RockFS: Cloud-backed File System Resilience to Client-Side Attacks

David R. Matos[1] Miguel L. Pardal[1] Georg Carle[2] Miguel Correia[1]
[1]INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal
[2]Technical University of Munich, Department of Informatics, Germany
{david.r.matos,miguel.pardal,miguel.p.correia}@tecnico.ulisboa.pt,carle@net.in.tum.de

ABSTRACT

Cloud-backed file systems provide on-demand, high-availability, scalable storage. Their security may be improved with techniques such as erasure codes and secret sharing to fragment files and encryption keys in several clouds. Attacking the server-side of such systems involves penetrating one or more clouds, which can be extremely difficult.

Despite all these benefits, a weak side remains: the client-side. The client devices store user credentials that, if stolen or compromised, may lead to confidentiality, integrity, and availability violations. In this paper we propose RockFS, a cloud-backed file system framework that aims to make the client-side of such systems resilient to attacks. RockFS protects data in the client device and allows undoing unintended file modifications.

CCS CONCEPTS

• **Security and privacy → Distributed systems security**; • **Computer systems organization** → *Maintainability and maintenance*;

KEYWORDS

File Systems, Cloud Computing, Privacy, Intrusion Recovery

ACM Reference Format:
David R. Matos[1] Miguel L. Pardal[1] Georg Carle[2] Miguel Correia[1]. 2018. RockFS: Cloud-backed File System Resilience to Client-Side Attacks. In *19th International Middleware Conference (Middleware '18), December 10–14, 2018, Rennes, France.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3274808.3274817

1 INTRODUCTION

Cloud storage is increasingly important to individuals and organizations. Currently it is being used as part of mission-critical systems like medical record databases and government software that have strong requirements in terms of security. However, cloud storage raises important concerns regarding security. First, data is stored and maintained by a third party, so it may be vulnerable to theft and corruption [45, 66]. Second, if the cloud provider has an outage (which is not that uncommon [41]), there is no way to access the data.

Several solutions have been proposed to enhance *cloud storage security* [2, 9, 10, 20, 34, 54, 68, 70]. An interesting approach is to encrypt and store data in several clouds, forming a *cloud-of-clouds*, allowing to improve data availability, while ensuring data integrity and confidentiality, even if some of the clouds fail [2, 9, 10, 70]. Some of these systems, and others that are not so concerned with security and dependability, may be designated *cloud-backed file systems*, in the sense that they provide client-side software that exports a POSIX file system interface [24]. This approach allows users to use seemingly local files that transparently store data in the clouds [10, 57, 68].

Despite all these benefits, a weakness remains: *the client device*. These devices store user credentials that if stolen or compromised may lead to violations of confidentiality (files are disclosed), integrity (files are modified), and availability (files are made inaccessible).

Both file systems that store data in a single cloud and in cloud-of-clouds assume that the client-side – the user's device or proxy used to access the cloud – is secure [2, 9, 10, 57, 68, 70]. This is a strong assumption considering that in many occasions users leave their personal devices unattended, use weak passwords, have outdated systems with known vulnerabilities, or fall prey to social engineering [18, 65]. Once an attacker gains access to a user account, he can read, modify or delete any file that the user has access to. Even if the file system stores several versions of the files, the attacker may delete the old versions or revert a file to the version prior to the attack, discarding a significant amount of work. Moreover, there is a reasonably recent surge of *ransomware* that encrypts device data, potentially making the user unable to use his credentials and access the files stored in the cloud(s) [35, 69].

To overcome attacks against client devices, we propose the *RockFS framework*,[1] a set of components that allow improving client-side security of cloud-backed file systems. RockFS provides two sets of security mechanisms to be integrated with the client-side of a file system: a *recovery service* capable of undoing unintended file operations without losing valid file operations that occurred after the attack; and *device data security mechanisms* to safely store encryption keys reducing the probability of having the credentials compromised by attackers and to protect cached data.

In this paper we present the RockFS framework, a prototype compatible with most cloud storage systems[2] and an experimental evaluation.

To the best of our knowledge the client-side protections provided by RockFS are not provided by any other cloud-backed storage systems, not even individually. Recovery services have been designed

[1]ROCKFS – RecOverable Cloud-bacKed File-System
[2]Available for download at: https://github.com/inesc-id/SafeCloudFS.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Middleware '18, December 10–14, 2018, Rennes, France
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5702-9/18/12…$15.00
https://doi.org/10.1145/3274808.3274817

for local file systems [26, 31, 36], cloud databases [46], and web applications in the cloud [47, 52], but not for cloud-backed storage. In relation to device data security mechanisms, backup and encryption schemes are widely adopted for protecting credentials in endpoints, but not secret sharing that both protects and allows recovering credentials [12, 61, 62], and not for cloud-backed storage. This paper presents new protections and demonstrates their usefulness by leveraging an existing cloud-backed file system, not by proposing a new one. Specifically, the RockFS prototype is based on the Shared Cloud-backed File System (SCFS) [10], which itself is based on the DepSky Byzantine fault-tolerant cloud-of-clouds storage service [9].

The remainder of the paper is structured as follows: Section 2 describes RockFS, presenting the system model, threat model and system architecture. Section 3 presents the file recovery mechanisms. Section 4 describes the mechanisms used to secure the user's credentials and cached files. Section 5 describes our implementation of RockFS. Section 6 presents the result of our experiments. Section 7 describes related work in literature. Finally, Section 8 concludes the paper.

2 ROCKFS

RockFS improves cloud-backed file systems in the two following ways: it provides logs that allow an administrator to analyze the usage of the file system and to recover user files stored in the clouds by undoing unintended operations; it secures user data that is stored on the client-side, namely, access credentials to the cloud service providers and cached files by encrypting this data with a key that is distributed using secret sharing.

2.1 Threats

We assume attackers cannot access the cloud back-ends, however they may gain total access to the user's device, therefore RockFS focuses on three security threats not addressed by current *cloud-backed file systems*, as they target the client device:

- **Threat T1:** *Adversary illegally modifies files in the cloud*, e.g. with a ransomware that overwrites the data files in the client device, which eventually are synchronized to the cloud.
- **Threat T2:** *Adversary prevents a user from accessing the cloud*, by corrupting or deleting the access credentials stored in the client.
- **Threat T3:** *Adversary accesses the locally cached files*, when he gets access to the user's device, as the client cache is not encrypted.

2.2 System Model

We assume that the client device may be compromised by an attacker that may get access to any data that is on the disk. On the contrary, we assume that adversaries have no access to data in memory. In relation to the services running in clouds, we make the same assumptions as cloud-backed file systems, e.g., that the cloud is not compromised [57] or that no more than a threshold of clouds is compromised [10].

We assume that communication over the Internet is done using secure channels that ensure communication authentication, confidentiality, and integrity (e.g., using SSL/TLS). We make the standard assumptions about cryptography, e.g. that encryption cannot be broken in practice, that hash functions are collision-resistant, and that signatures cannot be forged.

In RockFS there are two main actors:

- *Users:* contract cloud services to store files and access them remotely using a personal computer or a mobile device.
- *Administrators:* operate RockFS, i.e., maintain the coordination service, monitor the usage of RockFS, and trigger recoveries.

The authentication of both users (PU_U, PR_U) and administrators (PU_A, PR_A) relies on *asymmetric keys* (PUblic, PRivate). We assume that only the owner of a key pair knows the private key, whereas public keys are known and accessible to every user.

The access control to the cloud storage relies on *access tokens* (t_l, t_u). Access tokens are temporary credentials generated by cloud storage providers that authorize users and applications to use specific actions of their services without sharing passwords. In RockFS we use an access token for the log (t_l) that only authorizes to append data, and another one (t_u) that authorizes users to read and modify files but not the logs. These access tokens will typically include the identifier of the user, an expiration date, an identifier of RockFS and a signature to ensure integrity. We assume that these tokens are generated by the cloud storage providers in a non-predictable manner and that it is not possible for an attacker to re-use revoked tokens. RockFS requires several tokens and keys to ensure integrity and confidentiality of data. The list is presented in Table 1.

2.3 System Architecture

The system architecture is presented in Figure 1. RockFS can be deployed using a single cloud or using a cloud-of-clouds. On the top left of each figure there is a cloud storage service, where the data is stored (e.g., Amazon S3, Google Drive, etc.). On the top right of each figure there is a coordination service, which is responsible for storing logs and other metadata. It is possible to deploy each replica of the coordination service in a distinct cloud (cloud-of-clouds architecture), improving the availability of the service. RockFS may be deployed with coordination services like ZooKeeper [33] or DepSpace [8]. The client agent, on the bottom, is a middleware component that runs on the client-side and communicates with the storage cloud and the coordination service, transparently for the file system user.

Both RockFS' users and administrators access the cloud storage and coordination service, but with different privileges: users only access their files; administrators access the files and also the logs used to store recovery data. The users interact with the file system by invoking the POSIX operations (e.g. open, read, write and close). The administrator accesses logs to analyze usage and recover from unintended actions.

The interaction between the client, the cloud storage services and the coordination service is mediated by the RockFS agent and the cloud-backed file system (CBFS) agent, that are responsible for intercepting file system operations with the aid of the FUSE library[3]. The RockFS agent performs several encryption and encoding tasks.

[3]Filesystem in Userspace, https://github.com/libfuse/libfuse

Entity	Notation	Description	Generated by	Stored in
User	PU_U PR_U	Public key of user U Private key of user U	User U during setup	Shared between the user's device, coordination service and external storage
	A_i B_i	ith log entry secret key ith log entry secret key	RockFS agent	Coordination service
	t_l t_u	Access token to create log entries in the cloud storage Access token to manage files	Cloud providers	Shared between the user's device, coordination service and external storage
	SC_i CC_i	Cloud storage service credentials Coordination service credentials	User and admin. during setup Administrator during setup	Shared between the user and the administrator's device, coordination service and external storage
	S_U	Session key of user U for local cache	User U	Shared between the user's device, coordination service and external storage
Administrator	PU_A PR_A	Public key of administrator A Private key of administrator A	Administrator A during setup	Shared between the administrator's device, coordination service and external storage
Clouds	PU_{SC_i} PR_{SC_i}	Cloud storage service public key Cloud storage service private key	Administrator during setup	Each cloud storage service
	PU_{CC_i} PR_{CC_i}	Coordination service public key Coordination service private key	Administrator during setup	Each cloud hosting a coordination service replica

Table 1: Keys used in RockFS with description, who generated them, and where they are stored.

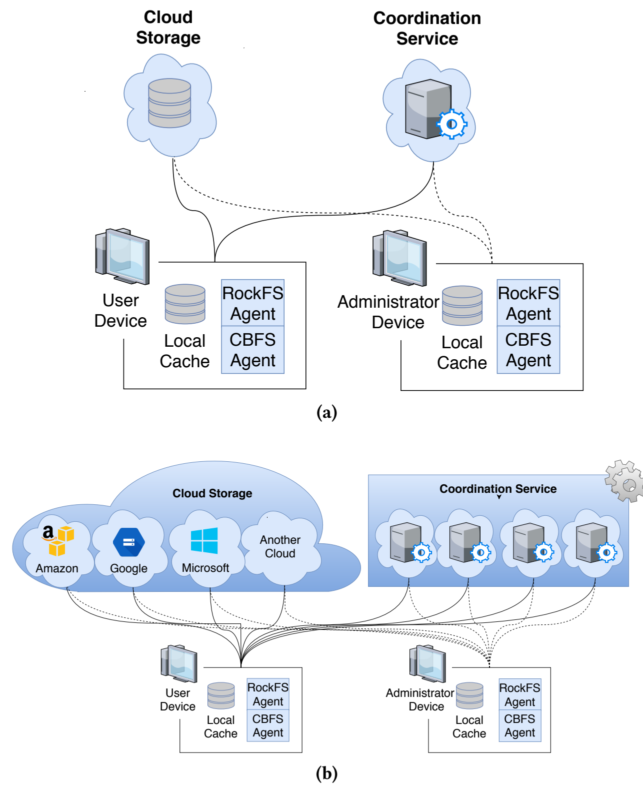

Figure 1: RockFS architecture (a) with a single cloud storage provider and coordination service (b) with four different cloud storage providers and four replicas of the coordination service.

2.4 Client Architecture

The user device contains the RockFS agent, the local cache and the keystore, including the private key of the user (PR_U). In order to access the cloud(s), the RockFS agent needs the access credentials of each cloud storage provider (SC_i) and cloud used by the coordination service (CC_i). These credentials are stored in a file called *keystore*. The keystore is kept in persistent storage. RockFS splits the keystore in shares and stores them in separate places so that, even if an attacker accesses one of the shares, he can neither read nor delete the keystore. This is achieved using secret sharing [12, 62].

When the user performs a login, the RockFS agent needs to combine some of the shares, e.g., 2 out of 3 shares, to obtain the credentials, which it keeps only in volatile memory. By default, it uses the share kept in the coordination service and the share in the client device. The share in the client device is protected by the user account access control mechanisms.

For recovery there is one or more additional shares stored in an external memory, like a USB flash drive, or a smart card, which must be kept at a secure location. The use of secret sharing is further detailed in Section 4.1.

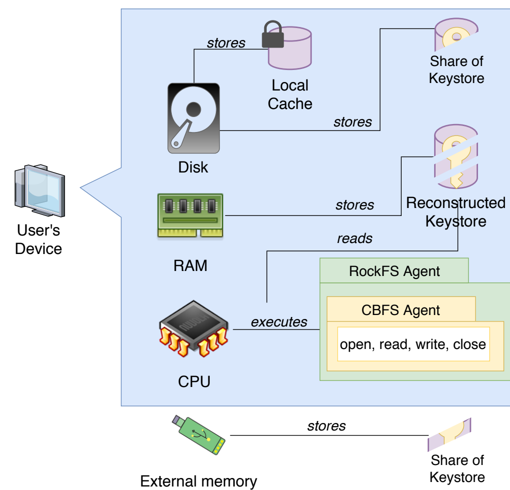

Figure 2: User device with client-side components and external memory with one of the shares of the keystore.

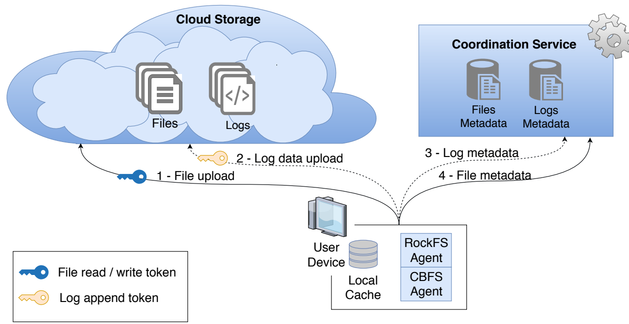

Figure 3: RockFS logging of operations in the file system.

The user device architecture is represented in Figure 2. The Disk stores the encrypted local cache and one of the shares of the keystore. The RAM stores the keystore that is reconstructed with secret sharing. This ensures that even if the user's device is stolen, the adversary cannot obtain the access credentials from the disk. The CPU executes the RockFS agent.

2.5 Logging Architecture

Figure 3 represents the architecture of the subsystem used to log file system operations in order to support recovery. The operations are logged in a coordination service (top-right of the figure). This coordination service will contain the metadata corresponding to the log entry, i.e., a timestamp, file identifier and user identifier. The log entry data is stored in the cloud storage.

The figure also represents what happens when the user closes a file after writing into it: the file and log data are uploaded to the cloud storage services. Then, a log entry is created in the coordination service. Finally, in some cloud-backed file systems (e.g., SCFS), the file metadata will be updated in the coordination service. This step must be performed at the end to ensure that all data – the file, log data and metadata – is stable before notifying the user that the operation was completed successfully. This step is done at the end rather than at the beginning to prevent other users from seeing data files that were not yet uploaded to the cloud, or may not be uploaded if the client crashes during the operation.

Each log entry has two parts: the data part and the metadata part. These parts are kept in different systems for two reasons: first, the data part requires more storage space, so it is more effectively stored in a cloud storage service; second, the metadata needs to be queried by the administrator, something that is supported by coordination services, but is harder with cloud storage services (typically the data would have to be downloaded before the query being processed).

3 STORAGE RECOVERY

This section details how RockFS handles threat T1: an adversary illegally modifying files as if he was their owner (see Section 2.1).

When an adversary hijacks a user's session, e.g., by stealing his computer, he can tamper with the user's files. Every action executed in the user's device that modifies a file is eventually synchronized to the storage cloud(s), so the effects of an attack that occurred on the client-side are propagated to the cloud-side. When this happens, the cloud(s) will have tampered files. To cope with this vulnerability we propose a recovery approach that enables the administrator to identify malicious operations and undo them.

3.1 Recovery and Logging Threats

The log is the resource that allows the administrator to recover from attacks. As such, it becomes a target for an attacker that wants to prevent the administrator from recovering the file system. After modifying user files, an attacker may also try to modify the log entries corresponding to his actions in order to make the effects of the attack permanent. More exactly, in RockFS an adversary may try the following attacks:

- **A1:** attacker illegally modifies user files in the cloud storage services;
- **A2:** attacker illegally modifies log entries;
- **A3:** attacker illegally modifies both the user files and corresponding log entries;

If an attacker succeeds in **A1** or **A3** it is still possible to revert what he has done by recovering the affected files, as explained further in Section 3.3. To cope with **A2** we propose a log integrity check mechanism explained in Section 3.2.

3.2 Logging

To enable the administrator to undo the faulty operations performed by adversaries, all modifications done to a file have to be registered in a log. The RockFS agent is the component responsible for recording these operations. The log is stored alongside with the files in the cloud or cloud-of-clouds, with the protections provided by the cloud-backed file system (more precisely by its cloud-access subsystem, such as the one available in DepSky [9]).

The metadata that is logged in the coordination service for an operation, lm_{fu}, contains a timestamp, the user identifier, the file name, the version identifier and the operation (create, update or delete).

The log data of the user's files, ld_{fu}, is stored encrypted in the cloud or cloud-of-clouds and consists of the differences between the new version of the file and the previous one or, if the differences are larger than the file, the file itself.[4]

Logging is triggered when a file is closed. Specifically, when the POSIX operation `close` is invoked on a file f_u, the following operations are executed by the RockFS agent:

- Compute ld_{fu}, a file with the differences between f_u and its previous version that is stored; if ld_{fu} is larger than f_u, then consider ld_{fu} to be f_u (a flag indicates which is the case);
- ld_{fu} is encrypted using a random secret key S_{fu};
- If a cloud-of-clouds is being used, the `encode` function is used to split ld_{fu} in n shares, where n is the number of clouds used;

[3]From time to time it is necessary to create a snapshot of the file system in order to clean old log entries (for instance, by moving them to cold storage, e.g., Amazon Glacier [4]) and save some storage space.

[4]More sophisticated policies might be devised but we simplify by considering that the best is to store whatever is the smallest between the two: differences or whole file. Nevertheless, we expect the differences file to be typically much smaller than the whole file, except for small files.

- ld_{fu} is sent to the cloud storage service or services (one share per service in the latter case) following the same scheme used in the cloud-backed file system;
- The key S_{fu} is stored in the same way the cloud-backed file system does, e.g., in the coordination service or by splitting it in shares using secret sharing and storing one per cloud (as in SCFS [10]);
- Log metadata lm_{fu} is stored in the coordination service;
- Finally, both the file and log data are uploaded to the cloud storage back-end after its corresponding metadata was updated in the coordination service.

Both the log entries and files are uploaded to the cloud or cloud-of-clouds by the agent. If an adversary gains access to the user's device, he could try to erase or modify the log entries in order to prevent the administrator from recovering the file. To prevent this from happening, RockFS uses two distinct access credentials for the cloud storage: the user's files token, t_u, and the logging token, t_l. These tokens are generated by the cloud storage providers and they restrict access in a way that t_l cannot be used to modify the user files and t_u cannot be used to tamper with the log entries. Most cloud storage services provide such control access mechanisms [5, 17, 38]. Figure 3 shows how RockFS agent uses these credentials.

By using the logging token, t_l, we ensure that even if an adversary gains access to the user's device he cannot tamper with the existing log entries. He can only create new log entries in order to try to interfere with the recovery process. To ensure the recovery is done correctly, RockFS has to provide *forward-secure stream integrity* [44], a property that ensures that logs, which are considered to be streams of sequential entries, can be verified for integrity. The scheme we leverage in RockFS is the forward-secure stream integrity scheme (FssAgg) presented in [44] that provides the following guarantees:

- *Forward Security:* the secret signing key used in the scheme is updated by a one-way function, making it computationally unfeasible for an attacker to recover previous keys from a current, stolen, key;
- *Stream Security:* the sequential aggregation process preserves the order of the log entries and provides stream-security;
- *Integrity:* illegal insertions, modifications and deletion of log entries are detected.

This scheme provides the following functions:

- `FssAgg.Kg`: generates pairs of asymmetric keys;
- `FssAgg.Asig`: generates an aggregate signature for a log entry. Takes as input the previous aggregate signature, the log entry and the current secret key.
- `FssAgg.Upd`: updates the secret key used to authenticate each log entry;
- `FssAgg.Aver`: verification algorithm that takes as input a log entry and the corresponding signature.

These protections are relevant for the security goals of RockFS and the system implements them. The forward-secure stream integrity scheme [44] allows the integrity of the entire log to be checked. This requires that, during setup, two random symmetric keys, A_1 and B_1, are securely exchanged between two parties. We assume the administrator himself is responsible for providing these keys to the cloud storage servers (stored in shares using secret sharing) and the coordination service. For every new log entry L_i that is created, RockFS agent will generate a hash value based on the previous ones:

$$\mathcal{U}_i = H(\mathcal{U}_{i-1} | mac_{A_i}(L_i))$$

where $\mathcal{U}_i$ is a FssAgg MAC of the ith entry of the log. It is calculated by the hash of the concatenation of two values: the previous $\mathcal{U}_{i-1}$, and the MAC of the current log entry with the current symmetric key A_i. A_i evolves in each new log entry by using an hash function ($A_i = H(A_{i-1})$).

The new hash values are then uploaded to the cloud storage servers and the coordination service. A more detailed explanation and proof of work can be found in [43].

3.3 Recovery

Before recovering a file, the RockFS agent needs to verify the integrity of the log entries using the forward-secure stream [43] verification algorithm. This is done by taking the symmetric key A_1 and and computing the corresponding hash values (A'_i) for each log entry (L_i). Then RockFS compares the computed hash values of each entry ($A'_i = A_i$). If they match then the log entries are valid and the recovery process can initiate. Otherwise, the invalid log entries are removed.

Recovering a file, as opposed to rolling it back to a previous version, allows users to erase illegal modifications while keeping the valid ones that occurred after the attack. To do so it is necessary to reconstruct the file by executing each valid action. This technique has been proposed in a different context and named as *selective re-execution* [36]. It can be executed from the first operation that created the file or by obtaining the latest valid version of a file. From this point, RockFS will apply every valid operation to it until the present time. In more detail:

1. The administrator fetches the first version of f_u and its corresponding log entries, LD_{fu} (LD_{fu} is an array with several ld_{fu});
2. The function `decode` is used to join the shares of both the f_u and every ld_{fu}
3. The administrator selects the malicious modifications from LD_{fu} and discards them;
4. For each ld_{fu} in LD_{fu}, the function *patch* is invoked to apply ld_{fu} to f_u;
5. File f_u is shared by the `encode` version and sent to the cloud storage services.

Steps 2 and 4 are only necessary if the file system is backed by a cloud-of-clouds offering. If the file system is backed by a single cloud, then there is no need to share the log entries and the file.

In relation to step 3, notice that we assume that there is some way of knowing which modifications have been compromised. This is a problem of intrusion detection to which there are many solutions [39, 58], so we simply take that for granted.

It is worth mentioning that the recovery operations will be logged as well. This means that, if a file gets corrupted a second time after it was already recovered, then the RockFS will execute the same operations on the file that were executed in the first recovery. This happens because we want to ensure integrity of the log entries and by not allowing even the administrator to erase or modify log entries we are also avoiding adversaries to do so.

4 SECURING THE USER DEVICE

In this section we propose solutions for the threats T2 (adversary corrupting access credentials) and T3 (adversary accessing files cached in the user device) presented in Section 2.1. The user device stores two types of sensitive information: the credentials for accessing the cloud(s), and the local cache with the most recently accessed files. By corrupting the access credentials an attacker may violate the availability of the service. If the local cache is accessed by an attacker both the integrity and confidentiality of the user files are at risk.

4.1 Securing User Credentials

The credentials are protected with *secret sharing* [62]. The idea is that, even if the user's machine is compromised and the keystore is corrupted, the user may still recover his credentials and resume the use of the cloud-backed file system.

In a secret sharing scheme there is a special entity, the dealer, that splits a secret among n parties. Each party gets a share of the secret, meaning that if an attacker succeeds in obtaining one of the shares he cannot reconstruct the secret. With such a scheme, $k < n$ shares of the secret are required to reconstruct it, therefore an attacker would need to be get k shares to recover it. More specifically, in this work we use a secret sharing scheme called *publicly verifiable secret sharing scheme* (PVSS) that allows verifying if the shares are corrupted [61].

In RockFS, the PVSS scheme is used to break the keystore with the credentials in shares. The client acts as a dealer, sharing, combining and verifying the shares. PVSS provides the following functions:

- `share`, invoked by the client to obtain the shares of the keystore;
- `combine`, invoked by the client to reconstruct the keystore using k shares;
- `verifyD`, invoked by the servers to verify if the received share is legitimate;
- `verifyS`, invoked by the client to verify if the shares sent by the servers are legitimate.

The shares of the keystore are generated during *setup*, in the following way:

- The RockFS agent asks the user for how many shares of the keystore (n) should be generated, and how many are needed to reconstruct the keystore (k);
- The RockFS agent invokes the `share` function of the PVSS with parameters n and k;
- One of the shares is sent to the coordination service while the remaining shares are given to the user so he can choose where to store them,
- The shares given to the user are erased from disk and RAM, and the setup is complete.

By default, the user keeps one secret share on his device, for making it simple to log in RockFS (assuming a scenario with $n = 3$ and $k = 2$) and another share in the coordination service (so the $k = 2$ that are needed are available online). However, the PVSS allows the user to choose a different way to split the secret (different parameters n and k) and different devices where to store them, for added security. The user's smartphone can be used for this purpose, or other more elaborate password stores [63].

A user recovers the keystore in two situations: every time he logs in, and when his device was compromised and he needs to recover the keystore using the share kept in external memory. For both cases the recovery works as follows:

- The user provides $k - 1$ shares (the remaining one is located in the coordination service);
- RockFS agent fetches the remaining share from the coordination service;
- RockFS agent executes the PVSS function `verifyS` to verify if all the shares are legitimate;
- RockFS agent executes the PVSS function `combine` and loads the keystore into memory. In any case the keystore is on the disk.

This scheme protects the keystore from deletion or ransomware encryption, unlike the alternative of encrypting the keystore with an user-provided password (which would be converted to an encryption key). It also protects the keystore from being read at the disk, unlike the alternative of backing it up. In relation to the alternative of both encrypting and backing up the keystore, the solution still provides the benefit of not requiring the user to introduce a password.

4.2 Securing the Device Local Cache

In RockFS we propose mechanisms to verify the integrity and ensure confidentiality of the local cache on the device, which stores the files recently accessed by the user. The existence of this cache in a cloud-backed file system is not mandatory, but in practice it is essential for the performance to be acceptable (otherwise, e.g., the client would block for long periods waiting for reads or writes in the storage clouds).

Figure 4 represents the extension of the POSIX file operations `open` and `close` that have to be modified for protecting the cache.

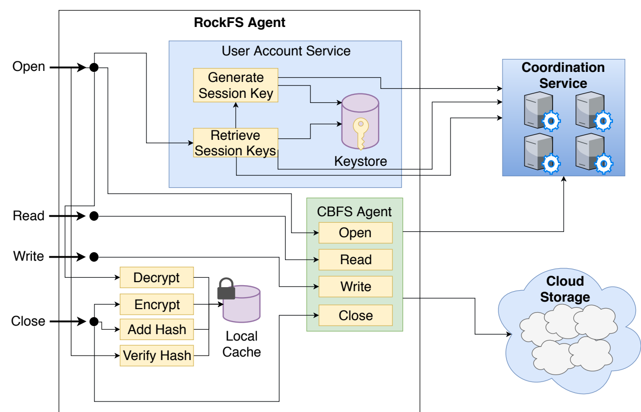

Figure 4: RockFS operations, the `read` and `write` operations from the cloud-backed file system remain unaltered. The operations `open` and `close` decrypt and encrypt the local cached files to ensure data confidentiality.

4.2.1 Confidentiality. Local cache files are encrypted using a session key, S_U, to ensure confidentiality. This key has a short validity (configurable by the administrator), and is associated with an entry in the coordination service, preventing an attacker from reusing old session keys. On open, the RockFS agent:

- Checks if S_U is still valid, if not the local cached file is discarded and a new S_U is generated and exchanged with the coordination service;
- Decrypts the opened file and loads it.

The file `close` operation:

- Creates a log entry for the file update, and uploads the log entry to the cloud;
- Uploads the file to the cloud;
- Fetches S_U from the keystore;
- Removes the log entry from the disk or memory;
- Encrypts the file that was closed.

Besides encrypting the locally cached file, the `close` operation also logs the modifications, as detailed in Section 3.

4.2.2 Integrity. The integrity of the local cache is ensured using cryptographic hash functions, which are encrypted with S_U together with the rest of the file. When a user invokes the open operation on a file f_u, the RockFS agent:

- Fetches h_{fu}, the hash value correspondent to f_u;
- Computes a hash value h'_{fu} of f_u;
- Compares both hash values, h_{fu} and h'_{fu}. If they match the file is opened, otherwise the file is discarded and a valid version of the file is fetched from the cloud.

When the user invokes the `close` operation on a file f_u:

- A new hash value, h_{fu}, is calculated and stored in the local cache alongside f_u;
- The file f_u and h_{fu} are encrypted.

5 ROCKFS IMPLEMENTATION

In our implementation of RockFS we used a modified version of SCFS as cloud-backed file system [10]. Most modifications were made to implement the log of operations and extend the compatibility of SCFS with more cloud storage back-ends by using JClouds [1]. SCFS works with a single cloud or with a cloud-of-clouds, but we considered only the latter configuration, as it provides stronger security and dependability assurances for the server-side (i.e., the cloud). The underlying protocol, DepSky, deals with the heterogeneity of the different clouds and is compatible with most commercial cloud storage. As coordination service we used DepSpace, also supported by SCFS. We start by presenting these components before presenting the implementation of RockFS itself.

5.1 DepSky Cloud-of-Clouds Storage

DepSky is a dependable and secure cloud-of-clouds storage system that addresses four limitations of existing cloud storage services: loss of availability, loss and corruption of data, loss of privacy, and vendor lock-in [9]. It works by using a collection of cloud storage services that are managed remotely by a client library. The user files are stored encrypted in public clouds.

It also uses erasure-codes [16, 27–29] to reduce the required storage for the user files. This way, instead of having n copies of a file distributed in n cloud providers, only a fraction of n (usually half) is needed. DepSky provides a consistency level similar to the weakest consistency level given by the cloud providers. For each user file in the clouds, there is the file itself and a signed metadata file that stores the version number of the file and verification data.

DepSky supports two protocols: A and CA. *A*, which stands for availability, and *CA*, which stands for confidentiality plus availability. The *A* protocol replicates the file in each cloud storage. This improves availability at the cost of the extra storage (n times the size of the file). The *CA* protocol encrypts each file with a symmetric key before sending it to the cloud. Then the key is split in n parts using secret sharing, requiring at least $k + 1$ parts to reconstruct the key. The file itself is also split in n shares using erasure-codes, which will require $k + 1$ shares to reconstruct it. The *CA* protocol requires less storage thanks to the erasure-codes ($\frac{n-k}{n}$ as opposed to n).

RockFS uses the CA protocol for two reasons: first, it requires less storage space, and, second, it encrypts the files and secures the encryption key with secret sharing, which fits our system model.

5.2 SCFS Cloud-of-Clouds File System

SCFS [10] is a distributed file system supported by DepSky, when configured for cloud-of-clouds (the case we consider). It provides consistency-on-close semantics [30]. Its architecture is more complex than DepSky because it requires a coordination service to keep the logical structure of the file system and to coordinate concurrent accesses from multiple writers.

SCFS provides confidentiality of data stored in cloud storage providers through the mechanisms of DepSky: the user's files are encrypted and the secret keys are protected using secret sharing. These encryption mechanisms ensure confidentiality of data in the cloud. However, the data stored in the user's personal device is not encrypted since it is assumed to be trusted. Metadata of SCFS is stored in the coordination service for three reasons: the coordination service offers consistent storage; it uses replication to cope with faults; and because it can be used to cope with faults and to implement synchronization operations, such as file locking.

5.3 DepSpace Dependable Coordination Service

DepSpace [8] is a Byzantine fault-tolerant distributed coordination service. It was designed to run on a cluster of commodity machines, making it possible to deploy in any infrastructure cloud service. Unlike other coordination services like ZooKeeper [33], DepSpace tolerates arbitrary faults. It uses Byzantine fault-tolerant protocols [6] to ensure correctness in the event of up to f arbitrary faults (also designated Byzantine faults), requiring $3f + 1$ replicas, possibly running in different cloud services, for that effect. DepSpace provides a *tuple space* which can be used to implement locks, timed tuples, partial barriers and secret storage.

One of the limitations of the original DepSpace was the fact that it maintained its state in volatile memory, making it impossible to recover if more than f replicas failed by crashing. An enhanced version was proposed in [11] and is being used by RockFS. It applies check-pointing and logging mechanisms that use external storage

to keep recovery data. These mechanisms also allow the migration of a DepSpace cluster to a different cloud provider, thereby avoiding vendor lock-in.

5.4 RockFS Implementation

RockFS was implemented in Java SE 8. We chose this programming language since all the components of SCFS were written in Java and every cloud storage service used in this work also provides Java APIs. Also, since Java is a multi-platform language, it is possible to deploy in distinct execution environments, making it more robust against operating system specific vulnerabilities.

The size of the keys as well as the algorithms used to generate them were chosen taking into account their security in the long term. According to ENISA, the recommended values are: SHA-512 for hash values, 512 bit elliptic curves for public keys and AES-256 for symmetric keys [22].

The keystore (depicted in Figure 4) is a text file with the access credentials for cloud services, the access credentials for the coordination service, the sessions keys (S_U) and the private key of the user (PR_U). This file is never stored on persistent storage (disk). It is split in, at least, three shares: with one of the shares in the coordination service, another in the user's device and an extra one in a external storage (USB memory or smartcard), especially for recovery. To recover the keystore it is not enough to reveal the secrets since this file is also encrypted (with AES-256), requiring a user password to decrypt it. Once it is decrypted it is loaded into memory (RAM).

The RockFS agent uses a variation of the UNIX *diff* command [32] called JBDiff [5]. This command is used to calculate the log entries of each file operation. Each log entry is in fact the difference between the old version and the new version of a file. Recovery is done by reconstructing a file, i.e., executing the corresponding *patch* command sequence.

6 EXPERIMENTAL EVALUATION

The performance of the protections for the credentials (threat T2) and cache (T3) was found to be below tens of milliseconds, so their cost can be considered negligible in the overall cloud-of-clouds solution. Therefore, our evaluation focused on T1, i.e., on the costs of the *recovery* scheme presented in Section 3. With the experiments performed we wanted to answer the following three questions: (a) What is the cost, in terms of performance, of having the RockFS agent log every file operation? (b) What is the cost, in terms of storage, of saving every file modification? (c) How long does it take to recover files depending of the number of modifications they suffered? The answer to this last question is relevant to assess how effective our solution is against *ransomware* attacks.

In the experiments we wanted to simulate a realistic scenario in which RockFS would be deployed in at least two different clouds. This way it would be possible to ensure that metadata and data are stored in different locations (logically and geographically) ensuring that even if one cloud gets compromised, it is not possible to read the users' files. We set up RockFS using two different clouds: Amazon S3 [5] for the cloud storage services; and Google Compute Engine [38], for the coordination service. Regarding the Amazon S3 storage services, we set up 4 storage buckets in Ireland. In the Google Compute Engine we created 4 instances (for the 4 replicas of DepSpace) with 1 vCPU and 3.75GB of memory for each machine. All 4 replicas were located in the Belgium data center.

For the client machine, we created an instance, again with 1 vCPU and 3.75GB of memory, in the London data center. This additional instance serves as a client of RockFS and will execute the RockFS agent code. We chose to execute the client on the cloud for two reasons: first, it provides a stable Internet connection; and second, this machine is as simple as possible, meaning that no extra software running in the background interferes with the execution of the experiments.

6.1 Latency Overhead of Log Operations

To calculate the latency of logging operations we created a workload that consisted in creating files and then updating these files with an extra 30% content. We vary the size of the files between 1 and 50MB, according to statistics from [3]. Given that SCFS offers two different approaches for file synchronization (blocking and non-blocking), we performed the experiments with both configurations. Each test was repeated 10 times and the values presented in the graphs correspond to the *average* values.

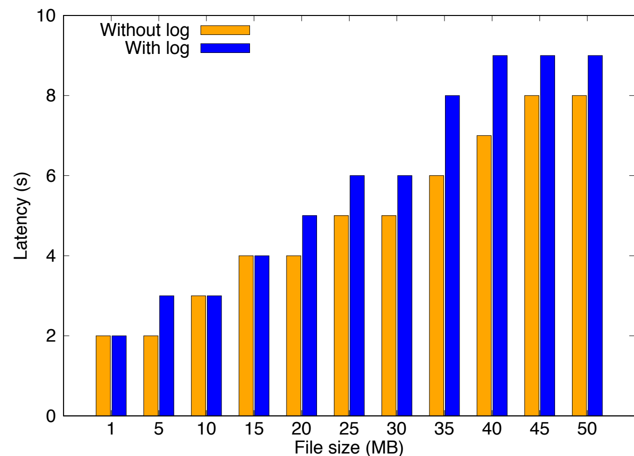

Figure 5: Latency of using RockFS with and without the log.

Figure 5 presents the average latency of logging file operations in RockFS. The latency is the time it takes since the user finishes updating the file, i.e. invoking the POSIX `close` function, and the time the coordination service finishes recording the file operation. The latency values without log were collected executing the workload on SCFS. The latency values with logging are, on average, 20% higher than the ones without logging. This overhead is expected, since it takes time for the RockFS agent to compute the log entry (differences between versions) and to upload these differences to the cloud. Several optimizations were performed to reach this value. The two most important ones were (1) both the logging and the file operation are processed in parallel by the coordination service, and (2) the file and log entry uploads are also done in parallel. This 20% overhead can be reduced by improving the network bandwidth (for instance, by using the same data center for the storage services

[5] Java Binary Diff https://github.com/jdesbonnet/jbdiff

and the coordination services) and by improving the computing hardware of the client (to process the differences quicker).

In a different experiment we used microbenchmarks from FileBench [23] to execute different workloads. Table 2 shows the results of three different workloads: *sequential write*, that appends data to the end of a file; *random write*, that modifies a random section of a file; and *create files*, that creates new files without modifying them afterwards. Each workload was tested in both SCFS and RockFS in two different modes: non-blocking (NB) which synchronizes files to the cloud in the background allowing the user to proceed his work, and blocking (B) which blocks the application until the modified file has been completely uploaded to the clouds. We tested these three workloads with both SCFS and RockFS to understand how much does it cost, in terms of performance, to log file operations. Unlike the original experiments in SCFS [10] that executed several workloads of read operations, in this case we are only interested in testing operations that modify files, since these are the only ones being logged by RockFS. The overhead of using RockFS according to the results shown in Table 2 is non-negligible but can be considered acceptable, especially in the non-blocking mode which is the recommended configuration.

6.2 Storage Overhead of Log Operations

We did experiments to find out how much more storage RockFS needs to keep all the logs of several file operations. To do so we executed 1, 10 and 100 file updates in files with sizes again varying from 1MB to 50MB.

Experiments show that the storage overhead of the log is significant. Every time a user appends 10MB to a file, an extra 10MB are added to the log. In this system model we are using the *CA* protocol of DepSky (described in Section 5.1) which uses erasure-codes to reduce the required storage to 2 times (as opposed to 4 times in the *A* protocol) i.e. a log entry of 10MB will occupy 20MB overall in the clouds.

It is also worth noting that a file that is modified several times will create a log history greater than a file that is created once and subsequently is not modified. In these experiments we wanted to evaluate how much storage does it take to store the log entries of files that are rarely updated (1 version), moderately updated (10 times) and intensively updated (100 times). Each modification to the file was in 30% of the original size of the file (e.g. a file with 10MB was updated with more 3MB every time).

Figure 6 presents the storage overhead of logging different files with different versions. The storage growth is linear. The red bars, labeled as "without log entries", represents the total storage occupied the file itself in the clouds. It is worth noticing that each file occupies twice its size in the clouds storage due to the use of several clouds and erasure codes. The increment in the total size for the blue bars (with 1 log entry) is only marginally superior to the red bars because it represents the size of the file plus the log entries, which only contains the delta of the modifications. For the 10 versions we can see that the required storage for the log is greater than the file itself. This motivates the adoption of a future *snapshot* mechanism to create backup versions of the files in order to discard log entries. The log size values for the 100 versions file are not in the chart. The sizes vary from 60MB (for the 1MB file) to around 3GB (for the 50MB file). It is worth noticing that existing cloud-backed file systems already employ a multi version approach to prevent data loss. In such systems the amount of storage required to save every version of the file would be greater than the RockFS logs, since it just stores the delta encoding of each version.

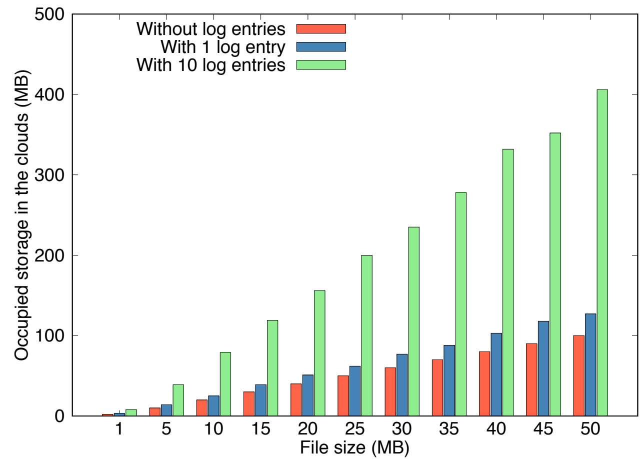

Figure 6: Required storage for the files and logs in the cloud storage services.

It is possible to calculate and approximation of the total required storage s for a file in version n with a percentage of modifications by computing the following equation:

$$s_n = 2 \times (s_{n-1} + \quad \times s_{n-1}) \tag{1}$$

The storage overhead is considerable but it is not detrimental of the adoption of RockFS for two reasons. First, cloud users and providers seem not to be too eager to minimize the storage space used as, for example, Microsoft OneDrive by default keeps in storage 500 versions of each file. Second, most cloud providers offer cheaper cold storage to which older versions can be moved, e.g., Amazon Glacier. Compression techniques could also be used to reduce the overall storage required by RockFS. This is a problem we intend to explore as future work.

6.3 Mean Time to Recover Files

The MTTR (Mean Time to Recover) a specific file varies according to the number of versions of that file. A file that was only modified once before being attacked can be recovered by executing a *patch* operation (i.e. applying the differences in the log to the original version), while a file that was modified 100 times requires 100 *patch* operations to be executed.

Here we are recovering the file system from a *ransomware* attack. In this type of attack, every file in the file system is corrupted (encrypted). First we did experiments to measure the recovery of a single file. Then we did experiments with the recovery of the set of all files.

To evaluate the MTTR of different files we took the files and log entries from the previous experiments and recovered each file 10 times (to reach an average value). Again, a file with 1 version

Micro-benchmarks	# Operations	File size	SCFS		RockFS		Overhead	
			NB	B	NB	B	NB	B
write	1	4MB	1.63	1.71	1.90	2.12	17%	24%
create	200	16KB	197.60	236.76	219.00	298.20	11%	26%

Table 2: Latency (in seconds) of Filebench micro-benchmarks for SCFS and RockFS.

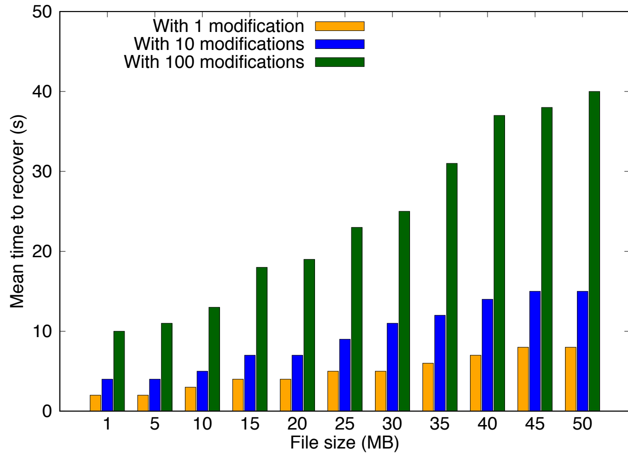

Figure 7: Mean time to recover files with 1, 10 and 100 versions.

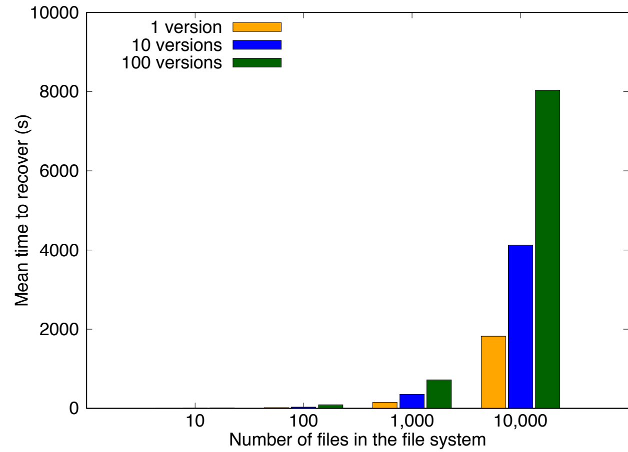

Figure 8: Mean time to recover a file system compromised by a ransomware attack varying in the number of files and versions of each file.

means that was created and modified just once while a file with 100 versions means that after its creation, it was modified 100 times.

Figure 7 presents the MTTR of several files with different versions. Although the MTTR grows linearly with the file size, in the 100 versions files the growth is steeper. The time varies from around 2 seconds (for 1 version file with 1 MB) to around 40 seconds (for the 100 versions file with 50MB). We optimize the recovery process by downloading every log entry of the file to be recovered at once, instead of downloading each entry at a time.

To evaluate how RockFS recovers a complete file system from such attack, we created 16KB files (from 10 to 10,000) and modified them several times (from 1 to 100 versions with each modification being a 4KB write in the file). Then RockFS recovered every file of the file system. The results are presented in Figure 8. The mean time to recover grows exponentially with the number of files in the file system. In the worst case of the experiments (10,000 files with 100 versions each) it took around 2 hours and 5 minutes to recover every file in the system. Considering the hindering effects of a ransomware attack, the full recovery time is acceptable. This is still a considerable time but once RockFS starts the recovery process, files become gradually available for the user. Because of this property of the system, the recovery can start with the most urgently needed files, as specified by the user. And assuming that a regular user does not access every file in the system at once, this allows him to resume working while RockFS continues the remaining recovery process in the background.

6.4 Network overhead

There is a network overhead every time a user modifies a file and when an administrator recovers a file. It is possible to calculate how much network traffic will be spend by computing a model for the upload and a model for the download. Moreover, this model allows the users of RockFS to calculate how much more will it cost, in terms of monetary costs, to use RockFS as opposed to a single cloud storage without recovery capabilities.

In the following models we assume a network overhead of that is depends on the size t of the file, the delta encoding percentage of the file – , the number of clouds – n and the number of versions of the file – .

6.4.1 Network overhead while logging. Every time a user updates a file it is automatically uploaded to the cloud (or clouds). RockFS will also append the delta encoding of the modifications to be stored as the log entry of that file operation. Given that a cloud-of-clouds configuration requires data to be sent to multiple clouds such cost must be multiplied by the number of clouds. In the end we can divide the total cost by 2 given that we use erasure codes to fragment data. The following equation can be used to calculate the total cost of using RockFS to log every file operation.

$$= \frac{((t + \quad \times t) \times n)}{2} \tag{2}$$

For example, by computing this model with a fixed = 30% and $n = 4$, uploading a 1MB file will result in 3MB traffic while a 50MB file will result in 130MB.

In terms of monetary costs, most cloud storage services do not charge for uploading data. Network-wise logging every file operation will not have an impact in the service bill.

6.4.2 Network overhead while recovering. When the administrator recovers a file, RockFS will download the first version of that file and every log entry. The following model can be used to compute the total network traffic required for recovering a file.

$$= \frac{((t + \quad \times t \times \quad) \times n)}{2} \tag{3}$$

For example, by computing this model with a fixed = 30% and $n = 4$, recovering a 1MB file with only 1 version will result in 3MB traffic while a 50MB file with 100 versions will result in 3.1GB. As of April, 2018, Amazon S3 charges about 9 cents of a US dollar for GB downloaded from their cloud. Other cloud storage services charge a similar price for their services. This means that the 50MB file with 100 versions will cost the user around 27 cents of US dollars to recover, while the 1MB file with only 1 version would cost less than a cent.

7 RELATED WORK

RockFS fits in a large class of file systems that aim to provide security guarantees when using the cloud. There are some solutions to provide confidentiality and integrity for files stored in a single cloud, leveraging different cryptographic schemes. Burihabwa et al. present an interesting survey and comparison of such schemes [14]. SafeFS [55] is a modular file system that offers security building blocks based on encryption, replication and coding. Like RockFS, SafeFS is implemented in user space using FUSE to intercept system calls for the file system. None of these systems considers client-side security aspects.

The system architecture of RockFS shares some similarities with Tiera [56], in the sense that it relies on a middleware that runs in the client to abstract the complexity of different cloud services.

Versioning filesystems. The idea of creating a new version for a file every time it gets updated was explored in [48, 50, 59, 67]. These versioning file systems ensure that it is possible to recover an older version of a corrupted file, however, they require a considerable amount of extra storage. In [19], the authors propose two space-efficient metadata structures for versioning file systems, *journal-based metadata* and *multiversion b-trees*, which reduce the required storage by 80% and 99%, respectively, when compared with [48, 59]. Journal-based metadata operates like a write-ahead log, recording the file changes of each file (instead of creating a new copy of the file when it gets updated). When an old version needs to be restored, each change is undone backward through the journal until the desired version is recreated. This process is called *journal roll-back*. Multiversion b-trees work by storing file versions in a binary tree. Each file version has a specific key and timestamp to identify it. When an old version of a file needs to be recovered it can be read from the tree without needing to be reconstructed. Although both structures are capable of recovering older versions of files, journal-based metadata is slow to fetch older versions while multiversion b-trees can fetch older versions almost immediately. This comes with the cost of extra storage and overhead when fetching files during normal operation, since it is required to navigate through an entire b-tree with several versions of files. Given the system model of RockFS, the journal-based approach was used, to reduce the required storage to store the files' metadata. In Git [67] every version of an immutable object is stored, allowing users to revert to previous versions. What differs between Git and RockFS is that RockFS keeps logs in an isolated storage, which is not accessible by any user of the system except for administrators. In Git, if an attacker manages to gain access to a user account with high privileges he can rewrite the history of any file, making the recovery process impossible. RockFS also employs mechanisms to ensure that if the log are tampered it is possible to detect such corruption and abort the recovery process.

Intrusion recovery. The problem of recovering from intrusions by undoing malicious operation has been explored in the literature for some years. Taser [26] is an intrusion recovery service for (local) file systems. It aims at reverting malicious actions taken by attackers while preserving legitimate actions. It operates in three steps: auditing, analysis and recovery. The auditing phase collects operation logs, the analysis phase is started by the administrator to obtain dependencies between operations, and the recovery phase reverts malicious operations by performing a selective redo of the legitimate operations. RockFS shares some similarities with Taser, but includes several mechanisms that are necessary to fit the cloud-of-clouds model: logging of the user who performed the action; recovery on both the client and cloud sides; and the assumption that the operations log is not a trustworthy component and, therefore, requires additional integrity protection mechanisms.

Back to the Future is a framework that uses an approach similar to Taser's for malware removal [31]. A *monitor* is responsible for intercepting all read and write operations from processes, then a *logger* records these operations so that later a *recoverer* is able to undo faulty operations. In order to preserve integrity, *trusted* and *untrusted* processes are taken into account by the monitor. This guarantees that trusted processes do not read untrusted data.

The Retro system [36] repairs desktops or servers that were compromised by applying selective re-execution of legitimate actions. This technique undoes malicious actions performed by adversaries while keeping legitimate files intact. Retro records an *action history graph* that describes actions and their dependencies. This graph allows to recover the system with minimal impact since only the malicious actions and their dependencies are recovered. Retro also employs a mechanism to execute compensating actions, which allow the propagation of recovery to external services. Retro solves one limitation of Taser when it comes to refining legitimate and illegal actions. Taser may incur false positives, i.e. undo legitimate actions. RockFS is fundamentally different from Retro, as it does not employ any mechanism to take into account how processes affect the file system.

The use of erasure-codes to recover data files has been proposed in other works [16, 27, 29]. More recently, Mitra et. al. [49] proposed Partial Parallel Repair (PPR), a novel method to recover data files using erasure-codes. As the authors show in their work, PPR is

able to reduce the required time to recover a data file by computing the repair process of the shares in parallel. Although this solution provider better performance when comparing to ours, because we are using storage containers without computation capacity, it is impossible to adopt the PPR mechanism.

Log integrity. As explained above, the recovery log has to provide *forward-secure stream integrity* [44]. The first solutions to enforce this property were based on forward-secure sequential aggregate signatures. These solutions generate a single digital signature or message authentication code to protect the whole log [42, 43]. However, such schemes have two problems: they are not secure against truncation attacks (the attacker can delete a contiguous segment of entries at the end of the log) and delayed detection attacks (the attack is not detected until the entire log is downloaded to a trusted server). RockFS uses the forward-secure stream integrity scheme (FssAgg) of Ma and Tsudik that solves these two problems [44]. There are similarities between forward-secure streams and blockchains [21, 51, 53], as both aim to provide integrity of a log. The original blockchain is the core of the Bitcoin cryptocurrency [51]. Bitcoin relies on a peer-to-peer network of *miners* that remove the need for a trusted central authority. Each miner is randomly assigned the right to add new blocks to the chain. To be able to do so, it has to solve a cryptopuzzle, which is a slow and power-hungry process. As of November 2017, Bitcoin can only handle 7 transactions per second with the maximum block size of 1 MB [51]. For this reason, *proof-of-work* blockchains like Bitcoin are not suitable for log integrity protection, as required by RockFS. Regarding other approaches, there are *proof-of-stake* blockchains that use alternative means to randomly assign the right to add blocks [7, 37]. These approaches require a large number of participants and activity level that is not a good fit for a user-driven file system like RockFS. Permissioned blockchains like Hyperledger Fabric would be a better fit, as they perform better than their permissionless counterparts, but they are arguably slow and complex in comparison to RockFS' solution [15]. There is much more work on making blockchains efficient, but they seem to be too complex to be part of a system like RockFS [13, 25].

Endpoint security and ransomware protection. Detecting malware in endpoints is an old problem [64], which fostered the appearance of a large industry focused on malware detection and anti-malware protection. Ransomware is a reasonably recent threat [35], so protection mechanisms are still limited. There is some work on detecting ransomware before it does its job, e.g. by verifying software in cloud services before it is downloaded [40, 60]. Another solution is to do backups and keep them offline. RansomSafeDroid improves this last approach for protecting mobile devices in real-time by running backup software inside a trusted execution environment, supported by ARM processors with the TrustZone extension [69]. RockFS uses an approach that is new in this context –secret sharing [12, 62]–, which allows recovering the cryptographic material and files in case ransomware or other malware encrypts or deletes it.

8 CONCLUSION

This paper presented RockFS, a recoverable cloud-of-clouds file system resilient to client-side attacks. It provides recovery mechanisms for the access credentials of the users and for files stored in the cloud storage services. RockFS improves on SCFS and other cloud-backed file systems by protecting against client-side attacks and allowing for recovery of unauthorized changes, in particular, recovery from ransomware attacks. The experimental evaluation results show that it is possible to recover intensively modified files (with 100 updates) in around 40 seconds. Using RockFS to log file system operations imposes a performance overhead in the order of 20%, a cost that can be further reduced by improving the computing characteristics of the clouds used in the system.

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 PTDC/EEI-SCR/1741/2014 (Abyss) and UID/CEC/50021/2013 (INESC-ID).

REFERENCES

[1] 2018. JClouds. http://jclouds.apache.org. (2018).
[2] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon. 2010. RACS: a case for cloud storage diversity. In *Proceedings of the 1st ACM Symposium on Cloud Computing*. 229–240.
[3] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch. 2007. A five-year study of file-system metadata. *ACM Transactions on Storage* 3, 3 (2007), 9.
[4] EC Amazon. 2015. Amazon web services. *Available in: http://aws.amazon.com/es/ec2/* (2015).
[5] EC Amazon. 2015. Amazon web services. *Available in: http://aws. amazon.com/es/ec2/(November 2012)* (2015).
[6] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr. 2004. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing* 1, 1 (Jan 2004), 11–33. https://doi.org/10.1109/TDSC.2004.2
[7] I. Bentov, C. Lee, A. Mizrahi, and M. Rosenfeld. 2014. Proof of Activity: Extending Bitcoin's Proof of Work via Proof of Stake. *ACM SIGMETRICS Performance Evaluation Review* 42, 3 (2014), 34–37.
[8] A. N. Bessani, E. P. Alchieri, M. Correia, and J. S. Fraga. 2008. DepSpace: a Byzantine fault-tolerant coordination service. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*. 163–176.
[9] A. N. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa. 2011. DepSky: dependable and secure storage in a cloud-of-clouds. *EuroSys'11 Proceedings of the 6th Conference on Computer Systems* (2011), 31–46.
[10] A. N. Bessani, R. Mendes, T. Oliveira, N. Neves, M. Correia, M. Pasin, and P. Verissimo. 2014. SCFS: A Shared Cloud-backed File System. In *Proceedings of USENIX Annual Technical Conference*. 169–180.
[11] A. N. Bessani, M. Santos, J. Felix, N. F. Neves, and M. Correia. 2013. On the Efficiency of Durable State Machine Replication. In *Proceedings of the USENIX Annual Technical Conference*. 169–180.
[12] G. R. Blakley. 1979. Safeguarding Cryptographic Keys. In *Proceedings of the AFIPS National Computer Conference*, Vol. 48. 313–317.
[13] E. Buchman. 2016. *Tendermint: Byzantine Fault Tolerance in the Age of Blockchains.* Master's thesis. The University of Guelph.
[14] D. Burihabwa, R. Pontes, P. Felber, F. Maia, H. Mercier, R. Oliveira, J. Paulo, and V. Schiavoni. 2016. On the cost of safe storage for public clouds: an experimental evaluation. In *Proceedings of the 35th IEEE Symposium on Reliable Distributed Systems*. 157–166.
[15] C. Cachin. 2016. Architecture of the Hyperledger blockchain fabric. In *Workshop on Distributed Cryptocurrencies and Consensus Ledgers*.
[16] C. Cachin and S. Tessaro. 2006. Optimal resilience for erasure-coded Byzantine distributed storage. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*. 115–124.
[17] B. Calder et al. 2011. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 143–157.
[18] Cloud Security Alliance. 2012. Security Guidance for Critical Areas of Mobile Computing. (2012).
[19] A. N. Craig, G. R. Soules, J. D. Goodson, and G. R. Strunk. 2003. Metadata Efficiency in Versioning File Systems. In *2nd USENIX Conference on File and Storage Technologies*.
[20] D. Dobre, P. Viotti, and M. Vukolić. 2014. Hybris: Robust hybrid cloud storage. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 1–14.
[21] Ethereum team. 2014-17. Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. (2014-17). White Paper.
[22] European Union Agency for Network and Information Security. 2014. *Algorithms, Key Size and Parameters Report – 2014.* ENISA.
[23] Filebench. 2017. FileBench. *https://github.com/filebench/filebench* (2017).

[24] B. Gallmeister. 1995. *POSIX. 4 Programmers Guide: Programming for the real world.* O'Reilly.
[25] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich. 2017. Algorand: Scaling Byzantine agreements for cryptocurrencies. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles.* 51–68.
[26] A. Goel, K. Po, K. Farhadi, Z. Li, and E. De Lara. 2005. The Taser Intrusion Recovery System. In *ACM SIGOPS Operating Systems Review*, Vol. 39. ACM, 163–176.
[27] R. R. Goodson, J. Wylie, G. R. Ganger, and M. K. Reiter. 2004. Efficient Byzantine-tolerant erasure-coded storage. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks.* 135–144.
[28] R. Halalai, P. Felber, A. M. Kermarrec, and F. TaÃŕani. 2017. Agar: A Caching System for Erasure-Coded Data. In *Proceedings of the IEEE 37th International Conference on Distributed Computing Systems.* 23–33. https://doi.org/10.1109/ICDCS.2017.97
[29] J. Hendricks, G. R. Ganger, and M. K. Reiter. 2007. Low-overhead Byzantine fault-tolerant storage. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles.* 73–86.
[30] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. 1988. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems* 6, 1 (1988), 51–81.
[31] F. Hsu, H. Chen, T. Ristenpart, J. Li, and Z. Su. 2006. Back to the future: A framework for automatic malware removal and system repair. In *Proceedings of the 22nd Annual Computer Security Applications Conference.* 257–268.
[32] H. W. Hunt and T. G. Szymanski. 1977. A fast algorithm for computing longest common subsequences. *Commun. ACM* 20, 5 (1977), 350–353.
[33] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. 2010. ZooKeeper: wait-free coordination for Internet-scale systems. In *USENIX Annual Technical Conference.*
[34] S. Kamara and K. Lauter. 2010. Cryptographic cloud storage. In *International Conference on Financial Cryptography and Data Security.* Springer, 136–149.
[35] A. Kharraz, W. Robertson, D. Balzarotti, L. Bilge, and E. Kirda. 2015. Cutting the Gordian knot: A look under the hood of ransomware attacks. In *Proceedings of the 12th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment.* 3–24.
[36] T. Kim, X. Wang, N. Zeldovich, and M. F. Kaashoek. 2010. Intrusion Recovery Using Selective Re-execution. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation.*
[37] S. King and S. Nadal. 2012. PPCoin: Peer-to-peer crypto-currency with proof-of-stake.
[38] S. P. T. Krishnan and J. L. U. Gonzalez. 2015. Google compute engine. In *Building Your Next Big Thing with Google Cloud Platform.* Springer, 53–81.
[39] C. Kruegel, G. Vigna, and W. Robertson. 2005. A multi-model approach to the detection of web-based attacks. *Computer Networks* 48, 5 (2005), 717–738.
[40] J. K. Lee, S. Y. Moon, and J. H. Park. 2017. CloudRPS: a cloud analysis based enhanced ransomware prevention system. *The Journal of Supercomputing* 73, 7 (2017), 3065–3084.
[41] Z. Li, M. Liang, L. O'Brien, and H. Zhang. 2013. The cloud's cloudy moment: A systematic survey of public cloud service outage. *arXiv preprint arXiv:1312.6485* (2013).
[42] D. Ma. 2008. Practical forward secure sequential aggregate signatures. In *Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security.* 341–352.
[43] D. Ma and G. Tsudik. 2007. Forward-secure sequential aggregate authentication. In *Proceedings of the IEEE Symposium on Security and Privacy.* 86–91.
[44] D. Ma and G. Tsudik. 2009. A new approach to secure logging. *ACM Transactions on Storage* 5, 1 (2009).
[45] T. Mather, S. Kumaraswamy, and S. Latif. 2009. *Cloud security and privacy: an enterprise perspective on risks and compliance.* O'Reilly.
[46] D. Matos and M. Correia. 2016. NoSQL Undo: Recovering NoSQL Databases by Undoing Operations. In *Proceedings of the 15th IEEE International Symposium on Network Computing and Applications.*
[47] D. R. Matos, M. Pardal, and M. Correia. 2017. Rectify: Black-Box Intrusion Recovery in PaaS Clouds. In *Proceedings of the 2017 ACM/IFIP/USENIX International Middleware Conference.*
[48] K. McCoy. 1990. *VMS file system internals.* Digital Press.
[49] S. Mitra, R. Panta, M. Ra, and S. Bagchi. 2016. Partial-parallel-repair (PPR): a distributed technique for repairing erasure coded storage. In *Proceedings of the 11th ACM European Conference on Computer Systems.*
[50] K. Muniswamy-Reddy, C. P. Wright, A. Himmer, and E. Zadok. 2004. A Versatile and User-Oriented Versioning File System. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies.* 115–128.
[51] S. Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system.
[52] D. Nascimento and M. Correia. 2015. Shuttle: Intrusion Recovery for PaaS. In *Proceedings of the 2015 IEEE 35th International Conference on Distributed Computing Systems.* 653–663.
[53] M. E. Peck. 2017. Blockchains: How they work and why they'll change the world. *IEEE Spectrum* 54, 10 (2017), 26–35.
[54] S. Pereira, A. Alves, N. Santos, and R. Chaves. 2016. Storekeeper: A Security-Enhanced Cloud Storage Aggregation Service. In *Proceedings of the 35th Symposium on Reliable Distributed Systems.*
[55] R. Pontes, D. Burihabwa, F. Maia, J. Paulo, V. Schiavoni, P. Felber, H. Mercier, and R. Oliveira. 2017. 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.*
[56] A. Raghavan, A. Chandra, and J. B. Weissman. 2014. Tiera: towards flexible multi-tiered cloud storage instances. In *Proceedings of the 15th International Middleware Conference.* ACM, 1–12.
[57] R. Rizun. [n. d.]. S3FS - FUSE-based file system backed by Amazon S3. http://code.google.com/p/s3fs/.
[58] W. Robertson, G. Vigna, C. Kruegel, and R. Kemmerer. 2006. Using generalization and characterization techniques in the anomaly-based detection of web attacks. In *Proceedings of the 13th Symposium on Network and Distributed System Security.*
[59] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. 1999. Deciding when to forget in the Elephant file system. In *Proceedings of ACM SIGOPS Symposium on Operating Systems Principles.* 110–123.
[60] N. Scaife, H. Carter, P. Traynor, and K. R. B. Butler. 2016. Cryptolock (and drop it): stopping ransomware attacks on user data. In *Proceedings of the IEEE 36th International Conference on Distributed Computing Systems.* 303–312.
[61] B. Schoenmakers. 1999. A simple publicly verifiable secret sharing scheme and its application to electronic voting. *Proceedings of the 19th International Cryptology Conference* (1999), 148–164.
[62] A. Shamir. 1979. How to share a secret. *Communications of ACM* 22, 11 (Nov. 1979), 612–613.
[63] M. Shirvanian, S. Jareckiy, H. Krawczykz, and N. Saxena. 2017. SPHINX: A Password Store that Perfectly Hides Passwords from Itself. In *IEEE 37th International Conference on Distributed Computing Systems (ICDCS).* 1094–1104. https://doi.org/10.1109/ICDCS.2017.64
[64] E. Skoudis. 2003. *Malware: Fighting Malicious Code.* Prentice Hall.
[65] J. M. Stanton, K. R. Stam, P. Mastrangelo, and J. Jolton. 2005. Analysis of end user security behaviors. *Computers & Security* 24, 2 (2005), 124–133.
[66] H. Takabi, J. B. D. Joshi, and G. Ahn. 2010. Security and privacy challenges in cloud computing environments. *IEEE Security & Privacy* 8, 6 (2010), 24–31.
[67] L. Torvalds and J. Hamano. 2010. Git: Fast version control system. *URL http://git-scm.com* (2010).
[68] M. Vrable, S. Savage, and G. M. Voelker. 2012. BlueSky: A cloud-backed file system for the enterprise. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies.*
[69] S. D. Yalew, G. Q. Maguire Jr., S. Haridi, and M. Correia. 2017. Hail to the Thief: Protecting Data from Mobile Ransomware with ransomSafeDroid. In *Proceedings of the 16th IEEE International Symposium on Network Computing and Applications.*
[70] R. Zhao, Yue, B. Tak, and C. Tang. 2015. SafeSky: a secure cloud storage middleware for end-user applications. In *Proceedings of the 34th IEEE Symposium on Reliable Distributed Systems (SRDS).* 21–30.