S-Audit: Efficient Data Integrity Verification for Cloud Storage

Filipe Apolinário*†, Miguel L. Pardal*, Miguel Correia*
*INESC-ID, Instituto Superior Técnico, Universidade de Lisboa †INOV-INESC Inovação
Email: filipe.apolinario@inov.pt, miguel.pardal@tecnico.ulisboa.pt, miguel.p.correia@tecnico.ulisboa.pt

***Abstract*—Commercial cloud storage services are being widely adopted. The most common integrity verification methods for data stored remotely are based on cryptographic hashes and digital signatures. These allow checking that the data has not been tampered while stored in the cloud. However, both require downloading all the data before doing the verification, with significant time and monetary costs.**

This paper presents S-AUDIT, a service that provides integrity verification of data stored in commercial clouds. S-AUDIT uses homomorphic authentication with digital signatures to avoid retrieving the protected data from the cloud. The service was integrated with a cloud-backed file system called SCFS to show how it can be used in practice. Our experimental evaluation shows that using S-AUDIT is 7.1% cheaper than using RSA signatures when the integrity of the data is verified monthly, and 34.9% when verified weekly, in a typical setting.

I. INTRODUCTION

Commercial cloud storage services such as Dropbox, Google Files, Microsoft OneDrive, and Amazon S3, are being widely adopted. With these services and together with research cloud-backed storage solutions [10], [39], [36], [31], [11], [15], [43], [32], users can store their data in remote datacenters and benefit from resource elasticity and world-wide access [25]. Datacenters are managed by the cloud providers, which are paid by the users for the resources they consume. Despite these benefits, users inevitably lose the control of their data, at least in part. Moreover, there is a recent growth on attacks against the integrity of data stored in public infrastructures [7], [5], [6]. The affected organizations, e.g., hospitals and companies, can fall victim of *ransomware* that encrypts their data. Next they are blackmailed to get it back, as in the recent case of WannaCry [30]. Similar attacks and other modifications to the data may also be done by malicious insiders [16], [24]. The damage caused by these changes may be visible immediately or may go unnoticed for a long time until data is used again.

Nowadays data owners use integrity control mechanisms based on *cryptographic hashes* [29], [20] to protect their outsourced storage. *Digital signatures* are used for *collaborative storage* when data is shared among several cloud users, and MACs (Message Authentication Codes) are used for *private storage* when data is used by a single cloud user. For performing the integrity control, users need to have some personal key: an asymmetric private/public key pair for digital signatures or a symmetric key for MACs. The user data is stored together either with a signature or a MAC, computed with the user's private key or symmetric key, respectively. Whenever the user wants to guarantee that the integrity of the data is preserved, the user must first download the data and the corresponding signature/MAC from the cloud, then verify if the data matches the signature/MAC. If they match, the integrity is verified and the user can rest assured that the data was not tampered while in the storage.

Notwithstanding the effectiveness of these mechanisms, if the users do not trust the cloud, they must download all the data to verify it. Therefore, when users are only interested in verifying the integrity of the data, not in reading it, each verification requires an unnecessary download that implies a potentially large bandwidth consumption, delay and monetary cost (downloads have a significant cost in most cloud storage services). For example, consider a user with 1000 files stored on the Amazon Web Services (AWS) cloud [4] in Ireland, each with 1GB of size. If the user wants to check the integrity of every file 4 times a month, he has to download a total of 4TB from the cloud monthly. In this scenario the user is subjected to the latency of downloading 1TB every time and a charge of 360 USD monthly (for every 1GB read, approximately 0.09USD is charged [3]).

In order to reduce delay and bandwidth consumption some works proposed more advanced integrity mechanisms that are *homomorphic*, i.e., that produce integrity control structures that have the same structure as the signed data [8], [42], [40], [13], [41], [18], [35]. These mechanisms provide *verifiability* (data integrity can be verified using proofs) and *unforgeability* (a malicious cloud cannot forge a proof without having the files) without needing to download the data to be verified. These new mechanisms fall into two categories: homomorphic digital signatures, that provide *public verifiability* (anyone can perform the integrity verification); and homomorphic message authentication codes, which provide *private verifiability* (only the user that possesses the secret key can perform the integrity verification). To understand the benefits of these mechanisms consider the previous example of 1000 1GB files stored at AWS. If homomorphic digital signatures with 40-byte public keys are used, a user would have to download only 60 bytes from the cloud during the verification process. Therefore, independently of the size of the data to be verified, integrity verification with these mechanism requires downloading a small proof, with the associated low communication delay and negligible cost.

In contrast with prior works that explore the potential of compact integrity proofs by presenting theoretical demonstrations of their feasibility and security analysis [8], [42], [40],

[13], [41], [18], [35], this work explores the practical applicability of these techniques for verifying data on commercial cloud storage. For that purpose, we present a service capable of being integrated on real world storage solutions, including commercial clouds and cloud-backed applications.

This paper presents S-AUDIT, a software service that improves the Shacham-Waters (SW) integrity verification scheme [35] and adapts it for use with cloud storage. S-AUDIT improves the original SW scheme by providing: an overall performance increase by carefully selecting *pairing-friendly elliptic curves* [9] for SW scheme parametrization; and a storage cost decrease of 50% in relation to the original scheme using *point compression* [27]. Moreover, it leverages the Function-as-a-Service (FaaS) model[1] [21], [26] to reduce cloud costs, by using computation resources in the cloud only when necessary. These improvements make S-AUDIT the most cost-efficient homomorphic verification mechanism for use in commercial clouds.

S-AUDIT was designed as a practical implementation that can be easily plugged into current commercial cloud services and cloud-backed applications. S-AUDIT is simple to use, as using it does not require advanced cryptography knowledge. To demonstrate its potential, the service was deployed at AWS and integrated with the SCFS cloud-backed file system [11]. Our experimental evaluation has shown that using S-AUDIT is 7.1% cheaper than using RSA signatures when the integrity of the data is verified monthly, and 34.9% cheaper when it is verified weekly in a typical setting in AWS.

The main contributions of this paper are: the design and implementation of the S-AUDIT integrity verification service; a protocol for verifying data stored in remote clouds; a proof-of-concept integration of S-AUDIT with a commercial cloud and a cloud-backed file system; and an experimental evaluation of the use of this service standalone and integrated with AWS and SCFS.

II. S-AUDIT

The goal of the S-AUDIT software service is to assure users that all the data they store in the cloud is retrievable with its integrity preserved, i.e., it has not been tampered. This service is envisioned to be easily integrated with: current commercial storage clouds, such as AWS [4], for providing integrity proofs on the stored data; and cloud-backed storage applications (such as [10], [11], [31], [36], [15], [43], [32]), to generate all the necessary digital signatures and automate the request and verification of the integrity proofs supplied by the clouds.

S-AUDIT leverages *homomorphic digital signatures* for integrity control of the stored data, and the computation resources of commercial clouds infrastructures for executing code and generating compact integrity proofs based on the data and signatures present in the cloud storage. Also, by requesting and verifying these small proofs, cloud-backed applications can perform storage integrity control without being constrained by network bandwidth limitations or having to download large quantities of data.

[1]The FaaS model is also called *serverless computing* because the developer does not need to manage the server where the function code is executed. The function code runs when desired and there is no need to have a server instance reserved and awaiting commands.

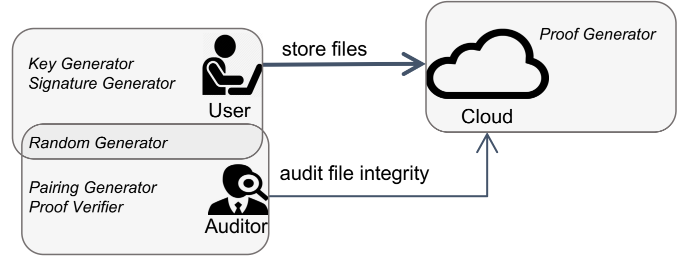

Figure 1. S-AUDIT components and entities.

A. Entities involved in S-Audit

In S-AUDIT there is interaction among three types of entities: *clouds*, *users* and *auditors* (Figure 1) . All these entities need to run S-AUDIT code at some point.

Clouds are commercial public infrastructures that provide to their users both data storage and code execution capabilities for integrity proofs. *Users* are the normal commercial cloud users, who store data on the cloud and perform operations on the stored data (read, write, delete, or set access control permissions). *Auditors* are entities trusted by the users for auditing the data stored in the cloud. They are responsible for issuing and verifying integrity proof requests to the cloud.

B. Threat Model and Assumptions

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

Since the purpose of S-AUDIT is to detect cloud integrity attacks, this service 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 work. The threat model assumes that all communication between entities is authenticated and secure at all times and that neither the user nor the auditor are malicious and their machine do not respond arbitrarily to requests from the other entities.

C. Preliminary Concepts

S-AUDIT is built on top of *multiplicative cyclic groups* and uses pairing-based cryptographic techniques, namely Boneh–Lynn–Shacham (BLS) homomorphic digital signatures [27] and the Shacham-Waters (SW) integrity verification scheme [35]. This section provides mathematical background and summarizes the aforementioned cryptographic techniques.

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* G every member is generated by powering the generator g with integers belonging to Z (the set of all 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 p 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 example, consider a multiplicative cyclic group of order $n = 6$ and generator $g = 2$. The multiplicative group is composed of six members $[g^0 = 1, g^1 = 2, g^2 = 4, g^3 = 8, g^4 = 16, g^5 = 32]$. Linear operations over members of the group are mapped as follows:

- $g^x = g^{x mod 6}$, for example $g^6 = g^0 = 1$ and $g^7 = g^1 = 2$
- $g^x \times g^y = g^{(x+y) mod 6}$, for example $g^1 \times g^2 = g^3 = 8$ and $g^7 \times g^8 = g^{(7+8) mod 6} = g^{(15) mod 6} = g^3 = 8$

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

2) Pairing-based cryptography: S-AUDIT leverages pairing-based cryptography to obtain homomorphism. 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., $e : G \times G \rightarrow G_T$. The pairing enforces 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 $e(u^a, v^b) = e(u, v)^{ab}$.

3) BLS Signature Scheme: In order to provide integrity control of a data file, S-AUDIT uses the *BLS signature scheme* [12] for constructing digital signatures over pairing-based cryptography. To do so, integrity control takes 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}.
- *Signature:* 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.
- *Verification:* 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 is verified, the integrity is assured.

4) SW Scheme for Homomorphic Verifiable Integrity Proofs: The use of BLS signatures ensures the *homomorphic property* for integrity verification and consequently allows the construction of homomorphic verification schemes, where data and signatures are aggregated using additions and multiplications into compact verifiable proofs. This is done because if each file and signature can be divided into blocks of a given size and these blocks can be mapped into multiplicative cyclic groups with $order = size$. 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. In S-AUDIT, the SW integrity verification scheme [35] is used in order to provide homomorphic generation and verification of compact integrity proofs. To do so, under this scheme, integrity control takes 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}.
- *Block Signature:* Given a block with the identifier $id \in Z$ and the data corresponding to the block $d_{id} \in Z_p$, a hash function that maps $H : Z \rightarrow Z_p$, the secret key $sk \in Z_p$, and the signature parameter w, compute the signature $\theta_{id} = (H(id) \times w_{id}^{d})^{sk} \in G$.
- *Proof Generation:* Given a collection of block identifiers $id_1...id_n \in Z$, 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 identifiers $i...n$, the public key $pk \in G$, the signature $\theta \in G$, the pairing e, the generator g, and the signature parameter w, by applying pairing verify that: $e(\beta, g) = e(\prod_{i=1}^{n} H(id_i) \times w^{\alpha}, pk)$. If the verification is positive, integrity is assured.

In short, the SW scheme combines pairings and BLS signatures with standard cryptographic hash functions like SHA-1 in a single integrity verification protocol. It allows users to select any arbitrary sample of data stored on a remote cloud location and verify that the integrity of that sample is kept by just downloading proofs of small and constant size that compress both the data and signatures. This is an innovation when compared with integrity verification schemes like RSA digital signatures because the SW scheme guarantees that the amount of data downloaded from the cloud remains constant

even as the size of the selected sample grows.

There are significant challenges in adopting the SW scheme for verifying the integrity of clouds that S-AUDIT mitigates. Namely: the SW scheme requires coordination between the user and the cloud for selecting the several underlying parameters used in the scheme (addressed in Section II-D by the S-AUDIT protocol); and the SW scheme increases the cloud data storage requirements (addressed in Section II-E by pairing parameter selection in S-AUDIT).

D. S-Audit Protocol

In order to preserve the integrity of the data stored on the cloud, the entities involved – cloud, user and auditor – need to follow the S-AUDIT protocol, described herein. The protocol is divided into four tasks: setup (Section II-D1), store data (Section II-D2), request and verify integrity proof (Section II-D3), and generate integrity proof (Section II-D4).

1) Setup: Before storing any data in the cloud, the user and auditor must perform the following protocol steps:

- The user and the auditor exchange data. The auditor provides two files[2] to the user for setting-up pairing-based cryptography: the '.param' file with all the secure public initialization parameters needed for configuring cyclic groups G, G_T and the pairing for mapping $G \times G \rightarrow G_T$; and the '.g' file with generator g of the cyclic group G. The user provides configuration information to the auditor about the time when each audit should be performed (e.g., daily, weekly), and other settings.
- 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 *key* and *random number generators* (explained in Section III).
- The user shares the public key and w with auditor and stores w on the cloud.
- The user configures the cloud for listening to requests from the auditor requests and for responding to them, with the execution of the *proof generator service* (detailed in Section III).

After these steps are performed users can now store their data in the cloud, as explained next.

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 *signature generator* (further explained in Section III) automates these tasks and produces a signature equivalent to the SW Block Signature step (described in Section II-C4). To do so, the client provides as *input* for the signature generator: the data and its identifier (e.g., the file content of the 'data.txt' file is used as the data and the identifier is the file name), 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. Data can now be verified.

[2]*Data structures* would be a more rigorous term but *files* is more concrete and is how data is stored in our implementation.

3) Request and Verify Integrity Proof: The auditor is responsible for integrity verification. To do so, whenever the auditor wants to obtain integrity proofs of a file stored on the cloud, it must perform the following steps:

- Select a file composed of x data elements (vector $[0, ..., x-1]$).
- Generate a random challenge (number belonging to Z_p) for each of the x data elements chosen, using the *random number generator*.
- Issue the integrity proof request to the cloud specifying the 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 *proof verifier* (further explained in Section III). The auditor provides the public key pk and the signature parameter w, alongside with the identifiers and challenges used on the integrity request, and obtains the integrity verification result. This step corresponds to the Proof Verification step of the SW scheme (described in Section II-C4).

4) Generate Integrity Proof: Whenever the cloud receives an integrity proof request for a given file, it performs the following steps:

1) Fetch all the data and signatures of the file from the storage cloud corresponding to the identifiers 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: aggregation of signatures provided (β); and aggregation of data provided (α), by using the *proof generator* (explained in III-E). The generator receives data, setup parameters ('.g' and '.param'), signatures, challenges, pairing cryptography parameters and the random initialization parameter related to the file; and produces the α and β. This step corresponds to the proof generation step of the SW scheme.
4) Respond to requester with the integrity proof (α and β).

E. SW Signature Size Reduction

The size of the block signatures produced by S-AUDIT is equal to the size of the multiplicative cyclic group G stipulated by the auditor (e.g., if G is equal to 128 bits then the block signatures are also 128 bits). Also, the size of the groups G are determined by the *elliptic curve* selected for its initialization and are always larger than the integers used for its generation Z_p. For example, when a type A elliptic curve [27] is used for the generation of multiplicative cyclic groups, with the recommended sizes where G and G_T are 128 bytes and Z_p is 20 bytes, the signatures produced are *6.4 times bigger* than the original file, raising the storage cost in that proportion. This large overhead would make the technique too costly for practical use. S-AUDIT introduces two techniques in the original SW scheme to address this issue.

The first is the selection of the pairing curve that produces the shortest multiplicative cyclic groups, which is the pairing-

friendly elliptic curves of prime order [9] (also named type F curves and described in [12]), as recommended by both BLS and SW authors in [35] and [27]. This optimization allows the creation of multiplicative cyclic groups G that are *two times the size* of the original data Z_p, producing signatures with twice the size of data.

The second technique is to incorporate a signature compression scheme in S-AUDIT using the *point compression* described in [27]. The improvement 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 value to select from the possibilities. 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 allows signatures to have half of the expected size of applying the signature step of SW scheme. In the best case, where type F elliptic curves [9] are used, signatures are of the *same size* of the original data.

With these two optimizations, S-AUDIT is able to produce signatures that are of the same size as the original data, which is the lowest possible value using the known homomorphic signature schemes.

III. IMPLEMENTATION OF S-AUDIT

The S-AUDIT service, represented in Figure 1, is composed of several components, each one implementing a task of the S-AUDIT protocol. This separation in components simplifies the integration with cloud-backed applications, commercial clouds, and auditors. S-AUDIT was developed in Java, so each component is essentially a Java class. The pairing-based cryptographic mechanisms were implemented using the Java Pairing-Based Cryptography Library (JPBC) [17], which implements multi-linear maps and the operations that manipulate them.

Auditors use the *Pairing Generator* component to generate the setup parameters for pairing-based cryptography. Users utilize the *Key Generator* component to generate their asymmetric secret/public key pair and signature parameter (w); and use the *Signature Generator* component to sign their data. Both these entities use the *Random Generator* component to generate random numbers belonging to any field of their choosing (Z_p, G or G_T).

Clouds run the *Proof Generator* component to generate integrity proofs. Auditors use the *Proof Verifier* component to verify the proofs obtained from the cloud.

Each of the mentioned components will be explained in detail in the following subsections.

A. Pairing Generator

This component allows auditors to construct setup parameters ('.param' and '.g') for initializing pairing-based cryptography, according to their security specification.Auditors provide as input the type of pairing curve[3] to be used for pairing generation, 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.

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

C. Signature Generator

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

D. 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.

E. Proof Generator

The *Proof Generator* component is the only one that is executed in the cloud (cf. Figure 1). It allows clouds to generate integrity proofs with the files they have stored whenever an auditor requests them. To do so, the algorithm first initializes pairing with the setup parameters, then calculates α and β based on the data's blocks present in the file. To simplify the deployment and to reduce the cost of running the *Proof Generator* in a cloud, we leverage recent services that

[3] See Section 4 of [27] for more information about the pairing curves and their selection.

implement the FaaS model [21], [26]. The alternative would be to have a virtual machine for this purpose in a cloud compute service (e.g., Amazon EC2), but it would be costly to run it permanently in the cloud, or to store an image there to run it when necessary. The FaaS model allows the execution of a code component (a function) in a cloud upon a certain event, in our case, the reception of a request through a REST API. In this model, the users pay only for the time and resources used when the function is executed, not when it is idle. Therefore, it is possible to have the *Proof Generator* component always ready to run in the cloud without costs when it is not running.

F. Proof Verifier

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

IV. Extending SCFS with S-Audit

S-Audit was designed to allow easy integration with existing cloud-backed applications. As a proof of concept, the S-Audit components described in Section III were integrated with the *Shared Cloud-backed File System* (SCFS) [11].

SCFS is a distributed file system that stores files in a cloud or a set of clouds (a *cloud of clouds*). Users *mount* the SCFS file system on a folder of their device, and the SCFS client-side component synchronizes files with the cloud storage services. SCFS supports data sharing among several users, automatically propagating users' modifications between them. In the integration we have to consider the three S-Audit entities:

- The *user* code is integrated with the client-side code of SCFS;
- The *auditor* code is a stand-alone Java program;
- The *cloud* code runs in a FaaS service such as Amazon Lambda [1].

Next we focus mostly on the first entity, as it is the one truly integrated with SCFS code.

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]. In this integration, SCFS was configured with DepSky. Data integrity is protected in SCFS and DepSky using RSA digital signatures [34]. 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, with both monetary costs and delays.

The user components of S-Audit were integrated in DepSky's component responsible for uploading data into the cloud. The logic for communicating with different commercial clouds is implemented in subcomponents called *cloud drivers*. Since the integration of S-Audit should not break any of the features currently supported by DepSky or SCFS, integrating both systems required code changes to DepSky, in a contained way. The followed approach was the addition of a new type of cloud driver: the *auditable cloud driver*. With these newly introduced cloud drivers, besides accessing and uploading data to the cloud, data is signed using S-Audit's signature generator and the signature is also stored on the cloud. As seen in Figure 2, for integrating these new drivers, DepSky was modified in two packages: core and drivers. Code was added to the core package of DepSky, in the DepSky initialization function (in *LocalDepSkySClient.java*) and to the DepSky driver constructor function (in *DriversFactory.java*).

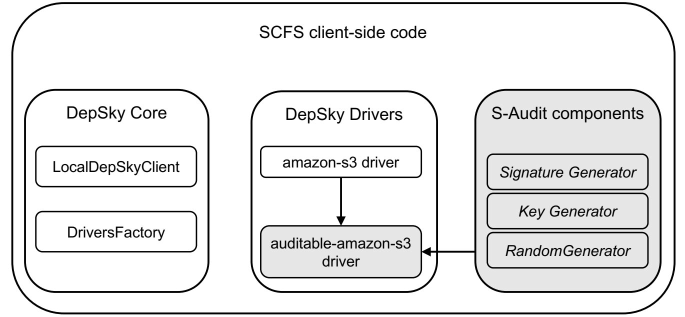

Figure 2. SCFS/DepSky client-side code. Components modified and added for S-Audit integration are shown in grey.

For using S-Audit, SCFS has to be configured with these *auditable cloud drivers*, which implement the logic of our system. 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* was used. Users can choose which drivers to use, by modifying the configuration file with the name of the desired drivers. The DepSky initialization function automatically reads the user's secret key, the setup parameters ('.param' and '.g') and the signature parameters ('.w') provided by the auditor; and uses the initialization function of the DepSky driver for initializing the driver with that information. Regarding the driver package, the auditable drivers extend the non-auditable drivers. Whenever data is uploaded to the commercial cloud using the auditable driver, data is signed by using S-Audit's sign data component and then both signature and data are uploaded to the commercial cloud by invoking the the non-auditable driver code.

V. Experimental Evaluation

The cloud used during the implementation was AWS [4]: *S3* [2] was used for storage and *Lambda* for executing the Proof Verifier.

With the experiments we wanted to answer the questions:
(1) What is the gain in terms of bandwidth consumption in using S-Audit instead of RSA (Section V-B)?
(2) What are the monetary costs of using S-Audit in comparison to the use of digital signatures (Section V-C)?
(3) What is the performance overhead observed by the user when writing files (Section V-D)?
(4) How long does it take to verify the integrity of a file (Section V-E)?

A. Experimental Settings

On the experiments, the *user* and the *auditor* components were executed on a computer with Intel Core i7-4500U CPU 1.80-2.40 GHz processor, 8 GB of RAM, and running Windows 10. The user and auditor were located in Portugal. The cloud was AWS located in Ireland. Lambda was setup to execute code with 128 MB of memory (the cheapest configuration) and 512MB of memory (when the file size required it).

The evaluation was performed using one file for each of the following sizes: 100 KB, 500 KB, 1 MB, and 10 MB. Each experiment was repeated 30 times.

Some of the experiments involved the execution of two schemes to serve as baseline: the original SW integrity verification scheme (implemented using a version of S-AUDIT that strictly follows the scheme without employing the point compression technique); and the RSA digital signature scheme. Both S-AUDIT and the original SW scheme were parameterized with type F pairing curves, where G had 40 bytes, G_T 80 bytes, and Z_p 20 Bytes, with SHA-1 as hash algorithm, and asymmetric keys used with a 20 byte secret key and a 80 byte public key. For RSA, 1024 bit keys and SHA-1 were used.

B. Bandwidth

An important goal of our work is to avoid the time and cost of downloading all the data from the cloud in order to verify if it was modified. In this section we measure the bandwidth consumed downloading data, measured in number of bytes.

Figure 3 shows the results for S-AUDIT and compares them with the original SW scheme (that provides identical results) and with the use of RSA digital signatures (that retrieves all data to be verified). The results show that as the storage size grows, S-AUDIT 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), the bandwidth consumption is always equal to the sum of these group's sizes and that it is always low (the cost for reading 60 bytes is negligible). On the contrary, for RSA, the use of bandwidth grows linearly with the size of the files.

C. Monetary Costs

To assess the monetary costs of verifying the integrity of the cloud storage two cases were considered: the additional storage taken with digital signatures; and the costs of generating proofs on the cloud.

a) Storage Costs: Using S-AUDIT for verifying data integrity on the cloud storage requires users to store on the cloud the data's digital signatures, which implies additional monetary storage costs.

Figure 4 compares the storage size – of the file and signature(s) – as data size grows, when using S-AUDIT, SW, and RSA signatures. As seen in the figure, storing SW signatures increases the storage size by 200%, but S-AUDIT manages to reduce this overhead to 100% with the signature reduction scheme of Section II-E. This reduction has great positive impact on storage monetary costs, but still requires twice the storage than the ideal case where signature sizes are negligible (the RSA case). The actual monetary costs tend to be proportional to the amount of data stored in most commercial clouds. For example, in Amazon S3 Ireland this cost is \$0.023 per GB per month, for the first 50 TB / month, using standard storage (half of that for infrequent access and \$0.004 in the Glacier service) [3].

b) Proof Generation Costs: In order to evaluate the monetary costs associated with integrity proof generation, S-AUDIT's proof generator was executed in the cloud.

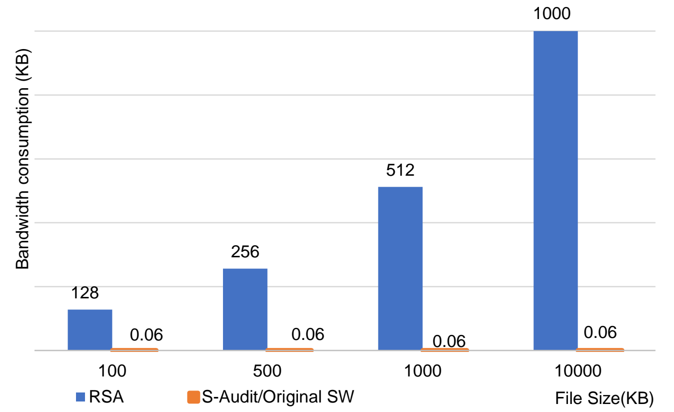

Figure 3. Bandwidth consumption comparison between requesting file integrity proofs using S-AUDIT, SW, and RSA digital signatures.

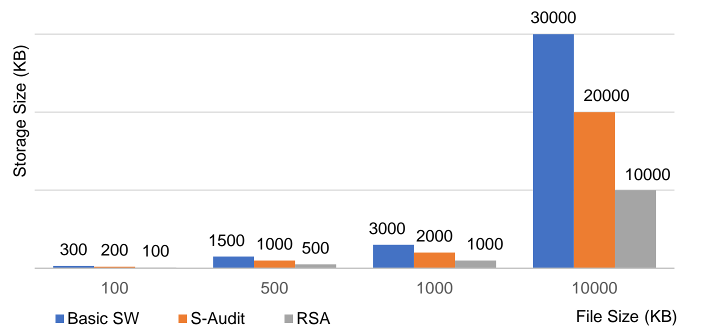

Figure 4. Storage size for storing data and signature when using S-AUDIT, SW, and RSA.

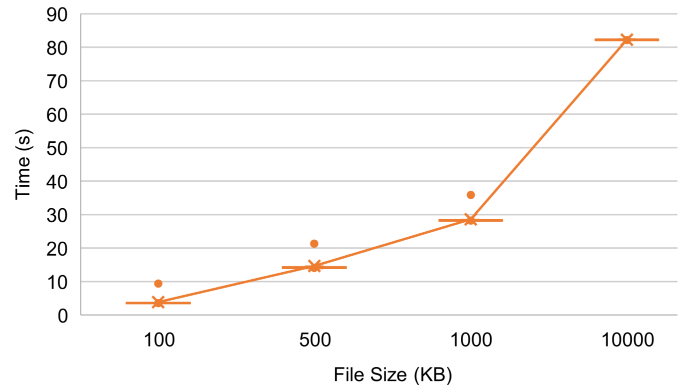

Figure 5. Time for the cloud to generate an integrity proof using S-AUDIT.

Table I
PRICES FOR GENERATING PROOFS AND READING DATA (BASED ON AMAZON IRELAND PRICES, STANDARD STORAGE).

	S-AUDIT		RSA signatures	
File Size (KB)	Average Execution Time (s)	Execution cost[1] (microUSD)	Read Costs[2] (microUSD)	Savings (%)
128	3.82	8.11	11.52	29.58
256	7.44	15.60	23.04	32.29
512	14.67	30.57	46.08	33.64
1000	28.55	59.48	90.00	33.90
10000	82.29	686.38	943.74	27.27

[1]Considering 0.208 microUSD for 0.1s of computation (Lambda w/128 MB RAM) for files up to 1 MB, and 0.834 microUSD for 0.1s of computation (Lambda w/512 MB RAM) for bigger files.
[2]Considering 0.09 microUSD for 1GB read from the cloud storage (S3)

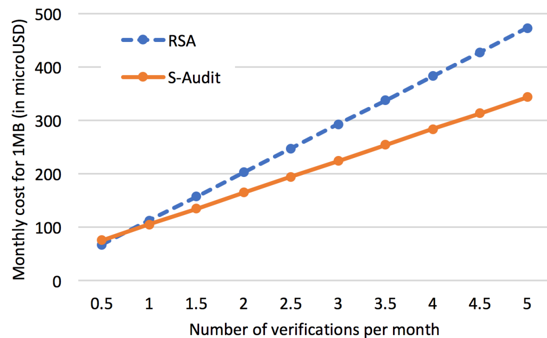

Figure 6. Monthly costs in microUSD for 1 MB of data depending on the number of verifications per month.

Figure 5 presents the time for generating integrity proofs in S-AUDIT. The figure makes clear that the time grows linearly with the storage size. Furthermore, as seen in Table I, when comparing price paid for generating a proof (execution time) with the cost of downloading the files entirely and performing the integrity verification on the auditors' device (as required by RSA), generating integrity proofs was cheaper than reading the data from the cloud and allowed a monetary saving of 30%, on average.

c) Cost Tradeoffs: The previous results show that using RSA signatures is costly (and slow) in terms of downloading data, but S-AUDIT doubles the cloud storage monetary costs. Therefore there is a trade-off that we now quantify. Figure 6 shows how the cost of verifying 1 MB varies with the number of verifications done per month. For S-AUDIT the cost has two main components: the cost of storage (again the values for standard storage in S3 Ireland), and the cost for generating proofs in the cloud (again in Lambda). For RSA signatures, the cost has also two main components: the cost for storing and downloading the data (also S3 Ireland).

The main conclusion from the graph is that the best option in terms of cost depends on how often the data is verified. If the data is verified once per month, the cost of using S-AUDIT is 7.1% lower than the cost of using RSA signatures. This cost becomes much lower – 34.9% – if the verification is done approximately every week (4 times per month).

Notice that the cost of S-AUDIT would be lower if cheaper storage services were used, e.g., Amazon S3 with infrequent access or Amazon Glacier [3].

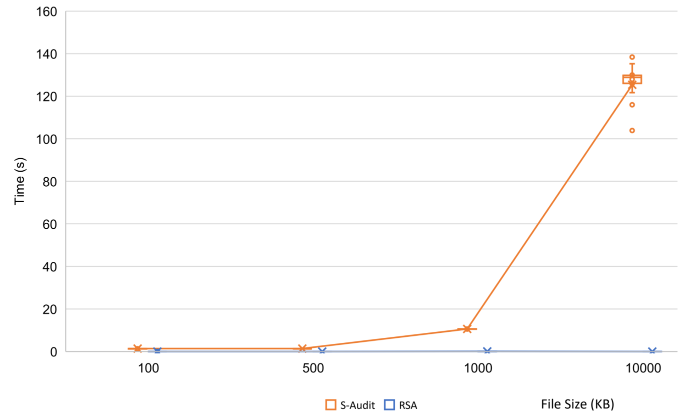

Figure 7. Time for signing data using S-AUDIT and RSA.

D. User: Client-Side Overhead

S-AUDIT should not have a great impact on the performance seen by the *user*, i.e., on the client-side software. However, such impact may exist when files are written in the cloud, as signatures have to be computed in order to allow verifying integrity later. In order to assess if S-AUDIT meets this criteria, two aspects were evaluated: the time taken to sign data using S-AUDIT and RSA; and the overhead on SCFS.

a) Signature Generation: The signature generation of S-AUDIT was evaluated in terms of the time required to compute a signature in the user's device. The results obtained are presented in Figure 7. The time required for signing data using S-AUDIT increases linearly and is much slower when compared to RSA digital signatures, which take around two milliseconds. This almost constant time is due to the fact that data signed using RSA digital signatures is first hashed (faster phase), then encrypted using RSA (slower but constant time). In S-AUDIT, the SW scheme, and all the other publicly verifiable schemes, all of the 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, which takes longer as data grows. This makes S-AUDIT slower than the usual signature generation mechanisms. This was expected due to the computational cost of the homomorphic signature generator. Nevertheless, this overhead can be masked by the application, as shown next.

b) SCFS with S-Audit's Signature Generator: In order to evaluate the performance impact of S-AUDIT integrated on cloud-backed applications, we evaluated the performance of writing a file in SCFS, both with and without S-AUDIT. The results differ much depending on the mode in which SCFS is executed: non-blocking or blocking.

The *non-blocking mode* is the one that is recommended [11]. In this mode, when a client closes a file by calling *close*, the file is written to the local disk and the call returns. Then, in the background, DepSky pushes the file to the clouds. In this mode, both versions of SCFS, with and without S-AUDIT, had the same performance from the client's perspective.

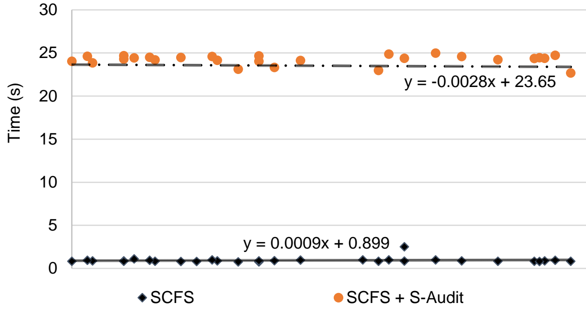

Figure 8. One dimensional scatter plot representing the time required for SCFS to upload a 1 MB file (dot) to the cloud with and without S-AUDIT (with linear interpolation).

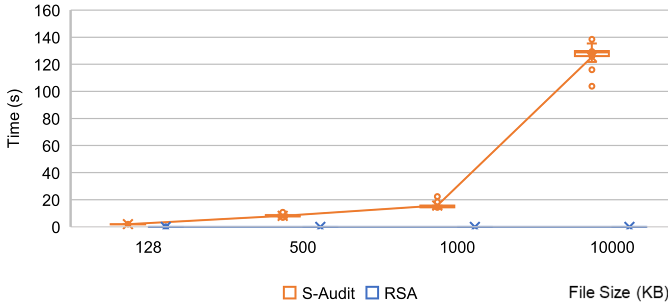

Figure 9. Time for verifying S-AUDIT file integrity proofs and RSA signatures.

In *blocking mode*, SCFS waits for the file to be stored in the cloud for the *close* call to return. This mode is slow even in the original SCFS, so it is not recommended [11]. Nevertheless, we did several experiments of uploading a 1 MB file to the cloud using SCFS with and without S-AUDIT. The results obtained are presented in Figure 8 and show that the integration with S-AUDIT increases time significantly. These results are similar to those presented in Figure 7 (for the 1 MB file).

E. Auditor: Proof Verification Time

The last set of experiments assessed the time required for the *auditor* component do to its job, i.e., to verify the proofs obtained from the cloud. This was compared with checking RSA signatures, excluding the time to download the files.

As seen in Figure 9, the time necessary for verifying a signature in S-AUDIT increases linearly and is slow compared with RSA digital signatures. This is due to the fact that for verifying a proof using S-AUDIT 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 the data grows. For example, verifying 1MB of data involves multiplying the identifiers of 25600 blocks, which increases time to the values obtained in the experiments.

F. Evaluation Outcomes

The main outcomes of the experimental evaluation are the following:

- In terms of monetary costs, using S-AUDIT is better or worse than RSA signatures depending on the periodicity of the verifications; in a typical environment (AWS) S-AUDIT is cheaper when data is verified monthly (7.1% cheaper) and is considerably cheaper when data is verified weekly (34.9%);
- When doing integrity verification, S-AUDIT requires downloading much less data than RSA signatures (only 60 bytes), but verifying the proofs takes time also;
- S-AUDIT requires computing signatures when data is uploaded so it has an impact on the performance of that operation, but this impact can be completely masked by the application, as seen with SCFS in non-blocking mode.

VI. RELATED WORK

S-AUDIT fits in the cloud integrity verification research area. More precisely, it is closely related with mechanisms that allows users to perform integrity verification on cloud storage. The available mechanisms can be divided in two categories: integrity verification with non-homomorphic proofs and with homomorphic proofs.

In systems that use *non-homomorphic proofs* [33], [37], [10], [19], [39], [22], [11], [31], [36], [15], performing integrity verification requires files to be downloaded, with the associated bandwidth consumption and monetary costs. S-AUDIT aims to reduce these penalties.

In the case of systems that use *homomorphic proofs*, such as [8], [42], [40], [13], [41], [18], [35], integrity verification requires less bandwidth consumption than systems that use non-homomorphic proofs. However all the mechanisms of this class studied were not yet practical and only had theoretical demonstrations of their feasibility. S-AUDIT is the first system to show that it is possible to integrate with conventional cloud-backed applications or commercial clouds and also it is the first to benchmark extensively a homomorphic scheme in a way that makes it possible to clearly understand the advantages and disadvantages of using homomorphic schemes.

S-AUDIT allows applications to use a cost-efficient homomorphic integrity verification scheme, based on the SW scheme and expanded with pairing-friendly elliptic curves [9] and point compression [27] optimizations.

Research on cloud storage security aims to provide a large set of properties and services [14], from which remote integrity verification is only a part. DepSky, SCFS, CYRUS, MetaSync, UniDrive, and RACS provide integrity and availability by storing data in several clouds [10], [11], [15], [23], [38], [43]. Some of these services provide also confidentiality [10], [11], [15], [43]. Depot provides fork-join-causal consistency even if most storage servers/clouds and clients are faulty, but no confidentiality [28]. CloudProof allows users to prove cloud violations of integrity, write-serializability, and freshness [33].

VII. CONCLUSION

This paper presents S-AUDIT, a cloud-storage verification service designed to be easily integrated with current cloud storage solutions, including cloud-backed applications and commercial storage clouds. S-AUDIT automates all the tasks

involved in storage integrity verification, including signature generation and verification.

S-AUDIT is targeted at providing integrity proofs without retrieving the data. S-AUDIT 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, e.g., when data is stored unmodified for reasonably long periods of time.

S-AUDIT was integrated with the SCFS cloud-backed file system and AWS. As shown by our evaluation: S-AUDIT is 34.5% cheaper than signatures when data is verified weekly in a typical setting: requires downloading only 60 bytes although proof verification takes seconds; and requires uploading much more data, but this overhead can be entirely masked by the application, as observed with SCFS.

Acknowledgements. 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). We warmly thank Alysson Bessani for feedback on a previous version of this work.

REFERENCES

[1] Amazon Lambda. https://aws.amazon.com/lambda/.
[2] Amazon S3. https://aws.amazon.com/s3/.
[3] Amazon S3 Pricing. https://aws.amazon.com/s3/pricing/.
[4] Amazon Web Services. https://aws.amazon.com/.
[5] Ransomware a threat to cloud services, too. http://krebsonsecurity.com/2016/01/ransomware-a-threat-to-cloud-services-too/.
[6] Top Cyber Threats for 2016; integrity attack on financial sector likely. http://www.carriermanagement.com/news/2015/11/30/148345.htm.
[7] W. Ashford. 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.
[8] 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*, pages 319–333, 2009.
[9] P. S. Barreto and M. Naehrig. Pairing-friendly elliptic curves of prime order. In *International Workshop on Selected Areas in Cryptography*, pages 319–331, 2005.
[10] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa. DepSky: Dependable and secure storage in a cloud-of-clouds. In *Proceedings of the 6th ACM European Conference on Computer Systems*, EuroSys'11, pages 31–46, 2011.
[11] 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*, pages 169–180, 2014.
[12] D. Boneh and X. Boyen. Short signatures without random oracles. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 56–73, 2004.
[13] 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*, pages 187–198, 2009.
[14] D. Burihabwa, R. Pontes, P. Felber, F. Maia, H. Mercier, R. Oliveira, J. Paulo, and V. Schiavoni. On the cost of safe storage for public clouds: an experimental evaluation. In *Proceedings of the 35th IEEE Symposium on Reliable Distributed Systems*, pages 157–166, 2016.
[15] 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 10th ACM European Conference on Computer Systems*, EuroSys'15, 2015.
[16] Cloud Security Alliance. The notorious nine: Cloud computing top threats in 2013, Feb. 2013.
[17] A. De Caro and V. Iovino. jPBC: Java pairing based cryptography. In *Proceedings of the 16th IEEE Symposium on Computers and Communications*, pages 850–855, 2011.
[18] N. T. F. de Carvalho. A practical validation of homomorphic message authentication schemes. Master's thesis, University of Minho, 2014.
[19] D. Dobre, P. Viotti, and M. Vukolić. Hybris: Robust hybrid cloud storage. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14, 2014.
[20] D. Eastlake 3rd and P. Jones. US secure hash algorithm 1 (SHA1). Technical report, 2001.
[21] G. C. Fox, V. Ishakian, V. Muthusamy, and A. Slominski. Status of serverless computing and Function-as-a-Service (FaaS) in industry and research. *arXiv preprint arXiv:1708.08028*, 2017.
[22] D. Grolimund, L. Meisser, S. Schmid, and R. Wattenhofer. Cryptree: A folder tree structure for cryptographic file systems. In *Proceedings of the IEEE 25th Symposium on Reliable Distributed Systems*, pages 189–198, 2006.
[23] S. Han, H. Shen, T. Kim, A. Krishnamurthy, T. E. Anderson, and D. Wetherall. Metasync: File synchronization across multiple untrusted storage services. In *USENIX Annual Technical Conference*, pages 83–95, 2015.
[24] M. Hanley, T. Dean, W. Schroeder, M. Houy, R. F. Trzeciak, and J. Montelibano. An analysis of technical observations in insider theft of intellectual property cases. Technical Note CMU/SEI-2011-TN-006, Carnegie Mellon Software Engineering Institute, Feb. 2011.
[25] Q. Hassan. Demystifying cloud computing. *The Journal of Defense Software Engineering*, pages 16–21, 2011.
[26] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Serverless computation with OpenLambda. *Elastic*, 60:80, 2016.
[27] B. Lynn. *On the implementation of pairing-based cryptosystems*. PhD thesis, Stanford University, 2007.
[28] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. *ACM Transactions on Computer Systems*, 29(4), 2011.
[29] R. C. Merkle. Secrecy, authentication, and public key systems. 1979.
[30] msft-mmpc. Wannacrypt ransomware worm targets out-of-date systems. https://blogs.technet.microsoft.com/mmpc/2017/05/12/wannacrypt-ransomware-worm-targets-out-of-date-systems/, May 2017.
[31] R. Padilha and F. Pedone. Augustus: scalable and robust storage for cloud applications. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys'13, pages 99–112, 2013.
[32] R. Pontes, D. Burihabwa, F. Maia, J. Paulo, V. Schiavoni, P. Felber, H. Mercier, and R. Oliveira. SafeFS: A modular architecture for secure user-space file systems (one FUSE to rule them all). In *Proceedings of the 10th ACM International Systems and Storage Conference*, 2017.
[33] 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*, pages 355–368, 2011.
[34] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.
[35] H. Shacham and B. Waters. Compact proofs of retrievability. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 90–107, 2008.
[36] 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*, pages 229–238, 2012.
[37] 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*, 2008.
[38] H. Tang, F. Liu, G. Shen, Y. Jin, and C. Guo. Unidrive: Synergize multiple consumer cloud storage services. In *Proceedings of the 16th Annual Middleware Conference*, pages 137–148, 2015.
[39] M. Vrable, S. Savage, and G. M. Voelker. Bluesky: a cloud-backed file system for the enterprise. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, 2012.
[40] 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*, 8(1):92–106, 2015.
[41] C. Wang, K. Ren, W. Lou, and J. Li. Toward publicly auditable secure cloud data storage services. *IEEE Network*, 24(4):19–24, 2010.
[42] C. Wang, Q. Wang, K. Ren, and W. Lou. Privacy-preserving public auditing for data storage security in cloud computing. In *Proceedings of the IEEE INFOCOM*, pages 1–9, 2010.
[43] R. Zhao, C. Yue, B. Tak, and C. Tang. SafeSky: A secure cloud storage middleware for end-user applications. In *Proceedings of the IEEE 34th Symposium on Reliable Distributed Systems*, pages 21–30, 2015.