SafeAudit: A Software Library for Efficient Data Integrity Verification on Commercial Clouds

Filipe Miguel Marcos Apolinário
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa
Email: filipe.apolinario@tecnico.ulisboa.pt

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 data has to be downloaded to perform the check.

This article explores homomorphic authentication with digital signatures for integrity verification of data stored in commercial cloud services 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 bandwidth consumption and low network 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.

I. INTRODUCTION

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 [20]. The data-centers are managed by the cloud provider who is paid 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 [30].

According to recent studies [5, 7] there is a growth tendency of integrity attacks in public infrastructures, such as hospitals and other public enterprises, who are subjected to these attacks in order to pay to the responsible 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 additional integrity control mechanisms they store on the cloud 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) based on cryptographic hashes. Whenever the user wants to guarantee that the integrity of the data is kept in the cloud, the user: first downloads the data and the corresponding cryptographic hash from the cloud; then verifies that the downloaded data’s cryptographic hash matches the signed hash 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 (as identified in [5]).

In order to reduce the users’ monetary charges and allow clients to obtain integrity assurances with low and constant bandwidth consumption costs, several works [5, 13, 14, 16, 25, 30, 31] 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 homomor-
phic message authentication codes, that provide private verifiability (only the person who possesses the secret key can verify it).

Contrary to prior works that explore the potential of compact integrity proofs by presenting theoretical demonstrations of their feasibility and security analysis [8], [13], [14], [16], [25], [30], [31], this article explores practical applicability of these techniques by presenting a fully deployable system, capable of being integrated as-is on real world storage solutions, including commercial clouds and cloud-backed applications.

The rest of this article presents SAFEAUDIT; a software library that implements the Shacham-Waters integrity verification (SW) scheme introduced in [25] and adapts it to commercial cloud solutions. Furthermore, SAFEAUDIT optimizes the original SW scheme and provides: an overall performance increase, by carefully selecting pairing-friendly elliptic curves [9] for the SW scheme’s parameterization; and a storage costs decrease by half of the demanded by the original scheme, with the point compression technique presented in [22]. Contrary to all 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 cryptography in order to fully understand its usage. To demonstrate its full potential, this library was integrated with Amazon AWS commercial cloud and the SCFS [11] cloud-backed storage application, and the results obtained show that SAFEAUDIT when compared to commonly used RSA digital signatures [24], is able to provide lower monetary costs (30% less) and lower bandwidth consumption costs.

The main contributions of this article 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 with discussion of the results obtained on using this library as standalone, or integrated with the SCFS and the Amazon AWS commercial cloud.

This work 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.

The remaining of this article is structured as follows: Section II explains the design and background concepts that were applied on the SAFEAUDIT software library; Section III covers the implementation of SAFEAUDIT stating its principal algorithms and their functionality. Section IV details what changes were necessary to be performed on SCFS for integrating the SAFEAUDIT software library; Section V explains how SAFEAUDIT was implemented and integrated with Amazon AWS, while presenting the evaluation and discussion of using this library as standalone or as an integrated solution with AWS and SCFS; Section VI compares the overall contributions of this article with the current works present in the state of the art; and Section VII concludes the article.

II. SAFEAUDIT

The goal of this article 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 [4]), for providing integrity proofs on the stored data; and cloud-backed storage applications (such as [10], [11], [26], [32]), 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 section the conceptual design of SAFEAUDIT will be explained in detail. Section II-A explains the entities involved and their roles for integrity preservation. Section II-B states which threats SAFEAUDIT is able to protect the users’ storage and which necessary conditions must hold for SAFEAUDIT to behave properly. Section II-C explains the preliminary security and mathematici concepts to which the SAFEAUDIT was based upon. Section II-D describes the interaction protocol required to be performed by each entity for integrity preservation while using SAFEAUDIT. Finally, Section II-E explains how SAFEAUDIT tries to reduce storage monetary costs necessary for storing homomorphic digital signatures.

A. 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 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.

B. 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.

C. Preliminary Concepts

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

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 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 of a multiplicative cyclic group of order 6 and generator $g = 2$. The multiplicative group is composed of six members $\{g^0, g^1, g^2, g^3, g^4, g^5\}$, generated with the number $g$, into another multiplicative cyclic group ($G_T$) of the same prime order ($p$), i.e., $e: G \times G \rightarrow G_T$. By using the pairing the cryptographic techniques ensure the following properties: Computability: 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 $e(u^a, v^b) = e(u, v)^{ab}$.

1) 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: Computability: 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 $e(u^a, v^b) = e(u, v)^{ab}$.

2) BLS Signature Scheme: In order to provide integrity control of a file SAFEAudit uses the BLS signature scheme 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 $g^s$.

- **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) 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) \cite{25} 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:

$$\alpha = \sum_{i=1}^{n} d_i \times chal_i \in Z_p \text{ and } \beta = \prod_{i=1}^{n} \theta_{id_i}^{chal_i} \in G.$$  

- **Proof Verification**: given the proof $(\alpha, \beta)$, 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:

$$e(\beta, g) = e(\prod_{i=1}^{n} H(id_i) \times w^d, pk)$$

If verification is positive integrity is assured.

4) **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 \cite{22}. While there is support for those six different types of curves, as recommended by both BLS and SW authors in \cite{25} and \cite{22}, the best suitable curves to be used in pairing initialization are the pairing-friendly elliptic curves of prime order \cite{9} (also called named type F curves and described in Section 4.14 of \cite{12}).

D. SafeAudit’s Interaction Protocol

In order to preserve the integrity of the data stored on the cloud using SafeAudit, the entities involved (cloud, user and auditor) need to follow the SafeAudit’s interaction protocol. As will be explained in the rest of this subsection, the interaction protocol is divided into four parts: set up (Section II-D1), store data (Section II-D2), request and verify of integrity proof (Section II-D3), and proof generation (Section II-D4).

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

- The user generates his secret/public asymmetric key pair and the signature parameter $(w)$ for signing and verifying data under the SW scheme, using respectively the SafeAudit’s key and random number generators (further explained in Section \cite{11}).

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

- The user configures the cloud for listening to the auditor’s requests and for responding to them, with the execution of SafeAudit’s proof generator service (further explained in Section \cite{11}).

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

2) **Store Data**: When the user stores data in the cloud, all data must be divided into blocks belonging to $Z_p$ and signed. The SafeAudit’s signature generator (further explained in Section \cite{11}) automates these tasks and produces a signature equivalent to the SW sign block step (as seen in Section II-C3). 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...
After the signature of the data is obtained, the user stores both the data and signature in the cloud.

3) Requesting and Verifying Integrity Proofs: In SafeAudit’s iteration protocol, the auditor is responsible for integrity verification. To do so, whenever the auditor wants to obtain integrity proofs of a dataset stored on the cloud, he must perform the following steps:

- Select a dataset composed of \( x \) data elements (vector \([0,...,x]\)) so that the cost of obtaining the proof for the \( x \) elements is at most the price the user wants to pay for the audit.
- Generate a random challenge (number belonging to \( Z_p \)) for each of the \( x \) data elements chosen, using the SafeAudit’s random number generator.
- Issue the integrity proof request to the cloud specifying: identifiers’ vector \([id_0, ..., id_x]\) and the corresponding challenge vector \([chal_0, ..., chal_x]\).
- Upon receiving a response from the cloud, with the requested integrity proof, the auditor verifies it using the SafeAudit’s proof verifier (further explained in Section III). 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.

4) Generating Integrity Proofs: Whenever the cloud receives an integrity proof request of a given dataset (as described in Section II-D3), 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 (\( \beta \)); and the aggregation of data provided (\( \alpha \)), by using SafeAudit’s proof generator (further explained in Section III). 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 \( \alpha \) and \( \beta \). This step corresponds to the proof generation step of the SW scheme.
4) Respond to the requester with the integrity proof (\( \alpha \) and \( \beta \)).

E. 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 [25], the best case scenario, where type F curves \cite{25} are used, the SW scheme produces signatures that are twice the size of the signed data. Further aggravating the problem if type A \cite{12} curves are used, signatures size is given by the following formula: \(|\text{signature}| = \frac{|G| \times |\text{data}|}{|Z_p|}\).

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 [22]. 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 \cite{9} are used are the same size of the original data.

III. SafeAudit’s Implementation

In order to simplify integration with the users’ cloud-backed applications, commercial clouds, and auditors, the SafeAudit software library is composed of several components, each automatizing a task of the SafeAudit’s interaction protocol. The Pairing Generator component, allows auditors to generate all the setup parameters, required to initialize pairing-based-cryptography. The Key Generator component allows users to generate their asymmetric secret/public key pair and signature parameter (\( w \)). The Signature Generator component allows users to sign their data. The Random Generator component allows entities to generate random numbers belonging to any field of their choosing (\( Z_p, G \) or \( G_T \)). The Proof Generator component allows clouds to generate integrity proofs. The Proof Verifier component allows auditors to verify the proofs obtained from the cloud. Finally, the Random Generator component allows generation of random numbers belonging to any of \( Z_p, G \) or \( G_T \) fields.

IV. Extending SCFS with SafeAudit

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) \cite{11}.
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 [21]), even if weak consistency storage cloud services are used (e.g., services that guarantee only eventual consistency [29]).

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 [10]. DepSky provides an API for uploading and operating with a set clouds, while enforcing fault tolerance, lock-in resilience, 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 RSA digital signatures [24]. 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 cloud, 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 cloud, data is also signed using the SAFEAUDIT’s signature generator and the signature is also stored on the cloud. As seen in Figure 2 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. 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.

V. 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) [15], that implements Multilinear Maps and all the operations required by these mechanisms for manipulating those maps.

The cloud used during the implementation was Amazon AWS [4]. S3 [3] was used as storage and Lambda [1] for executing Java code at the auditors request through a REST API (i.e., the executing steps in Section II-D4). 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 [2]) 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 V-A 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 V-B presents bandwidth consumption evaluation and Section V-C 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 V-D presents the performance benchmarks and comments the results achieved.

A. Testbed Details

The evaluation benchmarks were performed using a dataset containing 1 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 \cite{18} with 1024 bit keys and tuned to use SHA-1 \cite{18} 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.

B. Bandwidth Evaluation

Integrity proof generation was evaluated in terms of bandwidth consumption. Figure 3 shows the result of 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 proves 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 encoded for reading 60 bytes are negligible).

C. 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.

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 4 compares the storage increase when using SAFEAUDIT, SW, or RSA + SHA-1 digital signatures as

- Figure 3. 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 4. 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.
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).

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, the results obtained for generating integrity proofs increase linearly as storage grows. Furthermore, as seen in Table 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%.

D. 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 V-D1 comparing the time taken to sign data using SAFEAUDIT and RSA + SHA-1 digital signatures; and the other, detailed in Section V-D2 for evaluating the overhead SCFS has when integrated SAFEAUDIT.

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 6. The time required for signing data using the SAFEAUDIT increases linearly and is in the order of seconds and is very slow 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, reducing abruptly the time for signing. In SAFEAUDIT, the SW scheme and all the other publicly verifiable 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 extremely slow and does not fit the criteria of the fast signature generation mechanisms.

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 7 and show that the integration with SAFEAUDIT increases time severely. This is not different from the results presented in Figure 6 and prove that integrating this signature generator can be a source of overhead in cloud-backed applications.

<table>
<thead>
<tr>
<th>File Size (KB)</th>
<th>Average Execution Time (s)</th>
<th>Execution cost(^1) (microUS$)</th>
<th>Read Costs(^2) (microUS$)</th>
<th>Savings (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>8.82</td>
<td>11.14</td>
<td>29.58</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>7.44</td>
<td>10.56</td>
<td>24.84</td>
<td>32.29</td>
</tr>
<tr>
<td>512</td>
<td>14.67</td>
<td>30.57</td>
<td>46.08</td>
<td>33.64</td>
</tr>
<tr>
<td>1024</td>
<td>28.35</td>
<td>59.38</td>
<td>79.54</td>
<td>31.90</td>
</tr>
</tbody>
</table>

\(^1\)Assumed 0.09 microUS$ for each 0.1s of computation
\(^2\)Assumed 0.09 microUS$ for 1GB read from the cloud storage

Figure 5. 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.

Table 1. Price Comparison between Generating Proofs and Reading Data.

Figure 6. 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.

Figure 7. Comparison between using SCFS as standalone or integrated with SAFEAUDIT. The Y axis represents the time taken to upload data to the clouds.
that perform it using authentication metadata. The integrity verification with a voting scheme; and the others can be divided into two categories: those that perform verification on cloud storages. The studied mechanisms are closely related with normal users’ devices. More precisely, it is closely related to existing cloud-backed applications and limit compatibility with normal users’ devices. In case of the systems that use non-homomorphic proofs verifying a file using RSA digital signatures. As seen on Figure 8, the results obtained shows that time necessary for verifying a signature increases linearly and is very slow compared with RSA digital signatures. This is due to the fact that for verifying a proof using SAFEAUDIT and on 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 involves multiplying the ids of 25600 blocks which increase time to this values obtained.

E. 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 high computational power on the users and auditors devices, for signing the data and verifying the proofs. These problems lead to a critical performance impact on cloud-backed applications, which may further complicate this library’s integration with the existing cloud-backed applications and limit compatibility with normal users’ devices.

VI. Related Work

SAFEAUDIT is inserted on the cloud integrity verification research area. More precisely, it is closely related with mechanisms that allows users to perform integrity verification on cloud storages. 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.

In systems involving voting schemes for integrity verification that store data on several clouds, such as [28], besides being able to verify integrity, users can pinpoint which clouds were compromised. However since it requires reading data from several clouds, making the user more vulnerable to network latencies and demanding high bandwidth consumption costs when compared with systems such as SAFEAUDIT. 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.

In systems involving authentication metadata for performing integrity verification on the cloud, such as [8], [10], [11], [13], [14], [16], [19], [23], [25], [28], [30], [32], verifying the data’s integrity is performed by requesting integrity proofs containing both the data and control data-structures, that are then verified by the user using authentication metadata. The integrity proofs used can be non-homomorphic or homomorphic.

In case of the systems that use non-homomorphic proofs verifying the integrity of a dataset requires to obtain each pair of data and integrity control structure separately and perform verification individually. Due to that fact, using these proofs demands more bandwidth consumption, and an higher monetary costs, when comparing to SAFEAUDIT who is able to maintain bandwidth consumption low and constant as the verified dataset’s size increases and reduce monetary read costs by aggregating data and signatures before sending.

In case of the systems that use homomorphic proofs, such as [8], [13], [14], [16], [25], [30], [31] or any other who uses SAFEAUDIT (such as the integrated version of SCFS present in this article), the integrity proofs obtained is composed of two aggregation structures, one for the data and the other for integrity control data-structures. This allows data 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 were not yet practical and only had theoretical demonstrations of its feasibility. SAFEAUDIT is the first to prove it is possible integrate with conventional cloud-backed applications and also it is the first to benchmark extensively an homomorphic scheme in a way that is possible to clearly understand the advantages and disadvantages of using homomorphic schemes.

Among the several articles studied that focus on using integrity verification with homomorphic proofs, the articles [12], [23], [25] served as inspiration for developing SAFEAUDIT. [22] explains how to use pairing-based-cryptography for creating highly compact public verifiable digital signatures, the BLS signatures, that preserve homomorphism. [12] explains what are BLS signatures and how they should be constructed in practice. [25] explains the Shacham-Waters integrity verification (SW) scheme which expands the capabilities of BLS and integrates them in a protocol for obtaining homomorphic verification.
VII. Conclusion

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

SafeAudit was integrated with the SCFS cloud-backed application and Amazon AWS; and as shown on the evaluation presented in this article, an integrity verification performed with SafeAudit requires low and constant bandwidth consumption costs of 60 bytes and has about 30% lower monetary costs for integrity verification than the conventional RSA digital signatures, at the expense of storing additional metadata on the cloud storage and demanding more data processing on the verifiers’ devices.

The main achievements provided by this work are: the construction of the SafeAudit software library, that provides integrity verification on commercial clouds with low bandwidth consumptions and low monetary costs; and the first detailed study that identifies the situations where homomorphic integrity verification schemes are the preferential method of checking the commercial cloud storage with low monetary costs.

REFERENCES


