compact integrity proofs by presenting theoretical demonstrations of their feasibility and security analysis [2, 9–14], this thesis explores practical applicability of these techniques by presenting a fully deployable system, SafeAudit, capable of being integrated as-is on real world storage solutions, including commercial clouds and cloud-backed applications. Also contrary to the prior works that never evaluated the monetary costs paid for applying these techniques on commercial clouds, this thesis is the first to quantify the monetary costs involved on using homomorphic authentication on commercial cloud storage solutions.

The rest of this thesis presents SafeAudit; a software library that implements the Shacham-Waters (SW) integrity verification scheme introduced in [14] and adapts it to commercial cloud solutions. SafeAudit optimizes the original SW scheme and provides: an overall performance increase, by carefully selecting *pairing-friendly elliptic curves* [15] for the SW scheme's parametrization; and a storage cost decrease by half of the demanded by the original scheme, using the *point compression* technique presented in [16]. Contrary to the prior works that explore homomorphic integrity verification, SafeAudit was designed as a practical implementation that can be easily plugged to current commercial cloud services and cloud-backed applications and it is the first homomorphic library to do so. Also, contrary to these previous works, SafeAudit is simple to use and requires little knowledge of advanced cryptog-

[1] Considering the monthly download of 10TB, every 1GB read is charged 0.09US$ which implies the 900US$ fee per integrity check.

raphy in order to fully understand its usage. To demonstrate its full potential, this library was integrated with Amazon AWS commercial cloud and the SCFS [17] cloud-backed storage application. The results obtained on the demonstration show that SafeAudit requires lower monetary costs (30% less) and lower bandwidth consumption costs than performing integrity verification with the commonly used non-homomorphic integrity control mechanisms (such as RSA digital signatures [18]), that require clients to download all the data from the cloud storage.

1.2 Contributions

The main contributions of this dissertation are: the design and implementation of the SafeAudit integrity verification library; an interaction protocol for verifying data stored in remote clouds using SafeAudit; a proof of concept analysis of integrating SafeAudit with a commercial cloud and a cloud-backed storage application; and an evaluation of the results obtained using this library as standalone, or integrated with the SCFS and the Amazon AWS commercial cloud.

This dissertation also contributes for identifying the situations where homomorphic integrity verification schemes are preferable against non-homomorphic integrity verification schemes for reducing the overall monetary costs paid by the user when integrated on commercial cloud solutions.

1.3 Structure of the Document

The remaining of this dissertation is structured as follows: Chapter 2 compares the overall contributions of this thesis with the current works present in the state of the art; Chapter 3 explains the design and background concepts that were applied on the SafeAudit software library; Chapter 4 covers the implementation of SafeAudit stating its principal algorithms and their functionality, and demonstrates the compatibility between this library and the current cloud-backed applications by integrating it with the SCFS application; Chapter 5 presents the evaluation and discussion of using this library in commercial cloud solutions as standalone or integrated with other integrity protection mechanisms; and Chapter 6 concludes the dissertation.

Chapter 2

Related Work

Integrity protection on storage clouds can be classified into three phases: the prevention phase where users use mechanisms such as access control restrictions to prevent attackers from performing integrity attacks; the surveillance phase where storage is continually inspected for detecting any ongoing attacks; and the reaction phase where the user tries to stop the attack and restore the integrity of the affected data. SAFEAUDIT is an integrity verification to be used on the surveillance phase for detecting data that was corrupted by any ongoing or previously performed integrity attacks.

SAFEAUDIT's design was inspired by several integrity verification mechanisms [2,9–14,17,19,19–26]. Also, in order to allow users to benefit from all the three integrity protection phases, SAFEAUDIT was developed to have full compatibility with several prevention and reaction mechanisms.

The remainder of this chapter, presents the several works that influenced the SAFEAUDIT's design and assumes the following structure: Section 2.1 presents the currently available integrity verification mechanisms studied for SAFEAUDIT's development; Section 2.2 explains the several techniques that can be combined with SAFEAUDIT or other integrity verification mechanisms for raising integrity protection on storage clouds. Section 2.3 shows several currently available cloud-backed storage applications that were taken into account for allowing SAFEAUDIT to be compatible with a wide range of cloud-backed applications. Finally, Section 2.4 provides a summary of the concepts addressed throughout this chapter.

2.1 Integrity Verification Mechanisms

SAFEAUDIT is inserted on the integrity verification research area. More precisely, it is closely related with mechanisms that allow users to perform integrity verification on cloud storage. The studied mechanisms can be divided into two categories: those that perform integrity verification with a voting scheme; and the others that perform it using authentication metadata.

2.1.1 Integrity Verification with Voting Schemes

In systems involving voting schemes for integrity verification, such as [19], users store data on several clouds, where each cloud can be subjected to integrity attacks, but the majority is assumed to be unaffected. To verify data's integrity, the user retrieves the data from several clouds (each data collected represents a vote), until the same data content (same vote) was collected from the majority of clouds or all of the clouds responded to the user (there is a tie). If the user is able to obtain a majority of responses from the clouds with the same votes, data integrity is preserved on the majority of the clouds, and any cloud who responded with a different vote has integrity problems. If there is no majority of votes, the

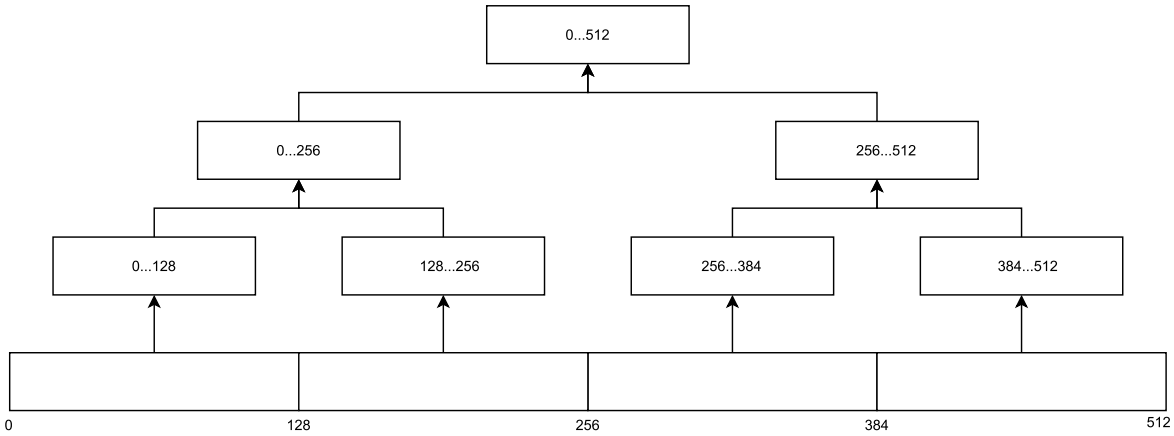

Figure 2.1: Example of an MHT being created over a dataset that contains four datablocks, each with 128 bits of size.

integrity is not preserved and the data is not retrievable. With this method besides being able to verify integrity, users can pinpoint which clouds were compromised and react accordingly. However since it requires reading data from several clouds, the user is more vulnerable to network latencies and high bandwidth consumption costs than simply storing data in one cloud, since multiple copies of the data are downloaded from different remote locations. Furthermore, when applied to commercial clouds there is a substantial increase of read monetary costs associated with the multiple read requests performed to the several clouds.

2.1.2 Verifying Integrity with Authentication Metadata

In systems involving authentication metadata for performing integrity verification on the cloud, such as [2, 9–14, 17, 19–26], the user is required to intervene in two phases. The first phase starts when storing data, the user first computes an integrity control data-structure (digital signature or MAC) based on the data and its authentication metadata (symmetric key in case of MAC or private key in case of digital signatures); and then it stores both the data and integrity control data-structure on the cloud. The second phase starts when the user requests an integrity proof. In this phase the cloud provides the proof containing both the data and control data-structures, and the user performs the verification using his authentication metadata (symmetric key in case of MAC or public key in case of digital signatures). Depending how these proofs are constructed, they can be categorized as non-homomorphic or homomorphic.

2.1.2.1 Non-homomorphic Integrity Proofs using Cryptographic Hashes

In the case of systems that use non-homomorphic proofs [17, 19–24], verifying the integrity of a dataset requires users to obtain each pair of data and integrity control structure (normally cryptographic hashes) separately and perform the verification individually. Due to that fact, using these proofs demands high bandwidth consumption for downloading the data, and a monetary fee charged by the cloud for reading the data. A possible way to reduce the amount of integrity control structures involved on the verification of a large dataset [26] is to use *Merkle Hash Trees (MHT)* [27].

MHT is a signature scheme that organizes a dataset in a binary tree form where the leaf nodes are the dataset's blocks and the root node is an hash of all those blocks, as exemplified in Figure 2.1. The traversal of the tree is done bottom-up where each parent is the result of hashing the concatenation of its

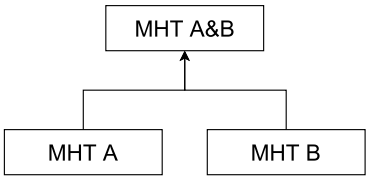

Figure 2.2: Example of an MHT built on top of two root nodes, one that authenticates the dataset A and the other the dataset B. The root node of this MHT authenticates both datasets.

children. One MHT root node that authenticates a given dataset can be combined with an another MHT root node that authenticates a different dataset to create one unique MHT root node that authenticates both datasets, by hashing the concatenation of the two root nodes, as exemplified in Figure 2.2. With the aggregation properties ensured in the MHT scheme, users can reduce the amount of integrity control structures, by storing one root node to authenticate the whole dataset. For example, consider a cloud-backed application that stores backups of an entire file system composed of several files on the cloud. In this example to reduce the amount of integrity control structures involved in the verification of a complete backup, instead of having an integrity control structure stored on the cloud for each file, users can store an MHT root node that authenticates all the files that belong to the backup. This way, when a backup is retrieved from the cloud, the integrity verification is performed on a single check, by reconstructing the MHT and comparing with the one retrieved from the cloud. Although this approach may be favoured, for confirming the integrity of large quantities of data read from the cloud, it still requires all the data to be downloaded from the cloud to perform the check. Therefore, when this approach is applied to cases where users are only interested in verifying the integrity of the stored data without particular interest on the information read, each verification demands users to perform an unnecessary download of large quantities of data from the cloud to perform the check. In these cases a more economic solution, that reduces the amount of data read from the cloud while assuring the same integrity guarantees, can be obtained using homomorphic integrity proofs.

2.1.2.2 Homomorphic Integrity Proofs

In case of the systems that use homomorphic proofs, such as [2, 9–14] or any other who uses SAFEAUDIT (e.g., the proof of concept presented in Chapter 4), the integrity proofs obtained is composed of two small aggregation structures, one that compresses all data and the other for the integrity control data-structures. This compression is performed on the cloud through code execution by solving linear computations, and this way allows proofs to be compact and requires less bandwidth consumption than the systems who use non-homomorphic proofs or voting schemes for integrity verification. However all the mechanisms studied [2, 9–14] were not yet practical to use and no work had project the monetary costs paid for applying these mechanisms on commercial clouds. SAFEAUDIT is the first to prove it is possible integrate with conventional cloud-backed applications and also project the monetary costs involved on using these proofs on cloud storage solutions.

Among the several articles studied that focus on using integrity verification with homomorphic proofs, the articles [14, 16, 28] served as inspiration for developing SAFEAUDIT. [16] explains how to build highly compact public verifiable digital signatures, the *BLS signatures*, that preserve homomorphism, as will be further explained in Section 3.3.3. [28] explains what are BLS signatures and how they should be constructed in practice. [14] explains the *Shacham-Waters integrity verification (SW) scheme* which explores the capabilities of BLS and integrates them in a protocol for obtaining homomorphic integrity proofs, as will be further explained in Section 3.3.4.

SAFEAUDIT provides a software library for any commercial cloud with code execution capabilities to

generate homomorphic integrity proofs, making it possible for applications to follow the example presented in [11] and benefit from using homomorphic integrity verification.

2.1.3 Proofs of Data Possession and Retrievability

In order to verify the integrity of storage clouds, without requiring high bandwidth consumption costs or having to pay high monetary costs for those verifications, some works [9, 14, 26, 29, 30] have proposed probabilistic mechanisms that periodically audit a dataset stored on the cloud, by strategically selecting a sample of blocks from the dataset for demanding the cloud to prove the integrity of those blocks. This way by diversifying the blocks audited, users can obtain probabilistic guarantees that their storage is kept safe. Depending on how these probabilistic mechanisms are constructed, they can be categorized as *proofs of data possession* or *proofs of retrievability*.

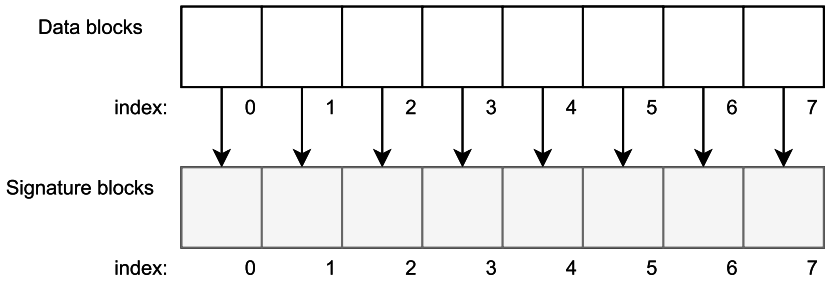

Figure 2.3: Example of authentication coverage when using proof of data possession mechanism. The grey coloured blocks represent signatures that can be used to verify the corresponding blocks.

Regarding proofs of data possession these mechanisms, that were first introduced by [9] and improved in [29], allow users to obtain probabilistic proofs that all their storage is kept safe in the cloud without having to download a large portion of data from the cloud to perform the check. To do so, all the data is authenticated with homomorphic integrity control structures, and upon each audit request, the user demands a proof of integrity for a given set of data blocks stored on the cloud. Clouds provide the proofs by compacting all the requested blocks and signatures into a small sized proof, that is then verified by the user, using his authentication key. As seen in Figure 2.3, since all data blocks are authenticated with integrity control structures, users can choose any combination of data blocks stored on the cloud to perform the audit. The bigger the amount of blocks selected to be audited the better the assurance of storage integrity provided.

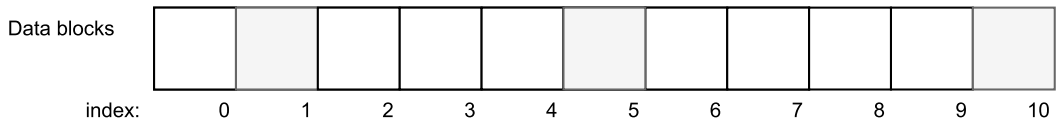

Figure 2.4: Example of authentication coverage when resorting to proof of retrievability mechanisms that uses additional sentinel blocks.The grey coloured blocks represent sentinels that are introduced on the data for integrity verification. When performing integrity verification only the grey coloured blocks can be checked.

Regarding proofs of retrievability introduced by [30] and improved in [14, 26], they are different from proofs of data possession mechanisms since only parts of the data are covered by the verification mechanism. As seen in Figure 2.4, instead of providing the complete verifiability of the data stored on the cloud, proof of retrievability mechanisms resort to additional random valued blocks, also called *sentinel blocks*, that are hidden on random positions of the data stored in the cloud. These blocks serve

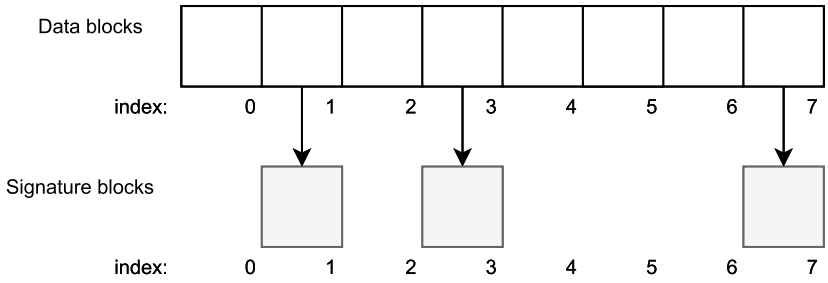

Figure 2.5: Example of authentication coverage when resorting to proof of retrievability mechanisms that uses integrity control structures. The grey coloured blocks represent signatures that can be used to verify the corresponding blocks. When performing integrity verification only the blocks that have corresponding signatures can be checked.

as traps for detecting data corruption in the cloud. Using these blocks and by keeping local copies on the verifiers' devices, proofs of retrievability can be obtained from the cloud, by revealing the position of a small sample of sentinel blocks and requesting their corresponding values. The proofs are then confronted with the content stored in the verifiers device and if they match storage integrity is probably kept. An alternative way to using additional sentinel blocks for building proofs of retrievability is, as seen in Figure 2.5, to use integrity control structures for providing authenticity of some blocks of the data stored on the cloud (these blocks assume the role of the sentinel block). In this approach, some strategic blocks of the original data are chosen as sentinels and for each block selected an integrity control structure is stored on the cloud. This way, verifiers do not have to store sentinel blocks on their device, and proofs of retrievability can be obtained, by revealing the position of a small sample of sentinel blocks and requesting their corresponding values and control structures, if they were not corrupted integrity is probably kept on all the nearby blocks.

The motivation for using proofs of retrievability instead of proofs of data possession is the economization of storage space. This is due to the fact that homomorphic integrity control structures, used in proof of data possession, normally consumes an high quantity of storage space (can be bigger than the data itself). Since proofs of retrievability do not store an integrity control structure for all the blocks stored on the cloud, only for the sentinels, the amount of additional storage used is reduced at the expense of reducing the detection coverage of the mechanism. Because of this trade-off, in order to detect effectively integrity attacks using proofs of retrievability, the position of the sentinel blocks needs to be carefully hidden and difficult to guess, for avoiding attackers to dodge those blocks when corrupting the data. To do so, proofs of retrievability mechanisms rely on obfuscation techniques, such as encryption of data, to mix sentinels with the common data blocks stored on the cloud, and pseudo-random functions to randomize the selection of positions where the sentinel blocks should be placed.

2.1.4 Discussion

In this section five integrity verification techniques were at focus: voting schemes; non-homomorphic verification schemes; homomorphic verification schemes; proof of data possession and proofs of retriveability. Table 2.1 presents a comparison of these techniques. As seen in this table, all mechanisms allow users to perform integrity verification with different benefits and limitations.

Voting schemes allow all users to verify data's integrity based solely on the data present on several clouds, without additional verification metadata, including integrity keys or additional integrity control structures (digital signatures or MAC). Due to that fact and since keys are not used, anyone can verify the data's integrity by simply downloading and comparing the data present on the several clouds,

Table 2.1: Comparison between using verification schemes, homomorphic/non-homomorphic authentication schemes, or the probabilistic methods, proof of data possession or proof of retrievability.

Scheme		Voting Scheme	Non-homomorphic Authentication	Homomorphic Authentication	Proof of Data Possession	Proof of Retrievability
Authentication Metadata		None, no integrity keys used	symmetric or assymetric keys	symmetric or assymetric keys	symmetric or assymetric keys	symmetric or assymetric keys
Cloud Code Execution		Not required	Not required	Required	Required [1]	Not required [2]
Commercial Cloud Costs	Verification	$\geq 2 \times \lvert data \rvert$	$\approx \lvert data \rvert$	$\leq 1Mb$, always constant	$\leq 1Mb$, always constant	$\leq \lvert data \rvert$
	Write	$\geq 3 \times \lvert data \rvert$	$\approx \lvert data \rvert$	$\geq 2 \times \lvert data \rvert$	$\geq 2 \times \lvert data \rvert$	$\approx \lvert data \rvert$[3]
Probabilistic		No	No	No	Yes	Yes

[1] Requires code execution since integrity verification is always based on homomorphic authentication schemes
[2] Only requires code execution if integrity verification is based on homomorphic authentication schemes.
[3] It may be bigger, depending on the amount of sentinels stored along with the data.

without having to worry about complex key management and distribution problems typically present on mechanisms that rely on authentication metadata. However, by applying voting schemes data must be present on at least three remote clouds, which results on higher monetary costs for storing data and for verifying the integrity, than the prices paid when authentication metadata is used.

On the other hand, the schemes that rely on authentication metadata require the users to worry about key management and distribution when working on collaborative environments. Nonetheless, the monetary savings provided in these schemes normally compensate the problems in hand, and may produce quite a substantial monetary reduction, depending on the type of authentication metadata chosen and its compatibility with the behaviour of the cloud-backed applications used for storing the data.

In the case of non-homomorphic authentication schemes, they are more compatible with cloud-backed applications that are write intensive and perform a low number of integrity verification checks, e.g. an online text editor that is constantly being updated on the cloud. This is due to the fact that they are write efficient with respect to the additional monetary costs associated with the integrity control structures stored on the cloud, which are despicable and constant as data grows (normally less than 1 MB of size regardless of the size of the data signed), but require monetary fees for verifying integrity that increases linearly as the data verified grows (since all data has to be downloaded to perform the check).

In the case of homomorphic authentication schemes they are more compatible with cloud-backed applications that do not perform data updates very often, i.e., after data is stored on the cloud the content of the data remains unaltered. For example a perfect candidate would be a back-up storage application used by an hospital or justice department where after data is stored on the cloud, users mostly perform read operations to consult the data. This is due to the fact that, they provide lower monetary costs for performing integrity checks, since executing code in the cloud for compacting the proofs is cheaper than reading the whole data (as will be shown in Chapter 5). However, to do so they require higher write monetary costs, related to the integrity control structures used in this schemes, which are always bigger or equal to the data's size.

This comparison between homomorphic and non-homomorphic schemes is of significant importance to this dissertation, and will be at focus on the Chapter 5 of this document, where the SafeAudit's homomorphic scheme will be compared with the RSA non-homomorphic scheme for better determining

the situations where each of these schemes should be used.

To further reduce the monetary costs paid while performing integrity verification using authentication metadata, the proofs of data possession or proof of retrievability can be used.

Regarding proofs of data possession, these mechanisms have the same benefits as homomorphic authentication metadata, since they use this integrity control structures to verify the data. However, the main advantage on using these probabilistic schemes comes on the amount of data used for generating the integrity proof. The integrity proof in the plane homomorphic authentication metadata schemes is generated using the entire dataset to guarantee an irrefutable verification result. This is not the case in proof of data possession that uses only a portion of the dataset to provide a probabilistic result. This leads to smaller computations needed to be performed for providing the integrity proofs, and reduces price paid for verification.

Regarding proofs of retrievability, these mechanisms are the more economic to use. In terms of storage they require low additional space to store the sentinel blocks, that only becomes noticeable when the user stores an high quantity of sentinel blocks per data. In terms of integrity verification, to check the integrity of the data users only have to download the data blocks that correspond to downloading a small portion of the data.

Although, SafeAudit does not directly provide a probabilistic mechanism to check the integrity of the data. This software library is intended to automate the generation of public homomorphic verifiable integrity proofs that should be compatible for both proof of retrieveability and proof of data possession mechanisms to use. This library adapts a public verifiable proof of retriveability scheme introduced in [14] and converts it into a more generic protocol to be capable of producing probabilistic or non-probabilistic integrity assurances. The original scheme uses sentinel blocks signed by homomorphic digital signatures, that are then compacted into an homomorphic compact proof of retriveability composed of two aggregations, one with all the sentinel blocks and the other with their corresponding signatures. The scheme introduced by SafeAudit additionally to what is already supported, allows the capability for proof of data possession schemes to sign all the data needed and to audit all the data they want.

2.2 Integrity Protection in Cloud Storage

SafeAudit was designed to help users protect cloud storage integrity. To this end, several works and techniques [10–12, 17, 19, 22, 23, 31–36] were studied for understanding how integrity protection is currently performed, and provide compatibility between SafeAudit and those techniques.

Based on the works and techniques studied, the remainder of this section presents an overview of the cloud storage characteristics related with integrity protection and states which works and techniques can be combined with SafeAudit for strengthening this protection. Section 2.2.1 presents the current consistency limitations found in cloud storages and how they influence integrity verification. Section 2.2.2 presents the challenges that should be addressed when using cloud storages in collaborative environments and how they should be addressed when performing integrity verification. Section 2.2.3 mentions the benefits of applying prevention mechanisms for access control in the cloud storage and what techniques should be used for combining these mechanisms with integrity verification. Section 2.2.4 explains how applying redundancy and fault-tolerance techniques alongside with integrity verification schemes increases cloud storage protection. Section 2.2.5 focuses on how to recover from integrity attacks after they are detected using surveillance mechanisms.

2.2.1 Integrity Verification in Eventually Consistent Clouds

Considering the *Brewer's CAP theorem* [37], which states that it is impossible for a distributed system to provide simultaneously consistency, availability and tolerance to network partitions, commercial clouds storage normally opt to provide high availability and tolerance to network partitions with weak consistency guarantees. In fact it is common for commercial clouds (e.g., Amazon [38]) to only provide an *eventually consistent* [39] storage, i.e., only guarantees that if no data updates are performed then all cloud storage servers will eventually have the most recent updated data. This consistency level and the possible inconsistency states it produces may be acceptable to user's tasks that do not depend on the linearizability of operations. However, in integrity verification mechanisms, data and signatures need to be consistent in order to properly verify the integrity of the data. If the data and signatures are not coherent with the write operation performed on the cloud storage verification errors will occur. For example, consider that a user updates the data and signature stored on the cloud storage, the cloud first confirms the update, the user then retrieves both the data and signature and performs the verification. In this example if a cloud provides to the user an updated data and corresponding outdated signature, the verification mechanism will accuse data corruption regardless of integrity state of the data, because the data and signature do not match. Although these errors do not affect the efficacy of the integrity mechanisms for detecting the attacks (since any inconsistent state will be considered as data corruption), they induce errors to the mechanism, i.e., inconsistent reads of data with its integrity preserved will be considered corrupted when in fact they are not. Another possible example of a situation that can be problematic is the case where an attacker performs a successful data update, and afterwards the verifier reads a previous version of the data that matches the version of the signature read. In this case the inconsistency of the data read creates an error that severely compromises integrity verification and allows attacks to go undetected until the verifier receives the correct and consistent version of both data and signatures.

To avoid the errors introduced by eventually consistent storage, there should be a way for cloud-backed applications to enforce higher levels of consistency on cloud storage. To do so, several works [17,23,26,29,31–33] use different techniques based on *versioning*, i.e., each data has an unique version identifier (identification token) that changes as data updates are performed. In cloud versioning techniques, all data and updates are stored separately on the cloud under an unique identification token, that changes as data updates are performed. For example, storing file 'hello.txt' on the cloud could be represented by the identification token 'hello.txt_v0' and performing data updates would generate tokens 'hello.txt_v1' for the first update, 'hello.txt_v2' for the second, so on and so forth. This technique allows users to have more control on which version of the data they want to read from the cloud, and allow users to obtain strong consistency using *coordination services*.

Coordination services (such as ZooKeeper [40] and DepSpace [41]) are entities that provide atomic consistent storage (also called linearizable storage [42]) and are used for storing and managing the versioning metadata. As reported on several works [17, 23, 33], strong consistency with coordination services is obtained as follows: whenever data is written on the weak consistent storage cloud, the user waits until the cloud confirms that the data has been successfully stored and then updates the coordination service's metadata with the information needed to obtain the last update of the file (typically the identification token of the data). Whenever the user wants to perform a read from the cloud, the user first lists the metadata stored on the coordination service and uses the most recent metadata version to obtain the corresponding data. Since coordination services are atomic and are consulted before performing any storage operation, users always see consistent states, i.e., all metadata available for reading in the coordination service corresponds to the data that the user expects.

Due to the particular importance that is reducing consistency errors when performing integrity verifi-

cation, SAFEAUDIT was designed to be fully compatible with coordination services, as will be explained in Chapter 5 in more detail.

2.2.2 Integrity Verification in Collaborative Environments

Several works tackle cloud storage integrity verification in collaborative environments, where the data is shared and modified by multiple trusted users.

In this type of environments, users may often experience concurrency problems when accessing to their data (also called *race condition*), i.e. two or more users can access the shared data and perform an operation (read or write) at the same time. When users are faced with these problems, the order of the operations performed is unpredictable and *serialization* [43] may not be possible, i.e., the result of the concurrent operations may not be equal to the outcome of executing them sequentially without overlapping in time. Thus, if write operations collude with no serialization possible, integrity of the data will be compromised. Also, if write and read operations collude with no serialization possible, the data read will not match the data stored on the cloud (and thus will introduce errors if checked with integrity verification mechanisms).

To avoid concurrency problems on collaborative environments, several works [17, 19, 22, 23, 26] propose using two possible alternatives: one, where all data operations are coordinated by a distributed lock protocol; and the other, where users are connected to a proxy for serializing data operations.

In the cases where distributed lock protocols are used [17, 22, 23], each data stored on the storage cloud is associated with a lock. Whenever the user wants to perform a data operation, the user firsts acquires the lock associated with the file, performs the data operation (read, write, delete) and when the operation finishes the user releases the lock. If another user has already acquired the lock, users that do not possess the lock wait until the lock has been released, to acquire the lock and perform the data operation. In these protocols, lock acquisition is normally represented as an empty file that is known by all users to be associated with the data (e.g., acquiring a lock for file ”a.txt” is made by creating the file ”a.txt.lock”), and lock release is made by deleting the lock file (e.g. releasing a lock associated with the file ”a.txt” is made by deleting ”a.txt.lock”).

In the cases where proxies are used for serializing data operations [19], the proxy should be placed between the user and the cloud, such that any communication between these two entities should pass by the proxy. Whenever users want to perform an operation to the cloud storage (e.g., list, delete, read, or write data), the users first contacts the proxy with an operation request, the proxy orders operations according to logical order (e.g., chronologically), and perform the operations on the cloud sequentially. When the cloud respond to the proxy operations, proxy reports back to the client with the answer.

The SAFEAUDIT library is developed to be used on collaborative environments and to that fact it is critical to be compatible with concurrency mechanisms. Chapter 5 will provide guidelines on how to use SAFEAUDIT alongside with the concurrency control mechanisms presented in this section.

2.2.3 Protecting Integrity with Access Control

In storage clouds, users can control the way they share the data with other users by implementing access control specifications, including which users can access the data and for each user the data operations allowed. For example, Amazon S3 access control [44] allows users to specify which Amazon S3 users can access the data, or make the data public for anyone to access (including users not registered in Amazon). Access control methods allow users to reduce the risk of being subjected to integrity attacks performed on their could storage by levering the exposure of data and controlling the operations that each entity can perform on the data. However, access control only protects the storage from external

attackers that attempt to access the clouds, and thus cloud storage may still be subjected to integrity attacks form internal cloud attackers. To further protect the integrity of the cloud storage from these attacks and also protect the storage from access control breaches, verification mechanisms, such as SAFEAUDIT or [2, 9–11, 14, 18, 20, 28], should be used to detect these threats and react accordingly.

This combination of access control with verification mechanisms that require key management, can be difficult when user's permissions are revoked and introduces the problem of trusting revoked users' signatures, i.e., since the user is no longer trusted to write on data, the user's signatures should be invalidated to avoid integrity attacks. There are two possible ways to handle this problem: force the remaining trusted users to resign the affected data; or rely on code execution performed on the cloud to perform the re-signature process, while the user's access control is being revoked.

The most common method of handling invalidated user keys, is to force trusted users to resign data. This method is straightforward to be implemented and after the re-signature is completed, the verifiability of the data sustains both internal or external integrity attacks. However, since the re-signature process requires the trusted user to download data from the cloud and upload the new signatures, there is a time range where the integrity of the data is uncertain, which starts from the moment the user access is revoked and ends with the upload of the new signatures. In this time range, since users have to trust the content of the data written by the revoked user, if the revoked user performs data updates before the download starts, the integrity of the data can be compromised without ever being detected.

In order to reduce this time range and resigning data with the minimum delay possible, [11] introduces a mechanism that uses code execution on untrusted clouds to re-sign invalid BLS [28] and SW [14] signatures. The conversion process of a signature under this mechanism, assumes the cloud is unaware of the revoked user's secret key and is performed by using *proxy re-signatures* [35], where users share keys with the untrusted cloud that convert bidirectionally the signatures from one user to another without compromising the signature's authenticity and verifiability. Thus, since certain clouds allow code execution to be triggered by cloud events [45], the re-signature process can start immediately after the access control is changed and this way reduce the time range to the minimum possible without involving the user or compromising the data's verifiability.

Both the aforementioned methods of revoking user access can be easily combined with SAFEAUDIT. Chapter 5 will provide guidelines on how to combine this software library with both these user revocation mechanisms and this way improve integrity protection on storage clouds that use access control mechanisms.

2.2.4 Sustaining Integrity Attacks with Redundancy

Verification mechanisms such as SAFEAUDIT and [2,9–11,14,18,20,28] are crucial for detecting integrity attacks and initiate the reaction phase, where the user attempts to stop the attack and undo the data corruption made by the attacker. However if the data affected by the integrity attack is damaged beyond repair, it may be difficult to recover its original state. In order to ensure that data is never damaged beyond repair, several works [10, 12, 21–23, 33, 34, 36, 46] propose using redundancy techniques for storing data in different storage clouds, and this way sustain, and possibly recover, from integrity attacks by collecting data from unaffected clouds. The techniques used on those works can be divided into three categories: those that store complete copies of the data; those who distribute small pieces of data with *erasure codes* [34]; and those who use *secret sharing* [36] techniques for providing redundancy and confidentiality of the data.

Regarding the redundancy techniques that store complete copies of the data in multiple storage clouds, the users must store complete copies of the data alongside with integrity control mechanisms, such as digital signatures or MAC, on a set of storage clouds that is bigger than the amount of integrity

attacks data they want to simultaneously sustain. For example: if the user wants to sustain integrity attacks to f copies of the file 'X.txt', this file must be stored on $f+1$ or more clouds must be used. With these techniques users can tolerate any type of storage availability faults (including byzantine faults [47]) or integrity attacks with the minimum communication required for retrieving the data, i.e., it is necessary to contact up to $f+1$ storage clouds to retrieve the data. The downside of these techniques is that monetary storage costs required are $f+1$ times higher than storing the file in one cloud.

To lower the monetary storage costs of storing data with redundancy, several works propose using erasure codes techniques. Erasure codes are a space efficient redundancy mechanisms that consist on dividing the data into x small chunks, and create y redundant copies based on the content present on the x chunks, such that any x chunks from the total of $x+y$ chunks created are necessary to collect in order to reconstruct the entire data. With erasure codes, users can sustain f integrity attacks with efficient monetary storage costs by applying the following steps: the user divides data in $f+1$ chunks and creates f redundant chunks; the user then stores each of the $2f+1$ chucks on a unique cloud (totalling $2f+1$ clouds used) accompanied by an integrity control mechanism of the stored chunk (digital signatures or MAC) [22]. This way, the user has to obtain up to $2f+1$ chunks from the clouds in order to recompute the data. Comparing the two approaches (storing complete copies of the data or erasure codes) in cases where users want to tolerate more than one fault, if the user chooses complete copies of data the communication involved and the amount of data required to download e always smaller than when erasure codes are chosen. However, erasure codes allow the users to pay less money for redundancy, since it requires more or less two times the storage monetary costs of storing an individual file ($\frac{|data|/}{f} * (2f+1) = (2+\frac{1}{f}) * |data|$) which is less than the $(f+1) * |data|$ required when complete copies used if the user wants to tolerate more than one failure simultaneously.

In some cases users may want to apply redundancy and also obtain confidentiality of the data stored on the cloud. In these cases applying any of the aforementioned redundancy mechanisms needs to be cyphered to enforce confidentiality of the data. To ensure that data is redundant and attackers cannot retrieve the original data, several works [22, 36, 46] propose using secret sharing techniques for ensuring both redundancy and confidentiality on the storage clouds. secret sharing is a technique similar to erasure codes that divides data into 2f+1 chunks, such that any $f+1$ chunks from the total of $2f+1$ chunks created are necessary to collect in order to reconstruct the entire data. However, when compared to erasure codes, the chunks produced by secret sharing have two main differences. First, the chunks are *information theoretic secure* [36], i.e., it is impossible to retrieve the original data (or any part of it) by obtaining less than the $f+1$ necessary chunks to reconstruct the original data. Second, they are all the same size of the original data. When comparing with the other approaches, the fact that each chunk generated by secret sharing is theoretical secure, makes it the only approach that allows data to be redundantly stored to several clouds while ensuring the confidentiality of the data with stronger guaranties than cyphering data. However, the fact that each chunk generated by secret sharing is typically the same size of the original data raises monetary storage costs to $2f$ times higher than storing the data in one cloud without redundancy, which makes it the most expensive of the approaches presented.

Due to the significant importance of combining integrity verification schemes with redundancy techniques for detecting and sustaining integrity attacks, SAFEAUDIT was integrated with the SCFS[17] cloud-backed application to demonstrate the practicability of these library and, since SCFS uses all the three redundancy mechanisms, the integration proves the compatibility between this library and the aforementioned redundancy techniques. Chapters 4 and 5 will explain in more detail how these integration can be performed.

2.2.5 Recover from Integrity Attacks

In spite of the fact that redundancy techniques provide capabilities for cloud-backed applications to sustain integrity attacks, users are still faced with the challenge of restoring data integrity on corrupted clouds to enforce correctness. The approach favoured by several works [10, 12, 33] for restoring data correctness affected by integrity attacks is having the users collect data from the unaffected clouds and re-upload the correct data to the affected clouds. To do so, this method requires data to be stored with redundancy to restore correctness.

The easiest redundancy approach to be used as basis for restoring data integrity is storing multiple complete copies on a set of storage clouds. Using this method, users can undo the damage of integrity attacks, by retrieving the original data from one of the unaffected clouds and overwriting the data on all of the corrupted clouds. Although this method can be easily performed by the users, if the users stores data on commercial storage clouds, they will have to pay a monetary fee for reading the data from the unaffected clouds and another fee for uploading the data to the corrupted clouds. There are works, such as [22, 33], that try to reduce monetary fees, by always choosing the cheapest cloud to read the data and substitute high price clouds with others that provide the lowest storage price available. These works could be adapted to this situation to reduce the overall monetary costs, however a better alternative solution would be to use erasure codes as basis for restoring data integrity [10, 12, 33]. Erasure codes also allow users restore data on clouds affected with integrity attacks, by collecting the chunks stored on the unaffected clouds, reconstructing the original content based on the chunks collected, and re-uploading to all of the clouds. Since re-applying erasure codes generates chunks that may be different from the previously created ones, all of the chunks stored on the cloud have to be overwritten with new ones. To avoid having to re-upload chunks to unaffected clouds, [10] proposes to substitute the computation of new erasure codes with a matrix solving technique to discover the original content of the chunks affected by the integrity attack, based on the chunks collected from the unaffected clouds and this way reducing the monetary costs, by only re-uploading the affected chunks to the respective clouds.

SafeAudit can be easily integrated with all the aforementioned methods of restoring data integrity affected by integrity attacks as will be described in Chapter 5.

2.3 Cloud-Backed Storage Solutions

In order to allow the compatibility between SafeAudit and the current cloud-backed solutions, particularly users' applications and cloud storages, the design of this software library leverage knowledge obtained from studying several cloud-backed storage applications. Section 2.3.1 presents an overview of the most important cloud-backed storage applications. Section 2.3.2 compares those applications and presents a rational for supporting the decision of using the SCFS application for the SafeAudit proof-of-concept (Chapter 4.2).

2.3.1 Cloud-backed Applications Overview

Currently it is common for applications to store data in the cloud. The SafeAudit library was created so that users can easily check the data stored by those applications and confirm its integrity.

In order to obtain a clear picture of how cloud-backed applications work and provide compatibility with the SafeAudit library, several examples of those applications were studied. This section will cover some examples of cloud-backed applications that can easily support this library.

2.3.1.1 Cumulus

Cumulus [48] is a cloud-backed application that allows users to store *incremental backups* of their data in any storage cloud, i.e., each backup performed over the data only contains the part of the data that changed after the previous backup was performed.

The main goal of Cumulus is to store data on the cloud with reduced storage costs. To do so, this application opts to aggregate small files belonging to the same snapshot into one large unique file (segment), to avoid the waste of storage space resulting from storing small files individually (including, higher metadata costs, and wasted space from rounding up to block boundaries). This way, a backup can be composed of several segments that aggregate the smaller data. For each segment created there is a metadata object. These objects contain information for mapping the snapshots, segments and the data within the segment, and also contain signed hashes to allow users to verify the integrity when the data is recovered.

Whenever a backup is added, the previous backup is compared with the new one. Old segments that contain common unchanged files are reused (adding a reference on the segment to the new backup) and the changed files are aggregated into a new segment. This way, by reusing old data Cumulus guarantees that all the user's data can be backed-up over time without wasting storage space with multiple data copies that remain unchanged over time.

2.3.1.2 RACS

RACS [33] is a system that allows users to disperse their data over several cloud providers using erasure code techniques, and this way become resilient against vendor lock-in problems, crash failures and data loss. RACS is designed to serve as a set of proxies, between the users and their cloud providers, and provide atomic guarantees in storage access using the Zookeeper coordination service [40], where each proxy is coordinated by an instance of this coordination service.

There are three scenarios where RACS plays an active role over the fate of the users data. First, regarding vendor lock-in problems, RACS is able maintain the lowest price by detecting when a cloud is about to raise price, compare the price over all combinations of cloud providers and migrate the data to a new clouds if viable. Second, because RACS disperses the data over the clouds in a redundant way using erasure coding, RACS is able to provide crash tolerance and file recovery (if active clouds are able to provide all the segments of data split by the erasure code).

2.3.1.3 DEPSKY and its Expansions

DEPSKY [22] is a cloud-backed storage library that allows applications to store data securely on several clouds with high availability, confidentiality and integrity guarantees. This library is compatible with many commercial clouds, including Amazon AWS [7], Windows Azure [49], Rackspace [50] and Google Cloud Storage [51], and requires no additional code to be executed in the cloud besides what is already provided by this storage entity. Regarding its usage, DepSky can be used for storing data in the cloud under two possible modes: the DepSky-A and the DepSky-CA.

The DepSky-A mode provides high availability guarantees to all the data stored on the cloud, by applying a redundancy technique that stores multiple copies of the data on different cloud storages.

The DepSky-CA mode, besides ensuring the availability guarantee provided by the DepSky-A, ensures that all data is confidentially stored on the cloud. To do so, in this mode all data is cyphered with a symmetric key to ensure confidentiality, and then applied redundancy with erasure codes for storing its content on different clouds and guarantee availability. Additionally, to allow the keys used for data encryption to also benefit from the confidentiality and availability guarantees, the keys are stored

on the clouds using a secret sharing technique and the resulting splits are placed on different storage clouds. Therefore, under this mode all the information stored on the cloud benefits from confidentiality and availability guarantees.

In both DepSky modes, integrity of all the information read from the cloud storage is ensured by using cryptographic hashes, which are placed on the storage cloud when the data is stored. These signatures allow DepSky to verify the integrity state of the information stored on the cloud when it is retrieved from storage, and resort to other clouds when faced with integrity problems.

Following the DepSky's work, two other works, CYRUS [52] and SafeSky [25], expanded the contributions made by this library.

CYRUS adapts DepSky to situations where users store their data on several commercial clouds, which some of them share the same infrastructure (e.g. iCloud [53] stores users data in Amazon AWS [7] and Google Cloud Storage [51], as stated in [54]), by developing an algorithm that detects and fixes these overlaps and selects which clouds to store the data based on their proximity to the users location and the data centres used by those clouds. This selection is of significant importance to security and availability of the data stored, since if redundant data is stored on the same data centre, redundancy becomes almost irrelevant since attackers that manage to penetrate the infrastructure can easily corrupt the data beyond repair (or disclose confidential information in case of secret sharing redundancy mechanisms). To avoid this problem the algorithm reduces latency without compromising the security offered when using DepSky by always storing the data on the clouds that are the nearest as possible to the user's location, while ruling out the clouds that store the data in the same infrastructure (e.g., if when selecting clouds between Amazon, Google Cloud Storage, Dropbox and iCloud, the iCloud is left out [1]).

SafeSky expands DepSky to provide a complete storage middleware that allows any host-based application to store the data on the cloud instead of the local disk, without any modification to its internal code. To do so, this middleware, located between the operating system and the applications, intercepts file system calls made from the application and simulates the operating systems' behaviour by reproducing the operation on the DepSky library to access the cloud storage and respond to the application with the data. This way SafeSky provides compatibility with a panoply of applications that could not be integrated with the original DepSky library, and also allows users to benefit from security and availability guarantees in all their applications.

2.3.1.4 Hybris

Hybris [23] is a cloud-backed storage application that stores data on a *hybrid cloud* set, and guarantees confidentiality, high availability, integrity and atomic consistency of all the data stored. The hybrid cloud set used by this application is defined as an aggregation of two cloud types: the *private cloud*, which is an infrastructure fully built and managed for a single individual or organization to use (e.g., HPE Helion OpenStack [56], Cisco [57], VMware's vSphere [58], Oracle private cloud [59]); and the *public cloud* (also known as commercial clouds), which is an infrastructure fully managed and built to serve multiple individuals or organizations simultaneously (e.g., AmazonAWS [7], Microsoft Azure [49], Google Cloud Storage [51]). The application orchestrates which information is stored on the private cloud or the public cloud regarding its sensitivity and this way guarantee its confidentiality.

The critical data (e.g., keys used for encryption) is placed on the private cloud for reducing the risk of being disclosed by attackers, since these clouds are assumed as less prone to being attacked than the public clouds. Hybris uses as private cloud the Zookeeper [40] infrastructure and takes advantage of its storage to store the sensitive data and the coordination service it provides for ensuring atomic

[1] Although the article considers Dropbox as an overlap with Amazon, this should no longer be the case, since in March of 2016 Dropbox now stores all its data on the MagicBox private cloud [55]

consistency of the information.

The less sensitive data is stored on the public clouds cyphered with a symmetric key to guarantee its confidentiality and then dispersed with erasure codes for additional availability assurances. This decision is driven by the monetary prices paid for storing data on private clouds, which is normally higher than the public cloud. With this orchestration, users can store information with confidentiality, availability and atomic consistency guarantees with reduced monetary storage costs.

To allow storage integrity protection, Hybris uses cryptographic hashes on all the information stored on the cloud. This decision allows data to be checked when read from the public and this way obtain integrity assurances that the integrity has not been compromised.

2.3.1.5 BlueSky

BlueSky [19] is a collaborative cloud-backed file system for storing and sharing data on a storage cloud. It uses a cache based proxy, placed between the users and the cloud, to serialize the users' data updates and this way solve concurrency problems of the collaborative environment.

Additionally, the BlueSky proxy provides confidentiality and integrity assurances to all the data stored on the cloud. Confidentiality is assured by cyphering all the data with a symmetric key before storing it on the cloud storage. Integrity is provided by non-homomorphic message authentication codes, which are stored on the cloud along with the signed data, and used for checking the integrity state when the data is read from the cloud.

2.3.1.6 SCFS

SCFS [17] is a collaborative cloud-backed-storage file system that ensures atomic consistency storage, concurrency control, availability, confidentiality and integrity of data stored over weakly consistent storage clouds.

Consistency and concurrency control is achieved by using DepSpace [41] or Zookeeper [40] coordination services for coordinating users' operations performed on the cloud-backed file system.

Integrity is achieved by storing data along with cryptographic hashes, and by using them for checking the integrity when data is read from the cloud.

The remaining assurances vary depending on the mode that this file system is configured. SCFS supports two modes: one where a single cloud is used for storage; and the other where several clouds are used. In the single cloud mode, SCFS assures confidentiality by cyphering all the data stored on the cloud with a symmetric key that is stored on the users device. In the second mode where SCFS is plugged into a set of storage clouds, the file system stores the data using the DepSky library under the DepSky-CA mode to achieve availability and confidentiality of the data.

2.3.1.7 HAIL

HAIL [12] is cloud-backed storage application that stores data on a set of clouds and assures availability, integrity verification and recovery of the data stored on the clouds. Availability of the data is assured by applying a redundancy technique based on erasure codes to disperse data into a set of storage clouds. Integrity verification is provided by requesting integrity proofs based on homomorphic message authentication codes. To do so, from time to time, users use HAIL to send proof requests to the storage clouds based on random samples of the stored data. Clouds resort to their code execution capabilities to generate integrity proofs that are then provided to the user upon each proof request. The user then validates the proofs received with the key used for generating the MAC and if attacks are detected HAIL initiates the recovery phase. To restore data's integrity, HAIL takes advantage of the erasure codes used

for replication, by reconstructing the data based on the unaffected clouds and then re-uploading the erasure codes to restoring integrity .

2.3.2 Discussion

This section provides a comparison between the aforementioned cloud-backed storage applications and the SAFEAUDIT's proof-of-concept that will be presented latter on Chapter 4.2 of this dissertation. As summarized in Table 2.3.2, to support the decision of using SCFS in conjunction with the SAFEAUDIT library for the proof-of-concept, three points of comparison were used: the type of integrity verification the applications provide; the techniques that each application provides for reducing data corruption and verification errors; and the resilience techniques they provide to reduce the damages of integrity attacks in the cloud storage.

2.3.2.1 Integrity Verification Comparison

Regarding the integrity verification capabilities offered, the applications have different characteristics with respect to the compatibility with collaborative environments and the requirements demanded on cloud storage selection to have code execution capabilities.

As seen in Table 2.3.2, most of the aforementioned applications that provide integrity verification opt in using cryptographic hashes to simultaneously guarantee that all the data read is stored correctly on the storage cloud and delivered to the user with its integrity preserved. The usage of cryptographic hashes, since no additional code has to be executed in the cloud for generating this integrity assurances, allows applications to remain compatible with a broad number of storage clouds. This is not the case of HAIL or the SAFEAUDIT's proof-of-concept, that opt to provide integrity verification assurances based on homomorphic structures for reducing the amount of data necessary to be downloaded from storage to perform the verification. Since data as to be transformed into compact proofs by resorting to additional code executing in the cloud, these applications are only compatible with clouds that provide both storage and code execution services. Due to that fact, the SAFEAUDIT library and its proof of concept has limited compatibility with respect to the clouds supported by design. Although this seems like a disadvantage for choosing this library to perform integrity verification on commercial clouds, the fact that almost all of them have code execution capabilities (e.g., AmazonAWS [7], Azure [49] and Google [51]), with few exceptions (e.g., iCloud [53] and DropBox [55]), shows how broad the diversity of clouds supported can be.

Also weighting on the comparison, the decision of whether to use private or public verifiable verification schemes on the application highly impacts its support for dynamic collaborative environments. As seen in Table 2.3.2, When using public verifiable schemes these applications are able to maintain the integrity verification assurances after revoking a user, by invalidating his key. In private verifiable schemes this is more difficult because all users share the same key and it is not trivial to revoke users access without all the previously performed signatures. BlueSky is the only studied application exception that allows the support of collaborative environments using MAC. This is due to the fact that integrity verification is not performed by the users directly, but by BlueSky' proxy. Since users do not have any knowledge of the MAC key, BlueSky can revoke users access without invalidating having to invalidate the key, and consequently support the collaborative environment change.

The idea of the SAFEAUDIT's proof-of-concept is to allow users to test all the capabilities of the library. Due to that fact, HAIL and Cumulus are not suitable for the SAFEAUDIT's proof-of-concept, since they cannot handle collaborative environments.

Application	Integrity Verification	Collaborative Environments	Concurrency Control	Consistency	Code Execution on the cloud	Sustains Integrity Attacks	Recovers from integrity attacks	Clouds used
Cumulus [48]	Cryptographic hashes	No	No	same as the cloud's consistency level	No	No	No	Single
RACS [33]	No	Yes	Yes	Atomic	No	Yes	Yes [1]	Multiple
DEPSKY [22]	Cryptographic hashes	Yes	Yes	Same consistency level as the weakest cloud present on the multiple cloud set.	No	Yes	Yes [1]	Multiple
Hybris [23]	Cryptographic hashes	Yes	Yes	Atomic	Yes	Yes	Yes [1]	Single or Multiple
BlueSky [19]	Cryptographic hashes	Yes	Yes	Same as the back-end cloud	No	No	No	Single
SCFS [17]	Cryptographic hashes	Yes	Yes	Atomic	No	Yes [2]	Yes [1]	Single or Multiple
HAIL [12]	Homomorphic MAC	No	No	same as the cloud's consistency level	Yes	Yes	Yes	Multiple
SCFS + SafeAudit	cryptographic hashes and homomorphic digital signatures	Yes	Yes	Atomic	Yes	Yes	Yes [1]	Multiple

[1] The user can recover the information but has to perform it manually, by reading the data and forcing a rewrite.

[2] Only sustains integrity attacks if DepSky mode is used

Table 2.2: Comparison between the integrated version of SCFS and SafeAudit and the other cloud-backed storage solutions.

2.3.2.2 Integrity Error Pruning Comparison

Regarding the integrity error pruning capabilities offered, the applications have different characteristics with respect to the consistency level guaranteed and their capabilities of handling concurrent operations performed on the storage cloud.

As seen in Table 2.3.2, the aforementioned applications that try to enforce concurrency control either resort to distributed lock management, through coordination services, or serialization of requests, using proxies. In the cases where coordination services are used, applications can also benefit from the higher levels of consistency offered by those services, and enforce strong consistency on all the storage operations.

To prove the compatibility between SAFEAUDIT, concurrency control and the mechanisms that enforce consistency guarantees, the application chosen for the proof-of-concept presented in Chapter 4 had to incorporate both those mechanisms. Due to that fact, Cumulus, DepSky [2], BlueSky and HAIL were not considered candidates for the SAFEAUDIT's proof-of-concept, since they do not provide strong consistency guarantees and concurrency control.

2.3.2.3 Resilience to Integrity Attacks

Some of the aforementioned applications can endure integrity attacks. To achieve this goal they opt to store data redundantly on several clouds, using multiple copies, erasure codes or secret sharing techniques. Furthermore since data is stored on multiple locations, all these applications allow users to recover the corrupted data from one cloud based on the content obtained from the unaffected clouds. Some are able to detect integrity attacks and restore the data automatically (e.g., HAIL) while the others require their users to perform this task manually (e.g. RACS, DepSky, Hybris, and SCFS).

Due to the importance of having systems that provide integrity verifiability and data recovery, the interoperability between SAFEAUDIT and these recovery methods was one of the major concerns surrounding its design. To show the compatibility between this library and the data recovery techniques, only applications that provide redundancy were considered as possible candidates for the proof-of-concept presented in Chapter 4.2. The list of candidates featured RACS, DepSky, Hybris, SCFS and HAIL, and had as favorits DepSky and SCFS since this systems included all the three methods of redundancy (multiple copies stored on the cloud, erasure codes and secret sharing).

2.3.2.4 Choosing SCFS for SafeAudit's proof-of-concept

As explained throughout this chapter, SAFEAUDIT was designed as an integrity verification software library capable of being compatible with other integrity protection mechanisms. To prove that this integration is possible a proof-of-concept was made by combining SCFS with the SAFEAUDIT library. The version of the application used was the one that stores data using DepSky-CA, since it was the only application considered that provides simultaneously: collaborative environments; concurrency control; atomic consistency; and redundancy techniques.

Besides the features offered by SCFS, two factors also lead to the creation of this proof-of-concept. The first factor is related to the fact that this application allows users to configure which of the features offered should be activated. For example, if a users only want to guarantee integrity verification, SCFS can be configured to only use cryptographic hashes and store the data in one cloud, without guaranteeing atomic consistency, redundancy or concurrency control. Even though that data is more vulnerable to

[2] Although the integration between SAFEAUDIT and DepSky was not considered for the proof-of-concept, since it lacked concurrency control and consistency guarantees, as will be seen in Chapter 4, the application used was the SCFS that stores data on multiple clouds. Therefore, to implement this proof-of-concept the SAFEAUDIT was integrated with the SCFS application through the DepSky library.

integrity attacks, the use of SCFS allowed a panoply of tests to be performed on SAFEAUDIT in the most flexible way possible. This allowed to relax the assumption model surrounding the use of this library, by using a basic model where only integrity verification is guaranteed, while proving that this library has full compatibility with the redundancy, atomic consistency and concurrency control features that are offered by this application. The other factor that lead to adopting SCFS is the fact that integrity verification is already implemented using cryptographic hashes. This permitted a more comprehensive evaluation and comparison between the two possible ways of performing integrity verification (the homomorphic checks provided by the SAFEAUDIT library or cryptographic hashes already offered by SCFS). Therefore, by comparing these two methods, the evaluation further presented in Chapter 5 shows which situations homomorphic verification should be chosen instead of cryptographic hashes to reduce the monetary price and bandwidth consumption cost, that users are affected when storing the data in commercial clouds.

2.4 Summary

This chapter presented the works that are related with the SAFEAUDIT library being proposed in this dissertation, by providing a global overview of the current mechanisms that allow cloud-backed applications to improve integrity protection on commercial cloud storage.

The first section of this chapter described the techniques that allow cloud-backed applications to perform integrity verification to commercial cloud storage. Since SAFEAUDIT is a software library for performing integrity verification on commercial clouds, this section provided a comparison of the currently available techniques and how the library being presented tries to solve their limitations.

The second section of this chapter explained the techniques that could be employed alongside with integrity verification mechanisms, including the SAFEAUDIT library, to improve the overall protection of the commercial cloud storage.

Finally, the third section presented some examples of cloud-backed storage applications compatible with the SAFEAUDIT library that employ integrity protection mechanisms for enforcing storage correctness on commercial clouds. The section concluded with a comparison between the applications presented, and explained that the SCFS application was chosen to be integrated with the SAFEAUDIT library as proof-of-concept since it was the one that provided the most integrity guarantees.

Chapter 3

SafeAudit

The goal of this dissertation is to provide a software library, SAFEAUDIT, for assuring users that all the data they store in the cloud is always retrievable with its integrity preserved. This library is envisioned to be easily integrated with: the current commercial storage clouds (such as Amazon-AWS [7]), for providing integrity proofs on the stored data; and cloud-backed storage applications (such as [17, 22, 25, 26]), to generate all the necessary digital signatures and automate the request and verification of the integrity proofs supplied by the clouds. To do so, SAFEAUDIT uses: homomorphic digital signatures for integrity control of the stored data; and the computational capabilities of commercial clouds infrastructures for executing code and generate compact integrity proofs based on the data and signatures present in the cloud storage. Also, by requesting and verifying those small sized proofs, cloud-backed applications can perform storage integrity control without being constrained with network bandwidth limitations or downloading large quantities of data.

In this chapter the conceptual design of SAFEAUDIT will be explained in detail. Section 3.1 explains the entities involved and their roles for integrity preservation. Section 3.2 states which threats SAFEAUDIT is able to protect the users' storage and which necessary conditions must hold for SAFEAUDIT to behave properly. Section 3.3 explains the preliminary security and mathematical concepts to which the SAFEAUDIT was based upon. Section 3.4 describes the interaction protocol required to be performed by each entity for integrity preservation while using SAFEAUDIT. Section 3.5 explains how SAFEAUDIT tries to reduce storage monetary costs necessary for storing homomorphic digital signatures. Finally, Section 3.6 provides a summary of the concepts addressed throughout this chapter.

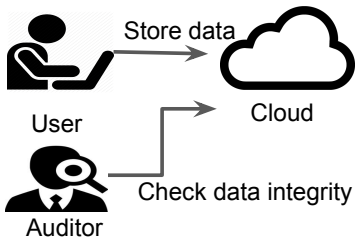

Figure 3.1: Entities involved on protecting user's data integrity in SAFEAUDIT and the interaction between those entities.

3.1 Entities Involved in SafeAudit

SAFEAUDIT is envisioned to be used in a model where there is interaction among three types of entities: *clouds*, *users* and *auditors* (see Figure 3.1).

Clouds are commercial public infrastructures that provide to their users data storage and code execution capabilities for providing integrity proofs.

Users are the normal commercial cloud users, who store data on the cloud and often perform operations (read, write, delete, or set access control permissions) to the stored data.

Auditors are security experts trusted by the users and responsible for auditing the users' data stored on the cloud. They are responsible for issuing and verifying integrity proof requests to the cloud, and advise users on how to protect their data. They provide all the necessary guidance and secure public configuration parameters so that users can be protected of integrity attacks.

3.2 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 to the users' data, particularly the operations that compromise integrity: write or delete. Under this scenario the attackers can be: an external entity who 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), but moved by malicious reasons has obtained control of one or more storage machines and 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 dissertation. 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.

3.3 Preliminary Concepts

SAFEAUDIT is built on top of *multiplicative cyclic groups* and uses several pairing based cryptographic techniques, BLS homomorphic digital signatures [16] and the Shacham-Waters integrity verification (SW) scheme [14]. In this section some mathematical background will be provided and the aforementioned cryptographic techniques will be explained for better understanding the remaining thesis.

3.3.1 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 powering the generator g with integers belonging to Z (the set of positive and negative integers). Multiplicative cyclic groups can be finite or infinite. The infinite ones are generated by powering with unbounded integers from Z. The finite ones

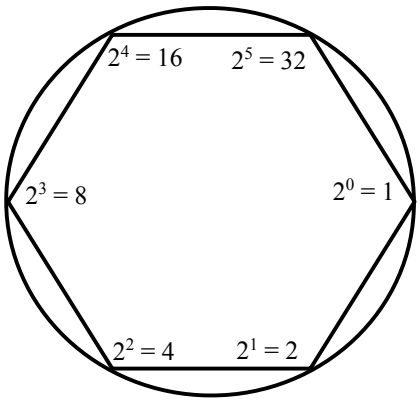

Figure 3.2: A multiplicative cyclic group representation of order 6 with 2 as its generator .

of order n are generated by powering 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 3.2, 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 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.

3.3.2 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 these cryptographic techniques ensure the following properties: *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 $e(u^a, v^b) = e(u, v)^{ab}$.

3.3.3 BLS Signature Scheme

In order to provide integrity control of a file SAFEAUDIT uses the *BLS signature scheme* [28] for constructing digital signatures over pairing based cryptography. To do so, using this digital signature scheme for 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 e and 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.

3.3.4 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 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 $order = size$ (e.g., 128 bits will generate group $0...128$), multiplications and additions will always produce elements of the same order. 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 128bit aggregation structure that represents the 10^6 bits file). In SAFEAUDIT the integrity verification (SW) [14] scheme 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 $d_{id} \in Z_p$, an hash function that maps $H : Z \rightarrow Z_p$, the secret key $sk \in Z_p$, and the signature parameter w, compute the signature $\theta_{id} = (H(id) \times w^d)^{sk} \in G$.

- *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 $chal_1...chal_n \in Z_p$, the hash function that maps $H : Z \rightarrow Z_p$, and the signature parameter w, 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.

3.3.5 Pairing Initialization with Elliptic Curves

Elliptic curves can be used in order to perform pairing initialization, i.e., initialize the multiplicative cyclic groups G and G_T, the generator g and the pairing function e needed for the BLS and SW setup phase. Currently there are several elliptic curves that can be used to perform this task and its selection is critical to obtain good performance with the BLS and SW schemes. SAFEAUDIT library currently supports pairing initialization with elliptic curves of the six different types described in Section 4 of [16]. While there is support for those six different types of curves, as recommended by both BLS and SW authors

in [14] and [16], the best suitable curves to be used in pairing initialization are the pairing-friendly elliptic curves of prime order [15] (also known as type F curves and described in Section 4.14 of [28]).

3.4 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 section, the interaction protocol is divided into four parts: set up (Section 3.4.1), store data (Section 3.4.2), request and verify of integrity proof (Section 3.4.3), and proof generation (Section 3.4.4).

3.4.1 Set up

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 to the user two files [1] 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 Chapter 4).

- 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 Chapter 4) using the code execution service provided by the cloud (e.g., AWS Lambda [45]).

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

3.4.2 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 Chapter 4) automates these tasks and produces a signature equivalent to the SW sign block step (as seen in Section 3.3.4). To do so, the client provides as input, for the signature generator, the data and its id (e.g., the file content of the 'hello.txt' file is used as the data and the id is the filename 'hello.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 the cloud.

[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.

3.4.3 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 one 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 Chapter 4). 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 SW scheme.

3.4.4 Generating Integrity Proofs

Whenever the cloud receives an integrity proof request of a given dataset (as described in Section 3.4.3), 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 4.1.5). 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).

3.5 SW Signature Size Optimization

Considering the elliptic curves supported by SAFEAUDIT, regardless of the elliptic curve chosen for pairing initialization, the SW scheme always produces signatures that are bigger than the original data. As reported in [14] the best case scenario, where type F curves [15] are used, the SW scheme produces signatures that are twice the size of the signed data. Further aggravating the problem if type A [28] curves are used, signatures size is given by the following formula: $|signature| = \frac{|G| \times |data|}{|Z_p|}$.

This means that SW scheme when applied directly to commercial cloud computing environments can imply huge storage overhead costs, since for example considering type A curves are used, with the recommended sizes where G and G_T are 128 bytes and Z_p is 20 bytes, the signatures produces are 6.4 times bigger than the original file, raising the storage costs 6.4 times high.

In order to cope with the storage overhead costs introduced by the SW scheme, and reduce it on SAFEAUDIT, signature compression is applied using the *point compression* technique described in [16]. 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, and 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 [15] are used are the same size of the original data.

3.6 Summary

In short the SAFEAUDIT library allows users to verify the integrity of the data stored on the cloud, without having to retrieve the whole data. The remainder of this section will recap the library's conceptual design presented throughout this chapter.

To provide integrity verification, the SAFEAUDIT library makes use of two cryptographic mechanisms built on top of pairings, the BLS homomorphic digital signatures and the SW integrity verification scheme. Although these mechanisms were already adaptable to commercial cloud usage, they required high storage consumption costs that increased the monetary costs paid on commercial clouds. To reduce these costs, SAFEAUDIT optimizes both underlying mechanisms by carefully selecting pairings that are generated with type F elliptic curves and by applying signature compression techniques. With these optimizations, the amount of storage consumed is reduced and produces significant monetary savings to the SW integrity verification scheme when commercial clouds are used.

Using the optimized verification mechanisms as basis, integrity preservation under SAFEAUDIT is performed in three phases. The first phase is the set-up phase, which happens before using the library and it is dedicated to configure all the information that the library depends. The second phase is the signature phase, that happens when storing data and which users resort to SAFEAUDIT to generate homomorphic digital signatures that are stored along with the data on the cloud. The last phase is the auditory phase where cloud storage is verified using the SAFEAUDIT, by requesting to the cloud an integrity proof for a set of data blocks, that is then computed by the clouds using the SAFEAUDIT for compacting the data blocks and signatures into a SW integrity proof.

To facilitate the interaction between the entities involved in integrity preservation during the three aforementioned phases, SAFEAUDIT provides an interaction protocol that encapsulates all the steps needed into a set of instructions that each entity should follow to ensure the integrity protection.

Chapter 4

Implementation

The SAFEAUDIT software library is designed to be easily integrated with the current cloud-backed storage solutions and to automate all the tasks needed for performing cloud storage integrity verification. This chapter focusses on the implementation aspects of the library and explains how this automation can be achieved, and how it is compatible with cloud-backed storage applications.

The remainder of this chapter is divided in two sections. Section 4.1 explains the internal construction of the library. Section 4.2 shows the compatibility of this library with cloud-backed applications in a proof-of-concept where SCFS was integrated with the library. Finally, Section 4.3 provides a summary of the concepts addressed throughout this chapter.

4.1 SafeAudit Library 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 automatizing a task of the SAFEAUDIT's interaction protocol. This chapter will cover all those components in full detail describing its overall behaviour and functionality.

This section is structured as follows. Subsection 4.1.1 presents the *Pairing Generator* component, which allows auditors to generate all the setup parameters required to initialize pairing-based-cryptography. Subsection 4.1.2 explains the *Key Generator* component, used to generate the asymmetric secret/public key pair and signature parameter (w). Subsection 4.1.3 covers the *Signature Generator* component that allows users to sign their data. Subsection 4.1.4 presents the *Random Generator* component which allows entities to generate random numbers belonging to any field of their choosing (Z_p, G or G_T). Subsection 4.1.5 explains the *Proof Generator* component which allows clouds to generate integrity proofs. Subsection 4.1.6 presents the *Proof Verifier* component that allows auditors to verify the proofs obtained from the cloud. Finally, Subsection 4.1.7 explains the overall benefits in using the implementation of this software library.

4.1.1 Pairing Generator

This component allows auditors to construct setup parameters ('.param' and '.g') for initializing pairing based cryptography, according to the their security specification. Interactions with this component are performed by this invocation:

```
1  <.param,.g> genPairing(char type,int parameters...)
```

Auditors provide as input the type of pairing curve to be used for pairing generation ($type = A|B|C|D|E|F$) [1], 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.

4.1.2 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.

Listing 4.1: Key generation algorithm.

```
1  <sk,pk,w> genKey('.param','.g') {
2     Pairing p = initializePairing('.param','.g');
3
4     //generate secret key from Zp
5     Element sk = p.getZp().getRandomElement();
6
7     Element g = p.getG().getGenerator();
8
9     //generate pk by powering g with sk
10    Element pk = g.pow(sk);
11
12    Element w = p.getG().getRandomElement();
13
14    return <sk, pk,w>
15 }
```

The generator is as described in Listing 4.1: the user inputs the setup parameters provided by the auditor '.param' and '.g'; the component initializes the pairing (line 2); generates the secret key by selecting a random number belonging to Z_p (line 5); generates the public key by computing g^{sk} (line 10); generate the signature parameter (*w*) by selecting a random number belonging to G (line 12) and returns the keys and the signature parameter to the user (line 14).

4.1.3 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 fix 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 is divided into two components: the Sign Block component, for signing data in Z_p; and the Sign Data component, that converts all the input data to one or several blocks $\in Z_p$, signs each block using Sign Block component, and returns the concatenation of all the blocks' generated signatures.

[1] See Section 4 of [16] for learning about the pairing curves and their selection

4.1.3.1 Sign Block

The Sign Block component is as described in Listing 4.2: 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 (line 1); the component initializes pairing (line 2); hashes the id to Z_p (line 4); multiplies the id's hash with w^d (line 6); signs the multiplication with the user's secret key (line 9); apply point compression (line 11) and return the resulting point as the signature (line 13).

Listing 4.2: Block signature generation algorithm.

```
1  <sig> signBlock('.param','.g', data, id, sk, w) {
2     Pairing p = initializePairing('.param','.g');
3
4     Element idHash = p.HashToZp(id);
5
6     Object blockToSign = (idHash * w.pow(data));
7
8     //sign with secret key
9     Element sig = blockToSign.pow(sk);
10
11    Point compressedSig = sig.getXCoordinate();
12
13    return compressedSig;
14 }
```

4.1.3.2 Sign Data

The Sign Data component is as described in Listing 4.3: the user inputs the setup parameters provided by the auditor '.param' and '.g', the data, id of the data id, the secret key sk and the signature parameter w (line 1); the component initializes the pairing (lines 1-3); the component divides the data into a vector of blocks that belong to Z_p (line 6 and 7); signs each block individually with an unique id (line 10 to 14); concatenates the blocks' signatures into one (line 17); and returns the concatenation (line 19).

Listing 4.3: Signature generator algorithm.

```
1  <sig> signData('.param','.g', data, id, sk, w) {
2     Pairing p = initializePairing('.param','.g');
3     Field zp = p.getZ_p();
4
5     //split data into Zp blocks
6     int blockLength = zp.getLength();
7     blocks[] = splitIntoBlocks(data, blockLength);
8
9     //sign each generated block, under a diferent id.
10    blockSigs[];
11    for(int i = 0; i < blocks.length; i++) {
12       idBlock = id + block.index;
13       blockSigs[i] = signBlock('.param','.g', block, idBlock, sk, w);
14    }
15
16    /*concatenate all blocks into a signature (with size Zp * blocks.length)*/
17    sig = aggregateBlocks(blockSigs);
```

```
18
19    return <sig>
20 }
```

4.1.4 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 belonging to the field specified.

4.1.5 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, as described in Listing 4.4 the algorithm first initializes pairing with the setup parameters (lines 2-4), calculates alpha (lines 7 to 11) based on the dataset and calculates beta based on the signature (lines 13 to 18). Note that since signatures are stored using point compression, it is necessary to calculate the y axis coordinate of the signature for generating the proof (line 18).

Listing 4.4: Proof generator algorithm.

```
1  <sig> generateProof('.param','.g', KeyMap<id, chal> blockChallenges, dataSetBlocks[],
       signatureSetBlocks[]) {
2    Pairing p = initializePairing('.param','.g');
3    Field zp = p.getZp();
4    Field g = p.getG();
5
6    //calculate alpha
7    Element alpha = zp.getNewElement().setToZero();
8    for(block : dataSetBlocks) {
9       chal = blocksChallenges.getChal(block.id);
10      alpha += chal*block.content;
11   }
12
13   //calculate beta
14   Element beta = g.getNewElement().setToOne();
15   for(block : sigBlocks){
16      chal = blocksChallenges.getChal(block.id);
17
18      Element sig = Element.createNewFromXCoordinate(block.content);
19
20      beta *= (sig).pow(chal);
21   }
22 }
```

4.1.6 Proof Verification

The Proof Verifier component allows users to verify integrity proofs, using the SW proof verification step. To do so, as described in Listing 4.5 the algorithm first initializes pairing with the setup parameters

('.param' and '.g') (lines 2-5); applies g pairing to beta (line 8), multiplies all ids present in the proof with w^{alpha} (lines 11-18), applies public key pairing to the id and alpha multiplication (line 21) and verifies if both pairings obtained match (lines 24-28). If so, integrity is preserved. Else some data has been tampered.

Listing 4.5: Proof verification algorithm.

```
1  <bool> isProofValid('.param','.g', Collection<id> ids, KeyMap<id, chal> blockChallenges,
       alpha, beta, pk, w) {
2    Pairing p = initializePairing('.param','.g');
3    Field zp = p.getZp();
4    Field g_field = p.getG();
5    Element g = p.getGenerator();
6
7    //apply g pairing to beta
8    Element p_beta = p.applyPairing(beta,g);
9
10   //calculate the multiplication of all the ids
11   Element id_mul = g_field.newElement().setToOne();
12   for(i = 0; i < ids.size; i++) {
13      id_mul.mul(ids[i]);
14   }
15
16   //calculate id_mul * w^alpha
17   Element pow_w_alpha = w.pow(alpha);
18   Element alpha = id_mul.mul(pow_w_alpha);
19
20   //apply public key pairing to final_alpha
21   Element p_alpha = p.applyPairing(_alpha, pk);
22
23   //compare both pairings
24   if(p_beta.equals(p_alpha)) {
25      return true;
26   }
27   else return false;
28 }
```

4.1.7 Library Takeaways

Throughout this first section the SafeAudit's library implementation was explained. This library is constructed into several components each optimizing a different task from the SafeAudit's interaction protocol. The decision to encapsulate these operations into different components was made to decouple its structure and make the library understandable. This way, all the tasks related to setting up the SafeAudit library are provided in two components: the Pairing Generator, that optimizes the construction of pairings used by library and the Key Generator component, that allows users to generate the integrity keys. To automatize signature generation, users may resort to the Signature Generator component. This component supports two signing modes, one where the data blocks are signed one by one using the Sign Block method, and the another that signs all the data blocks continuously and produces one signature for the whole data using the Sign Data method. Finally, to automatize the auditing process this library provides two components: the Proof Generator component, that allows clouds to generate

an integrity proof and the Proof Verifier component that verifies the integrity proofs.

4.2 SafeAudit Proof-of-concept

SafeAudit was developed for being easily integrated with the user cloud-backed applications. As a proof of concept, SafeAudit was integrated with the *Shared Cloud-backed File System* (SCFS) [17].

SCFS 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 (which requires atomic consistency or linearizability [42]), even if weak consistency storage cloud services are used (e.g., services that guarantee only eventual consistency [39]).

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

Regarding storage of data in the cloud, SCFS has two modes: the single cloud model, where files are stored in a service like Amazon S3; and the multiple cloud model, where files are stored on several clouds using the DepSky software library [22]. DepSky provides an API for uploading and operating with a set clouds, while enforcing fault tolerance, confidentiality, and integrity of all the overall set of the clouds storages as long as the clouds affected with the aforementioned problems do not reach the majority of the cloud set. In this integration, in order to allow easy expansion to several cloud services, SCFS was configured with DepSky and the integration with SafeAudit was made through DepSky.

Although data integrity is protected in SCFS and DepSky, they both use non-homomorphic digital signatures. This allows users to verify any data present in the cloud storage, but requires users to download the data and the signatures and perform the integrity verification on their device. This leads to scalability problems in terms of bandwidth consumption and performance implied on the users device and, is solved by integrating SCFS and DepSky with SafeAudit for automatically storing the homomorphic signatures whenever the user uses SCFS or DepSky to upload data to the commercial cloud.

As previously mentioned SCFS and SafeAudit integration was made on the component responsible for uploading data into the clouds, *DepSky*. This component receives the data from SCFS applies mechanisms that ensure confidentiality, integrity, and availability, and then stores the resulting data in the cloud. The logic for communicating with different commercial clouds is implemented in subcomponents called *cloud drivers*. Since the integration should not compromise any of the aforementioned properties, integrating both the systems required code changes to DepSky, and the addition of a new type of cloud drivers, the *auditable cloud drivers*. With these newly introduced cloud drivers, besides accessing and uploading data to the clouds, data is also signed using the SafeAudit's signature generator and the signature is also stored on the cloud. As seen in Figure 4.1, for integrating these new drivers, DepSky suffered changes in two packages: core and drivers. Regarding the DepSky's core package, code was added to the DepSky's initialization function (present on *LocalDepSkySClient.java*) and to the DepSky's driver constructor function (present on *DriversFactory.java*).

For using 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 be used, 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 DepSky's driver initialization function for initializing the driver with that information.

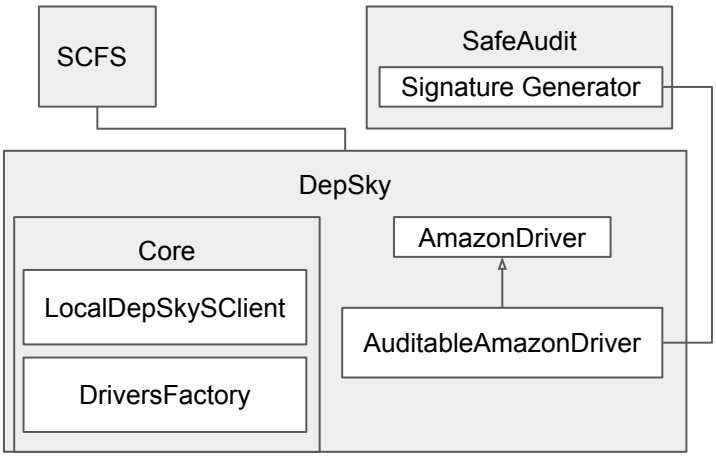

Figure 4.1: Components modified when integrating SCFS/DepSky with SafeAudit.

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.

This integration between SAFEAUDIT and SCFS, did not deteriorate the behaviour of both SCFS or DepSky systems and had no repercussion to the functionality of their internal mechanisms. Due to that fact and since both SCFS and DepSky already employed several integrity protection mechanisms, the proof-of-concept chosen demonstrates that SAFEAUDIT is compatible with a panoply of integrity protection mechanisms described in Chapter 2, including coordination services that strengthen consistency of the application storage and concurrency control, and redundancy mechanisms, including techniques that store multiple copies of the data; erasure codes; and secret spitting techniques. Even tough these mechanisms are not strongly related or required to perform integrity verification using SAFEAUDIT, providing compatibility with these mechanisms, particularly the redundancy techniques employed, is of critical importance to this work, since integrity protection is only enforced using SAFEAUDIT if the users can detect integrity attacks and recover the correctness of the data. By integrating SAFEAUDIT with these additional mechanisms users benefit from the integrity verification provided by this library and also have all the mechanisms necessary to restore the correctness of the data when integrity attacks are detected.

4.3 Summary

Throughout this chapter the SAFEAUDIT's implementation was at focus, explaining how it atomizes all the tasks described in SafeAudit's interaction protocol and demonstrating the compatibility with cloud-backed storage applications.

In the first section of this chapter the internal structure of the library was explained, describing its overall organization and how each entity can use the library for automatizing the tasks described in the SafeAudit's interaction protocol.

The second section of this chapter demonstrated the compatibility between the SAFEAUDIT and cloud-backed applications in a proof-of-concept, which integrated this library with the SCFS cloud-backed application. Besides proving the compatibility of the library with these applications, this proof-of-concept also proved that SAFEAUDIT is completely compatible with several integrity protection techniques described on Chapter 2 of this dissertation, including coordination services and redundancy mechanisms (multiple copies of the data, erasure codes and secret sharing).

Chapter 5

Evaluation

This chapter is centred on the evaluation conducted on the SAFEAUDIT software library. Section 5.1 will focus on the experimental evaluation and discuss the results obtained while performing integrity verification using this software library, regarding the monetary costs paid, the bandwidth consumed and the performance impact it implied on cloud-backed applications. Section 5.2 will evaluate the compatibility between SAFEAUDIT and the aforementioned integrity protection techniques described on Chapter 2, while providing guidelines to integrate these mechanisms. Finally, Section 5.3 summarizes the evaluation performed to the SAFEAUDIT library.

5.1 Experimental Evaluation

All SAFEAUDIT 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) [60], that implements Multi-linear Maps and all the operations required by these mechanisms for manipulating those maps.

The cloud used during the implementation was Amazon AWS [7]. S3 [61] was used as storage and Lambda [45] for executing Java code at the auditors request through a REST API (i.e., the executing steps in Section 3.4.4). Lambda was chosen since it allows code to be executed at a given event, in our case when HTTP request is received at the cloud. By using services like Lambda (instead of alternatives of running virtual machines like EC2 [62]) users only pay for the code execution, which does not contemplate idle costs, and therefore can leave the system always on and ready for executing the auditors proof generation request without additional monetary costs, this is not possible in EC2 since monetary costs are charged from the moment the machine boots until it is completely shut down.

In order to evaluate whether SAFEAUDIT is capable of being deployed to real-world scenarios, several experiments were ran for comparing this library with commonly used algorithms. Section 5.1.1 describes the conditions where the experiments were performed, and throughout the remaining of this section the results obtained will be discussed.

The main goal of SAFEAUDIT is to reduce bandwidth consumption without raising monetary costs related with integrity verification on commercial cloud storage. To do so, in order to evaluate this goal the signature and proof generators were benchmarked. Regarding proof generation, code execution in the cloud was benchmarked with bandwidth consumption and monetary costs as evaluation metrics. Regarding the signature generation, since monetary costs increase as the cloud storage capacity grows, signature generation mechanism was benchmarked to see the additional storage capacity required. These results are presented in two sections. Section 5.1.2 presents bandwidth consumption evaluation

and Section 5.1.3 presents the monetary cost projection.

Besides the aforementioned benchmarks, the signature generator and the proof verifier components were benchmarked with response time as evaluation metric, in order to understand the performance impact on the users devices for generating a signature and on auditors to verify the proofs. Section 5.1.4 presents the performance benchmarks of using the library on cloud-backed applications. Finally, Section 5.1.5 discusses the results obtained on the evaluation performed and summarizes the benefits and limitations of this software library.

5.1.1 Testbed Details

The evaluation benchmarks were performed using a dataset containing one file for each of the following sizes: 128, 256, 512, and 1000 Kilobytes. These benchmarks were tested under two testbeds: *user's device testbed* for evaluating the signature generation and proof verification; and the *cloud testbed* for evaluating the proof generation performed by the cloud.

1. *Users' device testbed.* In this testbed all the experiments were conducted on a Windows 10 computer with Intel Core i7-4500U CPU 1.80-2.40 GHz processor and 8 GB RAM, and repeated 30 times per experiment.

2. *Cloud's testbed.* In this testbed all the experiments were conducted on the Amazon's Ireland data center with user located in Lisbon (Portugal), Lambda AWS was configured with 128 MB memory for executing proof generation and Lambda S3 for storing data, and repeated 30 times.

3. *Baseline comparisons and algorithms parameterizations.* In the several experiments conducted for evaluating SafeAudit's library, two algorithms were used for baseline comparison: the original SW scheme implementation; and the RSA digital signature [18] with 1024 bit keys and tuned to use SHA-1 [6] as hash function. Both SafeAudit and the original SW scheme were parameterized with type F pairing curve where G is 40 bytes and G_T 80 bytes and Z_p of 20 Bytes, with SHA-1 as their hashing algorithm, and asymmetric keys was used with a 20 byte secret key and a 80 byte public key.

5.1.2 Bandwidth Evaluation

Integrity proof generation was evaluated in terms of bandwidth consumption. Figure 5.1 shows the bandwidth consumption when obtaining proofs using SafeAudit in the cloud's testbed and compares it with the original SW scheme and the RSA digital signatures where the entire data has to be retrieved in order to verify it. The results show that as the storage size grows, SafeAudit and the SW scheme are able to maintain constant bandwidth consumption. Also, since proofs are composed of an aggregation of blocks belonging to Z_p (20 bytes) and an aggregation of blocks belonging to G (40 bytes), this shows that bandwidth consumption is always equal to the sum of these group's sizes and that it is always low. When comparing to the RSA digital signatures, which implies linear bandwidth consumption costs as storage grows, SafeAudit allows lower network latencies and lower monetary read costs (monetary read costs encored for reading 60 bytes are negligible).

5.1.3 Monetary Costs

In order to test the monetary costs implied for verifying the integrity of the cloud storage, two tests were performed: one for benchmarking the additional monetary storage cost required for storing digital signatures; and the other for benchmarking the monetary costs of generating proofs on the cloud.

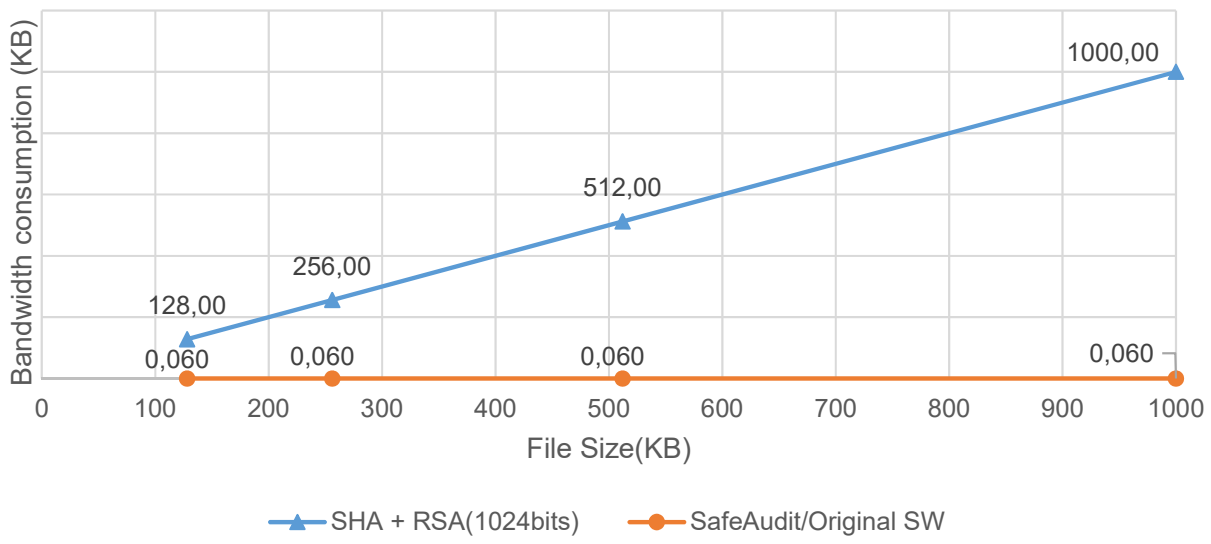

Figure 5.1: Bandwidth consumption comparison between requesting file integrity proofs using SafeAudit or RSA digital signatures. The X axis represents the size of the file in Kilobytes and the Y axis the bandwidth consumption in Kilobytes.

Figure 5.2: Storage increase comparison between using SafeAudit, SW, or RSA + SHA-1 digital signature sizes as data size grows. The X axis represents the size of the file in Kilobytes and the Y axis the signature size in Kilobytes.

5.1.3.1 Monetary Storage Costs

Using SafeAudit for verifying data integrity on the cloud storage requires users to store, on the cloud, the data's digital signatures and implies additional monetary storage costs.

Figure 5.2 compares the storage increase when using SafeAudit, SW, or RSA + SHA-1 digital signatures as data size grows. As seen in this figure, storing SW signatures increase storage size by 200%, SafeAudit is able to reduce this increase storage overhead to 100% due to the signature compression optimization. This optimization has great positive impact on storage monetary costs, but it still requires twice the storage than the ideal case where signature sizes are negligible (such is the case of RSA + SHA-1 digital signatures).

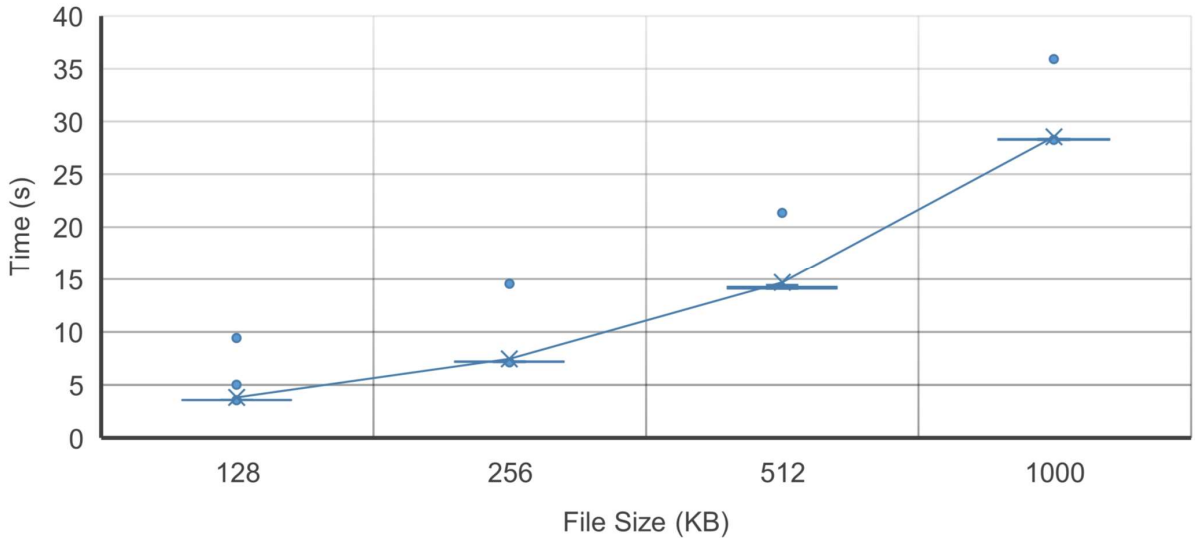

Figure 5.3: Execution time required for the cloud to generate an integrity proofs. The X axis represents the file size growth in Kilobytes and Y the execution time in seconds. In this chart, each dot represents the a sample of the 30 samples collected per file size, and each horizontal bar represents the mean value of the samples collected for that size.

Table 5.1: Price comparison between generating proofs and reading data.

File Size (KB)	Average Execution Time (s)	Execution Cost[1] (microUS$)	Read Costs[2] (microUS$)	Savings (%)
128	3,82	8,11	11,52	29,58
256	7,44	15,6	23,04	32,29
512	14,67	30,57	46,08	33,64
1024	28,55	59,48	90	33,90

[1]Assumed 0,208 microUS$ for each 0.1 seconds of computation
[2]Assumed 0,09 microUS$ for each 1 KB read from the cloud storage

5.1.3.2 Monetary Proof Generation Costs

In order to evaluate the monetary costs associated with integrity proof generation, SAFEAUDIT's proof generator was benchmarked under the cloud's testbed.

As seen in Figure 5.3, the results obtained for generating integrity proofs increase linearly as storage grows. Furthermore, as seen in Table 5.1, when comparing price paid for generating a proof (execution time) with the cost of downloading the files entirely and perform the integrity verification on the auditors device (as required by RSA + SHA-1), generating integrity proofs was cheaper than reading the data from the cloud and allowed a monetary saving rounding 30%. Even though these monetary savings are the same as only performing integrity verification with the original SW scheme, they were never discovered in the past since this work is first to perform monetary evaluations to SW scheme.

5.1.4 Performance Costs on User's Applications

Normally cloud users, and their cloud-backed applications, need fast signature mechanisms so that the process of storing data in the cloud is not delayed. In order to test if SAFEAUDIT meets this criteria, two performance benchmarks were used: one, detailed in Section 5.1.4.1, comparing the time taken to sign data using SAFEAUDIT and RSA + SHA-1 digital signatures; and the other, detailed in Section 5.1.4.2,

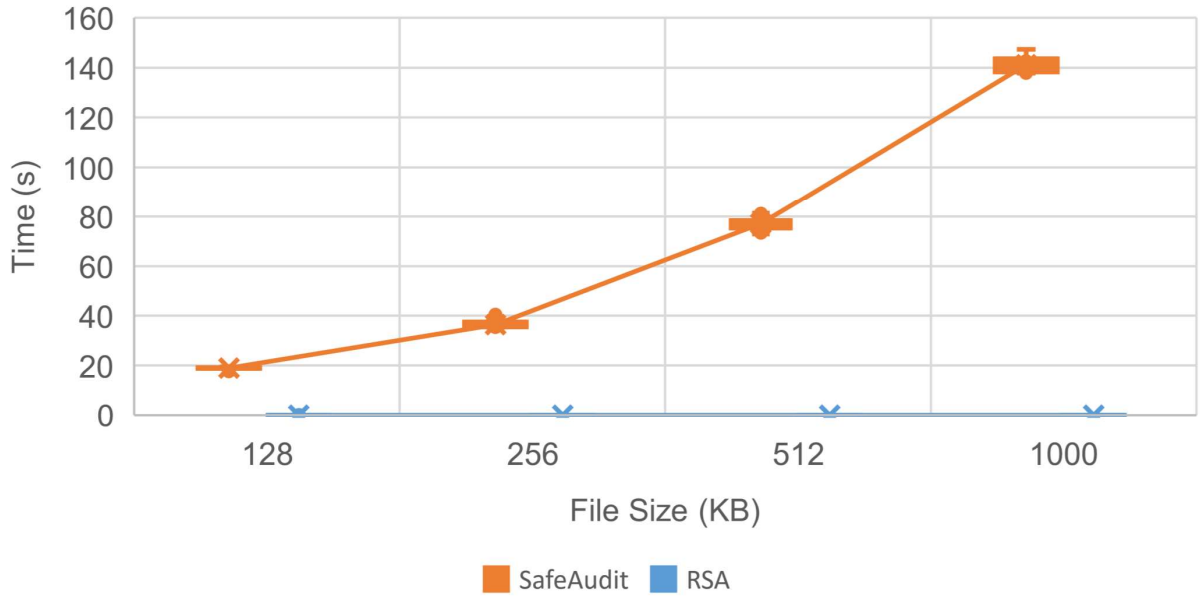

Figure 5.4: Comparison between signing data using SafeAudit and RSA + SHA-1 digital signatures. The X axis represents the size of the file in Kilobytes and the Y axis the signature size in Kilobytes.

for evaluating the overhead SCFS has when integrated SafeAudit.

5.1.4.1 Signature Generator Benchmark

SafeAudit's signature generation was benchmarked in terms of the time required to compute a signature. To do so, signature generation was tested using the user's device testbed and the results obtained are presented in Figure 5.4. The time required for signing data using the SafeAudit increases linearly and is in the order of seconds and is much slower when compared to the RSA + SHA-1 digital signatures which rounds the order of two milliseconds. This is due to the fact that data signed using RSA digital signatures is hashed into small sizes and then signed, abruptly reducing the time for signing. In SafeAudit, the SW scheme and all the other homomorphic schemes, all data has to be signed without using hashes, to avoid security problems related to generating proofs using precomputed hashes (i.e., an adversary at the cloud computes the hashes once, corrupts or discards the data, and later computes proofs using only the hashes). Furthermore, due to this limitation it is necessary to sign each block of data individually and it does not scale well as data grows. For example signing 1MB of data involves signing 25600 blocks which takes about 2 and half minutes. This makes SafeAudit much slower than the usual signature generation mechanisms, as it was expected since it is not different from the other homomorphic signature generators.

5.1.4.2 SCFS with SafeAudit's Signature Generator

In order to evaluate the performance impact of integrating SafeAudit on users' cloud-backed applications, two SCFS versions were tested to upload the 1000 Kilobyte file under the user's device testbed: one as standalone version of SCFS, and another with SafeAudit's integrated. Both the SCFS versions configured to store data into the local computer to eliminate possible network overheads. The results obtained are presented in Figure 5.5 and show that the integration with SafeAudit increases time significantly. This is not different from the results presented in Figure 5.4 and shows that integrating this signature generator can be a source of overhead in cloud-backed applications.

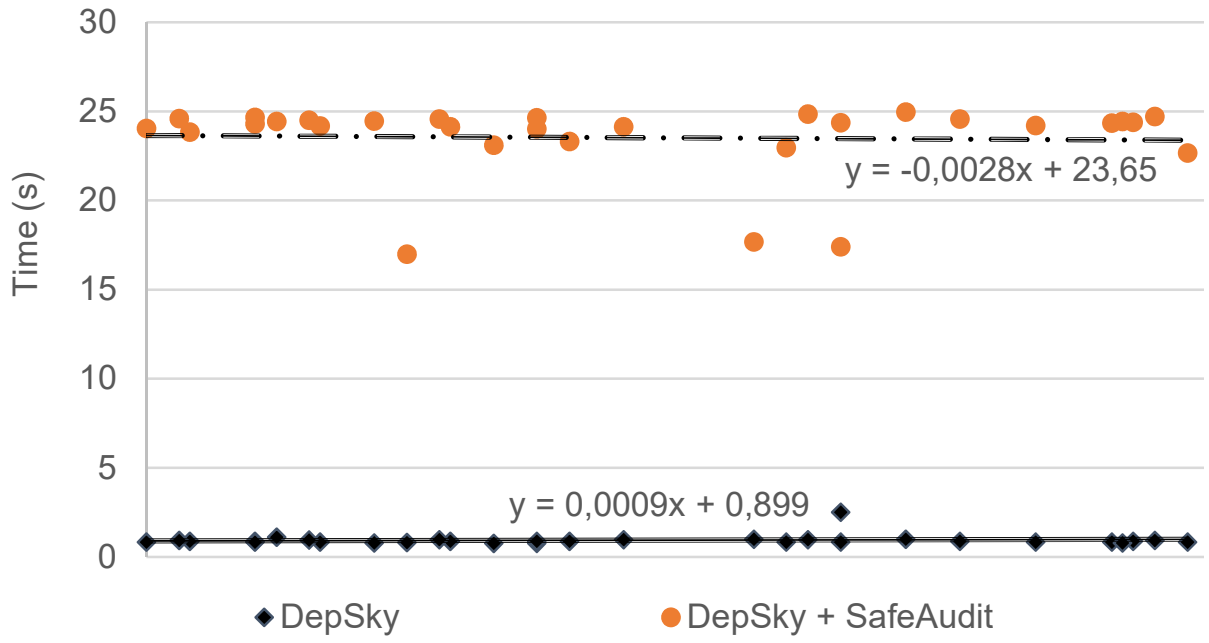

Figure 5.5: Comparison between using SCFS as standalone or integrated with SafeAudit. The Y axis represents the time taken to upload data to the clouds.

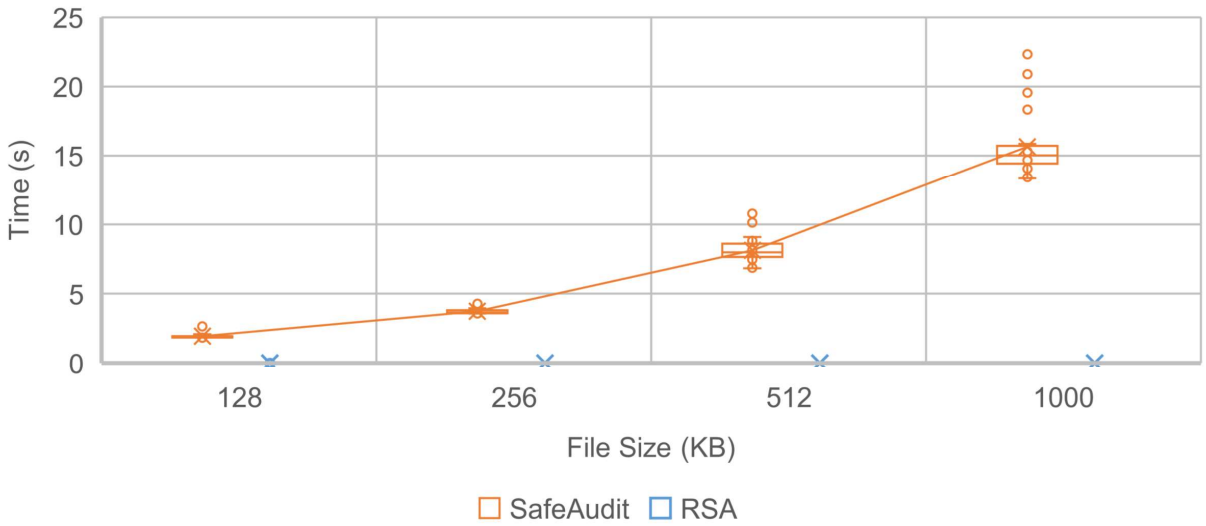

Figure 5.6: Comparison between verifying file integrity proofs using SafeAudit with verifying the integrity RSA+SHA-1 digital signatures. The X axis represents the size of the file in Kilobytes and the Y axis the time required for verification.

5.1.4.3 Proof Verification Benchmark

In order to test the performance impact required on the auditors' devices for verifying the proofs obtained from the cloud, proof verifier was tested on the users' device testbed and compared to verifying a file using RSA digital signatures. As seen on Figure 5.6 the results obtained shows that the time necessary for verifying a signature with SafeAudit increases linearly and is slower than RSA digital signatures. This is due to the fact that for verifying a proof using SafeAudit or the original SW scheme, it is necessary to multiply all the identifiers of the blocks audited and it does not scale well as data grows.

For example verifying 1MB of data involves multiplying the ids of 25600 blocks which increase time to these values obtained.

5.1.5 Experimental Evaluation Takeaways

The results discussed show that SAFEAUDIT is a good integrity verification mechanism for users that periodically want to verify their storage without having to suffer with high bandwidth consumption costs or are interested in optimizing the amount of money spent on integrity verification. Regarding the money charges, using SAFEAUDIT for verifying commercial clouds has two types of charges: an initial fee for storing data and signatures on the cloud (which is the double of only storing the data itself); and a fee payed whenever the integrity proofs are generated (which is 30% lower than downloading data from the storage cloud).

Besides the aforementioned benefits of using this library for integrity verification, the results also show that SAFEAUDIT requires considerable computational power on the users and auditors devices, for signing the data and verifying the proofs. This leads to a performance impact on cloud-backed applications.

5.2 Functional Evaluation

As discussed in Section 2, SAFEAUDIT was designed to help users protect cloud storage integrity and to be compatible with several integrity protection techniques. This section will evaluate the compatibility between SAFEAUDIT and those techniques, while providing guidelines for integration. The remainder of this section is structured as follows: Section 5.2.1 explains how to integrate SAFEAUDIT with consistency mechanisms. Section 5.2.2 states how SAFEAUDIT can be combined with concurrency control mechanisms. Section 5.2.3 mentions how use SAFEAUDIT and user revocation methods. Section 5.2.4 explains how to apply redundancy and recovery techniques alongside with SAFEAUDIT .

5.2.1 Integrity Verification in Eventually Consistent Clouds

SAFEAUDIT is compatible with two possible ways of enforce higher levels of consistency on cloud storages: using coordination services; or write-backs. Combining coordination services with SAFEAUDIT could be a plausible choice in a model where no integrity verification errors introduced by weak consistency cloud should be tolerated. To do so, we recommend that both users and auditors use the coordination service and recommend that this service is incorporated in the interaction protocol as follows: after the user stores the data (see Section 3.4.2) the user should register the metadata on coordination service; and when the auditor selects the data to be audited (see Section 3.4.3) this selection should be based on the metadata present on the coordination service. If these modifications are performed SAFEAUDIT becomes compatible with coordination services. Write-backs are also compatible with SAFEAUDIT and in this case it should be entirely compatible with the interaction protocol presented in Section 3.4.

5.2.2 Concurrency Control in Collaborative Environments

To avoid concurrency problems on collaborative environments, SAFEAUDIT is compatible with two possible alternatives: one, where all data operations are coordinated by a distributed lock protocol; and the other, where a users are connected to a proxy for serializing data operations.

If SAFEAUDIT is to be integrated with a distributed lock protocol, we recommend that all users use locks for accessing any data stored on the cloud (including the steps reported in Section 3.4.2). We also recommend for avoiding verification errors when requesting integrity proofs with SAFEAUDIT, that all auditors acquire the locks associated with the data chosen for verification prior to issuing a proof request, and release them after receiving the proof response (see Section 3.4.3).

If proxy solutions are preferred over distributed lock protocols, SAFEAUDIT can also be integrated in this cases. To do so, we recommend that all users and auditors use the same proxy for interacting with the cloud. This allows, users to perform storage operations without data corruption and auditors to request and obtain integrity proofs without being subjected to concurrency errors originated by concurrent write and read operations.

5.2.3 Protecting Integrity with Access Control

Due to the critical significance user revocation has on integrity protection, SAFEAUDIT is fully compatible with the two user revocation mechanisms presented in Chapter 2: forcing remaing trusted users to re-sign data; or use cloud code execution to re-sign the data.

In the case where users are forced to re-sign the data, this technique can be directly applied by re-sign repeating the store data step 3.4.2, but only uploading to the cloud the newly computed signatures.

In the case where cloud re-signatures are used, the compatibility more complex and it is recommendable to use proxy re-signatures implemented in PANDA [11], since it is built for SW and BLS signatures as is SAFEAUDIT. However since both systems are not entirely compatible, it is recommended to proceed with caution, since the mechanism chosen should produce a signature exactly as described in Section 4.1.3.2, including the point compression technique incorporated for reducing data's size.

5.2.4 Recovering from Integrity Attacks

SAFEAUDIT could be easily integrated with both aforementioned methods of restoring data integrity affected by integrity attacks (recovering data based on complete copies, or based on erasure codes dispersed to several clouds). To do so, we recommend that when the auditor detects and reports back to the user an integrity problem present on a cloud, the user immediately restores the corrupted data based on the unaffected clouds and for re-upload the data according to the Store Data step of the SAFEAUDIT's interaction protocol (see Section 3.4.2).

5.3 Summary

This chapter presented evaluation conducted on the SAFEAUDIT library.

The first section of this chapter described the experimental evaluation conducted on the SAFEAUDIT library and proved that performing integrity verification using this library requires low bandwidth consumption costs (always 60 bytes for any data verified using this library) and low monetary costs (30% lower than downloading data from the storage cloud). However, it also alerted to the fact users that use this library pay more monetary storage costs (which is the double of only storing the data itself) and also warned that when integrated with cloud-backed applications requires high computational power for generating homomorphic digital signatures or verifying the integrity proofs.

The second section of this chapter evaluated the compatibility between SAFEAUDIT and the techniques that could be employed alongside with integrity verification mechanisms to improve the overall

protection of the commercial cloud storage, previously mentioned in Chapter 2. The evaluation conducted shows that SafeAudit is entirely compatible with a panoply of integrity protection techniques described in this dissertation: consistency techniques (coordination services and write-backs); concurrency control (distributed lock protocols and proxies); signature revocation techniques (force re-signature of data by trusted user); and redundancy techniques (multiple copies of the data, erasure codes and secret sharing).

Chapter 6

Conclusion

This dissertation revolved on the problem of performing integrity verification on commercial cloud storage, which provide no provable guarantees to their costumers about the correctness of their storage state. Several works had already identified and partially solved this problem by incorporating additional protection mechanisms on cloud-backed applications for preventing, detecting and reacting against integrity attacks that compromise the cloud storage. Regarding the detection of these attacks the solutions proposed normally required high quantities of data to be downloaded to perform the verification and subjected the user to high bandwidth consumption costs, high monetary costs and latency problems.

To fix the aforementioned problems of performing integrity verification on commercial clouds this dissertation leveraged knowledge from previous homomorphic integrity verification schemes and proposed SafeAudit, which is the first software library that allows cloud-backed applications to obtain provable guarantees about the correctness of the data, without having to download it. To do so, the library resorts to code execution capabilities currently offered by commercial clouds for aggregating all the information about the data into a compact proof of integrity that serves as assurance for the cloud-backed applications.

Although SafeAudit is capable of providing integrity verification with lower monetary costs and bandwidth requirements than detection mechanisms that use cryptographic hashes, its usage is targeted at providing integrity proofs without retrieving the data. SafeAudit does not aim to substitute the current mechanisms that provide integrity assurances of data read from the cloud (MACs, signatures), but to provide proofs when downloading the data is not needed. SafeAudit should be used alongside those mechanisms for being used in these situations, e.g., when data is stored unmodified for long periods of time.

To prove the compatibility between SafeAudit and the current cloud-backed solutions, this library was integrated with the SCFS cloud-backed application and Amazon AWS. This integration has shown, not only that this library is capable of being integrated with the current cloud-backed solutions, but also that this library is also capable of being used alongside with other integrity protection mechanisms, including reaction mechanisms, such as erasure codes, that allow users to sustain and recover from the attacks detected by SafeAudit.

6.1 Achievements

This dissertation reached two important achievements. The first achievement is the construction of the SafeAudit software library, which is the first integrity verification mechanism for commercial cloud that requires low monetary costs for verification (30% lower than the conventional RSA digital signatures)

and has low and constant bandwidth consumptions (60 bytes regardless of the data verified).

The second achievement is the identification of the situations where homomorphic integrity verification schemes are preferable against other verification schemes for checking commercial cloud storage integrity with reduced monetary costs. Previous works never quantified the amount of money paid when performing integrity verification with homomorphic schemes. In this dissertation it was discovered that homomorphic verification schemes are preferable if the data verified over time is bigger than the amount of data being written to the storage cloud.

6.2 Future Work

As future work SAFEAUDIT could be continued by addressing the following points:

- One possible way to improve SAFEAUDIT is to allow this library to handle parallel computations on multi-core processors. This would greatly improve SAFEAUDIT, since one of the limitations is the high computational load on the cloud-backed applications for generating homomorphic digital signatures and checking integrity proofs. By making use of parallel computations, the time spent on performing this tasks would be greatly reduced. Furthermore since code execution on commercial clouds is paid by the time spent on computation, performing proof generation in parallel may also contribute substantially for reducing monetary costs.

- Another great possible contribution to SAFEAUDIT is to integrate with other commercial clouds that provide code execution capabilities. To test SAFEAUDIT an experimental evaluation was performed on Amazon AWS, it would be interesting to investigate the variation of monetary prices paid for proof generation in different commercial clouds.

- Another possible contribution for continuing the work of this dissertation is to integrate SAFEAUDIT with the integrity protection techniques presented in the Chapter 2 that were not contemplated on the proof-of-concept, demonstrating their compatibility on a practical standpoint. This can be achieved by following the guidelines proposed in the Chapter 5.

Bibliography

[1] Q. Hassan, "Demystifying cloud computing," *The Journal of Defense Software Engineering*, pp. 16–21, 2011.

[2] C. Wang, K. Ren, W. Lou, and J. Li, "Toward publicly auditable secure cloud data storage services.," *IEEE network*, vol. 24, no. 4, pp. 19–24, 2010.

[3] "Infosecurity Europe 2010: Data integrity attacks to become more common, say experts." `http://www.computerweekly.com/news/1280092630/Infosecurity-Europe-2010-Data-integrity-attacks-to-become-more-common-say-experts`. Accessed: 2016-05-29.

[4] "Ransomware a threat to cloud services, too." `http://krebsonsecurity.com/2016/01/ransomware-a-threat-to-cloud-services-too/`. Accessed: 2016-05-29.

[5] "Top Cyber Threats for 2016; integrity attack on financial sector likely." `http://www.carriermanagement.com/news/2015/11/30/148345.htm`. Accessed: 2016-05-29.

[6] D. Eastlake 3rd and P. Jones, "US secure hash algorithm 1 (SHA1)," tech. rep., 2001. Accessed: 2016-05-29.

[7] "Amazon Web Services." `https://aws.amazon.com/`. Accessed: 2016-05-29.

[8] "Amazon S3 Pricing." `https://aws.amazon.com/s3/pricing/`. Accessed: 2016-05-29.

[9] G. Ateniese, S. Kamara, and J. Katz, "Proofs of storage from homomorphic identification protocols," in *International Conference on the Theory and Application of Cryptology and Information Security*, pp. 319–333, Springer, 2009.

[10] C. Wang, Q. Wang, K. Ren, and W. Lou, "Privacy-preserving public auditing for data storage security in cloud computing," in *INFOCOM, 2010 Proceedings IEEE*, pp. 1–9, Ieee, 2010.

[11] B. Wang, B. Li, and H. Li, "Panda: public auditing for shared data with efficient user revocation in the cloud," *IEEE Transactions on services computing*, vol. 8, no. 1, pp. 92–106, 2015.

[12] K. D. Bowers, A. Juels, and A. Oprea, "Hail: a high-availability and integrity layer for cloud storage," in *Proceedings of the 16th ACM conference on Computer and communications security*, pp. 187–198, ACM, 2009.

[13] N. T. F. de Carvalho, "A practical validation of homomorphic message authentication schemes," Master's thesis, University of Minho, 2014.

[14] H. Shacham and B. Waters, "Compact proofs of retrievability," in *International Conference on the Theory and Application of Cryptology and Information Security*, pp. 90–107, Springer, 2008.

[15] P. S. Barreto and M. Naehrig, “Pairing-friendly elliptic curves of prime order,” in *International Workshop on Selected Areas in Cryptography*, pp. 319–331, Springer, 2005.

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

[17] A. Bessani, R. Mendes, T. Oliveira, N. Neves, M. Correia, M. Pasin, and P. Verissimo, “Scfs: a shared cloud-backed file system,” in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pp. 169–180, 2014.

[18] R. L. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems,” *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[19] A. Subbiah and D. M. Blough, “An approach for fault tolerant and secure data storage in collaborative work environments,” in *Proceedings of the 2005 ACM workshop on Storage security and survivability*, pp. 84–93, ACM, 2005.

[20] R. A. Popa, J. R. Lorch, D. Molnar, H. J. Wang, and L. Zhuang, “Enabling security in cloud storage slas with cloudproof.,” in *USENIX Annual Technical Conference*, vol. 242, 2011.

[21] M. W. Storer, K. M. Greenan, E. L. Miller, and K. Voruganti, “Potshards: secure long-term storage without encryption,” in *2007 USENIX Annual Technical Conference*, USENIX Association, 2008.

[22] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa, “Depsky: dependable and secure storage in a cloud-of-clouds,” *ACM Transactions on Storage (TOS)*, vol. 9, no. 4, p. 12, 2013.

[23] D. Dobre, P. Viotti, and M. Vukolić, “Hybris: Robust hybrid cloud storage,” in *Proceedings of the ACM Symposium on Cloud Computing*, pp. 1–14, ACM, 2014.

[24] D. Grolimund, L. Meisser, S. Schmid, and R. Wattenhofer, “Cryptree: A folder tree structure for cryptographic file systems,” in *2006 25th IEEE Symposium on Reliable Distributed Systems (SRDS’06)*, pp. 189–198, IEEE, 2006.

[25] R. Zhao, C. Yue, B. Tak, and C. Tang, “Safesky: A secure cloud storage middleware for end-user applications,” in *Reliable Distributed Systems (SRDS), 2015 IEEE 34th Symposium on*, pp. 21–30, IEEE, 2015.

[26] E. Stefanov, M. van Dijk, A. Juels, and A. Oprea, “Iris: A scalable cloud file system with efficient integrity checks,” in *Proceedings of the 28th Annual Computer Security Applications Conference*, pp. 229–238, ACM, 2012.

[27] R. C. Merkle, “Secrecy, authentication, and public key systems.,” 1979.

[28] D. Boneh and X. Boyen, “Short signatures without random oracles,” in *International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 56–73, Springer, 2004.

[29] C. C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia, “Dynamic provable data possession,” *ACM Transactions on Information and System Security (TISSEC)*, vol. 17, no. 4, p. 15, 2015.

[30] A. Juels and B. S. Kaliski Jr, “Pors: Proofs of retrievability for large files,” in *Proceedings of the 14th ACM conference on Computer and communications security*, pp. 584–597, Acm, 2007.

[31] D. Dobre, G. Karame, W. Li, M. Majuntke, N. Suri, and M. Vukolić, “Powerstore: proofs of writing for efficient and robust storage,” in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pp. 285–298, ACM, 2013.

[32] C. Băsescu, C. Cachin, I. Eyal, R. Haas, A. Sorniotti, M. Vukolić, and I. Zachevsky, "Robust data sharing with key-value stores," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pp. 1–12, IEEE, 2012.

[33] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon, "Racs: a case for cloud storage diversity," in *Proceedings of the 1st ACM symposium on Cloud computing*, pp. 229–240, ACM, 2010.

[34] W. Lin, D. M. Chiu, and Y. Lee, "Erasure code replication revisited.," in *Peer-to-Peer Computing*, pp. 90–97, 2004.

[35] M. Blaze, G. Bleumer, and M. Strauss, "Divertible protocols and atomic proxy cryptography," in *International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 127–144, Springer, 1998.

[36] H. Krawczyk, "Secret sharing made short," in *Annual International Cryptology Conference*, pp. 136–146, Springer, 1993.

[37] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *ACM SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002.

[38] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 205–220, 2007.

[39] W. Vogels, "Eventually consistent," *Communications of the ACM*, vol. 52, no. 1, pp. 40–44, 2009.

[40] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems.," in *USENIX Annual Technical Conference*, vol. 8, p. 9, 2010.

[41] A. N. Bessani, E. P. Alchieri, M. Correia, and J. S. Fraga, "Depspace: a byzantine fault-tolerant coordination service," in *ACM SIGOPS Operating Systems Review*, vol. 42, pp. 163–176, ACM, 2008.

[42] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 3, pp. 463–492, 1990.

[43] C. H. Papadimitriou, "The serializability of concurrent database updates," *Journal of the ACM (JACM)*, vol. 26, no. 4, pp. 631–653, 1979.

[44] "Overview of managing aws access." `http://docs.aws.amazon.com/AmazonS3/latest/dev/access-control-overview.html`. Accessed: 2016-05-29.

[45] "Amazon AWS Lambda." `https://aws.amazon.com/lambda/`. Accessed: 2016-05-29.

[46] R. Padilha and F. Pedone, "Belisarius: Bft storage with confidentiality," in *Network Computing and Applications (NCA), 2011 10th IEEE International Symposium on*, pp. 9–16, IEEE, 2011.

[47] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, no. 3, pp. 382–401, 1982.

[48] M. Vrable, S. Savage, and G. M. Voelker, "Cumulus: Filesystem backup to the cloud," *ACM Transactions on Storage (TOS)*, vol. 5, no. 4, p. 14, 2009.

[49] "Microsoft Azure Storage." `https://azure.microsoft.com/en-us/services/storage/`. Accessed: 2016-05-29.

[50] "RacketSpace Cloud Files." https://www.rackspace.com/cloud/files. Accessed: 2016-05-29.

[51] "Google Cloud Storage." https://cloud.google.com/storage/. Accessed: 2016-05-29.

[52] J. Y. Chung, C. Joe-Wong, S. Ha, J. W.-K. Hong, and M. Chiang, "Cyrus: Towards client-defined cloud storage," in *Proceedings of the Tenth European Conference on Computer Systems*, p. 17, ACM, 2015.

[53] "What is iCloud?." https://help.apple.com/icloud/?lang=en#/mmfc0efea4. Accessed: 2016-10-05.

[54] "Apple said to move part of cloud business from AWS to Google.." http://www.infoworld.com/article/3045438/cloud-computing/apple-said-to-move-part-of-cloud-business-from-aws-to-google.html. Accessed: 2016-10-05.

[55] "MagicBox, DropBox new storage infrastructure.." https://blogs.dropbox.com/tech/2016/05/inside-the-magic-pocket/. Accessed: 2016-10-05.

[56] "HPE Helion OpenStack." http://www8.hp.com/us/en/cloud/hphelion-openstack.html. Accessed: 2016-05-29.

[57] "Cisco Private Cloud." http://www.cisco.com/c/en/us/solutions/data-center-virtualization/private-cloud/index.html. Accessed: 2016-05-29.

[58] "VMWare's Vsphere." http://vcloud.vmware.com/uk/service-offering/virtual-private-cloud. Accessed: 2016-05-29.

[59] "Oracle Private Cloud." https://www.oracle.com/cloud/private-cloud.html. Accessed: 2016-05-29.

[60] A. De Caro and V. Iovino, "jPBC: Java pairing based cryptography," in *Proceedings of the 16th IEEE Symposium on Computers and Communications, 2011*, pp. 850–855, IEEE, 2011.

[61] "Amazon S3." https://aws.amazon.com/s3/. Accessed: 2016-05-29.

[62] "Amazon EC2 - Virtual Server Hosting." https://aws.amazon.com/ec2/. Accessed: 2016-05-29.