A Secure Distributed Cloud Storage System for Android Devices

Paulo Ribeiro, Rui Prior, and Sérgio Crisóstomo

Instituto de Telecomunicações & Departamento de Ciência de Computadores, Faculdade de Ciências da Universidade do Porto

Abstract. Digital data storage is essential nowadays. We store all types of data in our devices, either in local storage or using cloud storage services. Cloud storage services have several advantages, such as data sharing among devices, space-saving in local storage, and data preservation in case of user device's hardware failure. However, those services come with associated risks, which users often are not aware of, such as temporary/permanent data unavailability or loss of confidentiality. We propose a secure file storage system based on public cloud services that mitigates these risks by combining the use of multiple cloud providers with redundancy mechanisms and cryptographic techniques. The system ensures that, even if one provider is hostile or goes out of business, there is no loss of data or confidentiality. A performance penalty is paid when compared to a plain cloud storage service, but is acceptable in face of the additional guarantees.

Keywords: cloud storage, confidentiality, availability, integrity, mobile application, Android, file system

1 Introduction

Data storage is essential nowadays. We store everything, from personal data (like photos or text messages) to critical data (like medical records or financial reports). There are two main approaches to storing data: local storage or cloud storage. The concept of storing data in the cloud emerged in the past few years, embedded in the cloud computing concept. Cloud storage is a particular case of cloud computing where the computing resources are storage servers. A Cloud Storage Provider (CSP) provides a Cloud Storage Service (CSS) that enables users to store and manage the files in the remote infrastructure (cloud). Many CSSs are available, targeting individual users or organizations. For individual users, there are many free options providing a decent amount of space (at least 15GB), and for organizations, usually needing more space and better access speeds, there are a many paid services that usually charge for bandwidth and storage capacity.

Storing data in the cloud has several advantages: we can overcome the limitation of the local device storage capacity, the files are readily available from anywhere on any device, and the cost is typically lower due to scale factors. Despite all the advantages, there are some drawbacks to storing data in the cloud. Local storage is more secure [7,6]. When we use cloud storage services, our data is stored in a place we do not know, supervised by people we do not know. We cannot, therefore, be sure that the Cloud Storage Provider (CSP) will not lose, misuse, or leak our data. Some recent incidents remind us that we can not blindly trust CSPs to keep our data safe. CSPs can fail or be hacked, leading to data leaks or temporary (or even permanent) loss of stored data. In 2016, every major commercial CSP was offline for some time. The Google Cloud Platform was offline for more than 11 hours [3,4]. In 2012, two-thirds (68 million) of Dropbox user passwords were stolen by hackers and later made publicly available on the Internet [5]. In 2014, one of the most mediatic hacks ever occurred: hundreds of private photos of celebrities were leaked from the Apple iCloud storage service [12].

Another big concern about keeping data in the cloud is the vendor-lock-in issue. CSPs can at any instant raise the prices or start charging free users for space and/or bandwidth. This means that if we have the data stored at only one provider, we may either lose the access to the data or be forced to pay a perhaps significant fee to recover it [14]. Yet another concern is the lack of confidentiality or availability guarantees of data stored in public clouds.

This work describes the Multiple Cloud Overlay File System (MCOFS), an Android application that, using information partitioning and cryptographic techniques, allows users to keep folders in publi CSSs, like Dropbox, Google Drive or OneDrive, while ensuring the confidentiality, availability, and integrity of their contents. Caching techniques are used to improve application performance.

The rest of the paper is organized as follows. In Section 2 we overview systems that are somehow related to MCOFS, highlighting the similarities and differences. In Section 3 we make a top-down description of the system architecture, explaining its main components and the mechanisms used to achieve cloud storage confidentiality, availability and integrity. In Section 4 we briefly describe the development of the system, explaining the developed modules to achieve cloud storage confidentiality, availability, and integrity. In Section 5 we analyze experimentally the performance of MCOFS. Finally, Section 6 provides some concluding remarks.

2 Related Work

The MCOFS solution is unique for the Android platform: it the only one providing improvements in data availability, confidentiality, and integrity for every type of files. Some available applications combine multiple CSP accounts into one, basically merging the available space into one file system, but they do not add any security or redundancy mechanisms. Other applications ensure data confidentiality by encrypting the files before storing them remotely, but those applications do not provide availability mechanisms. MCOFS allows users to mount the File System (FS) as a local folder, therefore, integrating it into the

Android OS, which no other solution does. Below we describe some solutions that are somehow related to MCOFS:

EasySSHFS [2] is a solution that, like ours, allows users to mount a remote FS into a local storage folder, but it is not a cloud storage solution and does not add any security or redundancy mechanisms. The Android application allows mounting a remote FS using SFTP that most SSH servers support and enable this SFTP access by default;

SafeCloud Photos [15] is a solution that is similar to ours. However it is devoted only to image files. It is an Android application that allows users to combine the storage capacity of multiple public CSP accounts and makes use of use cryptographic techniques to enforce data privacy. This solution is focused on privacy and does not provide any redundancy mechanisms. MCOFS can be considered an extension of this solution, allowing any file type and adding availability and integrity mechanisms.

In contrast with the Android application ecosystem, the Linux OS has already some cloud storage based solutions fulfilling the confidentiality, integrity, and availability requirements:

DepSky [1] is a system that improves confidentiality, integrity, and availability of data stored in the cloud. It uses encryption, encoding, and replication of data over several commercial clouds. The goal is similar to ours, but DepSky uses different encoding and cryptographic techniques. The system behaves like a virtual disk where we can store data at a block level;

C2FS [13] is a system that uses DepSky as data storage service, but implements a POSIX compatible file system. It improves the data and metadata availability, integrity and confidentiality. The architecture of the system is similar to DepSky's, and it stores the data in a cloud of clouds (i.e., in multiple clouds). Its major advantages over DepSky are the ease of use (high level abstraction, in the sense that it behaves like a regular local folder) and the distributed directory services that ensure the meta-data availability and confidentiality.

3 System Architecture

MCOFS runs on Android smartphones and provides applications access to files stored in the cloud through a local folder. When the user saves a file in that folder, the file is not stored locally on the device, but sent to the cloud. When the user requests a file, it is downloaded from the cloud (multiple CSPs) and presented to the user. MCOFS implements mechanisms to keep the files secure: if one provider is hostile, goes out of business or an attacker manages to access or corrupt our files, there is no loss of data or confidentiality.

Figure 1 shows the architecture of the system and how its components interact. We have a FS implementation that intercepts and replaces the Android FS calls, and modules that help us achieve file availability, confidentiality, and

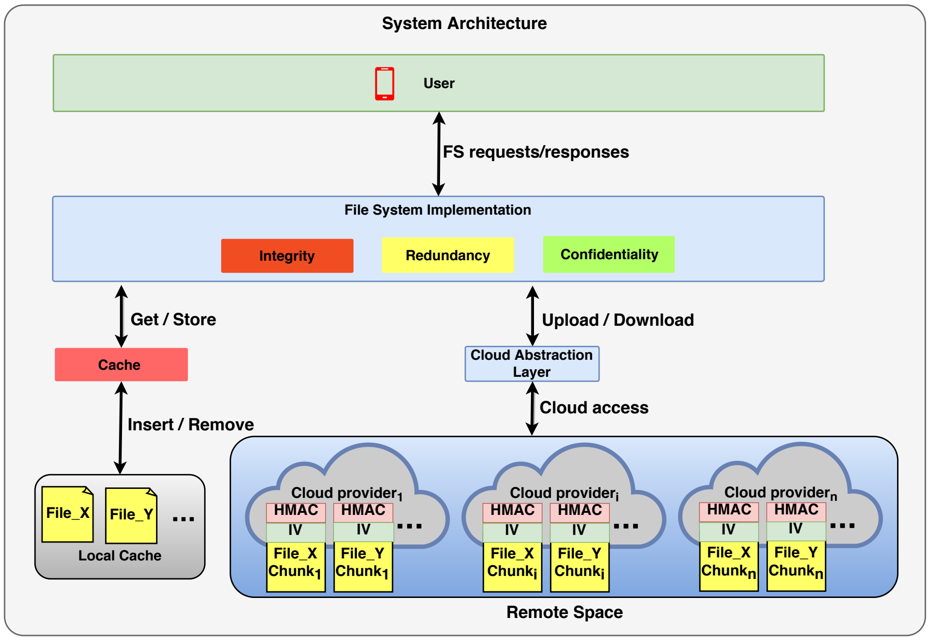

Fig. 1. System Architecture, with a configuration of three CSPs

integrity. The Integrity module is used to verify if an unauthorized entity has modified a file that we previously stored. The Redundancy module is responsible for giving the FS the ability to resist CSPs downtime, which is done by adding redundant information to the files and storing them split across various CSPs. The Confidentiality module is used to encrypt the files, so they remain confidential even if they are leaked. The Cloud Abstraction module is responsible for making the communication between our system and the CSPs transparent and uniform, and the Cache module is used to improve the system performance by maintaining local copies of the files under certain circumstances.

3.1 Cloud Abstraction

Since each CSP has its own Application Programming Interface (API) to manage the remote space, the Cloud Abstraction module provides an intermediate API that allows us to abstract from the individual CSP APIs and perform actions uniformly across the different CSPs. It provides generic interface with the necessary functions (store a file, get a file, list the files in a folder, among others) to interact with a CSP and independent implementations to interact with different CSPs. When a function is called, it automatically calls the specific implementation depending on the CSP we need to interact with.

3.2 Availability

In order to improve the availability of the stored files, we take advantage of redundancy techniques. The Availability module provides functions that add redundant information to the system and recover lost data if needed.

As every redundancy technique implies a performance overhead and a higher usage of storage space we need to go for a method that can balance those two aspects. Our system uses coding techniques because the alternative (information replication) would have a much higher storage space overhead. MCOFS uses simple parity because it is easy to implement and has a low impact on the system performance, as it is based on simple operations. The downside of this technique is that it only provides fault tolerance of one CSP at a time, but since the probability of more than one becoming inaccessible (or having files become corrupted) simultaneously is very low, it is a good choice.

Before storing the files in the remote space, we split them into $N-1$ chunks, where N is the number of CSPs in use, and then generate an extra chunk containing the bit by bit parity of the other chunks computed using XOR.

When all CSPs are accessible, we can recover the file by getting the first $N-1$ chunks. $chunk_N$, containing the parity, is only necessary for recovering a lost file chunk (due the CSP being inaccessible or the data being corrupted). The missing chunk is recovered simply by XORing the available chunks.

3.3 Confidentiality

Files remotely stored in CSPs are protected by a security mechanism (Authentication by Username and Password), and some CSPs claim that the files are encrypted. However, we do not know who has access to the cryptographic keys. We therefore assume the worst case scenario where the CSPs can be malicious. Moreover, the authentication mechanism is not strong enough to ensure the confidentiality of the stored files, as many examples unfortunately attest. Some redundancy techniques also provide confidentiality [9], but we want a robust system able to maintain the confidentiality of all the stored content even if the entire contents of a single CSP are leaked.

As we can not trust the CSPs to keep our files confidential, we use cryptographic techniques to that end. We opted for symmetric cryptography, that requires only one key, used in both the encryption and the decryption processes. Symmetric Cryptography is much faster and energy efficient when compared to Public Key Cryptography.

Within symmetric cryptography, there are some alternatives that could be used [11]. We opted for Advanced Encryption Standard (AES) cipher because it is widely accepted is the most utilized in the industry, and, most importantly, because it is considered secure. Moreover, some smartphones chipsets support AES hardware acceleration. Though we strongly recommend the use of AES, there is no impediment to the utilization of a different symmetric cipher, as long it is considered secure and resistant to cryptanalysis.

The AES cipher has many operation modes, but we recommend Cipher Block Chaining (CBC) mode, where we need a key and an Initialization Vector (IV). With CBC, the previous ciphertext block (or IV) is effectively random (and independent from the plaintext block), making the block ciphertext an effectively random string.

AES comes with three standard key sizes (128, 192 and 256 bits). The 256-bit version is a bit slower than the 128-bit version (by about 40%) but as the National Institute of Standards and Technology of U.S. (NIST) recommends the 256-bit version over the 128-bit version, we follow that recommendation.

The IV is used to add randomness at the start of the encryption process. When using CBC mode (where one block incorporates the prior block), we need a value for the first block, which is where the IV comes in.

If we had no IV (or always reused the same), using CBC with just a key, two files that begin with identical text would produce identical first blocks. If the input files changed midway through, then the two encrypted files would start to look different from that point until the end of the encrypted file. If someone noticed the similarity at the beginning, they could use that information to gain information on the key or the beginning of some files throughout cryptanalysis. Therefore, we cannot use the same IV for different files, and this is very important to keep the process secure.

Confidentiality is obtained by encrypting the files before storing them in the CSPs. The key used for encryption is the same for all files, and is generated from a user-provided password. The IV is randomly generated for each file, and changed every time it is updated. The IV is then stored in the header of the files as we need it for decryption.

3.4 Integrity

To ensure data integrity we resort to Hash-based Message Authentication Code (HMAC). A HMAC function accepts a secret key and an arbitrary-length message as input, and yields a Message Authentication Code (MAC). Anyone knowing the secret key may use the MAC value to check whether the message was altered, ensuring its integrity, and that the MAC was created by someone knowing the key, ensuring data authenticity [10].

MCOFS uses this method to protect data files. The process is simple: we compute a HMAC for each chunk of the file (one per CSP) and store it in the chunk file header before sending the chunk to the CSPs. After retrieving the file we calculate the HMAC value again and check that it matches the HMAC stored in the header. The key used in the HMAC is the same for all files, and is generated from a user-provided password in the initial configuration of the system, similarly to the key used for confidentiality. MCOFS allows the users to define both keys, and they should be different so that if one of the used algorithms is broken, the other stays safe.

The generation of HMAC values could be made using different hash functions. We used the Secure Hash Algorithm (SHA) 2 because it is widely used and it is defined as a standard by NIST, which means that the algorithm validity has been thoroughly tested. Many attack attempts have been made against the algorithm, but none is able to entirely compromise its security [8]. Though NIST already defined a new standard to replace SHA 2, as of now there are no available implementations, and SHA 2 is still considered secure to use.

3.5 Caching

The MCOFS incorporates a client cache in order to improve access performance (i.e., minimize delay and increase throughput) to data stored remotely across the several CSPs. The choice of writing policy, which defines when a cached content is written to the backing storage (i.e., the CSPs in the case of MCOFS), and cache replacement policy, which defines which cached content shall be replaced when the cache is full, is important for the system to perform optimally. We considered the following writing policies:

Write-through Any modification to a file is synchronously reflected both in the local storage (cache) and in the backing storage (CSPs);

Write-back When a file is modified, it is written only to the cache. The backing storage will only be updated when a cached file is about to be replaced by another cache block.

Write-back on close When a file is modified, the changes are only reflected in the local cache. Updates to the backing store will be made either when a file is closed or when a cached file is about to be replaced by another cache block.

The writing policy that we consider most appropriate for our system is Write-back on close because it has reduced communication overhead when compared with write-through and less failure-prone than pure write-back.

For the cache replacement policy, we chose Least Recent Used (LRU), which is usually recommended for networked file systems [16]. With this policy, when the cache becomes full and cache space is needed for new data blocks, the blocks to be evicted will be the least recently used ones. This cache policy is known to have good performance, in terms of cache hits, in access patterns in which data that has been used in the recent past is likely to be referenced again in the near future – a property known as temporal locality.

MCOFS is designed to operate in a scenario with only one device accessing and changing the data. However, in order to avoid file system inconsistencies, the system design had to consider the possibility of (1) failure writing to the CSPs; and (2) critical application failures. To address these failure scenarios, we incorporated a journal/file version manager in MCOFS: the previous version of the file stored in the backing store is deleted only after the confirmation that all chunks of the new version were successfully stored in the backing store.

4 System Implementation

The file system implementation is based on File System In User Space (FUSE), an interface for userspace programs to export a filesystem to the Linux kernel.

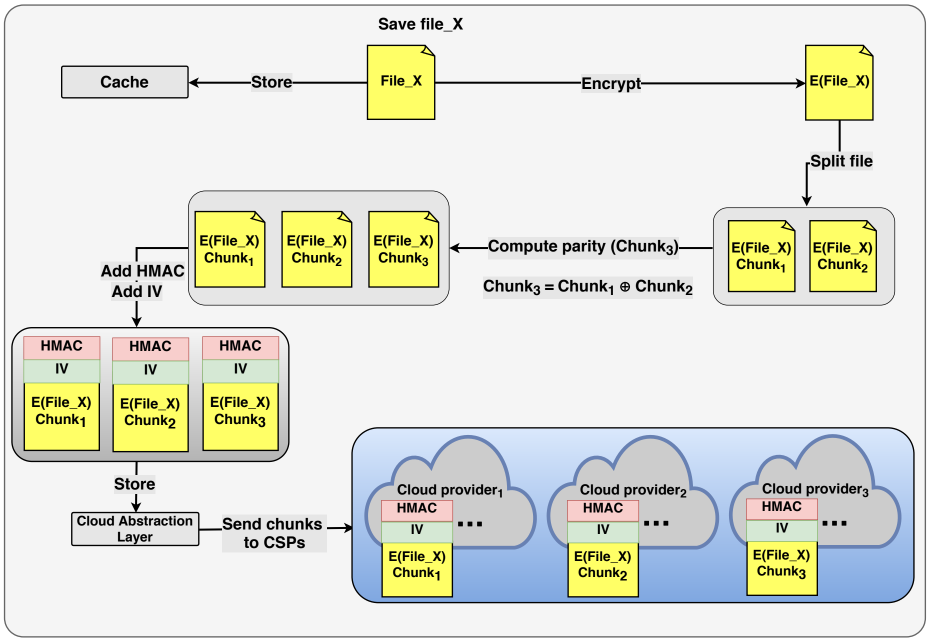

Fig. 2. Process of storing a file, with a configuration of $N = 3$ clouds, where the redundant chunk is stored in cloud number 3

Using FUSE, we intercept the Android FS calls and provide an implementation that interfaces with the CSPs through a layer that adds availability, confidentiality, and integrity, using the described techniques.

MCOFS requires at least three CSP accounts to be configured. This is done using a graphical interface. Currently, the CSP account that holds the parity chunks is fixed at configuration time. In a future version, we intend to aid this choice with a module that evaluates the access speed of each CSP in order to use the slowest one to store the parity chunks, which should be accessed less often than the others.

Figure 2, illustrates how the different module participates in the process of storing a new file in the FS. When the user creates a new file in the FS folder, a file is created in the local cache. Then, a sequence of system writes is performed (locally only, for now). In this process, other files may be removed from the cache, if necessary, to free up space. When the flush call is invoked (file saved/closed), we prepare the file to be uploaded: the Confidentiality module encrypts the file using the user-provided key and a randomly generated IV. The Redundancy module splits the file into $N-1$ chunks ($N = 3$ in the example) and computes the parity chunk. Then, for each chunk, the Integrity module calculates an HMAC, which is added to the file header, together with the previously generated IV. Finally, the Cloud Abstraction Module stores the resultant file chunks across the N CSPs. During this process ,if we detect that some of the used CSPs is

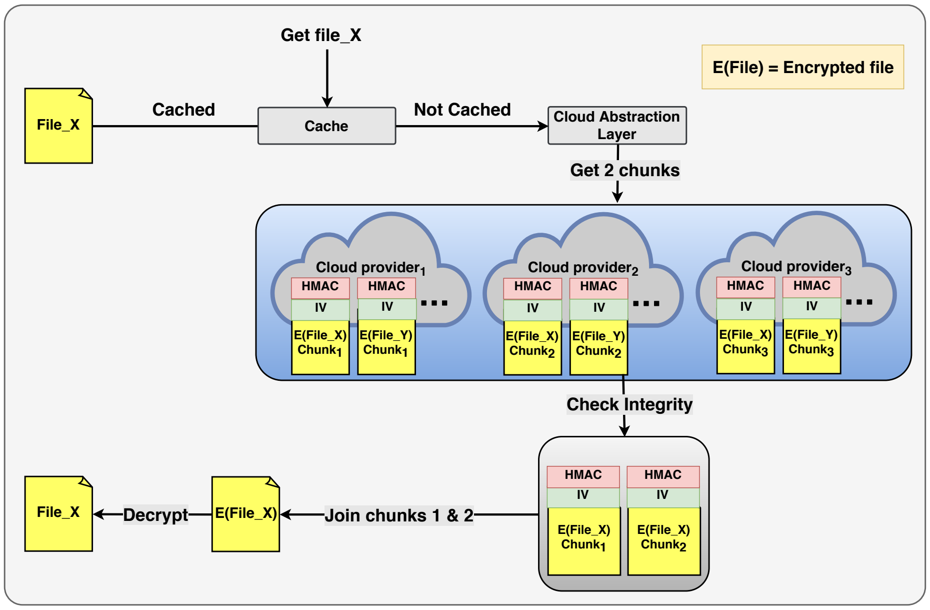

Fig. 3. Process of getting a file, with a configuration of $N = 3$ clouds, where the redundant chunk is stored in CSP 3. All CSPs available

unavailable, we store the other file chunks normally and notify the user of the acCSP state.

When accessing a file, first we check whether it is cached. If not, we proceed to download $N-1$ chunks, checking their integrity by comparing the computed HMAC values of the downloaded chunks with the HMACs stored in the file headers. Then, we merge the chunks (removing the headers first), reconstructing the complete original encrypted file. After that, using the IV in the file header and the encryption key, we decrypt the original file. Finally, we add the file to the local cache, so that the subsequent accesses to that file will not require traffic exchange with the CSPs. This process is illustrated in Figure 3, which highlights a scenario where the file integrity has not been violated and all the CSPs are available. Figure 4 illustrates the workflow for the case where a CSP is unavailable.

If we detect that the integrity of a downloaded chunk was violated, we can recover it by downloading a chunk from another CSP, and proceed as if the CSP of the violated chunk were offline (the case illustrated in Figure 4).

The current MCOFS implementation is not designed to support concurrent writes. However, it can be accessed by multiple devices at the same time in read-only mode. In case we wish to migrate the FS access to another device, we just need to configure the corresponding CSP accounts and passwords in the destination device.

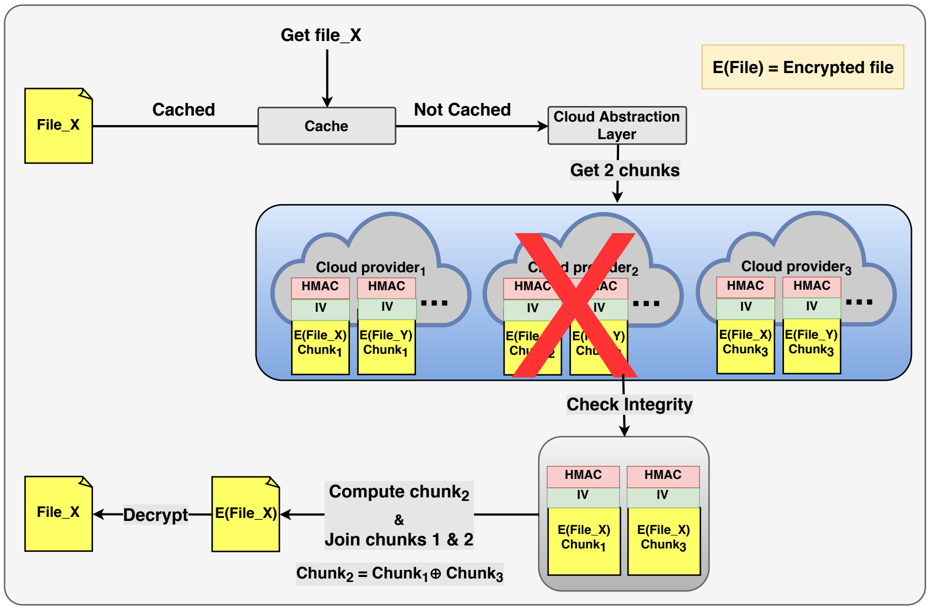

Fig. 4. Process of getting a file, with a configuration of $N = 3$ clouds, where the redundant chunk is stored in CSP 3; CSP 2 offline

To migrate the data, the user only needs to configure the same accounts and the same two passwords in the new device.

5 Performance evaluation

To assess the performance of MCOFS, we did some experiments where we measured the time to store and read files on the cloud. The experiments were done using three different Dropbox accounts in order to have this assessment earlier, since support for each CSP must be coded in the Cloud Abstraction module. The results were compared to a conventional single cloud application (Android Dropbox application).

We measured the times for storing and retrieving 10Mb files. For the file retrieval operations we tested different scenarios: without caching and with all CSPs available, without cache with one CSP out of service, and with file cache enabled. The measurements were repeated 20 times and the results averaged. The results of these experiments are shown in Figure 5.

Storing a file with MCOFS takes twice as long when compared to the Dropbox Android application. This difference is reasonable and expected, since we need to perform a lot more computation and need to store file chunks in different destinations (three different accounts, in this experiment).

Without caching, the average time it takes to store and get a file with MCOFS is about 2.6 times greater when compared to the Dropbox Android application.

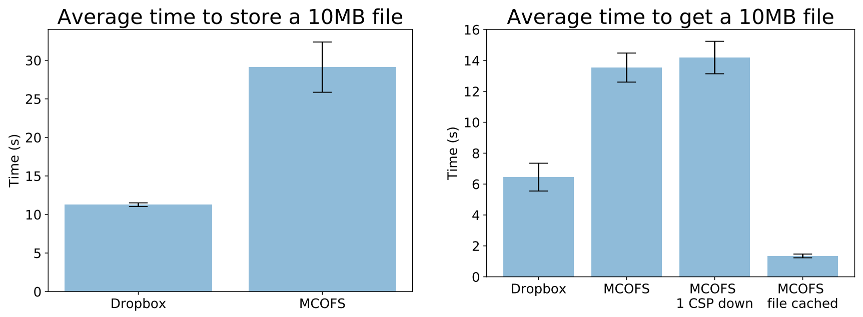

Fig. 5. Average time to get a 10Mb file with MCOFS and Dropbox Android application

Again, this is expected, for similar reasons. When we have one CSP unavailable, we have an extra step of computing the missing chunk from parity, but the time difference is almost insignificant. As can be seen, the caching mechanism plays a vital role in improving performance, reducing the access time to about one second.

6 Conclusion

MCOFS is an Android File System implementation that uses public CSPs as backing storage. It provides file confidentiality and is able to detect and recover from integrity violations, prevent data leaks, and recover from temporary or permanent failure of a CSP. Currently, no other Android cloud storage application provides these features.

Though its performance is slower than the Dropbox Android application (the most popular cloud storage application for Android), it is so by a small factor. We feel that this modest decrease in performance is a small price to pay for the unique availability and confidentiality features. As several recent incidents show us, we need to take measures ourselves to protect our data instead of just trusting some company to do it.

Since Android is moving away from FUSE (previously used to access the sdcard), in the future we may need to reimplement MCOFS using a different approach to user-space filesystems, or, as a last resort, by mirroring operations on local files monitored with FileObserver. Also, in the future the support for concurrent access should be added, possibly using a lock system.

Acknowledgments

This work was funded by Instituto de Telecomunicações, ref. UID/EEA/50008/2013.

References

1. Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. Depsky: dependable and secure storage in a cloud-of-clouds. *ACM Transactions on Storage (TOS)*, 9(4):12, 2013.
2. Bobrofon. EasySSHFS. Online, May 2017. Avaliable at https://play.google.com/store/apps/details?id=ru.nsu.bobrofon.easysshfs, Accessed September 05, 2017.
3. Brandon Butler. And the cloud provider with the best uptime in 2015 is.... *And the cloud provider with the best uptime in 2015 is*, 2016.
4. Maurice Gagnaire, Felipe Diaz, Camille Coti, Christophe Cerin, Kazuhiko Shiozaki, Yingjie Xu, Pierre Delort, Jean-Paul Smets, Jonathan Le Lous, Stephen Lubiarz, et al. Downtime statistics of current cloud solutions. *International Working Group on Cloud Computing Resiliency, Tech. Rep*, 2012.
5. Samuel Gibbs. Dropbox hack leads to leaking of 68m user passwords on the internet, Aug 2016.
6. Wenjin Hu, Tao Yang, and Jeanna N Matthews. The good, the bad and the ugly of consumer cloud storage. *ACM SIGOPS Operating Systems Review*, 44(3):110–115, 2010.
7. Iulia Ion, Niharika Sachdeva, Ponnurangam Kumaraguru, and Srdjan Čapkun. Home is safer than the cloud!: privacy concerns for consumer cloud storage. In *Proceedings of the Seventh Symposium on Usable Privacy and Security*, page 13. ACM, 2011.
8. Dmitry Khovratovich, Christian Rechberger, and Alexandra Savelieva. Bicliques for preimages: attacks on skein-512 and the sha-2 family. In *Fast Software Encryption*, pages 244–263. Springer, 2012.
9. Hugo Krawczyk. Secret sharing made short. In *Annual International Cryptology Conference*, pages 136–146. Springer, 1993.
10. Hugo Krawczyk, Ran Canetti, and Mihir Bellare. Hmac: Keyed-hashing for message authentication. 1997.
11. Shadi R Masadeh, Shadi Aljawarneh, Nedal Turab, and Aymen M Abuerrub. A comparison of data encryption algorithms with the proposed algorithm: Wireless security. In *Networked Computing and Advanced Information Management (NCM), 2010 Sixth International Conference on*, pages 341–345. IEEE, 2010.
12. Rich McCormick. Hack leaks hundreds of nude celebrity photos, Sep 2014.
13. Ricardo Mendes, Tiago Oliveira, Alysson Bessani, and Marcelo Pasin. C2fs: um sistema de ficheiros seguro e fiável para cloud-of-clouds. *INForum12, September*, 2012.
14. Justice Opara-Martins, Reza Sahandi, and Feng Tian. Critical review of vendor lock-in and its impact on adoption of cloud computing. In *Information Society (i-Society), 2014 International Conference on*, pages 92–97. IEEE, 2014.
15. INESC TEC. Safecloud photos. Online, May 2017. Avaliable at hhttp://www.safecloud-project.eu/consortium/inesctec, Accessed September 07, 2017.
16. Darryl L Willick, Derek L Eager, and Richard B Bunt. Disk cache replacement policies for network fileservers. In *Distributed Computing Systems, 1993., Proceedings the 13th International Conference on*, pages 2–11. IEEE, 1993.