Final Secure File System

Deliverable 2.8

Project reference no. 653884

February 2018

European Commission

Horizon 2020
European Union funding
for Research & Innovation

Document information

Scheduled delivery	01.03.2018
Actual delivery	01.03.2018
Version	1.0
Responsible partner	INESC-ID

Dissemination level

Public

Revision history

Date	Editor	Status	Version	Changes
15.01.2018	M. Correia	Draft	0.0	Initial draft
28.01.2018	D. Matos	Draft	0.1	Chapter 4 added
31.01.2018	M. Correia	Draft	0.2	Full draft
26.02.2018	M. Correia	Draft	0.3	Incorporate reviews
28.02.2018	M. Pardal	Final	1.0	Final version

Contributors

M. Correia (INESC-ID)
M. Pardal (INESC-ID)
D. Matos (INESC-ID)
F. Apolinário (INESC-ID)
L. Rodrigues (INESC-ID)
S. D. Yalew (INESC-ID)

Internal reviewers

S. Totakura (TUM)
D. Burihabwa (UniNE)

Acknowledgments

This project is partially funded by the European Commission Horizon 2020 work programme under grant agreement no. 653884

More information

Additional information and public deliverables of SafeCloud can be found at:
`http://www.safecloud-project.eu/`

Contents

1 Executive Summary **1**

2 Secure file system design **3**
2.1 Features . . . 3
2.2 Architecture . . . 5
2.3 Functions . . . 5
2.4 Summary . . . 7

3 Homomorphic coordination service **8**
3.1 MorphicLib . . . 8
3.2 DepSpace . . . 12
3.3 HomomorphicSpace . . . 13
3.3.1 Threat model . . . 13
3.3.2 Commands . . . 14
3.3.3 Architecture and functioning . . . 15
3.4 Summary . . . 16

4 Integrity verification service **17**
4.1 SafeAudit . . . 18
4.1.1 Entities involved in SafeAudit . . . 19
4.1.2 Threat Model and Assumptions . . . 19
4.1.3 Preliminary Concepts . . . 20
4.1.4 SafeAudit Protocol . . . 22
4.1.5 SW Signature Size Reduction . . . 24
4.2 Implementation of SafeAudit . . . 25
4.2.1 Pairing Generator . . . 25
4.2.2 Key Generator . . . 26
4.2.3 Signature Generator . . . 26
4.2.4 Random Number Generator . . . 27
4.2.5 Proof Generator . . . 27
4.2.6 Proof Verifier . . . 27
4.3 Extending SafeCloud-FS with SafeAudit . . . 27
4.4 Experimental Evaluation . . . 28
4.4.1 Experimental Settings . . . 29
4.4.2 Bandwidth . . . 29
4.4.3 Monetary Costs . . . 30
4.4.4 User: Client-Side Overhead . . . 32
4.4.5 Auditor: Proof Verification Time . . . 34
4.4.6 Evaluation Outcomes . . . 35
4.5 Summary . . . 35

5 Client-side security **36**
5.1 Client Extensions for SafeCloud-FS . . . 36
5.1.1 Threat Model . . . 37
5.1.2 System Model . . . 37
5.1.3 System Architecture . . . 37
5.1.4 Client Architecture . . . 39

5.1.5 Log Architecture . . . 39
5.2 Securing the User's Device . . . 40
5.2.1 Securing the User's Credentials . . . 41
5.2.2 Securing Client's Local Cache . . . 42
5.3 Storage Recovery . . . 44
5.4 Implementation of the Client Extensions . . . 46
5.5 Experimental Evaluation . . . 46
5.5.1 Latency Overhead of Log Operations . . . 47
5.5.2 Storage Overhead of Log Operations . . . 48
5.5.3 Mean Time to Recover Files . . . 49
5.6 Summary . . . 51

6 Mobile client trust 52
6.1 Background . . . 53
6.1.1 ARM TrustZone . . . 53
6.1.2 Android Architecture . . . 53
6.2 DroidPosture Architecture and Design . . . 54
6.2.1 Threat Model and Assumptions . . . 54
6.2.2 Architecture . . . 55
6.2.3 Posture Reports . . . 56
6.2.4 Application Analysis . . . 57
6.2.5 Kernel Analysis . . . 59
6.3 DroidPosture Implementation . . . 60
6.3.1 Runtime Environment . . . 60
6.3.2 DroidPosture Modules . . . 61
6.4 Performance Evaluation . . . 62
6.4.1 Micro-benchmarks: mechanism performance . . . 62
6.4.2 Macro-benchmarks: DroidPosture in a company . . . 63
6.5 Security Evaluation . . . 64
6.6 Summary . . . 64

7 Conclusion 66

List of Figures

2.1 SafeCloud-FS architecture. 6
2.2 DepSky read protocol. 7
2.3 DepSky write protocol. 7

3.1 MorphicLib API (summary) 9
3.2 DepSpace architecture with 4 server replicas 12
3.3 HomomorphicSpace architecture 15

4.1 SafeAudit entities and their interaction. 19
4.2 SafeAudit components and entities. 25
4.3 Components modified and added when integrating DepSky client-side with SafeAudit's user code are shown in grey. 29
4.4 Bandwidth consumption comparison between requesting file integrity proofs using SafeAudit, SW, and RSA digital signatures. 30
4.5 Storage size for storing data and signature when using SafeAudit, SW, and RSA. 31
4.6 Time for the cloud to generate an integrity proof using SafeAudit. 31
4.7 Monthly costs in millidolars for 1 MB of data depending on the number of verifications per month. 32
4.8 Time for signing data using SafeAudit and RSA. 33
4.9 Time for SafeCloud-FS to upload a 1 MB file to the cloud with and without SafeAudit (with linear interpolation). 34
4.10 Time for verifying file integrity proofs in SafeAudit and with RSA. 34

5.1 SafeCloud-FS architecture with four different cloud storage providers and four replicas for the coordination service DepSpace. 38
5.2 User device with client-side components and external memory with one of the shares of the keystore. 40
5.3 Logging of operations in SafeCloud-FS 40
5.4 SafeCloud-FS operations, the read and write operations remain unaltered. The operations open and close decrypt and encrypt the local cached files to ensure data confidentiality. 43
5.5 Latency of using SafeCloud-FS with and without the log in both modes of the file system: (a) blocking and (b) non-blocking. 48
5.6 Required storage for the files and logs in the cloud storage services. 49
5.7 Mean time to recover files with 1, 10 and 100 versions. 50
5.8 Mean time to recover a file system compromised by a ransomware attack varying in the number of files and versions of each file. 51

6.1 Architecture of a mobile device running DroidPosture. The grey boxes are components of the DroidPosture service. The applications in the normal world may use the SafeCloud-FS client to accees the cloud. The external services are the SafeCloud-FS coordination service in this case. 55
6.2 DroidPosture providing a posture report to an external service, such as the SafeCloud-FS coordination service. 57
6.3 DroidPosture delay when called locally with emphasis on the applications analysis modules (in seconds). 62

List of Tables

3.1 MorphicLib's main classes 10
3.2 Fields that may be used in tuples in HomomorphicSpace, besides values and wildcards. 14

4.1 Prices for generating proofs and reading data (based on Amazon Ireland prices, standard storage). 32

5.1 Keys used in SafeCloud-FS with description, who generated them, and where they are stored. 38
5.2 Latency (in seconds) of Filebench micro-benchmarks for SafeCloud-FS with and without extensions. 49

6.1 Lines of code for the DroidPosture modules. 61
6.2 DroidPosture delay when called locally (in seconds). 62
6.3 DroidPosture delay when called by a remote service (sec.). 63

1 Executive Summary

File storage is one of the most successful use cases for cloud computing. Services like Dropbox, Google Drive, Amazon S3, Microsoft OneDrive, and Apple iCloud Drive are widely used worldwide to store both personal and professional files. However, there have been security issues with these services and they cannot stand against the strong adversaries that project SafeCloud is concerned with. Therefore, it makes sense to provide the security assurances of SafeCloud to files stored in cloud computing services.

Some of these cloud storage services provide a web interface, often a RESTful interface, but this is not the usual interface for file storage. In fact, usually users access files through the interface provided by the operating system (e.g., through a windows interface or a command line console) or through applications that process these files (e.g., a text editor, spreadsheet editor, etc.). Several of the cloud storage services, e.g., Dropbox and Google Drive, recognise that this integration with the operating system is the natural way of accessing files, so they provide client software that provides such integration. Specifically, from the point of view of the user, the cloud storage space appears as if it is yet another folder in his disk. This integration is possible by mimicking a (local) file system by providing a Portable Operating System Interface (POSIX) file system interface, which is the standard in most current operating systems (e.g., Linux, Mac OS X, and Windows).

The objective of WP2's Task T2.3 is to provide a *secure cloud-backed file system*, more exactly a file system that stores files in clouds and provides SafeCloud's high degree of protection from strong adversaries. This deliverable presents the design of this file system. The final design will be presented in deliverable D2.8.

The SafeCloud filesystem – *SafeCloud-FS* – provides a POSIX interface and stores files in a set of clouds – in a *cloud-of-clouds* – in such a way that the files *integrity* and *availability* are guaranteed even if some clouds are compromised. These clouds may provide the SafeCloud block storage abstraction. The file system stores both the files and their metadata (e.g., file name, modification date, and directory) encrypted for *confidentiality* and *privacy*. Keeping metadata encrypted is particularly challenging as this data must be accessed by the cloud (e.g., to return a file with a certain name), so the file system has to resort to homomorphic encryption to support some operations without decrypting the data [Gen09]. Moreover, the file system allows a mechanism to audit if the files are actually stored and not modified in the clouds without the need to download the files, which may be costly. This mechanism provides a second layer of *integrity* and *availability*. Finally, SafeCloud-FS provides a set of mechanisms for protection from client-side attacks, e.g., ransomware that encrypts the user credentials or temporarily gets access to these credentials and modifies some files at the cloud. Communication may be done over SafeCloud's middleware (WP1) for higher security also at that level.

This deliverable is organized as follows:

- Chapter 2 describes the overall design of the file system and how it provides the main security properties by leveraging the notion of cloud-of-clouds.

- Chapter 3 details the coordination service used in the file system – HomomorphicSpace – and the library in which it is based – MorphicLib.

- Chapter 4 presents SafeAudit, which is the service that allows verifying if files stored in the cloud have not been modified.
- Chapter 5 presents the main client-side protection mechanisms, which protect from ransomware and support file recovery.
- Chapter 6 presents client-side protection mechanisms for clients running in mobile devices.
- Chapter 7 concludes the deliverable.

SafeCloud-FS is available online, at the SafeCloud project website.[1]

[1] http://www.safecloud-project.eu/results/platform/ss3

2 Secure file system design

SafeCloud's secure file system – SafeCloud-FS – is based on some of the design principles of the Shared Cloud-backed File System (SCFS) [BMO+14b]. In fact, SafeCloud-FS can be considered to be an extension of SCFS with several new mechanisms and attributes.

SafeCloud-FS is a distributed, POSIX-compliant, distributed file system that guarantees data confidentiality, integrity, and availability. It allows users to store files in a cloud or a set of clouds (a *cloud-of-clouds*) with the usual consistency of a file system, atomic consistency or linearizability [HW90], even if weak consistency storage cloud services are used. This is important as public clouds normally provide only eventual consistency [Vog09].

To use the file system, users *mount* it on a folder of their computer or device, and the SafeCloud-FS client-side library synchronizes files with the cloud storage services. SCFS supports data sharing among several users, automatically propagating users' modifications between them.

In SafeCloud-FS files are stored on several clouds using the DepSky software library [BCQ+13]. DepSky provides an API for uploading and operating with a set clouds, while enforcing fault tolerance, lock-in resilience, confidentiality, and integrity as long as the clouds affected with the aforementioned problems do not reach the majority of the cloud set.

2.1 Features

The main features of SafeCloud-FS are the following:

- It stores every file in a set of clouds, forming a *cloud-of-clouds*;
- It provides a *POSIX interface*, so files are manipulated using the standard functions, e.g., open, read, write, chmod, mkdir, flush, fsync, link, rmdir, symlink, chown, etc.;
- Similarly to local file systems, each file has an owner, but may be *shared* with other users that may also read and modify it;
- It provides *controlled sharing*, in the sense that it provides access control mechanisms that allow controlling who can use each file;
- It provides a *pay-per-ownership cost model*, meaning that each user pays for the storage of his files;
- It *runs mostly at the client* and does not require a cloud storage gateway (CSG);
- It uses *unmodified storage clouds* for storing the files;
- It provides strong consistency by leveraging a *consistency anchor*, which in implemented using a *coordination service*;
- It is *modular* in the sense that the service is composed by a set of parts that work

together but can be exchanged by others with similar functionality – coordination service, verifier, storage clouds;

- It uses *caching* extensively in order to provide a performance as close as possible to a local file system and to reduce monetary costs;
- It provides *consistency-on-close semantics*, i.e., when a user closes a file, all updates he did become observable to the rest of the users, and it provides *locks* to avoid write-write conflicts;
- It allows doing *integrity verification* of the files stored in the individual clouds without downloading them.
- It protects the clients' credentials from ransomware and other attacks against their integrity and availability.
- It protects the clients' cache confidentiality by encrypting the files stored there.
- It allows recovering files illegally modified in clouds using an intrusion recovery scheme.

SafeCloud-FS provides the following security and dependability properties:

- *Availability* – files continue to be usable even if some clouds stop working (the other clouds are still there) or user credentials are compromised (they can be recovered);
- *Integrity* – files continue to be usable even if some clouds corrupt them (the files are still at the other clouds and can be recovered);
- *Disaster-tolerance* – files continue to be usable even if some clouds suffer disasters such as earthquakes and floods (files may be stored in clouds geographically far apart);
- *Confidentiality (from clouds)* – neither files not their metadata can be read by external intruders or malicious insiders (they are encrypted);
- *Confidentiality/integrity (from users)* – files cannot be read or modified by unauthorized users (there is access control) and they can be recovered in case users are impersonated (using an intrusion recovery scheme).

SafeCloud-FS essentially adds several mechanisms to SCFS:

- *Encryption of file metadata* – file metadata such as names, directories, and timestamps may be private. Although encrypting metadata may seem as a trivial extension of file encryption, this is not the case. In fact it involves using homomorphic encryption because metadata must be searched and it is impractical to download and decrypt all metadata before accessing files. This encryption is supported by an homomorphic encryption library (MorphicLib) and an homomorphic tuple space (HomomorphicSpace) that are presented in Chapter 3.
- *Integrity verification mechanism* – if files are deleted or corrupted in a cloud, either due to accidental or intentional reasons, the degree of redundancy becomes lower

and files become more vulnerable to other issues. SCFS allows doing this verification but it requires downloading the files and checking a signature, which is both slow and expensive (downloading files from a cloud has a cost). The integrity verification mechanism (SafeAudit) is presented in Chapter 4.

- *Client-side protections* – user cloud-cloud access credentials are protected, cached files are encrypted, and a new mechanism allows recovering files in case they are compromised, e.g., due to stolen credentials (Chapter 5). Moreover, a mechanism allows assessing the security status of a mobile client (Chapter 6).

2.2 Architecture

Figure 2.1 presents the overall architecture of SafeCloud-FS. This architecture has mainly three parts: storage clouds, computing clouds, and clients.

In SafeCloud-FS, files are envisaged to be stored in public *storage cloud* services, such as Windows Azure, Google Files, rackspace, and Amazon S3. These services are not modified, i.e., there is no SafeCloud-FS code running in that part of the system. Alternatively, any device that provides the *SafeCloud block storage abstraction* can be used. This part is shown in the bottom-right of the figure.

SafeCloud-FS needs some code to run in the cloud, so it also resorts to *computing cloud* services like Windows Azure or Amazon EC2 (top of the figure). SafeCloud-FS runs two components in those services. First, it permanently runs a coordination service called HomomorphicSpace replicated in several of these services in order to support locks, access control, and storing file metadata. Second, when a user requires file integrity verification, they run a verifier (auditor).

The rest of the logic of SafeCloud-FS is implemented at the clients: *FS Client* in the figure (left). The clients do mainly four tasks. First, they manage file caching, which is extremely important from the point of view of performance and cost. Second, they access the storage clouds to read and write files. Third, they access the coordination service for reading and writing file metadata, and to access the files in a controlled way (locks, access control). Four, they launch and access verifiers in the computing clouds to do integrity verification.

2.3 Functions

In this section we describe how five important POSIX functions are implemented in SafeCloud-FS: open, read, write, flush, and close.

Open Before a file is read or written, it must be opened using function *open*. This function involves three main steps:

1. Access the coordination service to read the file metadata;

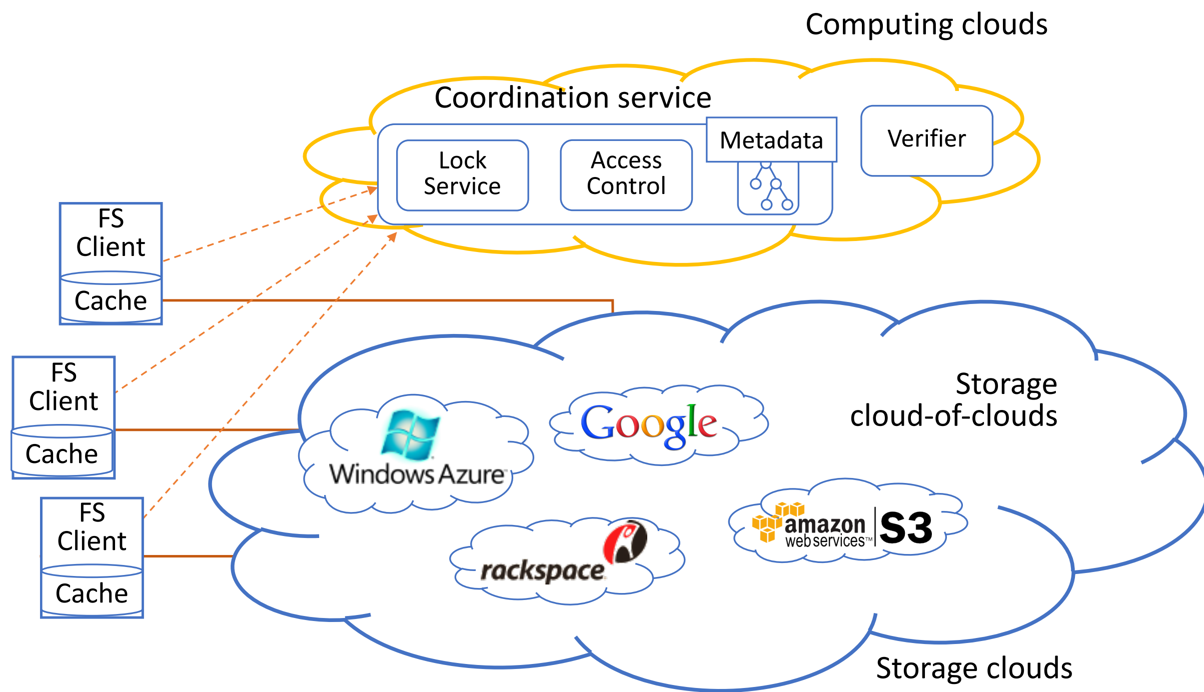


Figure 2.1: SafeCloud-FS architecture.

2. If the file is being opened for writing, access the coordination service to create a lock for the file and wait for the lock to be granted;

3. Access the storage cloud-of-clouds to read the file to the local cache.

The reading of the file in the last step is done using DepSky's read protocol (see Figure 2.2). In this protocol, the client accesses all storage clouds and gets the storage metadata of the file stored from a majority of them. Then it reads the file from one of the clouds that has the highest version of the file. If there is some problem with the file (e.g., the signature does not match the file or the cloud does not provide it), the file is read from another storage cloud.

Read and write As mentioned above, when the file is opened it is downloaded and stored in the local cache. Reads and writes are done in the version of the file stored locally, therefore they do not involve interactions with the computing clouds or the storage clouds.

A concern may be raised about the fact that writes done locally will not become visible to other users accessing the same file. However, this is not a problem, but a direct consequence of the consistency-on-close semantics provided by SafeCloud-FS.

Flush and close Flushing and closing a file involve pushing it from the local cache to the cloud. The main steps are:

1. Write the file to the storage cloud-of-clouds;

2. Access the coordination service to update its metadata (e.g., the version);

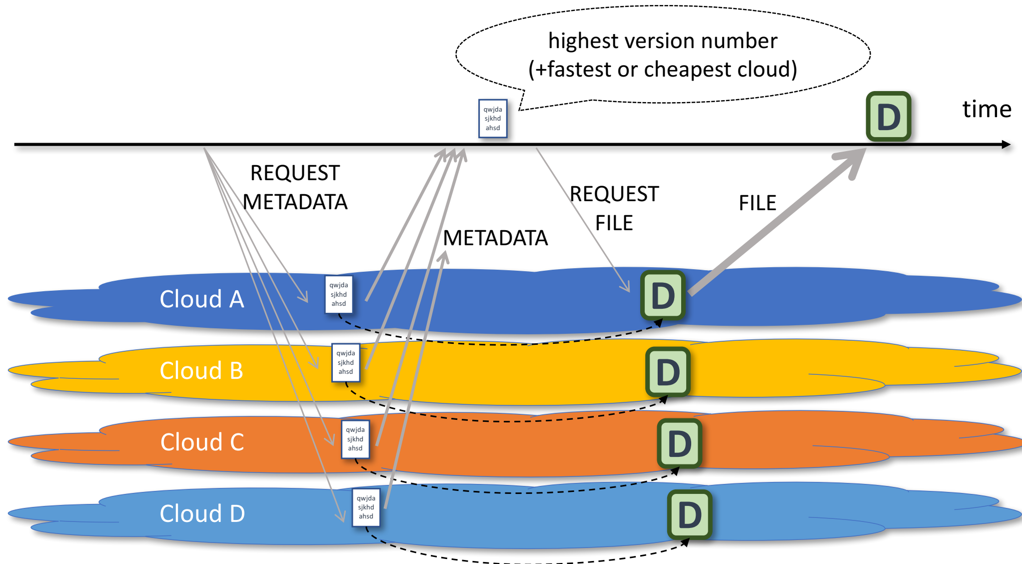


Figure 2.2: DepSky read protocol.

3. If the operation is *close*, access the coordination service to unlock the file.

The writing of the file to the cloud is done using DepSky's write protocol (see Figure 2.3). The client essentially uploads the file to all clouds, then writes the storage metadata.

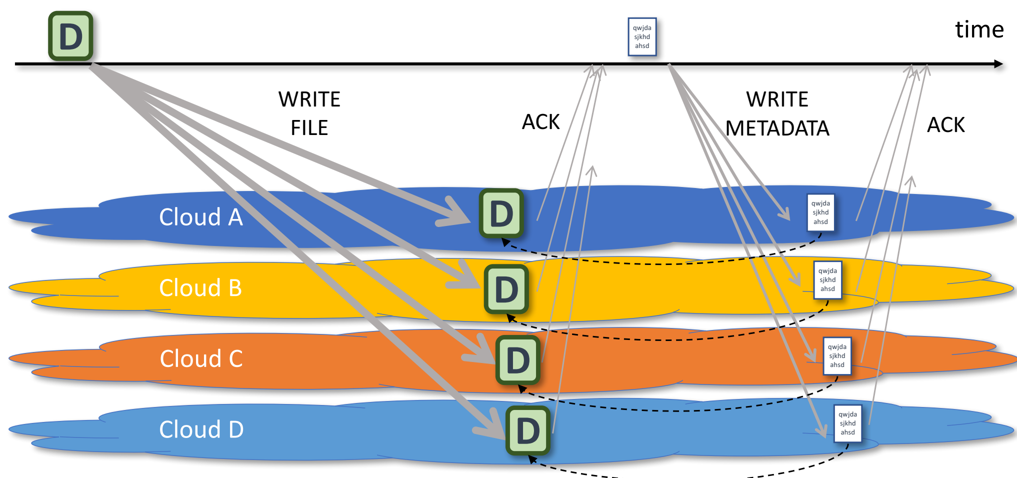


Figure 2.3: DepSky write protocol.

2.4 Summary

This chapter presented the overall design of SafeCloud-FS: its features, the properties it enforces, the architecture, and the operation of its main functions. All these functions are similar to SCFS'. The main difference is that metadata is stored encrypted in the coordination service, which is the topic of the following chapter. Later, Chapter 4 presents the integrity verification scheme.

3 Homomorphic coordination service

This chapter presents HomomorphicSpace, a coordination service that provides a *tuple space* abstraction [Gel85].

The HomomorphicSpace is based on a new library of homomorphic functions that we designed called MorphicLib, so we present it first (Section 3.1). The HomomorphicSpace is an extension of the DepSpace coordination service, so we introduce that system afterwards (Section 3.2). The rest of the chapter presents HomomorphicSpace itself.

3.1 MorphicLib

MorphicLib is a novel library of partial homomorphic cryptographic functions written in Java and providing a Java API. MorphicLib was not developed from scratch, but based on existing source code whenever possible. The objective was both to simplify the task and to avoid introducing bugs, which tend to appear due to the complexity of cryptographic code. This library can be used both at the client-side to encrypt and decrypt data, and at the server-side to do operations over encrypted data.

The code of the library is organized in classes, one per *homomorphic property*. One crucial difference between *partial homomorphic encryption* (PHE) and *fully homomorphic encryption* (FHE) is that in PHE data has to be encrypted taking into account the kind of operation that will be supported over the encrypted data. With FHE, on the contrary, arbitrary computation is possible over encrypted data (at a huge cost, in terms of performance). As we opted for PHE for efficiency (FHE is extremely slow), for each homomorphic operation we have four kinds of functions (or methods):

- Key generation function, typically used at client-side;
- Encryption function, typically used at client-side;
- Decryption function, typically used at client-side;
- Homomorphic operation functions, which allow doing operations over encrypted data, typically used at the server-side.

Next we explain the implementation of the functions for each homomorphic property. Information about the properties of the PHE algorithm, the operations supported, and the classes are in Table 3.1. Figure 3.1 shows a summary of the library API.

Random – Class HomoRand The cryptographic Random scheme is not homomorphic, but was included in the library for completeness. This scheme, is called Random because every time a given value is encrypted, it gives a different cyphertext. In fact, it is not an homomorphic encryption system, but can be used in a general homomorphic aware application precisely when no homomorphic property is required for certain data. In this case, Random is more secure than any of the homomorphic encryption schemes as it is not

```
1  public class HomoRand {
2      public static SecretKey generateKey()
3      public static byte[] encrypt(SecretKey key, byte[] IV, byte[] plaintext)
4      public static byte[] decrypt(SecretKey key, byte[] IV, byte[] ciphertext)
5  }
6  public class HomoDet {
7      public static SecretKey generateKey()
8      public static byte[] encrypt(SecretKey key, byte[] plaintext)
9      public static byte[] decrypt(SecretKey key, byte[] ciphertext)
10     public static boolean compare(byte[] op1, byte[] op2)
11         throws UnsupportedEncodingException
12 }
13 public class HomoOpeInt {
14     public static SecretKey generateKey()
15     public long encrypt(SecretKey key, int plaintext)
16     public int decrypt(SecretKey key, long ciphertext)
17 }
18 public class HomoSearch {
19     public static byte[] wordDigest(SecretKey key, String word)
20     public static SecretKey generateKey()
21     public static String encrypt(SecretKey key, String plaintext)
22     public static String decrypt(SecretKey key, String ciphertext)
23     public static boolean searchAll(String words, String ciphertext)
24 }
25 public class HomoAdd {
26     public static PaillierKey generateKey()
27     public static BigInteger encrypt(BigInteger m, PaillierKey pk)
28         throws Exception
29     public static BigInteger decrypt(BigInteger c, PaillierKey pk)
30     public static BigInteger sum(BigInteger a, BigInteger b, BigInteger
           nsquare)
31     public static BigInteger dif(BigInteger a, BigInteger b, BigInteger
           nsquare)
32     public static BigInteger mult(BigInteger a, int prod, BigInteger nsquare)
33
34 }
35 public class HomoMult {
36     public static KeyPair generateKey()
37     public static BigInteger encrypt(RSAKey key, BigInteger value)
38     public static BigInteger decrypt(RSAKey key, BigInteger ciphertext)
39     public static BigInteger multiply(BigInteger op1, BigInteger op2,
40         RSAPublicKey publicKey)
41 }
```

Figure 3.1: MorphicLib API (summary)

Table 3.1: MorphicLib's main classes

Property	Homomorphic Operations	Class	Input Data Types
Random	None (strong cryptanalisys resistance)	HomoRand	Strings, Byte Arrays
Deterministic	Equality an inequality comparisons	HomoDet	Strings, Byte Arrays
Searchable	Keyword search in text	HomoSearch	Strings
Order preserving	Less, greater, equality comparisons	HomoOpeInt	32 bit Integers
Sum	Add encrypted values	HomoAdd	BigInteger, String
Multiplication	Multiply encrypted values	HomoMult	BigInteger, String

vulnerable to a chosen plaintext attack [KL07].

For this scheme we have used the Advanced Encryption Standard (AES) implementation of the javax.crypto package with CBC mode and PKCS #5 padding. This algorithm is recommended for legacy and future use by ENISA [ENI14].

What gives this scheme the randomness property (same cleartext producing different ciphertexts) is the use of a random Initialization Vector (IV).

Deterministic – Class HomoDet In order to make possible equality comparison operations we need deterministic encryption, i.e., encryption in which the same plaintext originates always the same ciphertext. The deterministic scheme is essentially the same as the random encryption scheme, except that the IV takes a fixed value. In order to avoid that plaintexts with the same beginning have the same beginning on the correspondent ciphertext, we make a second encryption with the blocks in the reverse order, with the same IV. This form of encryption is weaker than the random scheme, but necessary for equality and inequality determinations [ENI14, PRZB11]. Needless to say, in this encryption system an attacker will be able to notice if two equal ciphertexts correspond to the same plaintext. Otherwise, this encryption scheme is as strong as AES encryption.

Searchable – Class HomoSearch The searchable scheme aims to produce a ciphertext that allows searching for words within it, without having to decrypt it. The trivial option would be to encrypt the text word by word with a deterministic encryption system. However, this approach would provide too much information to an attacker: frequency of words, position of the words in the text, and size of the words. To avoid those drawbacks we have built a scheme closely following the solution used in CryptDB [PRZB11]. The encryption for this scheme was implemented with the following sequence of steps:

1. It builds a list of distinct words found in the text (hides the frequency);
2. It encrypts each word with deterministic encryption;
3. It obtains a SHA 256 (also recommended by ENISA [ENI14]) hash of each encrypted word (hides the size of words);
4. It orders the obtained list randomly (hides the position in the text)
5. The text to be searched is encrypted with the random scheme and the list of hashes

is attached.

Searching for keywords in text consists in:

- The client encrypts and hashes the keyword(s) to be searched;
- The server searches for these hashes in the list and returns the encrypted text if there is a match.

To decrypt the text the list of hashes is not necessary.

Order Preserving – Class HomoOpeInt Order preserving encryption aims to allow comparisons of encrypted values such as *greater than*, *less than*, and *greater or equal to*. We implemented this scheme by supporting the encryption of 32-bit signed integers (Java's *int* primitive type). Encryption maps each value into a positive number in the range [0, MaxLong/2]. The algorithm implemented was the one described by Boldyreva et al. [BCLO09]. The implementation was based on CryptDB's C++ implementation obtained in GitHub [P^+15]. A challenge of the implementation was to find a reverse hypergeometric pseudo-random variate generator method, as CryptDB's code was too complex. Instead we used a Java implementation of the algorithm described in [KS85] available at GitHub [Der13].

Sum – Class HomoAdd As partial homomorphic scheme for the sum operation, we used the Paillier cryptosystem [KL07]. In order to be able to work with numbers as large as necessary, we decided to use as inputs big integers, namely Java's *BigInteger* class. For the implementation of Paillier we have adapted the Java code authored by Hassan found in the web [Has09].

The Paillier cryptosystem is an asymmetric scheme with the following two keys: public key – the pair (n, g); private key – the pair (λ, μ). The parameters n, g, λ, and μ are generated from two big prime numbers p and q. The parameter $n = p.q$, is part of the public key. So, the security of the system is based on the fact that an attacker cannot find p and q factorizing n. This is the same problem used by RSA, so the length of n, two times the length of p and q, should follow the recommendations for RSA, and have at least 2048 bits [ENI14].

This scheme also supports *multiplication* of encrypted values by constants. For that purpose, we raise the encrypted value to the constant (for a sufficiently large n):

$$Enc(a + b \, mod \, n) = Enc(a).Enc(b) \, mod \, n^2$$

$$Enc(k.m \, mod \, n) = Enc(m)^k \, mod \, n^2$$

Note that in PHE the operations performed with the encrypted data do not have to be the same that would be executed with plaintext. Those operations just need to produce the desired result, i.e., the result obtained must be the encryption of the result that would be obtained executing the original operation over the plaintext. This is the case with Paillier, in which to obtain the encryption of a sum, a product is made. The same way, the multiplication by a constant is determined by rising the encrypted value to that constant.

Multiplication – Class HomoMult For multiplication we used RSA, again with big integers. We used the standard Java functions in *javax.crypto* for encryption, decryption, and key generation. No padding is used to guarantee the homomorphic property.

We implemented encryption functions accepting inputs of the types *BigInteger* or *String* (containing an integer).

Two aspects should be noted:

1. In this way of using RSA both keys must be kept secret, otherwise chosen plaintext attacks would be possible;

2. The partial homomorphism for multiplication is valid for the modular multiplication. As the RSA keys have more than one thousand bits, that means that we can comfortably work with 32 bit integers or even 64 bit long integers. Actually we can work with BigIntegers of hundreds of bits provided that the multiplications do not exceed the value of the module used in the encryption.

3.2 DepSpace

DepSpace is a fault- and intrusion-tolerant *tuple space* service [BACF08]. Architecturally it is client-server system implemented in Java (see Figure 3.2). The server-side is replicated in order to tolerate arbitrary faults. The client-side is a library that can be called by applications that use the service. Clients communicate with the servers using a Byzantine fault-tolerant total order broadcast protocol called BFT-Smart. The most recent version supports extensions to the service [DBB+15]. A stable prototype is available online.[1]

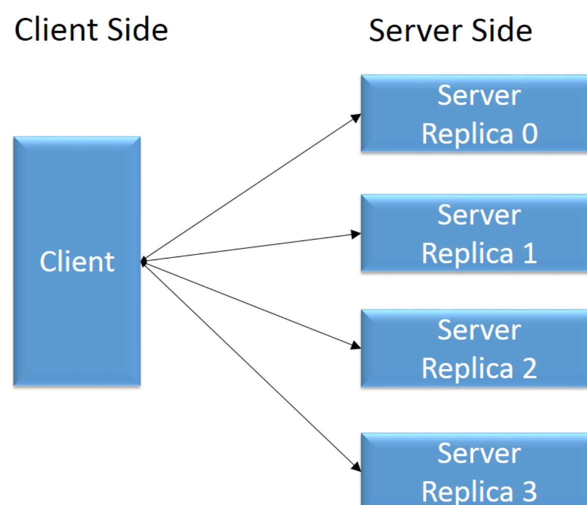


Figure 3.2: DepSpace architecture with 4 server replicas

The service provides the abstraction of tuple spaces. A tuple space can be understood as a shared memory that stores *tuples*, i.e., sequences of *fields* (data items) such as (1, 2, a).

[1] https://github.com/bft-smart/depspace

Tuples are accessed using *templates*. Templates are special tuples in which some fields have values and others have undefined values, e.g., wildcards meaning any value ("*"). A template *matches* any tuple of the space that has the same number of fields, in which the values in the same position are identical, and the undefined values match in some sense. For example, the template (1, *, a, *), matches the tuples (1, 2, a, b) and (1, 7, a, 14), but neither (1, 2, b, 4), where the 3rd field does not match, or (1, 2, a, b, 5), where the number of fields are different.

DepSpace supports a set of commands, issued by clients and executed by the servers. Here we consider the following commands:

- *out tuple* – inserts a tuple in the space;
- *inp template* – reads and removes from the space a tuple that matches the template;
- *rdp template* – reads but does not remove from the space a tuple that matches the template;
- *inAll template* – reads and removes from the space all tuples that match the template;
- *rdAll template* – reads but does not remove from the space all tuples that match the template.

DepSpace does not support homomorphic operations. However, it allows fields to be encrypted and basic equality matching by storing a hash jointly with the encrypted field. This solution however is vulnerable to trivial brute force and dictionary attacks. It does support the definition of access control policies using its policy-enforcement mechanism.

3.3 HomomorphicSpace

This section presents our homomorphic tuple space service.

3.3.1 Threat model

The threat model we consider for HomomorphicSpace is similar to the threat model for DepSpace except for one, crucial, difference: we consider that any server (or any cloud that contains the server) may be adversarial and try to read the content of the tuples it stores. We consider that all tuples whose fields' confidentiality has to be preserved are encrypted using homomorphic encryption, preventing malicious servers from doing such an attack.

Similarly to DepSpace, adversaries may compromise up to f out of $3f + 1$ servers and stop them or modify their behavior arbitrarily. This is tolerated using replication and the BFT-Smart protocol. Network messages may also be tampered with by the adversary, but the system tolerates this by using secure channels.

3.3.2 Commands

HomomorphicSpace extends DepSpace to allow commands over tuples with encrypted data items. More precisely in comparison with DepSpace, HomomorphicSpace:

- Supports the original match operations over encrypted data;
- Extends matching beyond the equality and wildcards with more complex matches, i.e., inequality, order comparisons (lower, greater), and keyword presence in a text, all over encrypted data;
- Allows addition and multiplication off encrypted fields.

Besides values and wildcards ("*"), HomomorphicSpace's *templates* can include the fields shown in Table 3.2.

Table 3.2: Fields that may be used in tuples in HomomorphicSpace, besides values and wildcards.

Field	Meaning
% $word_1 ... word_n$	matches a textual field containing all the words indicated
$>$ *val*	matches a numeric field containing a value greater than *val*
$>=$ *val*	matches a numeric field containing a value greater or equal to *val*
$<$ *val*	matches a numeric field containing a value lower than *val*
$<=$ *val*	matches a numeric field containing a value lower or equal to *val*

HomomorphicSpace adds three commands to those provided by DepSpace (Section 3.2): *crypt*, *rdSum* and *rdProd*.

The first is *crypt id template* and aims to define a *tuple encryption type*. The command takes as input an identifier (id) for the type it will create, and a template with the homomorphic operation desired for each of the fields, which will determine the homomorphic property. For example, if the template contains for a given field the operation "=", the system infers that the encryption to be used for that field is deterministic, which is the strongest that allows that operation. If no operation is indicated, the field will not be encrypted. The complete list of interpreted operations is:

- $=, <>$ – determinist encryption (notice that $<>$ means "different from")
- $>, >=, <, <=$ – order preserving encryption
- % - searchable encryption
- $+$ – Paillier
- & – RSA
- $.$ – random encryption
- other value – no encryption

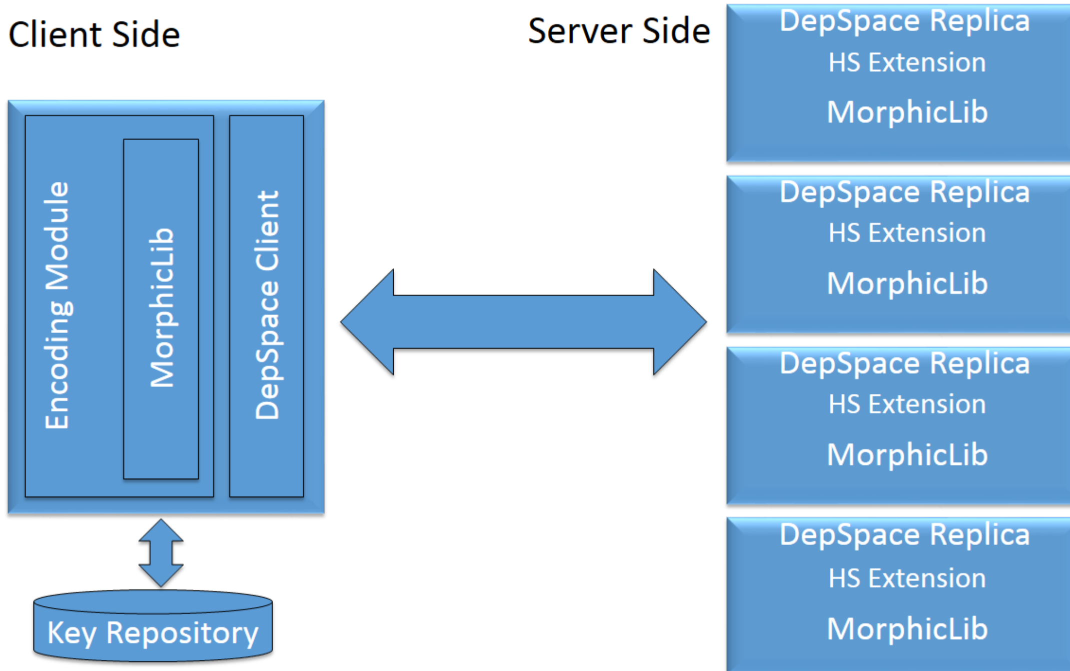


Figure 3.3: HomomorphicSpace architecture

The second command is *rdSum template*. This command starts by collecting all the tuples that match the template similarly to *rdAll*, then sums the (encrypted) fields with $+$ in the template. The function returns a single tuple with the result.

The third command is *rdProd template*, which works similarly to *rdSum* but does multiplications instead of sum.

This scheme allows a single type of encryption per field (unlike, e.g., CryptDB). However, with the tuple data structure this is not a restriction. For instance, for tuples with a single numeric field, two operations like equality and sum can be supported by transforming that field in two and using the tuple encryption type ($=$, $+$).

3.3.3 Architecture and functioning

Architecturaly the HomomorphicSpace is similar to DepSpace, with a client-side and a server-side. Figure 3.3 represents the system with 4 replicas, i.e., with $f = 1$. From the confidentiality point of view, the server-side is untrusted and the client-side trusted.

The server-side of the system is mostly DepSpace code with the server-side of the MorphicLib and with extensions to process the homomorphic operations. The client-side includes MorphicLib's and DepSpace's client-side libraries. The main functions of the client are to encrypt tuples and send them to the tuple space, and to decrypt them before they are delivered to the application. When a tuple is encrypted, the encryption keys are stored in a *key repository* (a folder with one file per key). Next we describe both sides in more detail.

Client side. When the *crypt* command is issued (i.e., that method is called), the library generates keys for every field of the tuple for which homomorphic properties are desired. These keys are stored jointly with the tuple encryption type (id and template) in the key repository.

All the other commands (*out*, *inp*, etc.) include an id that the library uses to retrieve the corresponding tuple encryption type and keys from the repository. If the operation indicated in a field is not compatible with the encryption defined with the *crypt* command, the command returns an error.

The library uses the DepSpace client library to send to the servers the command and the fields. If the command is an *out*, the fields are encrypted with the scheme defined in the tuple encryption type and the keys previously stored. If the command involves reading tuples it contains the operation and encrypted values. Note that each field of each id has its own key (or key pair for RSA), but the same field for the same id is always encrypted with the same key.

When the library receives a reply from the servers, it does the opposite, i.e., it decrypts the encrypted fields using the corresponding schemes and keys.

Server side. The server-side handles different commands in different ways. The *out* command is executed the same way as in DepSpace. The fields may be encrypted but they come encrypted from the client so the tuple is stored unmodified. The *inp* and *rdp* commands were modified using DepSpace's extension mechanism in order to support the $=$, $<>$, $>$, $>=$, $<$, $<=$, and text search operations over encrypted data, returning one of the matching tuples. The *rdall* and *inall* commands work similarly, as rdp and inp, but return all matching tuples. The *rdSum* and *rdProd* commands are implemented as a modification of the original *rdAll* command that returns a single tuple with the relevant fields respectively added or multiplied.

3.4 Summary

This chapter presented HomomorphicSpace, a coordination service capable of storing and processing encrypted data. The service can search encrypted data with matching operators like $=, <>, >, >=, <, <=$, find text based on keywords, and execute sums and multiplications. All those functions are performed without any decryption of the encrypted data.

4 Integrity verification service

This chapter presents SafeAudit, a software library that improves the Shacham-Waters (SW) integrity verification scheme [SW08] and adapts it for use with commercial clouds and SafeCloud-FS. SafeAudit improves the original SW scheme and provides: overall performance increase by carefully selecting *pairing-friendly elliptic curves* [BN05] for SW scheme parametrization; and a storage cost decrease of 50% in relation to the original scheme using *point compression* [Lyn07].

Nowadays data owners resort to integrity control mechanisms based on *cryptographic hashes* [Mer79, ErJ01] to protect their outsourced storage, often *digital signatures* for *collaborative storage* when data is shared among several cloud users, and MACs (Message Authentication Codes) for *private storage* when data is used by a single cloud user. To do so, users have some personal key: an asymmetric private/public key pair for digital signatures or a symmetric key for MACs. A user stores data together either with a signature or a MAC, computed respectively with the user's private key or symmetric key. Whenever the user wants to guarantee that the integrity of the data is preserved, the user must first download the data and the corresponding signature/MAC from the cloud, then verify if the data matches the signature/MAC. If they match, the integrity is verified and the user can rest assured.

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

In order to reduce delay and bandwidth consumption some works proposed more advanced integrity mechanisms [AKK09, WWRL10, WLL15, BJO09, WRLL10, dC14, SW08]. They are an evolution of the original mechanisms that are *homomorphic*, i.e., the integrity control structures they produce have the same structure as the signed data. These mechanisms provide *verifiability* (data integrity can be verified using proofs) and *unforgeability* (unauthorized modifications to proofs, data or control structures are always detected) without the need of downloading the data to be verified. These new mechanisms fall into two categories: homomorphic digital signatures, that provide *public verifiability* (anyone can perform the integrity verification); and homomorphic message authentication codes, which provide *private verifiability* (only the user that possesses the secret key can perform the integrity verification). To understand the benefits of these mechanisms consider the previous example of 1000 1GB files stored at AWS. If homomorphic digital signatures with 40-byte public keys are used, an user would have to download only 60 bytes from the cloud to verify the data integrity. Therefore, independently of the size of the data to be verified, integrity verification with these mechanism requires downloading a small proof, with the

[1] Considering that every 1GB read is charged approximately 0.09US$ [Amac].

associated low communication delay and negligible cost.

On the contrary of prior works that explore the potential of compact integrity proofs by presenting theoretical demonstrations of their feasibility and security analysis [AKK09, WWRL10, WLL15, BJO09, WRLL10, dC14, SW08], this work explores the practical applicability of these techniques for verifying data on commercial cloud storage. For that purpose, we present a service capable of being integrated on real world storage solutions, including commercial clouds and cloud-backed applications.

This paper presents SafeAudit, a software service that improves the Shacham-Waters (SW) integrity verification scheme [SW08] and adapts it for use with cloud storage. SafeAudit improves the original SW scheme and provides: an overall performance increase by carefully selecting *pairing-friendly elliptic curves* [BN05] for SW scheme parametrization; and a storage cost decrease of 50% in relation to the original scheme using *point compression* [Lyn07]. Moreover, it leverages the Function-as-a-Service (FaaS) or *serverless computing* model [FIMS17, HSH+16] to reduce cloud costs, by using computation resources in the cloud only when necessary. These improvements make SafeAudit the most cost-efficient homomorphic verification mechanism for use in commercial clouds.

SafeAudit was designed as a practical implementation that can be easily plugged into current commercial cloud services and cloud-backed applications. SafeAudit is simple to use, as using it does not require advanced cryptography knowledge. Our experimental evaluation has shown that using SafeAudit is 7.1% cheaper than using RSA signatures when the integrity of the data is verified monthly, and 34.9% when it is verified weekly in a typical setting in AWS.

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

4.1 SafeAudit

The goal of the SafeAudit software service is to assure users that all the data they store in the cloud is retrievable with its integrity preserved. This service is envisioned to be easily integrated with: current commercial storage clouds (such as AWS [Amad]), for providing integrity proofs on the stored data; and cloud-backed storage applications (such as [BCQ+11b, BMO+14b, PP13, SvDJO12, CJWH+15, ZYTT15b, PBM+17]), to generate all the necessary digital signatures and automate the request and verification of the integrity proofs supplied by the clouds.

SafeAudit leverages *homomorphic digital signatures* for integrity control of the stored data, and the computation resources of commercial clouds infrastructures for executing code and generate compact integrity proofs based on the data and signatures present in the cloud storage. Also, by requesting and verifying these small proofs, cloud-backed applications can perform storage integrity control without being constrained with network

bandwidth limitations or downloading large quantities of data.

4.1.1 Entities involved in SafeAudit

In SafeAudit there is interaction among three types of entities: *clouds*, *users* and *auditors* (Figure 4.1). All these entities need to run SafeAudit code at some point.

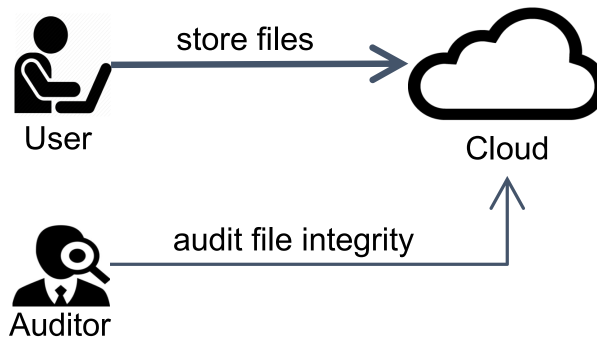


Figure 4.1: SafeAudit entities and their interaction.

Clouds are commercial public infrastructures that provide to their users data storage and code execution capabilities for providing integrity proofs.

Users are the normal commercial cloud users, who store data on the cloud and perform operations on the stored data (read, write, delete, or set access control permissions).

Auditors are entities trusted by the users for auditing the data stored in the cloud. They are responsible for issuing and verifying integrity proof requests to the cloud.

4.1.2 Threat Model and Assumptions

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

Since the purpose of SafeAudit is to detect cloud integrity attacks, this service is based on the assumption that the only way the attackers can compromise the users' data is by attacking the cloud. This assumption was made to isolate the threat model from problems related with network or identity spoofing attacks, which are outside of the scope of this

work. To do so, the threat model assumes that all communication between entities is authenticated and secure at all times (e.g., all entities communicate through HTTPS and use certificates signed by certificate authority trusted by all entities) and that neither the user nor the auditor suffer Byzantine faults, i.e. users and auditors are not malicious and their machine do not respond arbitrarily to requests from the other entities.

4.1.3 Preliminary Concepts

SafeAudit is built on top of *multiplicative cyclic groups* and uses pairing-based cryptographic techniques, namely Boneh–Lynn–Shacham (BLS) homomorphic digital signatures [Lyn07] and the Shacham-Waters (SW) integrity verification scheme [SW08]. This section provides mathematical background and summarizes the aforementioned cryptographic techniques.

4.1.3.1 Multiplicative Cyclic Group

A *cyclic group* is composed by members that are generated by a single *group generator* element g. In a *multiplicative cyclic group* G every member is generated by powering the generator g with integers belonging to Z (the set of all integers). Multiplicative cyclic groups can be finite of infinite. The infinite ones are generated by powering with unbounded integers from Z. The finite ones of order n are generated by powering g with a bounded set of integers belonging to Z that are modulo of p (also called group order p). For example, consider a multiplicative cyclic group of order $n = 6$ and generator $g = 2$. The multiplicative group is composed of six members $[g^0 = 1, g^1 = 2, g^2 = 4, g^3 = 8, g^4 = 16, g^5 = 32]$. Linear operations over members of the group are mapped as follows:

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

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

4.1.3.2 Pairing-based cryptography

SafeAudit leverages pairing-based cryptography to obtain homomorphism. In this type of cryptography, each cryptographic function uses a pairing e (also called bilinear map) to convert a multiplicative cyclic group (G) of prime order p, generated with the number g, into another multiplicative cyclic group (G_T) of the same prime order (p), i.e., $e : G \times G \to G_T$. The pairing enforces the following properties: *computabillity* – there exists an efficient algorithm to compute the pairing; *bilinearity* – for all u, v belonging to G, a, b belonging to Z_p and pairing $e : G \times G \to G_T$, it is guaranteed that $e(u^a, v^b) = e(u, v)^{ab}$.

4.1.3.3 BLS Signature Scheme

In order to provide integrity control of a file, SafeAudit uses the *BLS signature scheme* [BB04] for constructing digital signatures over pairing-based cryptography. To do so, integrity control takes the following steps:

- *Setup:* Choose two distinct multiplicative cyclic groups G and G_T of order p, and a generator g for G and generate pairing $e : G \times G \rightarrow G_T$.
- *Key Generation:* Using e and g compute an asymmetric secret/public key pair $sk \in Z_p$ and $pk \in G$. First compute sk, by selecting a random number that belongs to Z_p and then generate pk as g^{sk}.
- *Signature:* Sign the data $d \in Z_p$ using the secret key sk belonging to Z_p and by computing the signature $\theta = d^{sk}$ belonging to G.
- *Verification:* Using the public key $pk \in G$, the pairing e and the generator g, verify the signature $\theta \in G$ of the data $d \in Z_p$ by testing the following hypothesis: $e(\theta, g) = e(d, pk)$. If the hypothesis is verified, the integrity is assured.

4.1.3.4 SW Scheme for Homomorphic Verifiable Integrity Proofs

The use of BLS signatures ensures the homomorphic property for integrity verification and consequently allows the construction of homomorphic verification schemes, where data and signatures are aggregated using additions and multiplications into compact verifiable proofs. This is done because if each file and signatures can be divided into blocks of a given size (e.g., 128 bits) and these blocks can be mapped into multiplicative cyclic groups with *order* $=$ *size* (e.g., 128 bits will generate group $0...128$), multiplications and additions will always produce elements of the same order. Thus, files and signatures of unbounded size can be aggregated into compact structures of the multiplicative cyclic group (e.g., a file with 10^6 bits is divided into 128 bits blocks mapped to multiplicative cyclic group and multiplied each block, and therefore producing 128 bit aggregation structure that represents the 10^6 bits file). In SafeAudit, the SW integrity verification scheme [SW08] is used in order to provide homomorphic generation and verification of compact integrity proofs. To do so, under this scheme, integrity control takes the following steps:

- *Setup:* Choose two distinct multiplicative cyclic groups G and G_T of order p, and a generator g for G and generate the pairing $e : G \times G \rightarrow G_T$.
- *Key Generation:* Using e and g, compute: a signature parameter w, by selecting a random number that belongs to G; and an asymmetric secret/public key pair $sk \in Z_p$ and $pk \in G$. First, compute sk by selecting a random number that belongs to Z_p and then generate pk as g^{sk}.
- *Block Signature:* Given a block with the identifier $id \in Z$ and the data corresponding to the block $d_{id} \in Z_p$, an hash function that maps $H : Z \rightarrow Z_p$, the secret key $sk \in Z_p$, and the signature parameter w, compute the signature $\theta_{id} = (H(id) \times w^d)^{sk} \in G$.

- *Proof Generation:* Given a collection of block identifiers $id_1...id_n \in Z$, the corresponding data $d_1...d_n \in Z_p$ and numerical challenge vector of random numbers $chal_1...chal_n \in Z_p$, the hash function that maps $H : Z \to Z_p$, and the signature parameter w, compute the integrity proof:
  $\alpha = \sum_{i=1}^{n} d_i \times chal_i \in Z_p$ and $\beta = \prod_{i=1}^{n} \theta_i^{chal_i} \in G$.

- *Proof Verification:* given the proof (α and β), the identifiers $i...n$, the public key $pk \in G$, the signature $\theta \in G$, the pairing e, the generator g, and the signature parameter w, by applying pairing verify that: $e(\beta, g) = e(\prod_{i=1}^{n} H(id_i) \times w^{\alpha}, pk)$. If the verification is positive, integrity is assured.

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

There are significant challenges in adopting the SW scheme for verifying the integrity of clouds that SafeAudit mitigates. Namely: the SW scheme requires coordination between the user and the cloud for selecting the several underlying parameters used in the scheme (addressed in Section 4.1.4 by the SafeAudit protocol); and the SW scheme increases the cloud data storage requirements (addressed in Section 4.1.5 by pairing parameter selection in SafeAudit).

4.1.4 SafeAudit Protocol

In order to preserve the integrity of the data stored on the cloud, the entities involved – cloud, user and auditor – need to follow the SafeAudit protocol, described herein. The protocol is divided into four tasks: setup (Section 4.1.4.1), store data (Section 4.1.4.2), request and verify integrity proof (Section 4.1.4.3), and generate integrity proof (Section 4.1.4.4).

4.1.4.1 Setup

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

- The user and the auditor exchange data. The auditor provides two files[2] to the user for setting-up pairing-based cryptography: the '.param' file with all the secure public initialization parameters needed for configuring cyclic groups G, G_T and the pairing for mapping $G \times G \to G_T$; and the '.g' file with generator g of the cyclic group G. The user provides configuration information to the auditor about the time when each audit should be performed (e.g., daily, weekly), and other settings.

[2] *Data structures* would be a more rigorous term than *files*, but the word *file* is easier to understand as

- The user generates his secret/public asymmetric key pair and the signature parameter (w) for signing and verifying data under the SW scheme, using respectively the *key* and *random number generators* (further explained in Section 4.2).
- The user shares the public key and w with auditor and stores w on the cloud.
- The user configures the cloud for listening to requests from the auditor requests and for responding to them, with the execution of the *proof generator service* (further explained in Section 4.2).

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

4.1.4.2 Store Data

When the user stores data in the cloud, all data must be divided into blocks belonging to Z_p and signed. The *signature generator* (further explained in Section 4.2) automates these tasks and produces a signature equivalent to the SW Block Signature step (described in Section 4.1.3.4). To do so, the client provides as input, for the signature generator, the data and its identifier (e.g., the file content of the 'data.txt' file is used as the data and the identifier is the filename 'data.txt'), alongside with the pairing cryptography parameters ('.param' and '.g' files), secret key ('.sk'), and the signature parameter ('.w'), and obtains the signature of all the data blocks.

After the signature of the data is obtained, the user stores both the data and signature in the cloud. Data can now be verified.

4.1.4.3 Request and Verify Integrity Proof

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

- Select a file composed of x data elements (vector $[0, ..., x-1]$).
- Generate a random challenge (number belonging to Z_p) for each of the x data elements chosen, using the *random number generator*.
- Issue the integrity proof request to the cloud specifying the identifiers vector ($[id_0, ..., id_x]$) and the corresponding challenge vector ($[chal_0, ..., chal_x]$).
- Upon receiving a response from the cloud with the requested integrity proof, the auditor verifies it using the *proof verifier* (further explained in Section 4.2). The auditor provides the public key pk and the signature parameter w, alongside with the identifiers and challenges used on the integrity request; and obtains the integrity verification result. This step corresponds to the Proof Verification step of the SW scheme (described in Section 4.1.3.4).

they are indeed files in our implementation.

4.1.4.4 Generate Integrity Proof

Whenever the cloud receives an integrity proof request of a given file, it performs the following steps:

1. Fetch all the data and signatures of the file from the storage cloud corresponding to the identifiers specified.
2. Fetch from the storage cloud, the pairing cryptography parameters ('.param' and '.g'), and the signature parameter ('.w'), of the user requested.
3. Generate integrity proof, composed of: the aggregation of signatures provided (β); and the aggregation of data provided (α), by using the *proof generator* (further explained in 4.2.5). The generator receives data, setup parameters ('.g' and '.param'), signatures, challenges, pairing cryptography parameters and the random initialization parameter related to the file; and produces the α and β. This step corresponds to the proof generation step of the SW scheme.
4. Respond to the requester with the integrity proof (α and β).

4.1.5 SW Signature Size Reduction

The size of the block signatures produced by SafeAudit is equal to the size of the multiplicative cyclic group G stipulated by the auditor (e.g., if G is equal to 128 bits then the block signatures are also 128 bits). Also, the size of the groups G are determined by the elliptic curve selected for its initialization and are always larger than the integers used for its generation Z_p. For example, when a type A elliptic curve [Lyn07] is used for the generation of multiplicative cyclic groups, with the recommended sizes where G and G_T are 128 bytes and Z_p is 20 bytes, the signatures produced are *6.4 times bigger* than the original file, raising the storage cost in that proportion. This large overhead would make the technique too costly for practical use in commercial clouds. SafeAudit proposes two improvements to the original SW scheme to address this issue.

The first is the selection of the pairing curve that produces the shortest multiplicative cyclic groups, which is the pairing-friendly elliptic curves of prime order [BN05] (also named type F curves and described in [BB04]), as recommended by both BLS and SW authors in [SW08] and [Lyn07]. This optimization allows the creation of multiplicative cyclic groups G that are *2 times the size* of the original data Z_p, producing signatures with twice the size of data.

The second improvement is to incorporate a signature compression scheme in SafeAudit using the *point compression* technique described in [Lyn07]. This improvement comes from the fact that the multiplicative cyclic group G, where the signature belongs, is a two coordinate point (x, y) where y is one of the possible results of applying the elliptic curve function selected for pairing initialization. Due to this fact, the y coordinate of the signature can be computed solely based on the x coordinate, the elliptic function, and a one bit value indicating which of the possible values to select. Thus, the y coordinate can be completely discarded, and the signature is compressed always by half of the original size and

represented by its x coordinate and the one bit value necessary to recompute the y coordinate. This improvement allows signatures to have half of the expected size of applying the signature step of SW scheme and in the best case where type F elliptic curves [BN05] are used are the *same size* of the original data.

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

4.2 Implementation of SafeAudit

The SAFEAUDIT service is composed of several components, each one implementing a task of the SAFEAUDIT protocol, represented in Figure 4.2. This separation in components simplifies the integration with cloud-backed applications, commercial clouds, and auditors. SAFEAUDIT was developed in Java, so each component is essentially a Java class. The pairing-based cryptographic mechanisms were implemented using the Java Pairing-Based Cryptography Library (JPBC) [DI11], which implements multi-linear maps and the operations that manipulate them.

Auditors use the *Pairing Generator* component to generate the setup parameters for pairing-based cryptography. Users utilize the *Key Generator* component to generate their asymmetric secret/public key pair and signature parameter (w). Users resort to the *Signature Generator* component to sign their data. Both these entities use the *Random Generator* component to generate random numbers belonging to any field of their choosing (Z_p, G or G_T). Clouds run the *Proof Generator* component to generate integrity proofs. Auditors use the *Proof Verifier* component to verify the proofs obtained from the cloud. The rest of the section explains each component in detail.

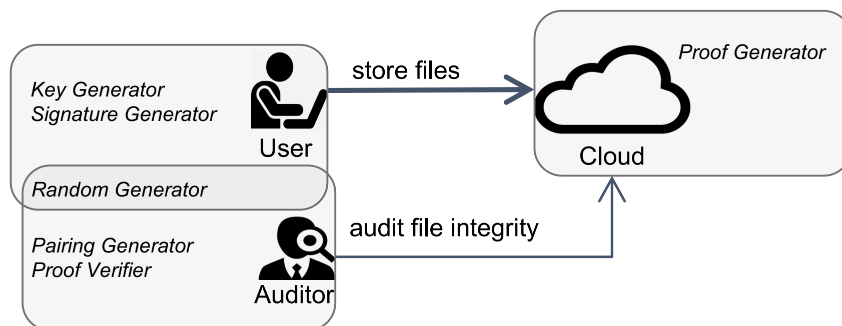


Figure 4.2: SAFEAUDIT components and entities.

4.2.1 Pairing Generator

This component allows auditors to construct setup parameters ('.param' and '.g') for initializing pairing-based cryptography, according to the their security specification.

Auditors provide as input the type of pairing curve[3] to be used for pairing generation

($type = A|B|C|D|E|F$), and the parameters needed for initializing the curves.

The *Pairing Generator* outputs: a specifier file ('.param') detailing all the information about the multiplicative cyclic groups G and G_T, the integer range of the Z integers used for generating elements, and the pairing specifications for mapping G to G_T; and the generator file '.g' containing the absolute value of the element used for generating the multiplicative group G.

4.2.2 Key Generator

The *Key Generator* component allows users to generate their own asymmetric key pair and signature parameter according to the security information provided by the auditor. The generated keys are used for the BLS and SW schemes.

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

4.2.3 Signature Generator

The *Signature Generator* component allows clients to sign their data using the SW scheme and compute the digital signatures.

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

4.2.3.0.1 Sign Block The Sign Block mode works as follows: the user inputs the setup parameters provided by the auditor '.param' and '.g', the block d, the identifier of the block id_d, the secret key sk and the signature parameter w; the component initializes pairing; hashes the identifier to Z_p; multiplies the id's hash with w^d; signs the multiplication with the user's secret key; and returns the signature of the block.

4.2.3.0.2 Sign Data The Sign Data mode works as follows: the user inputs the setup parameters provided by the auditor '.param' and '.g', the data d, identifier of the data id_d, the secret key sk and the signature parameter w; the component initializes the pairing;

[3] See Section 4 of [Lyn07] for more information about the pairing curves and their selection.

the component divides the data into a vector of blocks that belong to Z_p; signs each block individually with an unique id; concatenates the blocks' signatures into one; and returns the concatenation.

4.2.4 Random Number Generator

This component allows generation of random numbers belonging to any of Z_p, G or G_T fields. To do so, this generator receives as inputs the desired field, the pairing '.param' and the '.g' and outputs the random number.

4.2.5 Proof Generator

The *Proof Generator* component is the only one that is executed in the cloud (cf. Figure 4.2). It allows clouds to generate integrity proofs with the files they have stored whenever an auditor requests them. To do so, the algorithm first initializes pairing with the setup parameters, then calculates α and β based on the data's blocks present in the file.

To simplify deployment and to reduce the cost of running the *Proof Generator* in a cloud, we leverage recent services that implement the FaaS, serverless computing, or lambda model [FIMS17, HSH$^+$16]. The alternative would be to have a virtual machine for this purpose in a cloud compute service (e.g., Amazon EC2), but it would be costly to run it permanently in the cloud, or to store an image there to run it when necessary.

The FaaS model allows the execution of a code component (a function) in a cloud upon a certain event, in our case, the reception of a request through a REST API. In this model, the users pay only for the time and resources used when the function is executed, not when it is idle. Therefore, it is possible to have the *Proof Generator* component always ready to run in the cloud without costs when it is not running.

4.2.6 Proof Verifier

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

4.3 Extending SafeCloud-FS with SafeAudit

The SafeAudit components described in Section 4.2 were integrated with SafeCloud-FS.

In the integration we have to consider the three SafeAudit entities:

- The *user* code was integrated with the client-side code of SafeCloud-FS;
- The *auditor* code is a standalone Java program;
- The *cloud* code runs in a FaaS service such as Amazon Lambda [Amaa].

Next we focus mostly on the first as it is the one truly integrated with SafeCloud-FS code.

The user components of SafeAudit were integrated in DepSky's component responsible for uploading data into the cloud. This component receives the data from SafeCloud-FS, applies mechanisms that ensure confidentiality, integrity, and availability, then stores the resulting data in the cloud. The logic for communicating with different commercial clouds is implemented in subcomponents called *cloud drivers*. Since the integration of SafeAudit should not compromise any of the aforementioned properties, integrating both the systems required code changes to DepSky, in a contained way. The approach followed was the addition of a new type of cloud driver: the *auditable cloud driver*. With these newly introduced cloud drivers, besides accessing and uploading data to the cloud, data is signed using the SafeAudit's signature generator and the signature is also stored on the cloud. As seen in Figure 4.3, for integrating these new drivers, DepSky suffered changes in two packages: core and drivers. Code was added to the core package of DepSky, in the DepSky initialization function (in *LocalDepSkySClient.java*) and to the DepSky driver constructor function (in *DriversFactory.java*).

For using SafeAudit, SafeCloud-FS has to be configured with these *auditable cloud drivers*, which implement our system's logic. For instance, to use Amazon S3 as cloud storage, instead of using the original (non-auditable) driver *amazon-s3*, the corresponding auditable driver *auditable-amazon-s3* was used. Users can choose which drivers to use, by modifying the configuration file with the name of the desired drivers. The DepSky initialization function automatically reads the user's secret key, the setup parameters (.param and .g) and the signature parameters (.w) provided by the auditor; and uses the initialization function of DepSky driver for initializing the driver with that information. Regarding the driver package, the auditable drivers extend the non-auditable drivers. Whenever data is uploaded to the commercial cloud using the auditable driver, data is signed by using SafeAudit's sign data component and then stored both signature and data on the commercial cloud by invoking the superclass' non-auditable driver upload data function.

4.4 Experimental Evaluation

The cloud used during the implementation was AWS [Amad]: S3 [Amab] was used as storage and Lambda for executing the Proof Verifier.

With the experiments we wanted to answer the following questions: (1) What is the gain in terms of bandwidth consumption in using SafeAudit instead of RSA (Section 4.4.2)? (2) What are the monetary costs of using SafeAudit (Section 4.4.3)? (3) What is the performance overhead observed by the user when writing files (Section 4.4.4)? (4) How long does it take to verify the integrity of a file (Section 4.4.5)?

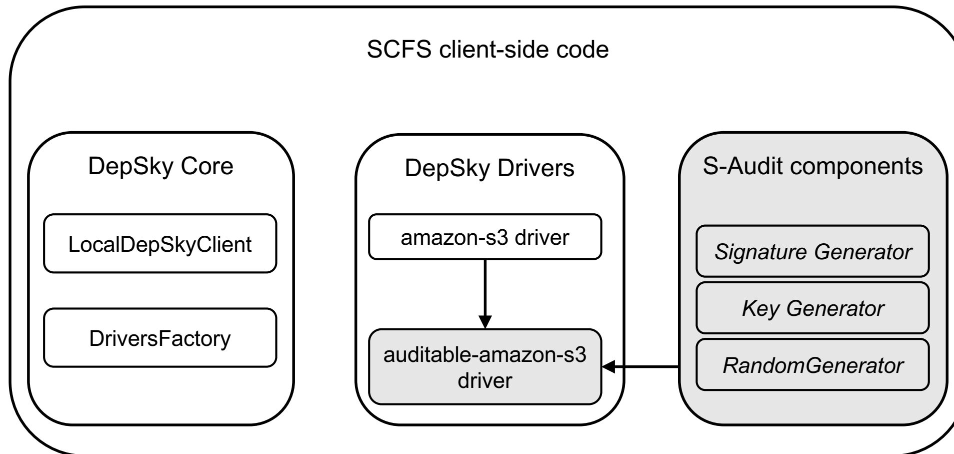


Figure 4.3: Components modified and added when integrating DepSky client-side with SafeAudit's user code are shown in grey.

4.4.1 Experimental Settings

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

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

Some of the experiments involved the execution of two schemes to serve as baseline: the original SW integrity verification scheme; and the RSA digital signature scheme. Both SafeAudit and the original SW scheme were parameterized with type F pairing curves, where G had 40 bytes, G_T 80 bytes, and Z_p 20 Bytes, with SHA-1 as hash algorithm, and asymmetric keys used with a 20 byte secret key and a 80 byte public key. For RSA, 1024 bit keys and SHA-1 were used.

4.4.2 Bandwidth

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

Figure 4.4 shows the results for SafeAudit and compares them with the original SW scheme (that provides identical results) and with the use of RSA digital signatures (that retrieves all data to be verified). The results show that as the storage size grows, SafeAudit and the SW scheme are able to maintain constant bandwidth consumption. Also, since proofs are composed of an aggregation of blocks belonging to Z_p (20 bytes) and an aggregation of blocks belonging to G (40 bytes), the bandwidth consumption is always equal to the sum of these group's sizes and that it is always low (the cost for reading 60 bytes is negligible). On the contrary, for RSA the bandwidth grows linearly with the size of the files.

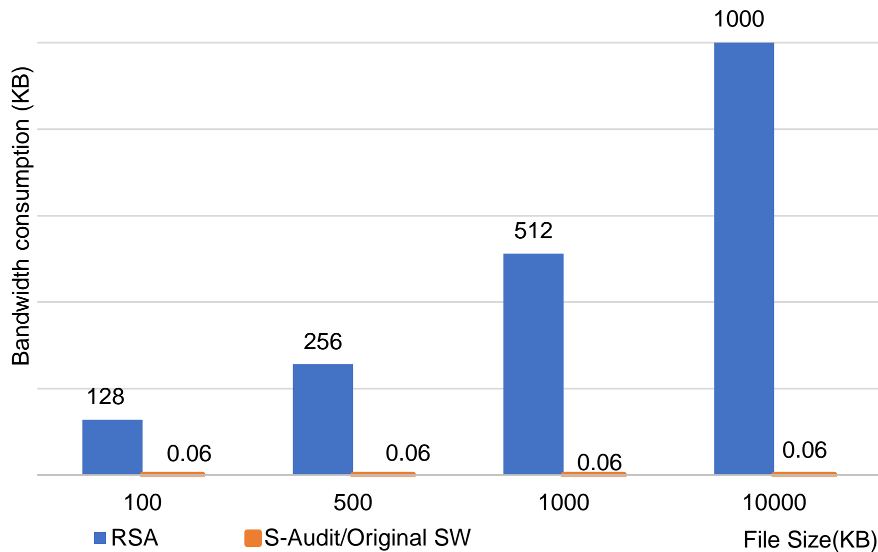


Figure 4.4: Bandwidth consumption comparison between requesting file integrity proofs using SafeAudit, SW, and RSA digital signatures.

4.4.3 Monetary Costs

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

4.4.3.0.1 Storage Costs Using SafeAudit for verifying data integrity on the cloud storage requires users to store on the cloud the data's digital signatures, which implies additional monetary storage costs.

Figure 4.5 compares the storage size (file plus signature(s)) as data size grows when using SafeAudit, SW, and RSA signatures. As seen in the figure, storing SW signatures increases the storage size by 200%, but SafeAudit manages to reduce this overhead to 100% with the signature reduction scheme of Section 4.1.5. This reduction has great positive impact on storage monetary costs, but still requires twice the storage than the ideal case where signature sizes are negligible (the RSA case). The actual monetary costs tend to be proportional to the amount of data stored in most commercial clouds. For example, in Amazon S3 Ireland this cost is $0.023 per GB per month, for the first 50 TB / month, using standard storage (half of that for infrequent access and $0.004 in the Glacier service) [Amac].

4.4.3.0.2 Proof Generation Costs In order to evaluate the monetary costs associated with integrity proof generation, SafeAudit's proof generator was executed in the cloud.

Figure 4.6 presents the time for generating integrity proofs in SafeAudit. The figure makes clear that the time grows linearly with the storage size.

Furthermore, as seen in Table 4.1, when comparing price payed for generating a proof (execution time) with the cost of downloading the files entirely and perform the integrity

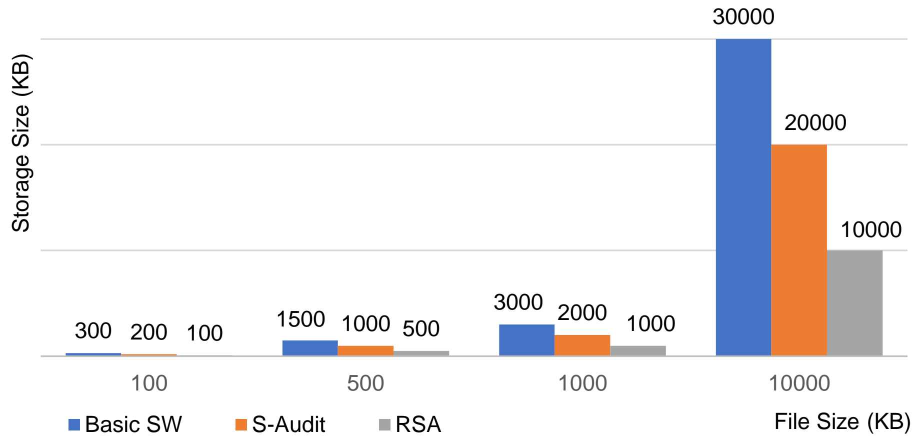


Figure 4.5: Storage size for storing data and signature when using SafeAudit, SW, and RSA.

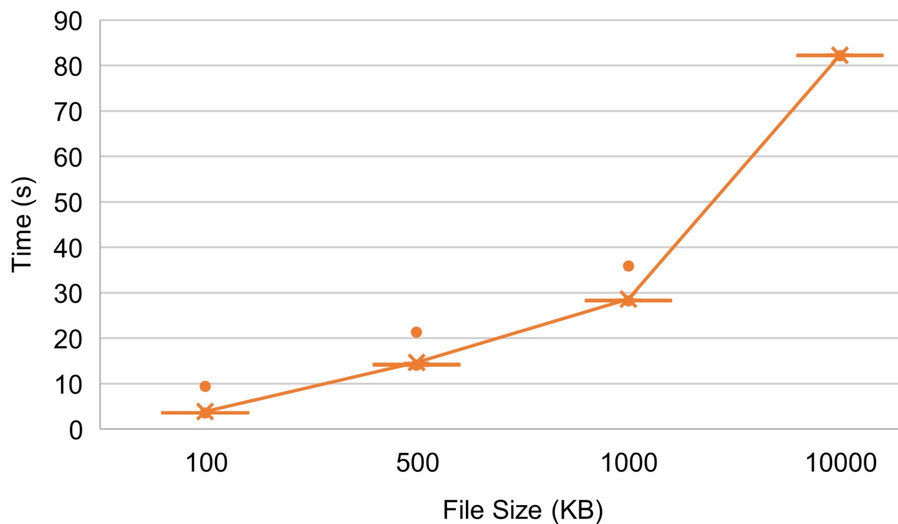


Figure 4.6: Time for the cloud to generate an integrity proof using SafeAudit.

verification on the auditors device (as required by RSA), generating integrity proofs was cheaper than reading the data from the cloud and allowed a monetary saving of 30%, on average.

4.4.3.0.3 Cost Tradeoffs The previous results show that using RSA signatures is costly (and slow) in terms of downloading data, but SafeAudit has additional costs in terms of storage. Therefore there is a tradeoff that we now quantify.

Figure 4.7 shows how the cost of verifying 1 MB varies with the number of verifications done per month. For SafeAudit the cost has two main components: the cost of storage (again the values for standard storage in S3 Ireland) and the cost for generating proofs in the cloud (again in Lambda). For RSA signatures, the cost has also two main components: the cost for storing and downloading the data (also S3 Ireland).

The main conclusion from the graph is that the best option in terms of cost depends on how often the data is verified. If the data is verified once per month, the cost of using

Table 4.1: Prices for generating proofs and reading data (based on Amazon Ireland prices, standard storage).

	SafeAudit		RSA signatures	
File Size (KB)	Average Execution Time (s)	Execution cost[1] (microUS$)	Read Costs[2] (microUS$)	Savings (%)
128	3.82	8.11	11.52	29.58
256	7.44	15.6	23.04	32.29
512	14.67	30.57	46.08	33.64
1024	28.55	59.48	90	33.90
10000	82.29	171.18	900	80.98

[1]Considering 0.208 microUS$ for each 0.1s of computation (Lambda)
[2]Considering 0.09 microUS$ for 1GB read from the cloud storage (S3)

SafeAudit is 7.1% lower than the cost of using RSA signatures. This cost becomes much lower – 34.9% – if the verification is done approximately every week (4 times per month).

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

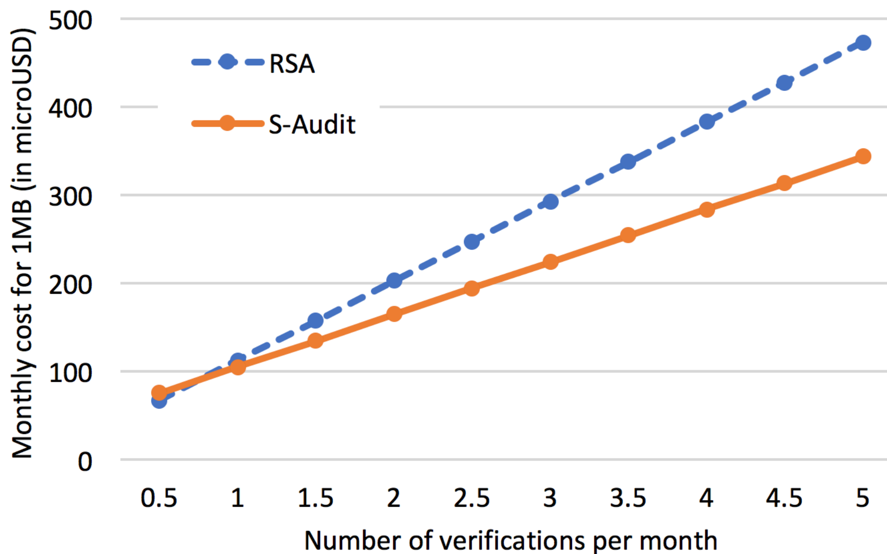


Figure 4.7: Monthly costs in millidolars for 1 MB of data depending on the number of verifications per month.

4.4.4 User: Client-Side Overhead

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

4.4.4.0.1 Signature Generation The signature generation of SafeAudit was evaluated in terms of the time required to compute a signature in the user's device. The results

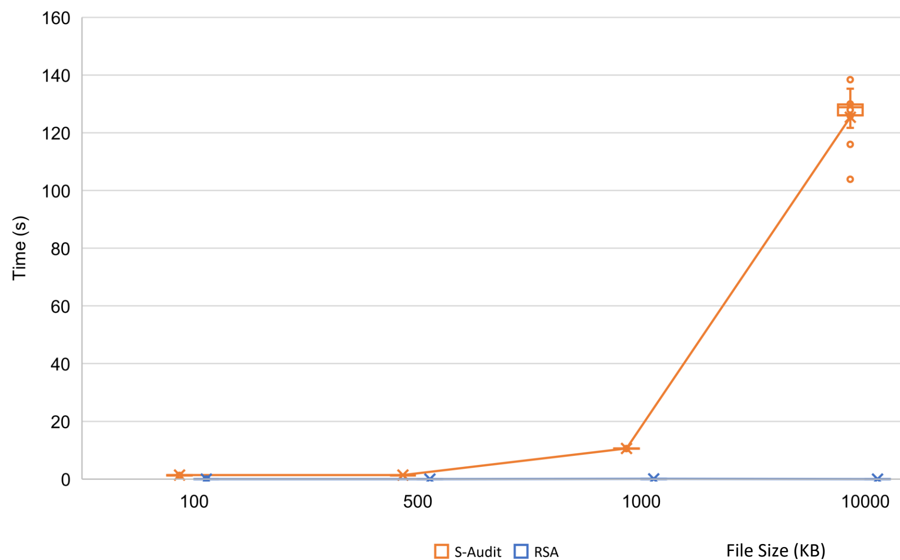


Figure 4.8: Time for signing data using SafeAudit and RSA.

obtained are presented in Figure 4.8. The time required for signing data using SafeAudit increases linearly and is much slower when compared to RSA digital signatures, which take around two milliseconds. This almost constant time is due to the fact that data signed using RSA digital signatures is first hashed (faster phase), then encrypted using RSA (slower but constant time). In SafeAudit, the SW scheme, and all the other publicly verifiable schemes, all data has to be signed without using hashes, to avoid security problems related to generating proofs using precomputed hashes (i.e., an adversary at the cloud computes the hashes once, corrupts or discards the data, and later computes proofs using only the hashes). Furthermore, due to this limitation it is necessary to sign each block of data individually, which takes longer as data grows. This makes SafeAudit slower than the usual signature generation mechanisms. This was expected due to the computational cost of the homomorphic signature generator. Nevertheless, this overhead can be masked by the application, as shown next.

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

The *non-blocking mode* is the one that is recommended [BMO+14b]. In this mode, when a client closes a file by calling *close*, the file is written to the local disk and the call returns. Then, in the background, DepSky pushes the file to the clouds. In this mode, both versions of SafeCloud-FS, with and without SafeAudit, had the same performance from the client's perspective.

In *blocking mode*, SafeCloud-FS waits for the file to be stored in the cloud for the *close* call to return. This mode is slow even in the original SafeCloud-FS, so it is not recommended [BMO+14b]. Nevertheless, we did several experiments of uploading a 1 MB file to the cloud using SafeCloud-FS with and without SafeAudit. The results obtained are

presented in Figure 4.9 and show that the integration with SAFEAUDIT increases time significantly. These results are similar to those presented in Figure 4.8 (for the 1 MB file).

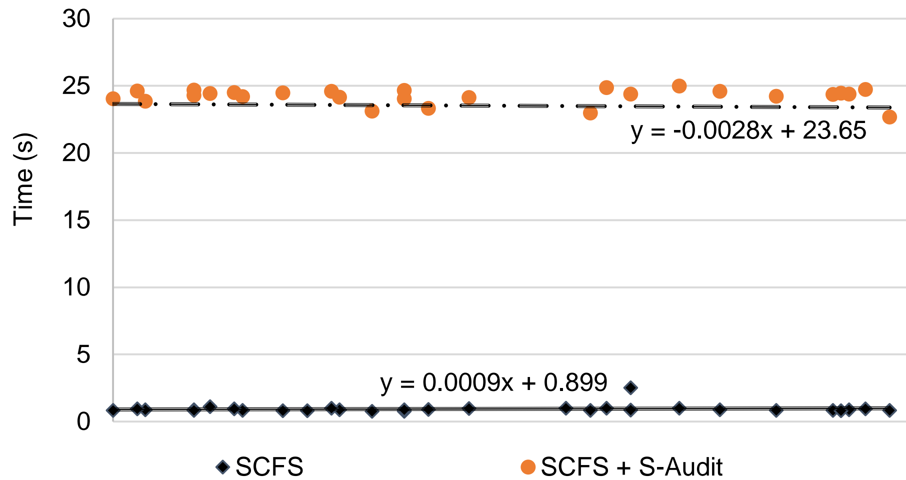


Figure 4.9: Time for SafeCloud-FS to upload a 1 MB file to the cloud with and without SAFEAUDIT (with linear interpolation).

4.4.5 Auditor: Proof Verification Time

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

As seen in Figure 4.10, the time necessary for verifying a signature in SAFEAUDIT increases linearly and is slow compared with RSA digital signatures. This is due to the fact that for verifying a proof using SAFEAUDIT and on the original SW scheme, it is necessary to multiply all the identifiers of the blocks audited and it does not scale well as data grows. For example, verifying 1MB of involves multiplying the identifiers of 25600 blocks which increase time to the values obtained in the experiments.

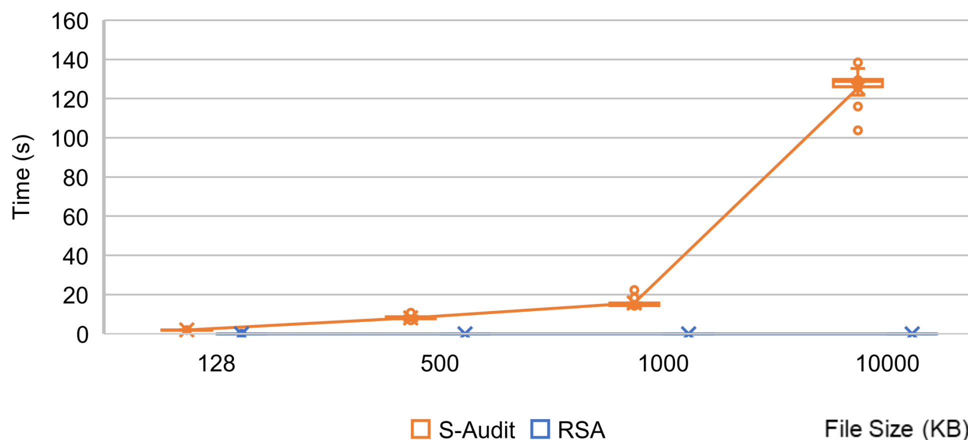


Figure 4.10: Time for verifying file integrity proofs in SAFEAUDIT and with RSA.

4.4.6 Evaluation Outcomes

The main outcomes of the experimental evaluation are the following:

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

4.5 Summary

This chapter presents SafeAudit, a cloud-storage verification service designed for being easily integrated with current cloud storage solutions, including cloud-backed applications and commercial storage clouds. SafeAudit automates all the tasks involved in storage integrity verification, including signature generation and verification.

5 Client-side security

The cloud-of-clouds file system paradigm increases integrity and availability of data by using several cloud service providers coordinated by a client. Such file systems may employ erasure-codes and secret sharing schemes to fragment the files and encryption keys, so that they can be securely distributed in diverse clouds. To attack the server side of a cloud-of-clouds file system, a malicious agent needs to penetrate the majority of the used clouds. However, a weak spot remains: the client device. Existing systems ultimately rely on this device to store the master credentials and if the attacker gains access to it, the system will be compromised.

This chapter presents client side extensions for SafeCloudFS providing recovery capabilities and resilience to client-side attacks. These extensions keep the resiliency of existing systems and adds encryption and secret sharing mechanisms to protect the client device and audit logs to allow file recovery after, e.g., ransomware attacks.

The initial version of SafeCloud-FS presented in deliverable D2.4 and some storage services in the literature provide high availability and integrity using public clouds [BCQ+11a, ALPW10, BMO+14a, ZYTT15a]. In these solutions several different cloud providers are used to distribute data. This *cloud-of-clouds* paradigm solves this problem and increases the integrity and availability of data. However, the client device remains as a point of failure. Existing solutions store master credentials on this device, and if an attacker penetrates it by exploiting some vulnerability, the file system is compromised.

The limitation is the assumption that the user's device or proxy used to access the file system is secure. This is a weak assumption considering that in many occasions users leave their personal devices unattended, use weak passwords, or fall prey to social engineering techniques [SSMJ05]. Once an attacker gains access to a user account, he can read, modify or even delete any file that the user has access to.

A specific concern are *ransomware* attacks [KRB+15]. If a device is compromised in this way, every file of the user is encrypted with a key known only to the attacker. Furthermore, the now encrypted files will be synchronized to the cloud storage making the remote copies useless for recovery.

To overcome possible attacks against a client device, we propose some client extensions that improve SafeCloudFS. These extensions add new protections that allow increased *availability* (an adversary can no longer corrupt the user credentials to prevent access to the clouds), *integrity* (it becomes possible to recover from file tampering, including ransomware attacks), and *confidentiality* (the files in the local cache are also encrypted). In this section we present the design of the client extensions, the working prototype, and an experimental evaluation of the costs of the recovery mechanisms.

5.1 Client Extensions for SafeCloud-FS

The developed work improves SafeCloud-FS in the following ways: it secures user data that is stored on the client side, namely, access credentials to the cloud service providers

and cached files; it provides logs that allow an administrator to analyze the usage of the file system and recover user files stored in the clouds.

5.1.1 Threat Model

The new extensions focus on three security threats, not addressed by SCFS or other cloud-backed file systems, as they originate from the client device:

Threat T1 *Adversary prevents a user from accessing the cloud*, by corrupting or deleting the access credentials stored in the client.

Threat T2 *Adversary accesses the locally cached files*, when he gets access to the user's device, as the client cache is not encrypted.

Threat T3 *Adversary illegally modifies files in the cloud*, e.g. through a ransomware attack that overwrites the data files in the client, which eventually are synchronized to the clouds.

5.1.2 System Model

In this new version of SafeCloud-FS there are two main actors: user and administrator. *Users* contract cloud services to store files and access them remotely using a personal computer or a mobile device. *Administrators* maintain the coordination service and monitor the usage of SafeCloud-FS .

Asymmetric keys – PUblic and PRivate – are used to authenticate both Users (PU_U, PR_U) and Administrators (PU_A, PR_A). 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. We assume that only the owner of a key pair knows the private key, whereas public keys are known and accessible to every user. SafeCloud-FS requires several keys, listed in Table 5.1, to ensure integrity and confidentiality of data.

5.1.3 System Architecture

Figure 5.1 presents the system architecture of SafeCloud-FS with the new client extensions. On the top left there are four different cloud storage services, where the data is stored. On the top right there are four replicas for the Byzantine fault-tolerant coordination service which should be deployed on four distinct cloud computing services. The coordination service stores the location of file data, locks for multiple writers and other metadata. The coordination service also communicates with the storage cluster in order to backup its state, ensuring later recovery when facing a system failure [BSF+13]. The client agent, on the bottom, communicates with both clusters when necessary.

Both the users of SafeCloud-FS as well as the administrator access the cloud storage and coordination services, but with different privileges: users only access their files; administra-

Entity	Notation	Description	Generated by	Stored in
User	PU_U	Public key of user U	User U during setup	Shared between the user's device, coordination service and external storage
	PR_U	Private key of user U	User U during setup	
	S_U	Session key of user U for local cache	User U	
Admin	PU_A	Public key of admin A	Admin A during setup	Shared between the admin's device, coordination service and external storage
	PR_A	Private key of admin A	Admin A during setup	
Log	A_i	ith log entry secret key	SafeCloud-FS agent	Coordination service
	B_i	ith log entry secret key	SafeCloud-FS agent	
Clouds	SC_i	Cloud storage service credentials	User and admin during setup	Shared between the user and the admin's device, coordination service and external storage
	CC_i	Coordination service credentials	Admin during setup	
	PU_{SC_i}	Cloud storage service public key	Admin during setup	Each cloud storage service
	PR_{SC_i}	Cloud storage service private key	Admin during setup	
	PU_{CC_i}	Coordination service public key	Admin during setup	Each cloud hosting a coordination service replica
	PR_{CC_i}	Coordination service private key	Admin during setup	

Table 5.1: Keys used in SafeCloud-FS with description, who generated them, and where they are stored.

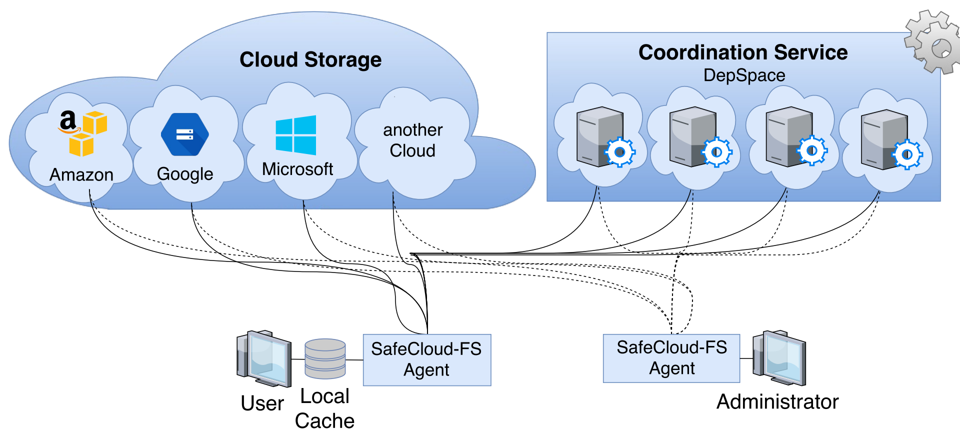


Figure 5.1: SafeCloud-FS architecture with four different cloud storage providers and four replicas for the coordination service DepSpace.

tors access also the logs. The users interact with the file system by invoking the POSIX [Gal95] operations (open, read, write and close). The administrators access 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 SafeCloud-FS agent, that is responsible for intercepting file system operations with the aid of the FUSE library[1]. It also performs the encryption and encoding tasks.

5.1.4 Client Architecture

The user device contains the SafeCloud-FS agent, the local cache and the keystore, including the private key of the user (PR_U). In order to access the cloud-of-clouds, the SafeCloud-FS agent needs the access credentials of each cloud storage provider (SC_i) and cloud used by the coordination service (CC_i). These credentials are also stored in the keystore file.

The keystore is protected with a user password and is kept in persistent storage. SafeCloud-FS splits the keystore in shares and stores them in separate places in a way that, even if an attacker obtains one of the shares, he cannot recover the entire keystore. This is achieved with secret sharing [Sha79].

When the user logs in, the SafeCloud-FS agent needs to combine some of the shares, e.g. 2 out of 3 shares. 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 an additional share 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 5.2.1.

The user device is represented in Figure 5.2. The disk stores the encrypted local cache and one of the shares of the keystore. The RAM stores the keystore that was 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 logic of SafeCloud-FS which is compiled in a software library providing the POSIX file operations (open, read, write and close).

5.1.5 Log Architecture

Figure 5.3 shows how the file system operations are logged to support recovery (as described in Section 5.3). The figure presents what happens when the user closes a file after writing to it. The file and the log data are uploaded to the cloud storage services. Then, the log entry in the coordination service is created. Finally, the file metadata is updated in the coordination service. This step must be performed at the end to ensure that everything – the file, log data and metadata – is available before notifying the user that the operation was completed successfully.

If the coordination service fails to register the file operation (e.g. because of a lack of consensus in the Byzantine fault-tolerant coordination service) then the log entry needs to be

[1] File system in user space https://github.com/libfuse/libfuse

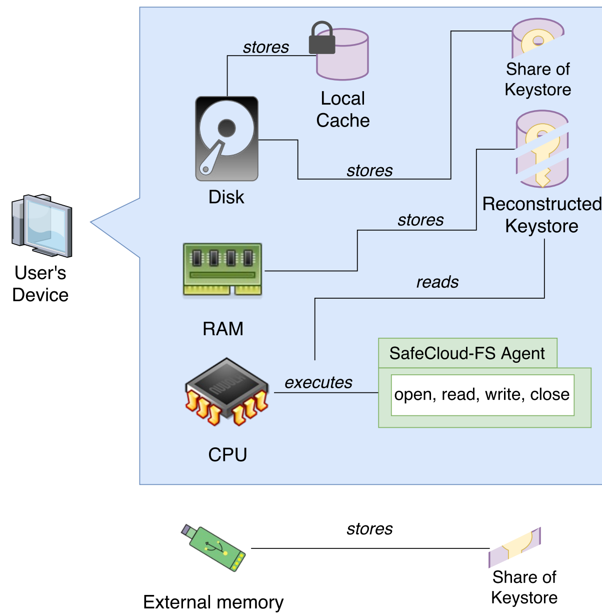


Figure 5.2: User device with client-side components and external memory with one of the shares of the keystore.

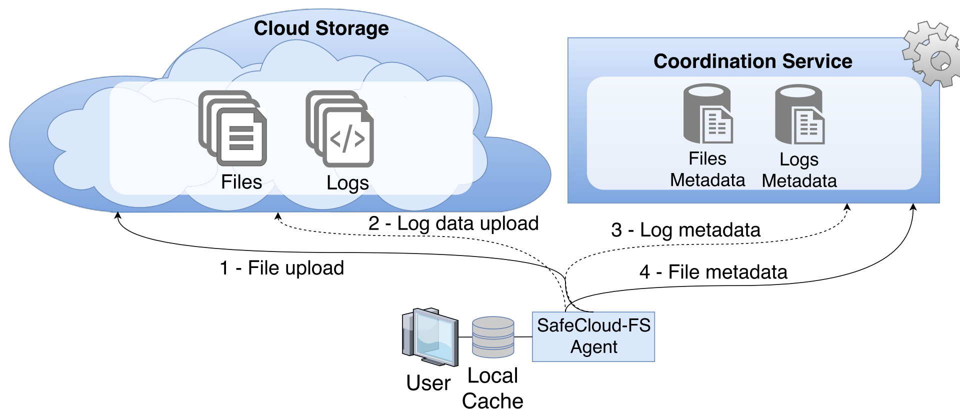


Figure 5.3: Logging of operations in SafeCloud-FS .

removed.

Each log entry is composed by 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 more effectively stored it in a cloud storage service; second, the metadata needs to be queried in as many ways as allowed by the coordination service's tuplespace.

5.2 Securing the User's Device

In this section we propose solutions for threats T1 (adversary prevents a user from accessing SafeCloud-FS) and T2 (adversary illegally accesses locally cached files) presented in Section 5.1.1.

The user's device stores two types of sensible information: the access credentials for the cloud providers, and the local cache with the most recently accessed files. With 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's file are at risk.

5.2.1 Securing the User's Credentials

The credentials are protected with secret sharing [Sha79] so, 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 SafeCloud-FS . In a secret sharing scheme a special entity, the dealer, distributes a secret to 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) [Sch99].

In SafeCloud-FS , the PVSS scheme is used to share two types of secrets: the keystore and the secret keys used to encrypt files and metadata. In both cases the client acts as a dealer, sharing, combining and verifying the secrets. More specifically, 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;
- `prove`, invoked by the server to create a proof for the share it owns;
- `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 SafeCloud-FS agent asks the user for how many shares of the keystore (n) should be generated, and how many are needed (k) to reconstruct the keystore;
- The SafeCloud-FS agent executes 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 easier log in to SafeCloud-FS (assuming a scenario with $n = 3$ and $k = 2$). However, the PVSS allows the user to chose 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 [SJKS17].

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);
- SafeCloud-FS agent fetches the remaining share from the coordination service;
- SafeCloud-FS agent executes the PVSS function `verifyS` to verify if all the shares are legitimate;
- SafeCloud-FS agent executes the PVSS function `combine` and loads the keystore into memory. In any case the keystore is in the disk.

5.2.2 Securing Client's Local Cache

In the proposed extensions SafeCloud-FS we propose mechanisms to verify the integrity and confidentiality of the local cache on the device, which stores the files recently accessed by the user. The client cache uses a least recently used (LRU) policy by default, but other policies could also be used, especially suited for erasure coded data [HFKT17].

Figure 5.4 represents the extension of the POSIX file operations `open` and `close` that was done for SafeCloud-FS .

5.2.2.1 *Integrity*

The integrity of the local cache is ensured by cryptographic hash functions. When a user invokes the `open` operation on a file f_u, the SafeCloud-FS agent executes:

- 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 is encrypted, as explained in the next section.

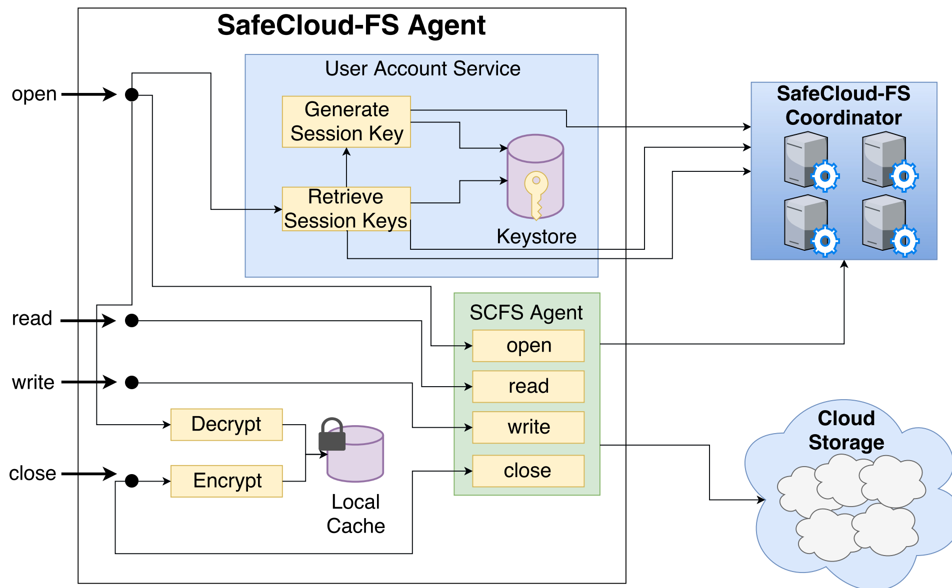


Figure 5.4: SafeCloud-FS operations, the read and write operations remain unaltered. The operations open and close decrypt and encrypt the local cached files to ensure data confidentiality.

5.2.2.2 *Confidentiality*

A session key, S_U, used to encrypt the local cache files, ensures their 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 SafeCloud-FS agent:

- Checks if S_U is still valid, if not the local cache is discarded and a new S_U is generated and exchanged with the coordination service;
- Decrypts the opened file and loads it into memory. If the large is too large to be loaded into memory an alert is given to the user that the file will be temporarily stored on disk.

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 the next section.

5.3 Storage Recovery

In this section we propose a solution for the remaining problem, T3 (when an adversary illegally modifies files on behalf of the user), of Section 5.1.1.

When an adversary hijacks a user's session, for example, by stealing the computer, he can tamper with the files on behalf of the user. Every action (create, modify and delete files) that the user executes in the user's device is eventually synchronized to the cloud, so the effects of the attack that occurred on the client side are propagated to the cloud side. When this happens the cloud-of-clouds 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.

To enable the administrator to undo the faulty operations performed by adversaries, every file operation (`open`, `read`, `write` and `close`) needs to be registered in a log. The SafeCloud-FS agent is the component responsible for recording these operations. The log is stored alongside with the files in the cloud-of-clouds, with the protections provided by DepSky.

The log metadata, lm_{fu}, contains a timestamp, the user id, the file name, the version id and the operation (create, update or delete). lm_{fu} is stored in the coordination service.

The log data of the user file, ld_{fu}, consists of the differences between the new version of the file and the previous one. ld_{fu}, is encrypted, split in shares using erasure-codes and stored in the cloud-of-clouds.

The erasure-codes library provides the following functions:

- `encode`(d, n, t) - encodes d on n blocks in such way that t are required to recover it;
- `decode`(db, n, t) - decodes array db of n blocks, with at least t valid to recover d.

When the POSIX operation `close` is invoked on a file f_u, the following operations are executed by the SafeCloud-FS agent:

- Compute a *diff* to calculate ld_{fu}, the differences between the new version of f_u and the previous version;
- f_u and ld_{fu} are encrypted using a random secret key S_{fu};
- The `encode` function is used to split f_u and ld_{fu} in n shares requiring $t = f + 1$ to be restore;
- Both f_u and ld_{fu} are send to the cloud storage services. Each cloud storage service gets on share of each;
- S_{fu} is split in shares with the `share` function of the PVSS with parameters n equal to the number of clouds and k equal to $f_u + 1$;
- Each share of S_{fu} is sent to each cloud storage service;

- Log metadata lm_{fu} is added in a tuple and sent to the coordination service.

One could assume that logs were stored in a secure environment, so they could not be illegally modified by adversaries. However, in the cloud environment, such an assumption is not realistic. To ensure the recovery is done correctly, it must provide *forward-secure stream integrity* [MT09], a property that ensures that logs, which are a stream of sequential entries, can be verified for integrity.

Forward-secure sequential aggregate [MT07, Ma08] is an authentication scheme that uses forward-secure signatures (or message authentication codes) to generate a single aggregate signature. However, this scheme has two problems: it is 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). The forward-secure stream integrity scheme presented in [MT09] solves these two problems by providing 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.

These protections are relevant for the security goals of SafeCloud-FS and the system implements them. The forward-secure stream integrity [MT09] 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.

Recovery is done by obtaining the last valid version of a file, and applying every valid modification 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.

5.4 Implementation of the Client Extensions

SafeCloud-FS and the new client extensions were 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 [Eur14].

The keystore (depicted in Figure 5.4) 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 SafeCloud-FS agent uses a variation of the UNIX *diff* command [HS77] called *JBDiff* [2]. 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.

5.5 Experimental Evaluation

The overhead of the protections for the credentials (threat T1) and cache (T2) 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 in detail on the costs of *recovery* from T3 described in Section 5.3. With the experiments performed we wanted to answer the following three questions: (a) What is the cost, in terms of performance, of having the SafeCloud-FS 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 SafeCloud-FS 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 SafeCloud-FS using two different clouds: Amazon S3 [Ama15] for the cloud storage services; and Google Compute Engine [KG15], for the coordination service. Regarding the

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

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.

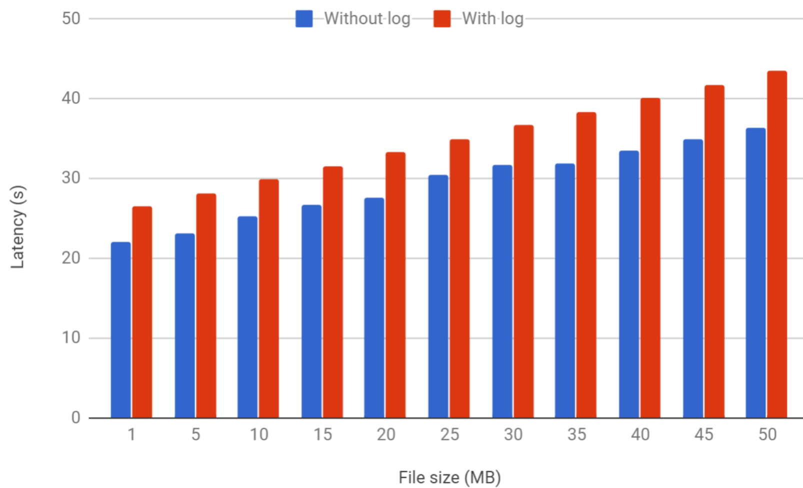
For the client machine, we created an instance, again with 1 vCPU and 3.75GB of memory, in the London data center. This additional instance serves as a client of SafeCloud-FS and will execute the SafeCloud-FS agent code. We chose to execute the client on the cloud for two reasons: first, it provides 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.

5.5.1 Latency Overhead of Log Operations

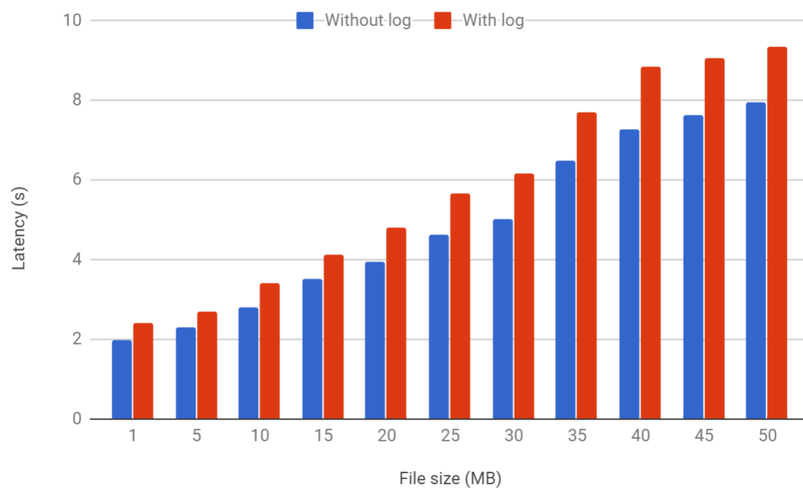
To calculate the latency of logging operations we created a workload that consisted in creating files and then updating these files with an extra 30% content. We vary the size of the files between 1 and 50MB, according to statistics from [ABDL07]. Given that SCFS offers two different approaches for file 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 5.5 presents the average latency of logging file operations in SafeCloud-FS . 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 with logging are, on average, 20% higher than the ones without logging. This overhead is expected, since it takes time for the SafeCloud-FS 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 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 [Fil17] to execute different workloads. Table 5.2 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 SafeCloud-FS with and without extensions 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 in SafeCloud-FS with and without extensions to understand how much it costs, in terms of performance, to log file operations. Unlike the original experiments in SCFS [BMO+14a] 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 SafeCloud-FS . The overhead of using the client extensions according to the results shown in Table 5.2 is non-negligible but can be considered acceptable, especially in the non-blocking mode



(a) Blocking API of the file system.



(b) Non-blocking API of the file system.

Figure 5.5: Latency of using SafeCloud-FS with and without the log in both modes of the file system: (a) blocking and (b) non-blocking.

which is the recommended configuration.

5.5.2 Storage Overhead of Log Operations

We did experiments to find out how much more storage SafeCloud-FS 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 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.

Micro-benchmarks	# Operations	File size	Without extensions NB	Without extensions B	With extensions NB	With extensions B	Overhead NB	Overhead 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 5.2: Latency (in seconds) of Filebench micro-benchmarks for SafeCloud-FS with and without extensions.

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).

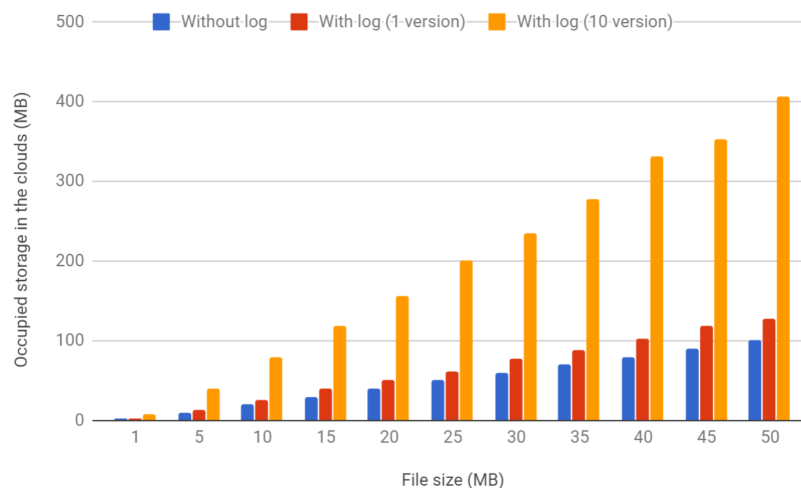


Figure 5.6: Required storage for the files and logs in the cloud storage services.

Figure 5.6 presents the storage overhead of logging different files with different versions. The storage growth is linear. For the 10 versions we can see that the storage required to store 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).

5.5.3 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

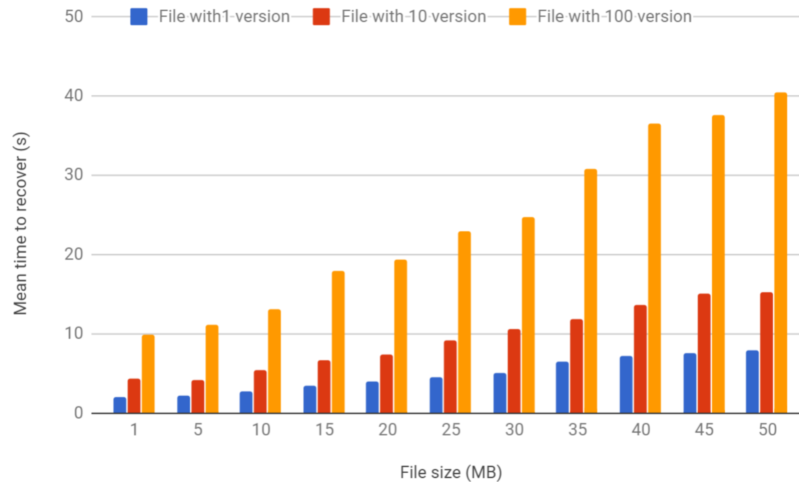


Figure 5.7: Mean time to recover files with 1, 10 and 100 versions.

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 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 5.7 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 one entry at a time.

To evaluate how SafeCloud-FS 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 SafeCloud-FS recovered every file of the file system. The results are presented in Figure 5.8. 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 SafeCloud-FS 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 SafeCloud-FS continues the remaining recovery process in the background.

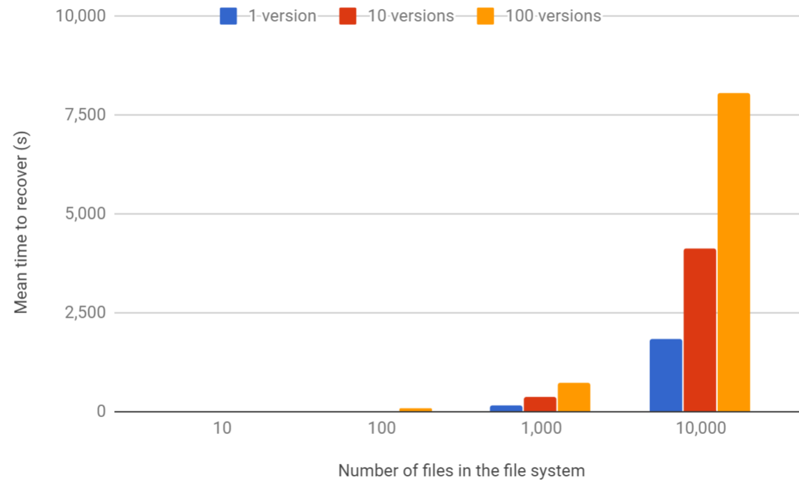


Figure 5.8: Mean time to recover a file system compromised by a ransomware attack varying in the number of files and versions of each file.

5.6 Summary

This section presented presented new client extensions for SafeCloud-FS, 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. The proposed client extensions improve SafeCloud-FS 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 SafeCloud-FS 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.

6 Mobile client trust

Mobile devices have gradually become an integral part of our daily lives. Currently, financial and healthcare institutions offer services to their clients using smartphone and tablet applications, which are important use cases for SafeCloud. Although this is convenient, it also leads people to rely on these applications to access and process privacy-sensitive data, e.g., financial data and medical records. Many of these applications run on Android, the most adopted mobile operating system (OS) today [K+16]. However, the popularity of Android and the open nature of its application marketplaces make it a prime target for malware [GZJS12, ZWZJ12, ZJ12, CFGW11]. This situation puts data stored in smartphones in jeopardy, as it can be stealthily stolen or modified by malware that infects the device.

Several companies provide anti-malware software for mobile OSs. Moreover, the research community has investigated mechanisms for detecting malware on Android using static and dynamic analysis [EGC+10, HHJ+11]. However, these tools and mechanisms run in the device and assume that the mobile OS is trusted, i.e., that it is part of the trusted computing base (TCB) [Nat83]. However, current malware often disables anti-malware software when it infects a device or computer. For mobile phones this trend started more than a decade ago with malware such as the Metal Gear Trojan and Cabir.M [Lea05] and continues, e.g., with HijackRAT [Gre14].

TrustZone is a hardware security extension incorporated in recent ARM processors, much used in smartphones and other mobile devices [ARM09]. This extension partitions the system resources (e.g., memory, peripherals, etc.) in two logical parts: the secure world and the normal world. The *secure world* runs trusted applications on top of a small trusted OS, whereas the *normal world* runs the normal applications on top of a normal OS such as Android. TrustZone protects the secure world resources from the normal world, whereas the secure world can access resources of the normal world. This hardware separation protects the confidentiality and integrity of computation and data in the secure world, while permitting the secure world to inspect the normal world.

Most uses of TrustZone in the literature are based on *measurements* of the normal world, i.e., on hashes of the software running in the normal world obtained using a collision-resistant hash function. This way of using TrustZone is interesting, but the versatility of TrustZone suggests it is possible to obtain richer information about the normal world than just hashes, which are simply numbers with limited semantics. An approach is to analyze the *posture* or *compliance* of the device. The notion of posture assessment was introduced in RFC 5209 [SKM+08] for *network access control* [Hof08], which proposed having a software agent running on endpoint devices (such as laptops and desktops) to evaluate and report the posture/compliance of the device to the network owners (e.g., anti-virus software running on the device or not, updates installed or not). The network owner has validation software that determines the device's compliance with the security policies, allowing it to connect to the company's network, to block it, or to connect it to some lower trust VLAN (e.g., one that connects only to the Internet).

The previous chapter assumes that the SafeCloud-FS client is executed in a terminal, e.g., in a personal computer (PC). However, it is possible to execute it in a mobile device and, such devices, are arguably less protected from malware than PCs. This chapter presents the design and implementation of DroidPosture, a *posture assessment service* for mobile

devices. DroidPosture runs in the devices (e.g., smartphones) and evaluates the security status of their OS (e.g., Android) and applications. DroidPosture is protected from the OS, applications, and malware by leveraging the TrustZone extension and running in the secure world. DroidPosture does introspection of the normal world and can be configured with a variety of assessment mechanisms, e.g., static analysis of applications and detection of rootkits. Posture data obtained with DroidPosture can be sent to *external services*, such as a SafeCloud-FS' coordination service that may be configured to use this information to grant or deny access.

In this chapter we propose two classes of assessment mechanisms – application and kernel analysis mechanisms – and provide two example of each: signature-based detector, learning-based detector, syscall table checker, and kernel integrity checker. These mechanisms illustrate the forms of posture analysis that can be implemented in DroidPosture, although others may also be used.

6.1 Background

This section provides background on the technologies underlying the design and implementation of DroidPosture.

6.1.1 ARM TrustZone

ARM TrustZone [ARM09] is a security extension supported by recent ARM processors, e.g., ARM Cortex-A and Cortex-M. It provides two logically isolated execution domains: the secure world for security-sensitive computation and storage, and the normal world for conventional processing. A program running in the normal world can make the processor switch to the secure world using a special *secure monitor call* (SMC). This technology is not yet widely-adopted, but it is used in Samsung's KNOX enterprise mobile security solution [Sam].

TrustZone partitions the system memory into two worlds, i.e., each world has its own address space. The secure world can see all the physical memory in the system, but the normal world can see only its own. Cache memories are tagged as secure or non-secure to protect them from accesses from the normal world. Individual hardware peripherals can be assigned to the secure world, and for these peripherals hardware interruptions are configured to be directly routed to, and handled by, the secure world. As a result, it is possible to secure peripherals such as memory, storage (e.g., an SD card), keyboard and screen to ensure they are protected from software attacks. In general, TrustZone protects the confidentiality and integrity of any computation and data in the secure world, so untrusted code running in the normal world cannot access these resources [SRSW14, YMHC17].

6.1.2 Android Architecture

Android is a Linux-based open source software stack for mobile devices. It consists of a modified Linux kernel, a middleware layer, and an application layer. The Linux kernel provides OS services like memory management, process scheduling, device drivers, file system support and network access. The middleware layer consists of native Android libraries, the Android run-time environment, and an application framework. The application framework consists of applications that provide system services, e.g., the Activity Manager that manages the life cycle of applications, the Application Installer that installs new applications, and the Package Manager that maintains information about all applications loaded in the system.

Android applications are implemented in Java, but they can also incorporate C/C++ native libraries through the Java native interface (JNI). Java code is compiled to a custom bytecode format, the Dalvik EXecutable (DEX) format. Android applications are comprised of four basic types of components: Activities, Services, Content Providers and Broadcast Receivers. Activities are responsible for the (graphical) user interface. Services execute background processes without user interaction. Content Providers support SQL-like data management for storing and sharing application data. Broadcast Receivers receive event notifications from the system and other applications.

Android applications are distributed as files in the Android Application Package format (APK or `.apk`). An APK file is essentially a zip archive containing all the application resources: bytecodes (`.dex`), manifest file, media files, etc. After a successful installation of an application, its APK file is stored in the file system (in `/data/app/`). For static analysis of an Android application, one has to unzip the APK file, then decompile the `.dex` file or translate it into Java source code using appropriate tools (e.g., dex2jar, APKtool). Every APK file includes a manifest file (`AndroidManifest.xml`) with essential information about the application: list of components that compose it; permissions that the application needs to run; permissions that other applications must have in order to interact with the application's components.

6.2 DroidPosture Architecture and Design

DroidPosture gives external service providers a mechanism to evaluate the posture of a smartphone and restrict access to critical data – files in the case of SafeCloud-FS – based on the evaluated posture. DroidPosture is a software component that runs in the secure world of a mobile device. The posture information is requested by external services when smartphone applications request to use a service or by local application to understand the posture of the smartphone.

6.2.1 Threat Model and Assumptions

We assume that DroidPosture runs in an ARM processor with TrustZone. We assume that in the normal world the mobile OS and the applications it executes are untrusted, i.e., that

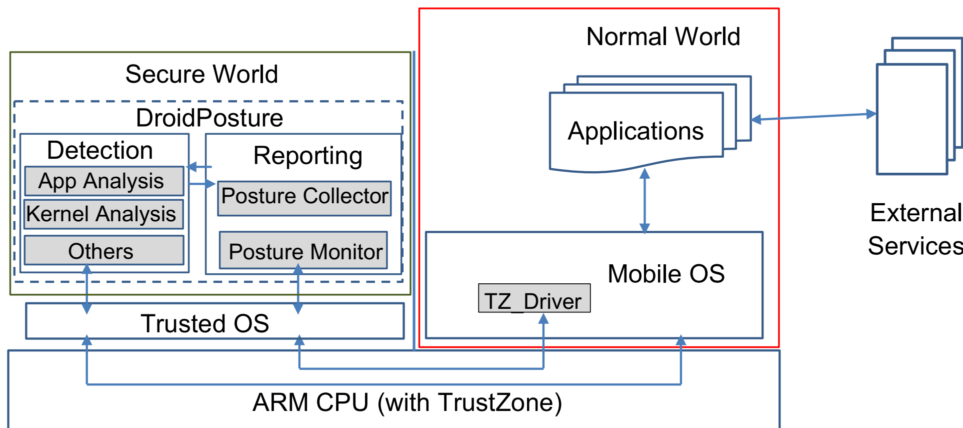


Figure 6.1: Architecture of a mobile device running DroidPosture. The grey boxes are components of the DroidPosture service. The applications in the normal world may use the SafeCloud-FS client to accees the cloud. The external services are the SafeCloud-FS coordination service in this case.

they may be malicious or compromised by malware or hackers. In contrast, we assume that the software running in the secure world, including the DroidPosture software, is trustworthy. In order to reduce the size of the TCB, the size of the software executed in the secure world has to be as small as possible, so it should not include for example a network stack. The size of the API is also as small as possible to reduce the attack surface, and all inputs are validated. These measures make software attacks against the secure world unlikely to be successful, so in this chapter we assume they are not. We also assume that the device (i.e., the normal world) is not yet compromised when DroidPosture is first installed (it can come pre-installed with the device).

Each DroidPosture instance in a device has an identifier *id* and a public-private key pair (Ku, Kr) for some public-key cryptographic scheme (e.g., RSA). It has also a certificate containing the public key, signed by some trusted certification authority (CA). We also assume the existence of a collision-resistant hash function (e.g., SHA-256).

6.2.2 Architecture

Figure 6.1 represents a mobile device running DroidPosture, SafeCloud-FS client applications, and communicating with the SafeCloud-FS coordination service. The *normal world* runs the usual mobile device software: a mobile OS and applications. It includes also a driver (`TZ_Driver`) that allows software in the normal world to call functions in the secure world, DroidPosture in our case. This driver allocates a shared memory zone that is used for the application to pass inputs to DroidPosture, and for DroidPosture to return outputs to the application (assessment results in our case).

The *secure world* runs the DroidPosture service. This world includes a small trusted OS that provides basic functions for software running in that world (processes, file access, etc.). Besides its private memory space, it is also configured to have a private persistent storage partition (either part of the internal memory or of an SD card). DroidPosture itself is composed of two modules, for detecting and reporting the posture of the normal world.

The *detection module* is further divided in two modules that we implemented in our prototype – *application analysis* and *kernel analysis* – although others may be designed and used (*others* in the figure). The *reporting module* provides an interface between the normal world and the detection modules. The *posture monitor* receives, validates (for protection against buffer overflows and other input attacks), and replies to requests for posture data from the normal world. The *posture collector* collects information from the detection module(s) and signs it.

The bootstrapping of the device starts by running the kernel of the secure world, so this kernel is the *static root of trust for measurement* [PMP11]. Before passing the control to the normal world and starting the execution of the normal world, the *kernel analysis* module calculates and stores a hash (measurement) of the normal world operating system. This is process is denominated *trusted boot* [PMP11] and may involve storing hashes of other modules, if needed.

6.2.3 Posture Reports

DroidPosture provides information about the posture of the device in the form of a *posture report*. The format of a posture report is: $\langle id, posture_data, nonce \rangle_{S_{Kr}}$, where *id* is the identifier of the DroidPosture instance, *posture_data* the posture data itself, *nonce* a nonce (a random number used only once for replay protection) that comes with the request for posture data, and S_{Kr} a signature obtained using the instance private key.

Posture reports can be delivered to applications or transferred to the SafeCloud-FS coordination service via the normal world. In both cases an application running in the normal world requests posture data from the *posture monitor* module. This request contains the above-mentioned nonce. The module then invokes the *posture collector* module that requests the *detection* modules to collect the posture of the device. The posture collector gets the result, creates and signs the posture report, and sends it to the posture monitor. The latter sends the posture report to the application, which may optionally send it to the SafeCloud-FS coordination service.

As the mobile OS and the applications may be compromised, we do not trust them to deliver the posture report unmodified to the application or external service that requested it. The authenticity and integrity of the report are verified using the digital signature S_{Kr} calculated using the private key of the DroidPosture instance in the device. Such an attacker might still do a denial of service attack by deleting or modifying all reports, but this would be understood by the service provider as consequence of a compromised device.

Figure 6.2 illustrates the steps for providing a posture report to an external service (e.g., the SafeCloud-FS coordination service). An application starts the interaction with the external service, e.g., the backend of the application (step 1). The external service replies and provides a nonce (step 2). The application forwards the nonce to DroidPosture in the secure world and asks for posture assessment (step 3). DroidPosture performs the posture assessment, creates and signs the posture report, then sends it to the application (step 4). The application sends this report to the external service (step 5). Finally, the external service verifies the nonce and the signature, using the certificate of that DroidPosture instance (Section 6.2.1). If they are correct, it then interprets the posture data. If it finds the

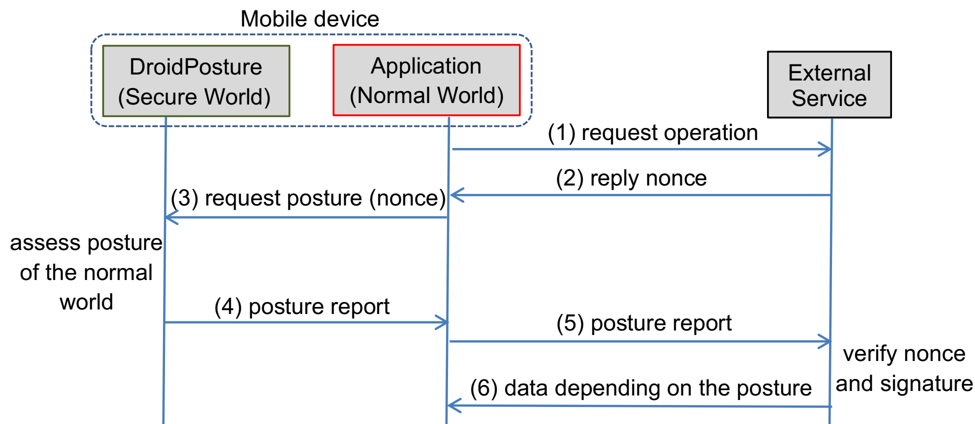


Figure 6.2: DroidPosture providing a posture report to an external service, such as the SafeCloud-FS coordination service.

posture acceptable it continues to interact the application, e.g., sending it some data (step 6).

6.2.4 Application Analysis

We designed two *detection* modules, although others may be used. As mentioned before, these mechanisms illustrate how posture analysis may be implemented in DroidPosture.

This section is about the *application analysis* module (Figure 6.1). This module provides two static analysis mechanisms to detect the presence of malware in Android applications: *signature-based detector* and *learning-based detector*.

By static analysis we mean analysis of code, without executing it. The application analysis module first unpacks the application APK and obtains the bytecodes. Then, the bytecode file is passed as input to the signature-based and learning-based detection mechanisms. Prior to this process, the hash of the APK file is compared with the hashes of the APKs already analyzed stored in the secure world persistent storage partition, in order to avoid re-executing the analysis. If it has already been analyzed, the result obtained previously is returned. Notice that if the APK changes, then its hash also changes (collision resistance property of the hash function).

6.2.4.1 Signature-Based Detector

The first and simplest malware detection technique is based on pattern matching. Malware detectors have a database of distinctive patterns – *signatures* – of malicious code and they look for them in applications. Malware has to be public for a period of time so that signatures can be generated for that specific malware family.

Our *signature-based* mechanism detects malicious applications or injected malicious code based on similarities of control flow graphs (CFGs). A control flow graph represents the control flow of a program. The signature-based mechanism takes the bytecode file and

converts each function in the bytecode into a string that represents the CFG of the function. The comparison of similarity of the CFGs is done by using a similarity algorithm such as Kolmogorov distance and normalized compression distance (NCD). The CFG string of each function is compared against the CFGs (the signatures) of known malware in the database to verify if they are similar.

6.2.4.2 Learning-Based Detector

As the *signature-based* detection mechanism only detects malware for which it has signatures, we propose a complementary mechanism to detect malware in applications. The *learning-based* detection mechanism relies on a machine learning classifier. This mechanism has two phases. In the first phase, *training*, the mechanism statically examines and extracts selected features from known malware samples and benign applications to build feature vectors; then, these features are used to train a machine learning classifier to distinguish malware from normal code. In the second phase, *detection*, the classifier is used to check applications for malware. Notice that the detection phase is the one that is executed by DroidPosture itself; the training phase is executed beforehand, by whoever manages the DroidPosture service.

The selection of features is essential for the efficiency of the detection mechanism. Redundant or relevant features may present several problems such as misleading the learning algorithm, and increasing model complexity and run time. We use the features below, extracted from the `AndroidManifest.xml` file and the `.dex` file. All features are binary, i.e., either the application has it or not:

- Requested permissions. Android uses permissions for restricting access to the device resources. Permissions are granted by the user during application installation, or later in the latest versions of Android. Malicious applications request certain permissions (e.g., `SEND_SMS`, `READ_SMS`) more often than benign applications. Requesting a security sensitive permission is a feature.

- Sensitive function calls. Among thousands of Android API functions, we consider API calls invoked by applications that allow to access sensitive data or resources. For example, APIs for accessing the user's personal information, network details, device ID, and sending SMSs.

- Suspicious intents. An intent is an abstract description of an operation to be performed. Application components are activated using intents. We consider as features intents that perform sensitive actions as features, e.g., `android.intent.action.CALL` or `android.intent.action.DIAL`.

- Suspicious content URI. A content URI is used to locate data in a content provider. It can be used to leak user's personal data or to access another application's data. For example, `content://sms/inbox` can be used to read SMS messages from inbox.

- Arbitrary code execution. Execution of native code using JNI or Linux commands. For example, `Ljava/lang/Runtime;->exec()` executes command `exec()` in a separate process.

In the training phase, to construct feature vectors, we retrieve the selected features from each malware sample and benign application and store them as binary numbers, 1 or 0, respectively for presence or absence of the feature. Furthermore we assign a class to each feature vector, *M* for malware and *B* for benign application. Feature vectors are then provided to the machine learning classifier.

We use the *k-Nearest Neighbours* (kNN) algorithm as classifier [Alt92]. Given vectors of N classes as training samples, kNN classifies an unknown sample by searching the entire set of training samples for the k nearest based on a distance metric, then the unknown sample is assigned to the class most common among its k nearest. k is a positive integer and if $k = 1$, then the unknown sample is simply assigned to the class of the single nearest sample.

6.2.5 Kernel Analysis

This section presents the *kernel analysis* module of Figure 6.1. Again, this module provides two kinds of analysis.

A rootkit is a piece of malware that gains privileged access to a system, hides itself from the user and the OS, then stealthily carries out some kind of malicious activity. Rootkits may be roughly classified in two classes. *User-level rootkits* replace system binaries and libraries with customized versions. *Kernel-level* rootkits modify the kernel, for example by adding code into the running kernel memory image (`/dev/mem`) or by injecting a Loadable Kernel Module (LKM).

The *kernel analysis* module starts by calculating a hash of the normal world kernel and by comparing it with the hash obtained during the boot of the device (Section 6.2.2). If the hashes are different, the kernel analysis fails immediately.

6.2.5.1 Syscall Table Checker

The Android kernel provides system calls (*syscalls*) that allow applications in user mode to interact with the kernel. Syscalls are one of the primary targets for kernel-level rootkit writers. The kernel uses a *syscall table*, an array of pointers mapping each syscall number to the corresponding function in kernel memory. Modifying a syscall table entry is a popular way to intercept the execution flow of any system service. Kernel-level rootkits often modify syscall table entries to point to new, malicious, system calls. Therefore, in order to detect a kernel-level rootkit, the first step is to verify the integrity of the system call table.

Each time the kernel is compiled, a file containing the map of kernel symbols and addresses is created (`System.map`). Comparing the addresses of syscalls in the `System.map` with the addresses in the *syscall table* during runtime detects if system calls have been redirected, which may be an indication that the kernel has been compromised by a rootkit.

When DroidPosture is installed, our *kernel analysis* mechanism starts by making a copy of the addresses of system calls in `System.map` and storing them in the secure world. Then during runtime the mechanism simply compares that copy with the values in the syscall

table in the normal world. Recall that we assume that the system is not compromised when DroidPosture is installed.

6.2.5.2 Kernel Integrity Checker

Besides syscall table integrity checking, the *kernel analysis* module is capable of checking the kernel code for modifications to detect rootkits. As the kernel is not supposed to change during runtime, changes are probably a sign of malware. For example, a rootkit can replace the first few bytes of some system call functions with a `jmp` instruction that redirects the execution to malicious code.

In order to verify the kernel integrity, the kernel integrity checker calculates a hash of the kernel code memory pages of the Android OS running in the normal world and compares it against a hash calculated when the system was in a pristine state, which is stored in the secure world persistent storage partition. To calculate a hash value, the start address and length of the target memory pages are required. The kernel integrity checker finds the virtual address of the kernel code in the copy of the `System.map` file stored in the secure world and translates this address to the secure world address space before evaluating the hash value.

6.3 DroidPosture Implementation

We implemented a prototype on an i.MX53 QSB board equipped with a Cortex-A8 single core 1 GHz processor, 1 GB DDR memory, and a 4GB MicroSD card. Most TrustZone-enabled smartphones are locked in such a way that it is not possible to use the secure world, so we opted for this board.

6.3.1 Runtime Environment

Genode is a framework for building special-purpose operating systems [Gen14]. It provides a collection of OS building blocks, e.g., kernels, device drivers, and protocol stacks. Genode can reduce OS complexity for security-sensitive scenarios, which makes it an appealing foundation for an OS to run in the secure world. Genode Labs has released a TrustZone virtual machine monitor (VMM) demo for our board, which enables the execution of Genode in the secure world, while a guest OS such as Linux, monitored by a Genode hypervisor, runs in the normal world. We used this demo as a starting point to implement our prototype.

In the secure world, we implemented DroidPosture based on a program called `tz_vmm` that runs on top of the Genode kernel. In the normal world, we run Android for the i.MX53 series from Adeneo, previously freescale (`http://witekio.com/cpu/i-mx-53/`). We used the Linux/Android kernel modified by Genode Labs for this board. The kernel is modified so as to prevent the normal world from directly accessing certain resources such as hardware, persistent storage and memory that are set as secure within the central security unit

Table 6.1: Lines of code for the DroidPosture modules.

Modules	Code Size (LOC)
Application Analysis	30484
Kernel Analysis	142
Posture Collector	86
Posture Monitor	121

(CSU) initialization. To create the secure world persistent storage partition, we used the Genode partition manager (`part_blk`) that supports partition tables and provides a block session for each partition of a SD card. This allows the partitions to be addressable as separate block sessions and makes it is easy to grant or deny access. We used this scheme to reserve a partition for the secure world.

We run `TZ_Driver` in the kernel for an application in the normal world to issue a hypercall to exit the normal world and trap into the secure world, using the SMC instruction. A shared buffer in RAM allows passing data between the two worlds. Some of the general purpose CPU registers are used to store information about the shared buffer between the two worlds, including its address and length.

In our prototype we used components written in Python, which required installing Python 2.6 in the secure world using the Genode libports repository. This is undesirable because it increases the size of the TCB. However, this is not a limitation of our proposal, but of the current prototype. DroidPosture itself does not need to use Python code.

6.3.2 DroidPosture Modules

Table 6.1 shows the code size of each module implemented in the DroidPosture service.

The *application analysis* module is based on Androguard [Anda], an open source tool written in Python. It is able to unzip an APK file, obtain its metadata and bytecodes. Androguard has a module to create the control flow graph (CFG) for each function in a bytecode file. In addition, Androguard has several built-in signatures that are able to detect known malicious applications. Since Androguard is a complete feature-rich framework, we use its modules to disassemble an application's Dalvik bytecode, then create a CFG for each function, and compare these CFGs with the malware CFGs (the signatures) that are stored in the secure world persistent storage partition. In addition to Androguard, we modified Androwarn [Andb] to extract the features (Section 6.2.4.2) from malware and benign applications to build feature vectors for the learning-based detector.

The *kernel analysis* module needs to access the normal world memory. TrustZone configuration within Genode partitions the DDR RAM between the secure world and the normal world using the multi-master multi-memory interface (M4IF) [Gen14]. `tz_vmm` is able to read the normal world's RAM via an IOMEM session during its start-up routine. The memory is mapped as uncached to the secure world's address space, thus the whole normal world memory can be accessed by the kernel analysis module in the secure world.

Table 6.2: DroidPosture delay when called locally (in seconds).

Size		Calls						
APK	.dex	ii	iii	iv	v	vi	vii	viii
12KB	5KB	1.81	0.8	2.23	0.15	1.64	1.75	4.01
19MB	39KB	14.12	1.51	14.92	0.14	1.63	1.77	16.26
4MB	67KB	31.84	1.57	32.26	0.14	1.63	1.77	33.19
250KB	103KB	29.03	1.55	29.70	0.14	1.63	1.77	30.40
803KB	153KB	66.28	5.85	69.96	0.14	1.63	1.77	71.21
401KB	305KB	206.21	7.86	206.54	0.14	1.63	1.77	208.42

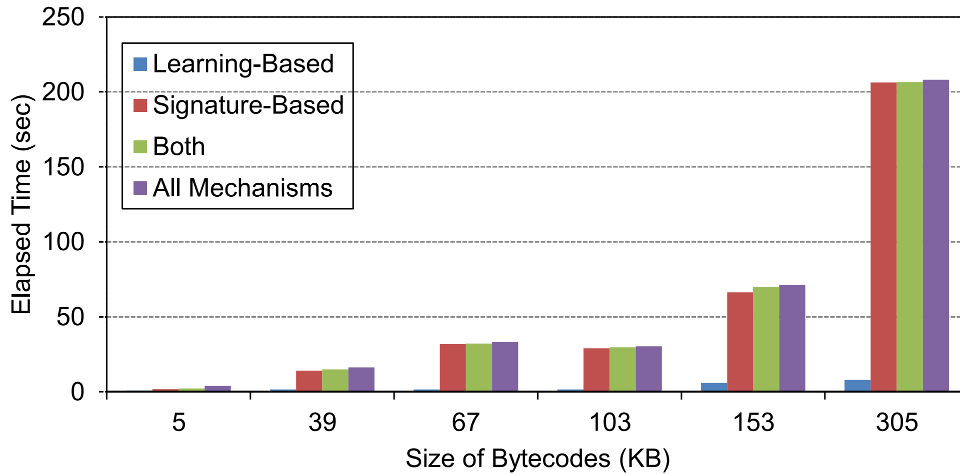


Figure 6.3: DroidPosture delay when called locally with emphasis on the applications analysis modules (in seconds).

6.4 Performance Evaluation

To evaluate the performance of DroidPosture, we used a set of micro- and macro-benchmarks by considering calls to the DroidPosture service that: (i) return immediately (baseline); (ii) do application analysis, only signature-based; (iii) do application analysis, only learning-based; (iv) do application analysis, both mechanisms; (v) do kernel analysis, only syscall table checker; (vi) do kernel analysis, only kernel integrity checker; (vii) do kernel analysis, both mechanisms; (viii) do all the detection mechanisms.

In the *micro-benchmarks*, an application (in the normal world) sends a request for posture and gets a reply back from DroidPosture (in the secure world). The *macro-benchmarks* are used to evaluate the posture assessment transmission protocol. For this purpose, we used a remote server which runs on a standard laptop. The server listens for incoming requests from the application in the normal world and sends requests for posture to the DroidPosture service running in the secure world of our board via the application.

6.4.1 Micro-benchmarks: mechanism performance

We used the calls mentioned above to evaluate the overhead of DroidPosture. To measure the time for the baseline (i), the application in the normal world sends a request for posture to the DroidPosture service in the secure world that does not execute any analysis

Table 6.3: DroidPosture delay when called by a remote service (sec.).

Size		Calls						
APK	**.dex**	**ii**	**iii**	**iv**	**v**	**vi**	**vii**	**viii**
12KB	5KB	1.85	0.89	2.29	0.17	1.67	1.78	4.59
19MB	39KB	14.27	1.56	14.94	0.16	1.65	1.78	16.74
4MB	67KB	31.86	1.60	32.38	0.16	1.65	1.78	34.05
250KB	103KB	29.07	1.57	29.77	0.16	1.65	1.78	31.24
803KB	153KB	66.39	5.87	69.12	0.16	1.65	1.78	72.32
401KB	305KB	206.61	7.88	207.11	0.16	1.65	1.78	208.89

module. We repeated the experiment 1000 times and obtained an average of 0.082 *ms*, with standard deviation of 0.0061.

For the rest of the calls the process is similar, except that DroidPosture executes a subset of the analysis modules. We expected calls to the application analysis modules to depend on the size of the applications, so we considered a set of applications with different sizes (downloaded from Google Play Store). We did experiments for the combinations of calls (ii) to (viii) and all bytecode sizes. The results of these experiments are shown in Table 6.2. Moreover, in Figure 6.3 we represent the same values but only for the combinations of application analysis modules and the total.

These results allow us to extract several conclusions. First, in the table it is clear that calls to the kernel module have a delay that is *independent* of the size of the application, as expected (columns *v-vii*). Second, both the table and the figure show that delay of the signature-based analysis grows with the size of the `dex` file, to the point of becoming unusable (column *ii*). This was expectable as it converts all the functions in the bytecodes into CFGs, which increase with the size of the code. Third, the table and the figure also show that learning-based analysis grows slowly with the size of the `dex` file, showing that this form of analysis is much simpler and faster than the signature-based (column *iii*). Fourth, they also show that these two delays depend on the size of the `dex` files, not on the size of the APK files, which often contain many files that are not analyzed, e.g., images and video (columns *ii-iii*). Fifth, all mechanisms and their combination seem to be usable, except the form of signature-based analysis we considered.

6.4.2 Macro-benchmarks: DroidPosture in a company

To evaluate the performance of the DroidPosture service in the context of a realistic use case, we measured the total time for the remote server to send a request for posture and to get a reply back from the service, e.g., from the coordination service (see Figure 6.2). We used a LAN network to emulate the case of posture being provided inside a company.

We measured a round trip time (RTT) between our board and the remote server of 0.497 *ms*. We used the same calls as before. In this case, the time for a baseline call was 1.92 *ms*, with standard deviation of 0.096. The results of these experiments are shown in Table 6.3. The trends are essentially the same that were observed with the micro-benchmarks, with the additional delay of the network.

6.5 Security Evaluation

As previously mentioned, the specific modules we implemented in DroidPosture serve mainly to demonstrate the kinds of analysis it can make and that it can support several. Nevertheless, we evaluated experimentally the quality of the detection made by our four modules, which we present here.

We used 500 malware samples from the Drebin datasets [ASH+14]. These datasets contain samples from 179 different malware families collected between August 2010 and October 2012. We balanced the number of samples from different malware families. For benign applications, we randomly downloaded 30 applications from 8 different categories on Google Play Store and verified them through VirusTotal that runs samples through around 10 anti-virus products, in order to get some confidence that they had no malware.[1].

To evaluate the detection performance of the *learning-based detection* mechanism, we randomly split our datasets into a training set (66%) and a test set (33%). The training set was used to determine the classification model, whereas the test set was used for measuring the detection performance. We use as metric *accuracy*, which evaluates the ratio of applications correctly classified (it is given by the number of applications correctly classified as good or bad, divided by the total number of applications evaluated). The result shows that the learning-based mechanisms using *kNN* with $k = 3$ had accuracy of 89.4% with a false positive rate (i.e., percentage of samples wrongly identified as malware) of 4%. The detection performance is relatively good, although our dataset is not large. This suggests that our features effectively model malicious code.

The *signature-based detector* achieves better detection performance for samples that have signatures in the database. To test its performance, we created signatures from over 100 different malware families, such as DroidDream, DroidKungfu, DogoWar and foncy. The signature-based detector was able to detect malware samples from those malware families correctly with approximately 100% accuracy. However, the learning-based mechanism is more effective than the signature-based mechanism for applications that contain unknown malware.

To illustrate the effectiveness of *kernel analysis* modules, we deployed the Mindtrick kernel-level rootkit on our board [Min14]. The Mindtrick rootkit replaces the entry for the read syscall (`sys_read`) to instead point to the address of a malicious function injected into the kernel. It allows attackers to obtain a reverse TCP shell on Android devices. Our kernel analysis module in the *secure world* is able to detect this rootkit by reading each address in the system call table from the *normal world* memory, and compare it with each syscall address listed in `System.map`. It inserts an error for the `sys_read` syscall entry in the posture report.

[1] https://www.virustotal.com

6.6 Summary

This chapter presents the DroidPosture service for mobile devices running SafeCloud-FS's client applications. The service aims to securely detect intrusions in an Android device leveraging the ARM TrustZone extension, and report posture information for external services, such as the coordination service. We implemented a set of application and kernel analysis mechanisms to exemplify the kind of posture assessment that our service can do, although the specific analysis to do are probably specific to different scenarios. The performance of these mechanisms seems to be adequate for many applications, with the exception of the signature-based analysis that is slow for large applications.

7 Conclusion

This deliverable presents SafeCloud's cloud-backed secure file system, SafeCloud-FS. This system is based on SCFS, a modular cloud-backed file system.

SafeCloud-FS contains several new features. First, it stores metadata encrypted, hiding from the clouds information such as the names of the files, the directory tree, and their timestamps (time of creation and of the last change).

Second, it allows verifying the availability and integrity of the files stored at the cloud without downloading it.

Third, it provides a set of mechanisms for client-side protection.

Finally, it may be used with DroidPosture in mobile devices to ensure the user can trust the device.

Bibliography

[ABDL07] Nitin Agrawal, William J Bolosky, John R Douceur, and Jacob R Lorch. A five-year study of file-system metadata. *ACM Transactions on Storage*, 3(3):9, 2007.

[AKK09] Giuseppe Ateniese, Seny Kamara, and Jonathan Katz. Proofs of storage from homomorphic identification protocols. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 319–333, 2009.

[ALPW10] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon. RACS: a case for cloud storage diversity. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, pages 229–240, 2010.

[Alt92] N. S. Altman. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician*, 46(3):175–185, 1992.

[Amaa] Amazon Lambda. `https://aws.amazon.com/lambda/`.

[Amab] Amazon S3. `https://aws.amazon.com/s3/`.

[Amac] Amazon S3 Pricing. `https://aws.amazon.com/s3/pricing/`.

[Amad] Amazon Web Services. `https://aws.amazon.com/`.

[Ama15] EC Amazon. Amazon web services. *Available in: http://aws. amazon.com/es/ec2/(November 2012)*, 2015.

[Anda] Androguard. `http://code.google.com/p/Androguard`.

[Andb] Androwarn. `https://github.com/maaaaz/androwarn`.

[ARM09] ARM. ARM security technology, building a secure system using TrustZone technology. `http://www.arm.com`, 2009.

[ASH+14] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. Drebin: Effective and explainable detection of Android malware in your pocket. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2014.

[BACF08] A. N. Bessani, E. P. Alchieri, M. Correia, and J. S. Fraga. DepSpace: a Byzantine fault-tolerant coordination service. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Systems Conference*, pages 163–176, April 2008.

[BB04] Dan Boneh and Xavier Boyen. Short signatures without random oracles. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 56–73, 2004.

[BCLO09] Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O'Neill. Order-preserving symmetric encryption. In *Proceedings of the 28th Annual International Conference on Advances in Cryptology: The Theory and Applications of*

Cryptographic Techniques, pages 224–241, 2009.

[BCQ+11a] A. N. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa. DepSky: dependable and secure storage in a cloud-of-clouds. *EuroSys'11 Proceedings of the 6th Conference on Computer Systems*, pages 31–46, 2011.

[BCQ+11b] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. DepSky: Dependable and secure storage in a cloud-of-clouds. In *Proceedings of the 6th ACM European Conference on Computer Systems*, EuroSys'11, pages 31–46, 2011.

[BCQ+13] 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*, 9(4):12, 2013.

[BJO09] Kevin D Bowers, Ari Juels, and Alina Oprea. Hail: a high-availability and integrity layer for cloud storage. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 187–198, 2009.

[BMO+14a] A. N. Bessani, R. Mendes, T. Oliveira, N. Neves, M. Correia, M. Pasin, and P. Verissimo. SCFS: A shared cloud-backed file system. In *Proceedings of USENIX Annual Technical Conference*, pages 169–180, 2014.

[BMO+14b] Alysson Bessani, Ricardo Mendes, Tiago Oliveira, Nuno Neves, Miguel Correia, Marcelo Pasin, and Paulo Verissimo. SCFS: a shared cloud-backed file system. In *2014 USENIX Annual Technical Conference*, pages 169–180, 2014.

[BN05] Paulo SLM Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. In *International Workshop on Selected Areas in Cryptography*, pages 319–331, 2005.

[BSF+13] A. N. Bessani, M. Santos, J. Felix, N. F. Neves, and M. Correia. On the efficiency of durable state machine replication. In *Proceedings of the USENIX Annual Technical Conference*, pages 169–180, 2013.

[CFGW11] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in Android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, pages 239–252, 2011.

[CJWH+15] Jae Yoon Chung, Carlee Joe-Wong, Sangtae Ha, James Won-Ki Hong, and Mung Chiang. CYRUS: Towards client-defined cloud storage. In *Proceedings of the 10th ACM European Conference on Computer Systems*, EuroSys'15, 2015.

[DBB+15] Tobias Distler, Christopher Bahn, Alysson Bessani, Frank Fischer, and Flavio Junqueira. Extensible distributed coordination. In *Proceedings of the 10th ACM SIGOPS/EuroSys European Systems Conference*, pages 10:1–10:16, 2015.

[dC14] Nuno Tiago Ferreira de Carvalho. A practical validation of homomorphic message authentication schemes. Master's thesis, University of Minho, 2014.

[Der13] Masih H. Derkani. Hypergeometric.java. `https://github.com/masih/sina/blob/master/src/main/java/DistLib/hypergeometric.java`, 2013.

[DI11] Angelo De Caro and Vincenzo Iovino. jPBC: Java pairing based cryptography. In *Proceedings of the 16th IEEE Symposium on Computers and Communications*, pages 850–855, 2011.

[EGC+10] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, pages 1–6, 2010.

[ENI14] ENISA. Algorithms, key size and parameters report – 2014, November 2014.

[ErJ01] D Eastlake 3rd and Paul Jones. US secure hash algorithm 1 (SHA1). Technical report, 2001.

[Eur14] European Union Agency for Network and Information Security. *Algorithms, Key Size and Parameters Report – 2014*. ENISA, 2014.

[Fil17] Filebench. Filebench. *https://github.com/filebench/filebench*, 2017.

[FIMS17] Geoffrey C Fox, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. Status of serverless computing and Function-as-a-Service (FaaS) in industry and research. *arXiv preprint arXiv:1708.08028*, 2017.

[Gal95] B. Gallmeister. *POSIX. 4 Programmers Guide: Programming for the real world*. O'Reilly, 1995.

[Gel85] David Gelernter. Generative communication in Linda. *ACM Transactions on Programing Languages and Systems*, 7(1):80–112, January 1985.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing*, pages 169–178, 2009.

[Gen14] Genode Labs. ARM TrustZone, an exploration of ARM TrustZone technology. `http://genode.org/news/an-exploration-of-arm-trustzone-technology`, 2014.

[Gre14] Adam Greenberg. Sneaky Android RAT disables required anti-virus apps to steal banking info. SC Magazine, https://www.scmagazine.com/sneaky-android-rat-disables-required-anti-virus-apps-to-steal-banking-info/article/538770/, July 2014.

[GZJS12] Michael C Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 101–112, 2012.

[Has09] Omar Hasan. Paillier.java. `http://liris.cnrs.fr/~ohasan/pprs/paillierdemo/Paillier.java`, 2009.

[HFKT17] R. Halalai, P. Felber, A. M. Kermarrec, and F. Taïani. Agar: A caching system for erasure-coded data. In *Proceedings of the IEEE 37th International Conference on Distributed Computing Systems*, pages 23–33, June 2017.

[HHJ+11] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These aren't the droids you're looking for: retrofitting Android to protect data from imperious applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pages 639–652, 2011.

[Hof08] Daniel V. Hoffman. *Implementing NAP and NAC Security Technologies: The Complete Guide to Network Access Control*. John Wiley & Sons, 2008.

[HS77] James W Hunt and Thomas G Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350–353, 1977.

[HSH+16] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Serverless computation with OpenLambda. *Elastic*, 60:80, 2016.

[HW90] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[K+16] M. Kitagawa et al. Gartner inc. market share: Final PCs, ultramobiles and mobile phones, all countries, 3Q16, 2016.

[KG15] S. P. T. Krishnan and J. L. U. Gonzalez. Google compute engine. In *Building Your Next Big Thing with Google Cloud Platform*, pages 53–81. Springer, 2015.

[KL07] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography: Principles and Protocols*. Chapman & Hall/CRC, 2007.

[KRB+15] A. Kharraz, W. Robertson, D. Balzarotti, L. Bilge, and E. Kirda. 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*, pages 3–24, 2015.

[KS85] V. Kachitvichyanukul and B. Schmeiser. Computer generation of hypergeometric random variates. *Statistical Computation and Simulation*, 22:127–145, 1985.

[Lea05] Neal Leavitt. Mobile phones: the next frontier for hackers? *IEEE Computer*, 38(4):20–23, 2005.

[Lyn07] Ben Lynn. *On the implementation of pairing-based cryptosystems*. PhD thesis, Stanford University, 2007.

[Ma08] D. Ma. Practical forward secure sequential aggregate signatures. In *Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security*, pages 341–352, 2008.

[Mer79] Ralph Charles Merkle. Secrecy, authentication, and public key systems. 1979.

[Min14] Mindtrick rootkit. `https://github.com/ChristianPapathanasiou/defcon-18-Android-rootkit-Mindtrick`, 2014.

[MT07] D. Ma and G. Tsudik. Forward-secure sequential aggregate authentication. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 86–91, 2007.

[MT09] D. Ma and G. Tsudik. A new approach to secure logging. *ACM Transactions on Storage*, 5(1), 2009.

[Nat83] National Computer Security Center. Trusted computer systems evaluation criteria, August 1983.

[P+15] Raluca Ada Popa et al. CryptDB webpage. `https://css.csail.mit.edu/cryptdb/`, 2015.

[PBM+17] R. Pontes, D. Burihabwa, F. Maia, J. Paulo, V. Schiavoni, P. Felber, H. Mercier, and R. Oliveira. SafeFS: A modular architecture for secure user-space file systems (one FUSE to rule them all). In *Proceedings of the 10th ACM International Systems and Storage Conference*, 2017.

[PMP11] Bryan Parno, Jonathan M. McCune, and Adrian Perrig. *Bootstrapping Trust in Modern Computers*. Springer, 2011.

[PP13] Ricardo Padilha and Fernando Pedone. Augustus: scalable and robust storage for cloud applications. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys'13, pages 99–112, 2013.

[PRZB11] Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, pages 85–100, 2011.

[Sam] Samsung. Samsung Knox. `https://www.samsungknox.com`.

[Sch99] B. Schoenmakers. A simple publicly verifiable secret sharing scheme and its application to electronic voting. *Proceedings of the 19th International Cryptology Conference*, pages 148–164, 1999.

[Sha79] Adi Shamir. How to share a secret. *Communications of ACM*, 22(11):612–613, November 1979.

[SJKS17] M. Shirvanian, S. Jareckiy, H. Krawczykz, and N. Saxena. Sphinx: A password store that perfectly hides passwords from itself. In *IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 1094–1104, June

2017.

[SKM$^+$08] P. Sangster, H. Khosravi, M. Mani, K. Narayan, and J. Tardo. Network endpoint assessment (NEA): Overview and requirements. RFC 5209, RFC Editor, June 2008.

[SRSW14] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Using ARM TrustZone to build a trusted language runtime for mobile applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 67–80, 2014.

[SSMJ05] Jeffrey M Stanton, Kathryn R Stam, Paul Mastrangelo, and Jeffrey Jolton. Analysis of end user security behaviors. *Computers & Security*, 24(2):124–133, 2005.

[SvDJO12] Emil Stefanov, Marten van Dijk, Ari Juels, and Alina Oprea. Iris: A scalable cloud file system with efficient integrity checks. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 229–238, 2012.

[SW08] Hovav Shacham and Brent Waters. Compact proofs of retrievability. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 90–107, 2008.

[Vog09] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[WLL15] Boyang Wang, Baochun Li, and Hui Li. Panda: public auditing for shared data with efficient user revocation in the cloud. *IEEE Transactions on Services Computing*, 8(1):92–106, 2015.

[WRLL10] Cong Wang, Kui Ren, Wenjing Lou, and Jin Li. Toward publicly auditable secure cloud data storage services. *IEEE Network*, 24(4):19–24, 2010.

[WWRL10] Cong Wang, Qian Wang, Kui Ren, and Wenjing Lou. Privacy-preserving public auditing for data storage security in cloud computing. In *Proceedings of the IEEE INFOCOM*, pages 1–9, 2010.

[YMHC17] Sileshi D. Yalew, Gerald McGuire, Seif Haridi, and Miguel Correia. T2Droid: A TrustZone-based dynamic analyser for Android applications. In *Proceedings of the 16th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, pages 25–36, August 2017.

[ZJ12] Yajin Zhou and Xuxian Jiang. Dissecting Android malware: Characterization and evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, pages 95–109, 2012.

[ZWZJ12] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*, pages 5–8, 2012.

[ZYTT15a] R. Zhao, Yue, B. Tak, and C. Tang. SafeSky: a secure cloud storage middleware for end-user applications. In *Proceedings of the 34th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 21–30, 2015.

[ZYTT15b] Rui Zhao, Chuan Yue, Byungchul Tak, and Chunqiang Tang. SafeSky: A secure cloud storage middleware for end-user applications. In *Proceedings of the IEEE 34th Symposium on Reliable Distributed Systems*, pages 21–30, 2015.