Secure File Storage for Android Devices on Public Clouds

Paulo Ribeiro, Rui Prior, Sérgio Crisóstomo
Instituto de Telecomunicações & Departamento de Ciência de Computadores,
Faculdade de Ciências da Universidade do Porto
Email: up201103884@fc.up.pt, rcprior@fc.up.pt, slcrisos@fc.up.pt

***Abstract*—We propose MCOFS a secure cloud storage system for Android that combines the use of multiple public cloud providers with cryptography and redundancy mechanisms to ensure the confidentiality and availability of files even if one of the providers becomes hostile or suffers catastrophic data loss. Experimental results show that, while there is a performance penalty in comparison to the plain use of a single provider, it is small enough and a fair price to pay for the added guarantees.**

***Keywords*—cloud storage, confidentiality, availability, integrity, Android, filesystem.**

I. Introduction

There are few things we value more than our data. Be it confidential work documents, medical records, or simply vacation photos, we want our files available anytime, anywhere, without risking their loss, but ensuring their privacy.

Cloud storage has emerged as an increasingly popular means for keeping our files. Compared to local storage, it 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. However, it also brings some concerns. It is generally considered less secure [6], [5]. In 2012, two-thirds (68 million) of Dropbox user passwords were stolen by hackers and later made publicly available on the Internet [4]. In 2014, hundreds of private photos of celebrities were leaked from Apple iCloud [13]. Outages are another concern. All major Cloud Storage Providers (CSPs) have had outages of several hours [3], and the reliability of Cloud Storage Services (CSSs) is not as high as CSPs claim [10]. Yet another concern vendor lock-in. CSPs may, at any moment, raise the prices or start charging free users for space and/or bandwidth. In that case, the user may lose access to data or be forced to pay a perhaps significant fee to recover it [15].

This work describes the Multiple Cloud Overlay File System (MCOFS), an Android application that, using information partitioning and cryptographic techniques, allows users to keep their files in public 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.

II. Related Work

While there are several Android softwares for cloud storage, MCOFS is the only one ensuring 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, unlike any other solution.

We now give a general overview of some solutions that are somehow related to MCOFS. The EasySSHFS [2] Android application allows users to mount a remote FS using SFTP, which that most SSH servers support, but it is not truly a cloud storage solution and does not add redundancy. SafeCloud Photos [11] 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. It is focused on privacy and does not provide any redundancy mechanisms, and works only with image files.

In contrast with the Android application ecosystem, GNU/Linux already has some cloud storage based solutions fulfilling the confidentiality, integrity, and availability requirements, such as DepSky [1] and C2FS [14]. DepSky uses encryption, encoding, and replication of block level data over several commercial clouds, thus behaving like a virtual disk [1]. C2FS [14] is a system offering a POSIX compatible file system built on top of DepSky data storage service.

III. System Architecture

MCOFS runs on Android smartphones and gives 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 highlighting how its components interact. We have an FS implementation that intercepts and replaces the Android FS calls, and modules that help us achieve file availability, confidentiality, and integrity. The Integrity module is used to verify that an unautho-

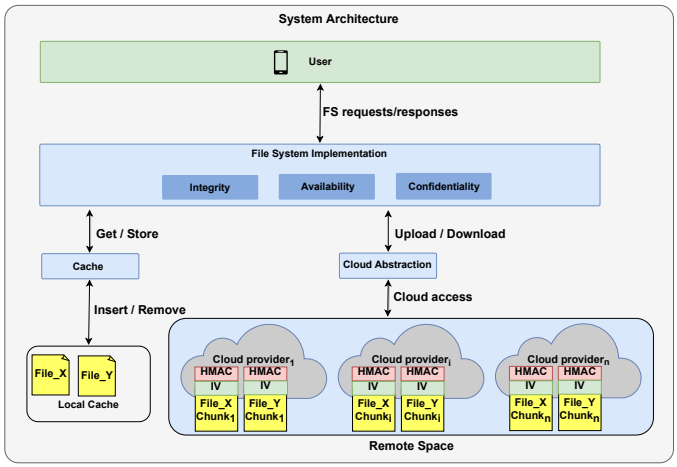

Figure 1. System Architecture, with a configuration of three CSPs

rized entity has not modified a file that we previously stored. The Availability module is responsible for giving the FS the ability to withstand the downtime of CSPs, 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. We will now describe these components in more detail.

A. Cloud Abstraction

Each CSP provides a specific Application Programming Interface (API) for file access. The Cloud Abstraction module defines an intermediate API that abstracts the differences. It provides all necessary functions (store file, get file, list files in folder, etc.) and maps them to CSP-specific calls so that file actions can be performed uniformly across the different CSPs.

B. Availability

The Availability module provides adds redundant information to the system and recovers lost data when necessary.

Different redundancy techniques provide varying degrees of fault tolerance and add varying amounts of storage and CPU overhead. Aiming at a good balance between these aspects, MCOFS uses simple parity. It is easy to implement and has a low impact on the system performance, as it is based on simple operations. The downside is that it only provides fault tolerance for one CSP at a time. However, since the probability of more than one becoming inaccessible (or corrupting files) simultaneously is very low, it is a good choice.

Before storing the files in the cloud services, 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 (XOR) of the other chunks.

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

C. Confidentiality

Files remotely stored in CSPs are protected by authentication, and some CSPs claim that the files are encrypted. However, we do not know who has access to the cryptographic keys, and the authentication mechanism may not be secure enough, as many examples unfortunately attest. Though some redundancy techniques also provide confidentiality [8], we used cryptographic techniques because we wanted a robust system able to maintain the confidentiality of all stored content even if the entire contents of a single CSP are leaked.

MCOFS uses symmetric cryptography, which is faster and more energy-efficient than public-key cryptography. Among the different alternatives [12], we selected Advanced Encryption Standard (AES) because it is widely accepted and considered secure. Moreover, some smartphone chipsets provide AES hardware acceleration.AES has three standard key sizes (128, 192 and 256 bits). Though it is about 40% slower than 128-bit, we used 256-bit version which is recommended by NIST.

AES has many operation modes. We used the 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.

The IV is used to add randomness at the start of the encryption process. If we had no IV (or always reused the same), using CBC with just a key, two files beginning with identical text would produce identical first blocks, differing only afterwards. This similarity could be explored to gain information on the plaintext or the key throughout cryptanalysis. Thus, using a different IV for different files is very important to keep the process secure.

Confidentiality is obtained by encrypting the files before storing them in the CSPs. The same key, generated from a user-provided password, is used for encrypting all files. The IV is randomly generated for each file, and stored in the header. It changes every time the file is updated.

D. Integrity

To ensure data integrity we resort to Hash-based Message Authentication Codes (HMACs). A HMAC function accepts, as input, a secret key and an arbitrary-length message, yielding, as output, a HMAC. Anyone knowing the secret key may use the HMAC value to verify the message integrity and authenticity [9].

MCOFS uses this method to protect data files: we compute a HMAC for file chunk (one per CSP) and store it in the chunk file header before sending it to the CSPs. After retrieving the file we recompute the HMAC value, and check if it

matches the HMAC stored in the file header. The key used for computing HMACs is the same for all files. It is generated from a user-provided password in the initial configuration of the system, similarly to the key used for confidentiality. Both keys are derived from the user password using OpenSSL EVP module functions.

The generation of HMAC values could be made using different hash functions. We used the Secure Hash Algorithm (SHA) 2 because it is a widely used National Institute of Standards and Technology of U.S. (NIST) standard, 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 [7]. 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.

E. 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 of utmost importance for the system to perform optimally.

We considered the *Write-through*, *Write-back*, and *Write-back on close* writing policies. The most appropriate writing policy for our system is *Write-back on close* (updates to the backing store will be made when a file is closed) because it has reduced communication overhead when compared with *write-through* and it is 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 failed writes to one CSP (or more), we we incorporated a journal/file version manager in MCOFS, and added a constraint to the cache replacement policy that a modified file can only be removed from cache after being successfully committed to the backing stores.

IV. System Implementation

MCOFS implementation is based on File System In User Space (FUSE), an interface for userspace programs to export a filesystem to the Linux kernel. Using FUSE, we intercept the

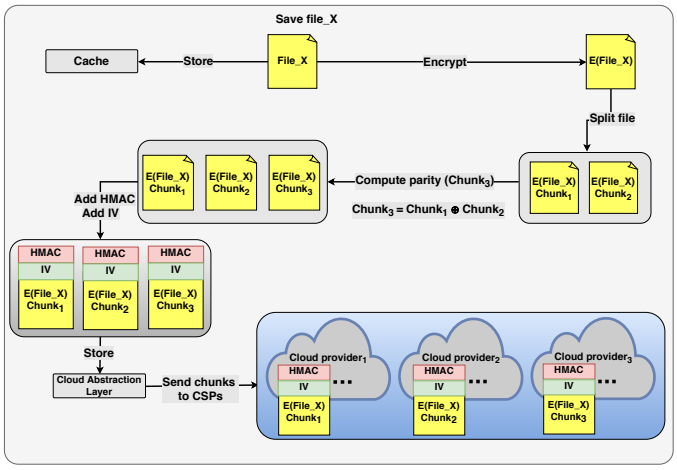

Figure 2. Process of storing a file, with a configuration of $N = 3$ clouds, where the redundant chunk is stored in cloud number 3

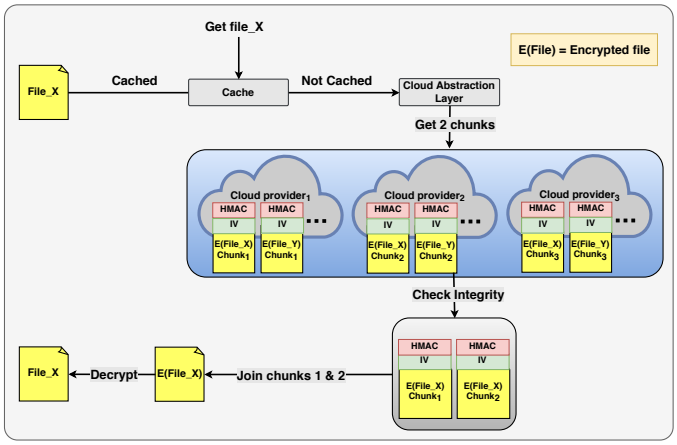

Figure 3. Process of getting a file, with a configuration of $N = 3$ clouds, where the redundant chunk is stored in CSP 3. All CSPs available

Android FS calls and translate them to CSPs calls 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 mapping of chunks to CSPs (notably including the account that holds the parity chunks) is fixed at configuration time. In a future version, we intend to improve this aspect with the addition of 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 modules interact for storing a new file. When an application creates a new file in the MCOFS folder, a file is created in the local cache. For now, file writes are performed locally. 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 it with the user-provided key and a randomly generated IV. The Availability module splits the file into $N-1$ chunks ($N = 3$ in

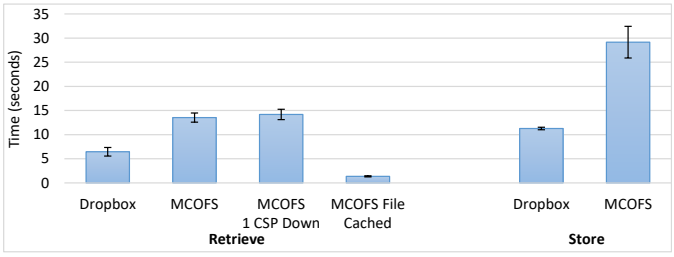

Figure 4. Time to retrieve and store a 10Mb file

the example) and computes the parity chunk. For each chunk, the Integrity module computes an HMAC, which is added to the file header along with the generated IV. Finally, the Cloud Abstraction Module stores the resultant file chunks across the N CSPs. If one of the used CSPs is unavailable, we store the other file chunks normally and notify the user. The file stays in cache until successfully written to all CSPs.

When reading a file, MCOFS first checks whether it is cached. If not, it downloads $N-1$ chunks, checking their integrity by comparing the computed HMAC values of the downloaded chunks with the HMACs stored in the file headers. It then merges the chunks (removing the headers first) to reconstruct the complete encrypted file. The file is then decrypted using the IV in the file header and the encryption key. The file is then written to the local cache, so that the subsequent reads will not require interaction with the CSPs. Figure 3 illustrates this process in a scenario where the file integrity has not been violated and all CSPs are available.

If one CSP is down or we detect that the integrity of a downloaded chunk was violated, we can recover it by downloading the parity chunk from the other CSP and use it jointly with the other ckunks to compute the missing chunk.

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

V. Performance evaluation

To assess the performance of MCOFS, we measured the time for storing and retrieving 10Mb files. Instead of different CSPs, we used three different Dropbox accounts, since support for each CSP must be coded in the Cloud Abstraction module. The results were compared to a conventional single cloud app (Dropbox Android). We tested different file retrieval scenarios: (1) no cache, all CSPs available; (2) no cache, one CSP down; and (3) with cache. We did 20 repetitions and averaged the results, which are shown in fig. 4. The wiskers show the 95% c.i. for the mean.

Storing a file with MCOFS takes twice as long as in Dropbox Android. This is reasonable and expected, since we need to perform much more computation and to store more data — due to parity chunks, MCOFS writes $\frac{N}{N-1}\times$ more than the file size, where N is the number of CSPs. Without caching, MCOFS takes about 2.6 longer than Dropbox Android to retrieve files, for similar reasons. With 1 CSP unavailable, this factor is raised by a minimal amount. As expected, caching immensely improves the performance of MCOFS.

VI. Conclusion

MCOFS allows Android to store folders in public CSPs and ensures data availability and confidenciality even if the CSPs become hostile. Though its performance is slower than the Dropbox Android app, it is so by a small factor. We believe this modest decrease in performance to be a small price to pay for the availability and confidentiality guarantees.

Since android is dropping FUSE (previously used to access the sdcard), we want to reimplement it using a different approach, and also add a locking mechanism to allow for concurrent access from multiple devices.

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] EasySSHFS. Online. https://github.com/bobrofon/easysshfs, Accessed September 17, 2019.

[3] Christophe Cérin et al. Downtime statistics of current cloud solutions, 2013. International Working Group on Cloud Computing Resiliency.

[4] Samuel Gibbs. Dropbox hack leads to leaking of 68M user passwords on the Internet. Online, August 2016. https://www.theguardian.com/technology/2016/aug/31/dropbox-hack-passwords-68m-data-breach, Accessed September 17, 2019.

[5] 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.

[6] 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.

[7] 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.

[8] Hugo Krawczyk. Secret sharing made short. In *Annual International Cryptology Conference*, pages 136–146. Springer, 1993.

[9] Hugo Krawczyk, Ran Canetti, and Mihir Bellare. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, February 1997.

[10] Liang Luo, Sa Meng, Xiwei Qiu, and Yuanshun Dai. Improving failure tolerance in large-scale cloud computing systems. *IEEE Transactions on Reliability*, 68(2):620–632, 2019.

[11] Francisco Maia. Data management and privacy in a world of data wealth. In *13th European Dependable Computing Conference (EDCC)*, pages 6–7, September 2017.

[12] 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.

[13] Rick McCormick. Hack leaks hundreds of nude celebrity photos. Online, November 2014. http://www.theverge.com/2014/9/1/6092089/nude-celebrity-hack, Accessed September 17, 2019.

[14] Ricardo Mendes, Tiago Oliveira, Alysson Bessani, and Marcelo Pasin. C2fs: um sistema de ficheiros seguro e fiável para cloud-of-clouds. *INForum12, September*, 2012.

[15] 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.

[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.